DBASE ⟶ CLIPPER ⟶ HARBOUR

Introdução a Programação com Harbour

Vlademiro Landim Júnior

2ª Edição - 2021

OBS-1: Esse trabalho pode ser copiado e distribuído livremente desde que sejam dados os devidos créditos. Muitos exemplos foram retirados de outros livros e sites, todas as fontes originais foram citadas (inclusive com o número da página da obra) e todos os créditos foram dados. O art. 46. da lei 9610 de Direitos autorais diz que "a citação em livros, jornais, revistas ou qualquer outro meio de comunicação, de passagens de qualquer obra, para fins de estudo, crítica ou polêmica, na medida justificada para o fim a atingir, indicando-se o nome do autor e a origem da obra" não constitui ofensa aos direitos autorais. Mesmo assim, caso alguém se sinta prejudicado, por favor envie um e-mail para vlademirolandim@gmail.com informando a página e o trecho que você julga que deve ser retirado. Não é a minha intenção prejudicar quem quer que seja, inclusive recomendo fortemente as obras citadas na bibliografia do presente trabalho para leitura e aquisição.

OBS-2: Caso alguém encontre algum erro nesse material envie um e-mail com as correções a serem feitas para vlademirolandim@gmail.com com o título "E-book Harbour". Eu me esforcei para que todas as informações contidas neste livro estejam corretas e possam ser utilizadas para qualquer finalidade, dentro dos limites legais. No entanto, os usuários deste livro, devem testar os programas e funções aqui apresentadas, por sua própria conta e risco, sem garantia explícita ou implícita de que o uso das informações deste livro, conduzirão sempre ao resultado desejado, sendo que há uma infinidade de fatores que poderão influir na execução de uma mesma rotina em ambientes diferentes.

INTRODUÇÃO A PROGRAMAÇÃO COM HARBOUR

Vlademiro Landim Júnior

2021

Sumário

Parte I

Introdução a programação

1 Introdução

> Nós todos pensamos por meio de palavras e quem não sabe se servir das palavras, não pode aproveitar suas ideias.
>
> Olavo Bilac

Objetivos do capítulo

- Descobrir se esse livro foi escrito para você.
- Entender a metodologia de ensino adotada.
- Visualizar o plano geral da obra.
- Ser apresentado a linguagem Harbour.

1.1 A quem se destina esse livro

Este livro destina-se ao iniciante na programação de computadores que deseja utilizar uma linguagem de programação com eficiência e produtividade. Os conceitos aqui apresentados se aplicam a todas as linguagens de programação, embora a ênfase aqui seja no aprendizado da Linguagem Harbour. Mesmo que essa obra trate de aspectos básicos encontrados em todas as linguagens de programação, ela não abre mão de dicas que podem ser aproveitadas até mesmo por programadores profissionais. O objetivo principal é ensinar a programar da maneira correta, o que pode acabar beneficiando ao programador experiente que não teve acesso a técnicas que tornam o trabalho mais produtivo. Frequentemente ignoramos os detalhes por julgar que já os conhecemos suficientemente.

Esse livro, portanto, foi escrito com o intuito principal de beneficiar a quem nunca programou antes, mas que tem muita vontade de aprender. Programar é uma atividade interessante, rentável e útil, mas ela demanda esforço e longo tempo de aprendizado. Apesar desse tempo longo, cada capítulo vem recheado de exemplos e códigos para que você inicie logo a prática da programação. Só entendemos um problema completamente durante o processo de solução desse mesmo problema, e esse processo só se dá na prática da programação.

Por outro lado, existe um objetivo secundário que pode ser perfeitamente alcançado : nas organizações grandes e pequenas existem um numero grande de sistemas baseados em Harbour e principalmente em Clipper. Isso sem contar as outras linguagens

do dialeto *xbase* [1]. Esse livro busca ajudar essas organizações a treinar mão-de-obra nova para a manutenção dos seus sistemas de informações. Se o profissional já possui formação técnica formal em outras linguagens ele pode se beneficiar com a descrição da linguagem através de uma sequencia de aprendizado semelhante a que ele teve na faculdade ou no curso técnico. Dessa forma, esse profissional pode evitar a digitação de alguns códigos e apenas visualizar as diferenças e as semelhanças entre a linguagem Harbour e a linguagem que ele já conhece.

1.2 Pré-requisitos necessários

Como o livro é voltado para o público iniciante, ele não requer conhecimento prévio de linguagem de programação alguma. Porém, algum conhecimento técnico é requerido, por exemplo :

1. familiaridade com o sistema operacional Microsoft Windows: navegação entre pastas, criação de pastas, cópia de conteúdo entre uma pasta e outra, etc.
2. algum conhecimento matemático básico: expressões numéricas, conjunto dos números naturais, inteiros (números negativos), racionais e irracionais. Você não precisará fazer cálculo algum, apenas entender os conceitos.

Os conhecimentos desejáveis são :

1. criação de variáveis de ambiente do sistema operacional: para poder alterar o PATH do sistema.
2. conhecimento de comandos do Prompt de comando do Windows: para poder se mover entre as pastas e ir para o local onde você irá trabalhar.

Caso você não tenha os conhecimentos desejáveis, nós providenciamos um passo a passo durante os primeiros exemplos. Basta acompanhar os textos e as figuras com cópias das telas.

1.3 Metodologia utilizada

Durante a escrita deste livro, procurei seguir a sequencia da maioria dos livros de introdução a programação e algoritmos. Contudo, verifiquei que existem pelo menos dois grupos de livros de informática: o primeiro grupo, o mais tradicional, procura apresentar os tópicos obedecendo uma estrutura sequencial, iniciando com as estruturas mais básicas até chegar nas estruturas mais complexas da linguagem. Pertencem a essa categoria livros como “Conceitos de computação usando C++” (Cay Horstman), “C++ como programar” (Deitel e Deitel), “Princípios e práticas de programação com C++” (Bjarne Stroustrup) , “Clipper 5.0 Básico” (José Antonio Ramalho) e a coleção “Clipper 5” (Geraldo A. da Rocha Vidal). Nesses livros não existem muitas quebras de sequencia, por exemplo: os códigos envolvendo variáveis são muito poucos, até que o tópico variável seja abordado completamente. Essa abordagem pode não agradar a algumas pessoas, da mesma forma que um filme de ação pode entediar o expectador

[1] Flagship, FoxPro, dBase, Sistemas ERPs baseados em *xbase*, XBase++, etc.

se demorar demais a exibir as cenas que envolvem batidas de carro e perseguições. Essa abordagem é a ideal para o estudante que deseja iniciar a leitura de um código entendendo exatamente tudo o que está lá. A grande maioria dos elementos que aparecem nos exemplos já foram abordados em teoria.

Por outro lado, existem os livros do segundo grupo, tais como : “Clipper 5.2” (Rick Spence), “A linguagem de programação C++” (Bjarne Stroustrup) e “C: A linguagem de programação” (Ritchie e Kerninghan). Esses livros enfatizam a prática da programação desde o início e são como filmes de ação do tipo “Matrix”, que já iniciam prendendo a atenção de quem assiste. Essa abordagem agrada ao estudante que não quer se deter em muitos detalhes teóricos antes de iniciar a prática.

O presente livro pertence aos livros do primeiro grupo. Desde o início foi estimulada a prática da programação com vários códigos, desafios e exercícios, sem“queimar etapas”. Algumas exceções podem ser encontradas, por exemplo, ainda no primeiro contato com a linguagem alguns conceitos avançados são vistos, mas de uma forma bem simples e sempre com um aviso alertando que esse tópico será visto mais adiante com detalhes. Os livros que pertencem aos do segundo grupo são ótimos, inclusive citei três nomes de peso que escreveram obras desse grupo : Dennis Ritchie (criador da linguagem C), Bjarne Stroustrup (criador da linguagem C++) e Rick Spence (um dos desenvolvedores da linguagem Clipper). Porém procurei não fugir muito da sequencia dos cursos introdutórios de programação. Se por acaso você não tem experiência alguma com programação e deseja iniciar o aprendizado, essa livro será de grande ajuda para você, de modo que você não terá muitas surpresas com os códigos apresentados. Se você praticar os códigos apresentados, o aprendizado será um pouco lento, mas você não correrá o risco de ter surpresas no seu futuro profissional por usar um comando de uma forma errada. Caso você se desestimule durante o aprendizado isso não quer dizer que você não tem vocação para ser um programador, nem também quer dizer que eu não sei escrever livros de introdução a programação. Não se sinta desestimulado caso isso aconteça, parta para os livros que estimulam uma abordagem mais prática, sem fragmentos isolados de código ou tente outra vez com outra publicação do primeiro grupo.

Portanto, esse livro pretende ser um manual introdutório de programação, ele presume que você não tem conhecimento algum dos conceitos básicos, tais como variáveis, operadores, comandos, funções, loops, etc.

1.4 Material de Apoio

Esse livro baseia-se em uma simples ideia : programar é uma atividade que demanda muita prática, portanto, você será estimulado desde o início a digitar todos os exemplos contidos nele. É fácil achar que sabe apenas olhando para um trecho de código, mas o aprendizado real só ocorre durante o ciclo de desenvolvimento de um código. Digitar códigos de programas lhe ajudará a aprender o funcionamento de um programa e a fixar os conceitos aprendidos. O material de apoio é composto por todos os códigos usados no livro e também por uma listagem parcial do código impresso, ou seja, você não precisará digitar o código completamente, mas apenas a parte que falta para completar a listagem. Cada fragmento que falta no código diz respeito ao assunto que está sendo tratado no momento.

1.5 Plano da obra

O livro é organizado como se segue. O **capítulo 1** é esse capítulo, seu objetivo é passar uma ideia geral sobre as características do livro e dos métodos de ensino adotados. O **capítulo 2** ensina conceitos básicos de computação, compilação e geração de executáveis. O objetivo principal desse capítulo é lhe ensinar a instalar o ambiente de programação e criar alguns programas de teste para verificar se tudo foi instalado corretamente. O **capítulo 3** inicia os conceitos básicos de programação, o objetivo principal é permitir que você mesmo crie seus primeiros programas de computador. São abordados conceitos básicos de indentação, comentários, quebras de linha, padrão de codificação, strings, uso de aspas e estimula a busca de pequenos erros em códigos. O **capítulo 4** aborda as variáveis de memória, ensina alguns comandos básicos de como se receber os dados e trás alguns exemplos bem simples envolvendo operadores e funções básicas . O **capítulo 5** trás os tipos de dados da linguagem e seus respectivos operadores. Aborda também o valor NIL e os SETs da linguagem. Ele é importante, porém os exemplos ainda não são muitos desafiadores. O **capítulo 6** introduz o conceito de algoritmo e inicia um assunto central em qualquer linguagem de programação: as estruturas de controle. Esse capítulo, especificamente, aborda o conceito de algoritmo e aplica esse conceito nas estruturas sequenciais. O **capítulo 7** introduz o uso das estruturas de decisão. As estruturas de decisão são vistas juntamente com os operadores de comparação, de modo a passar para o leitor um conjunto coeso de definições. O **capítulo 8** é uma extensão do capítulo 7, e continua abordando as estruturas de decisões. Nesse capítulo também são abordados as expressões lógicas complexas com os operadores E e OU. O leitor experiente talvez discorde da abordagem que foi utilizada nesses dois capítulos, pois nós apresentamos dois conceitos que, tradicionalmente, são ensinados em separado: os operadores relacionais e as estruturas de decisão. Nós preferimos essa abordagem porque, na prática, esses conceitos dependem um do outro, e o programador iniciante tem alguns problemas para criar situações úteis onde esses conceitos são aplicados [2]. Ao finalizar esse capítulo você já poderá escrever pequenos programas realmente funcionais. O **capítulo 9** introduz o uso das estruturas de repetição, também conhecidas como “laços” ou “loops”. O capítulo aborda essas estruturas através de um problema histórico envolvendo um comando obsoleto chamado de GOTO. Como sempre os algoritmos são usados para a representação dessas estruturas juntamente com o equivalente em Harbour. O **capítulo 10** é inteiramente dedicado as funções. Ele introduz o tema e também trás uma pequena listagem das funções do Harbour e vários exemplos que devem ser praticados, mas você ainda não irá aprender a criar as suas próprias funções. O **capítulo 11** aborda a interface modo texto do Harbour, ensina como funciona o sistema de coordenadas, as cores e os comandos @, que trabalham com coordenadas. Também aborda o sistema de gets da linguagem. O **capítulo 12** introduz o conceito de rotinas e ensina a criar programas com mais de um arquivo. Aqui você aprenderá a criar as suas próprias rotinas. O **capítulo 13** finaliza o aprendizado sobre variáveis, abordando historicamente a solução de problemas com acoplamento de rotinas. Nesse capítulo você irá aprender o que é uma variável public, private, local e static. É importante que você tenha aprendido bem o capítulo anterior para que esse

[2] Eu tive esse problema há mais de vinte anos e ainda me lembro da dificuldade que eu tive para entender a estreita relação entre um tipo de dado lógico, um operador relacional e uma estrutura de decisão.

capítulo seja inteiramente entendido. O **capítulo 14** analisa o sistema de erros do Harbour e nos trás algumas técnicas usadas por programadores profissionais para a manipulação eficiente de erros em tempo de execução. O **capítulo 15** aborda as estruturas complexas da linguagem, como arrays e hashs. Se você acompanhou tudo direitinho até o capítulo 9 não haverá problemas aqui, pois a maioria dos tópicos que ele aborda já foram vistos. O **capítulo 16** é dedicado as macros e aos blocos de código, dois recursos cujo conceito está presente em muitas linguagens atuais. O **capítulo 17** aborda o banco de dados padrão do Harbour, os arquivos DBFs, e alguns comandos para a manipulação desses arquivos. O **capítulo 18** aborda em detalhes o pré-processador do Harbour. O **capítulo 19** apresenta uma pequena introdução ao paradigma orientado a objeto através do desenvolvimento de uma aplicação simples. O **capítulo 20** apresenta algumas dicas para estudo futuro e conclui o livro. O livro também possui uma lista de apêndices que podem ser consultados posteriormente, tais como modelos de fluxogramas, lista de SETs, resumo de programação estruturada, etc.

1.6 Porque Harbour ?

A linguagem Harbour é fruto da Internet e da cultura de software-livre. Ela resulta dos esforços de vários programadores, de empresas privadas, de organizações não lucrativas e de uma comunidade ativa e presente em várias partes de mundo. O Harbour surgiu com o intuito de preencher a lacuna deixada pela linguagem Clipper, que foi descontinuada e acabou deixando muitos programadores orfãos ao redor do mundo. Harbour é uma linguagem poderosa, fácil de aprender e eficiente, todavia ela peca pela falta de documentação e exemplos. A grande maioria das referências encontradas são do Clipper, esse sim, possui uma documentação detalhada e extensa. Como o Harbour foi feito com o intuito principal de beneficiar o programador de aplicações comerciais ele acabou não adentrando no meio acadêmico, o que acabou prejudicando a sua popularização. As universidades e a cultura acadêmica são fortes disseminadores de cultura, opinião e costumes. Elas já impulsionaram as linguagens C, C++, Pascal, Java, e agora, estão encantadas por Python. Por que então estudar Harbour ? Abaixo temos alguns motivos :

1. **Simplicidade e rapidez** : como já foi dito, Harbour é uma linguagem fácil de aprender, poderosa e eficiente. Esses três requisitos já constituem um bom argumento em favor do seu uso.

2. **Portabilidade** : Harbour é um compilador multi-plataforma. Com apenas um código você poderá desenvolver para o Windows, Linux e Mac. Existem também outras plataformas que o Harbour já funciona : Android, FreeBSD, OSX e até mesmo MS-DOS.

3. **Integração com a linguagem C** : internamente um programa escrito em Harbour é um programa escrito em C. Na verdade, para você ter um executável autônomo você necessita de um compilador C para poder gerar o resultado final. Isso confere a linguagem uma eficiência e uma rapidez aliada a uma clareza do código escrito. Além disso é muito transparente o uso de rotinas C dentro de um programa Harbour.

4. **Custo zero de aquisição** : Por ser um software-livre, o Harbour lhe fornece todo o poder de uma linguagem de primeira linha a custo zero.
5. **Constante evolução** : Existe um número crescente de desenvolvedores integrados em fóruns e comunidades virtuais que trazem melhorias regulares.
6. **Suporte** : Existem vários fóruns para o programador que deseja desenvolver em Harbour. A comunidade é receptiva e trata bem os novatos.
7. **Facilidades adicionais** : Existem muitos produtos (alguns pagos) que tornam a vida do desenvolvedor mais fácil, por exemplo : bibliotecas gráficas, ambientes RAD de desenvolvimento, RDDs para SQL e libs que facilitam o desenvolvimento para as novas plataformas (como Android e Mac OSX)
8. **Moderna** : A linguagem Harbour possui compromisso com o código legado mas ela também possui compromisso com os paradigmas da moderna programação, como Orientação a Objeto e programação Multithread.
9. **Multi-propósito** : Harbour possui muitas facilidades para o programador que deseja desenvolver aplicativos comerciais (frente de loja, ERPs, Contabilidade, Folha de pagamento, controle de ponto, etc.). Por outro lado, apesar de ser inicialmente voltada para preencher as demandas do mercado de aplicativos comerciais a linguagem Harbour possui muitas contribuições que permitem o desenvolvimento de produtos em outras áreas: como bibliotecas de computação gráficas, biblioteca de jogos, desenvolvimento web, funções de rede, programação concorrente, etc.
10. **Multi-banco** : Apesar de possuir o seu próprio banco de dados, o Harbour pode se comunicar com os principais bancos de dados relacionais do mercado (SQLite, MySQL, PostgreSQL, Oracle, Firebird, MSAccess, etc.). Além disso, todo o aprendizado obtido com o banco de dados do Harbour pode ser usado nos bancos relacionais. Um objeto de banco de dados pode ser manipulado diretamente através de RDDs, tornando a linguagem mais clara. Você escolhe a melhor forma de conexão.
11. **Programação Windows** : Apesar de criticado por muitos, o Microsoft Windows repousa veladamente nos notebooks de muitos mestres e doutores em Ciência da Computação. A Microsoft está sentindo a forte concorrência dos smartphones e da web, mas ela ainda domina o mundo desktop. Com Harbour você tem total facilidade para o desenvolvimento de um software para windows. Além das libs gráficas para criação de aplicações o Harbour possui acesso a dlls, fontes ODBC, ADO e outros componentes (OLE e ActiveX). Automatizar uma planilha em Excel, comunicar-se com o outros aplicativos que possuem suporte a windows é uma tarefa simples de ser realizada com o Harbour.

Existem pontos negativos ? Claro que sim. A falta de documentação atualizada e a baixa base instalada podem ser fatores impeditivos para o desenvolvedor que deseja ingressar rapidamente no mercado de trabalho como empregado. Esses fatores devem ser levados em consideração por qualquer aspirante a programador. Sejamos claros : se você deseja aprender programação com o único objetivo de arranjar um emprego em pouco tempo então não estude Harbour, parta para outras linguagens mais populares como Java, PHP e C#. Harbour é indicado para os seguintes casos :

1. **O programador da linguagem Clipper** que deseja migrar para outra plataforma (o Clipper gera aplicativos somente para a plataforma MS-DOS) sem efetuar mudanças significativas no seu código fonte.

2. **Profissional que quer ser o dono do próprio negócio desenvolvendo aplicativos comerciais** : Sim, nós dissemos que você pode até desenvolver jogos e aplicativos gráficos com Harbour. Mas você vai encontrar pouco sobre esses assuntos nos fóruns da linguagem. O assunto predominante, depois de dúvidas básicas, são aqueles relacionados ao dia-a-dia de um sistema de informação comercial, como comunicação com impressoras fiscais e balanças eletrônicas, acesso a determinado banco de dados, particularidades de alguma interface gráfica, etc. O assunto pode, até mesmo, resvalar para normas fiscais e de tributação, pois o programador de aplicativos empresariais precisa conhecer um pouco sobre esses assuntos.

3. **Profissional que quer aprender rapidamente** e produzir logo o seu próprio software com rapidez e versatilidade.

4. **O usuário com poucos conhecimentos técnicos de programação** mas que já meche com softwares de produtividade (Excel, Calc, MS-Access, etc.) podem se utilizar do Harbour para adentrar no mundo da programação pela porta da frente em pouco tempo.

5. **O estudante** que quer ir além do que é ensinado nos cursos superiores. Com Harbour você poderá integrar seus códigos C e C++ (Biblioteca QT, wxWidgets, etc.) além de fuçar as “entranhas” de inúmeros projetos livres desenvolvidos em Harbour. Por trás do código simples do Harbour existem montanhas de código em C puro, ponteiros, estruturas de dados, integração com C++, etc.

6. **O aprendiz** ou o *hobbista* que quer aprender a programar computadores mas não quer adentrar em uma infinidade de detalhes técnicos pode ter no Harbour uma excelente ferramenta.

7. **O Hacker**[3] que quer desenvolver ferramentas de segurança (e outras ferramentas de análise) pode obter com o Harbour os subsídios necessários para as suas pesquisas.

Se você deseja aprender a programar e quer conhecer a linguagem Harbour, as próximas páginas irão lhe auxiliar nos primeiros passos.

[3] Hacker é um indivíduo que se dedica, com intensidade incomum, a conhecer e modificar os aspectos mais internos de dispositivos, programas e redes de computadores. Graças a esses conhecimentos, um hacker frequentemente consegue obter soluções e efeitos extraordinários, que extrapolam os limites do funcionamento "normal"dos sistemas como previstos pelos seus criadores. (Fonte : `https://pt.wikipedia.org/wiki/Hacker`. Acessado em 14-Ago-2016) O termo Hacker foi injustamente associado a crimes, invasões e ao esterótipo do jovem com reduzida atividade social. O termo usado para caracterizar os criminosos cibernéticos é *cracker*, suas atividades envolvem : desenvolvimento de vírus, quebra de códigos e quebra de criptografia. Outras categorias confundidas com hackers são : pichadores digitais, ciberterroristas, estelionatários, ex-funcionários raivosos, revanchistas e vândalos.

2 O processo de criação de um programa de computador

Homem de bem. É preciso que não se possa dizer dele nem que é matemático, nem pregador, nem eloquente, mas que é homem de bem. Quando, ao ver um homem, a gente se lembra de seu livro, é mau sinal.

Blaise Pascal - Pensamentos

Objetivos do capítulo

- Entender o que é um computador.
- Compreender o que é um programa de computador.
- Saber a diferença básica entre um montador, um compilador e um interpretador.
- Categorizar a linguagem Harbour entre as linguagens existentes.
- Instalar o material de aprendizado e apoio no seu computador.
- Compilar um exemplo que veio com o material de apoio.

2.1 O que é um programa de computador ?

Para entender o processo de programação, você precisa entender um pouco sobre o que é um computador. O computador é uma máquina que armazena dados, interage com dispositivos e executa programas. Programas são instruções de sequencia e de decisão que o computador executa para realizar uma tarefa. Todos os computadores que você já utilizou, desde um relógio digital até um sofisticado smartphone são controlados por programas que executam essas operações extremamente primitivas. Parece impossível, mas todos os programas de computador que você já usou, todos mesmo, utilizam apenas poucas operações primitivas :

1. **entrada :** consiste em pegar dados do teclado, mouse, arquivo, sensor ou qualquer dispositivo de entrada;
2. **saída :** mostra dados na tela, imprime em duas ou três dimensões, envia dados para um arquivo ou outro dispositivo de saída.
3. **cálculo :** executa operações matemáticas simples.
4. **decisão :** avalia certas condições e executa uma sequencia de instruções baseado no resultado.
5. **repetição :** executa uma ação repetidamente.

Isso é praticamente tudo o que você precisa saber por enquanto. Nos próximos capítulos nós abordaremos cada um desse itens separadamente.

2.2 Linguagens de programação

Chamamos de programador o profissional responsável por dividir um problema da "vida real" em blocos de tarefas contendo as operações citadas na seção anterior. Para realizar esse intento ele utiliza uma linguagem de programação.

Antigamente, programar era uma tarefa muito difícil, porque ela se resumia a trabalhar diretamente com essas operações primitivas. Esse processo era demorado e enfadonho, pois o profissional tinha que entender como a máquina funcionava internamente para depois poder "entrar" com essas *instruções de máquina*. Essas instruções são codificadas como números, de forma que possam ser armazenados na memória. Como se não bastasse, elas diferiam de um modelo de computador para outro, e consistiam basicamente de números em sequencia, conforme o exemplo a seguir :

```
10011001 10011101 01101001 10011001 10011101 11111001
11111001 10011101 10001001 10011001 10011101 10011001
10011111 10011111 11111001 10011001 10011101 11111001
10011001 10011101 10000001 00011001 10011101 11100001
10011001 10011101 01111001 10011001 10011101 00111001
10011001 10011101 11111001 00011001 10011101 11111001
```

Ainda hoje os computadores, todos eles, funcionam através desses códigos. O que mudou foi a forma com a qual eles são gerados. Antes eles eram introduzidos diretamente na memória da máquina, através da alteração na posição de chaves ou "jumpers". Era uma tarefa tediosa e sujeita a erros.

Figura 2.1: Cena do filme "O jogo da imitação"

Com o tempo, os cientistas desenvolveram linguagens de programação cuja função era simplificar a criação desses códigos. Esse tipo de programa, recebeu o nome de "montador" (*assembler*). Os montadores representaram um importante avanço sobre a programação em código de máquina puro, e a linguagem utilizada recebeu o nome de "Assembly"[1]. A sequencia abaixo representa um código em Assembly.

```
move int_rate, %eax
sub 100, %eax
jg  int_erro
```

A linguagem Assembly é classificada como uma linguagem de baixo nível. Essa classificação significa "o quão próximo da linguagem de máquina pura" uma linguagem de programação está. Nesse caso, a relação entre uma instrução em Assembly e uma instrução em código de máquina é praticamente de um para um. Os montadores (assemblers) também possuem "baixa portabilidade", ou seja, elas são dependentes do tipo de computador ao qual ele se destina. Se surgir um novo modelo de computador,

[1]Assembler é a categoria da linguagem, Assembly é o nome da linguagem. Maiores detalhes em `https://crg.eti.br/post/assembly-assembler-e-linguagem-de-maquina/` (Acessado em Jul-2021)

o programa gerado não irá funcionar. Ou seja, existem casos onde para cada modelo de máquina existe uma linguagem Assembly específica.

Com o passar dos anos as linguagens de alto nível surgiram, tornando o processo de programar cada vez mais simples e acessível. Surgiu uma nova categoria de gerador de programa, chamado de "compilador". Isso porque a relação entre o código gerado e o codigo de máquina não era mais de um para um, mas de um para “n”. O nome se justifica, pois uma linha de código representa o compilado de várias linhas em uma linguagem de baixo nível.

Por exemplo, em Harbour, a seguinte instrução gera um código que equivale a várias linhas em uma linguagem de baixo nível. :

```
a := 10
```

Outros exemplos de linguagens de alto nível : C, C++, Pascal, Visual Basic, Java, Clipper, COBOL, C#, etc. Existem também as linguagens de “médio nível” que, como o nome já sugere, é um meio termo entre os dois tipos citados. Nessa categoria, apenas a Linguagem C se encontra[2].

Em resumo, o resultado do trabalho do programador sempre é em linguagem de máquina (que é a única linguagem que um computador entende), e quando ele utiliza uma linguagem de alto nível, muitas operações são simplificadas e até ocultadas. É por isso que o termo “programa compilado em Assembly” não é tecnicamente correto. O correto é : "programa **montado** em Assembly".

2.3 Tipos de linguagens de programação

As linguagens de alto nível também se ramificaram quanto ao seu propósito. Por exemplo, a linguagem C é classificada como uma linguagem de propósito geral, ou seja, com ela nós podemos desenvolver aplicações para qualquer área. Existem programadores C que se especializaram na criação de sistemas operacionais, outros que desenvolvem software para controlar robôs industriais e outros que desenvolvem jogos de computador. Pode-se desenvolver praticamente de tudo com a linguagem C, mas todo esse poder tem um preço : ela é uma linguagem considerada por muitos como de difícil aprendizado. A linguagem Harbour deriva diretamente da linguagem C, mas ela não é assim tão genérica, e também não é de difícil aprendizado. O nicho da linguagem Harbour são aqueles aplicativos comerciais que compõem os sistemas de informação, desde pequenos sistemas para gerir uma pequena empresa até gigantescos ERPs podem ser construídos com Harbour. Você ainda tem o poder da linguagem C embutida nela, o que permite a criação de outros tipos de aplicativos, mas o foco dela é o ambiente organizacional.

Uma outra classificação usada é quanto a forma de execução do código. A grosso modo existem dois tipos de linguagens : a linguagem compilada e a linguagem interpretada[3]. Nós já vimos o que é uma linguagem compilada : o código digitado pelo programador é compilado em um código de máquina. As etapas que todo programador precisa passar para ter o seu código compilado pronto estão ilustrados na figura 2.2.

[2]Essa classificação de médio nível para a Linguagem C foi dada pelos criadores da linguagem. A Linguagem C, portanto, abrange duas categorias : alto nível e médio nível.

[3]Existem nuances entre essas duas classificações, mas ainda é cedo para você aprender sobre isso. Por hora, tudo o que você precisa saber são as diferenças básicas entre um compilador e um interpretador

Figura 2.2: O processo de compilação

O que um compilador faz é basicamente o seguinte : lê todo o código que o programador criou e o transforma em um programa derivado, mas independente do código original. O código que o programador digitou é conhecido como “código fonte” e o código compilado é chamado de “código objeto”. Depois que o código objeto é criado existe uma etapa final chamada de “ligação” (ou *linkedição*). O resultado final desse processo é um novo programa totalmente independente do compilador (esse arquivo é chamado de “executável”). As consequências práticas disso são :

1. O programa final é independente do compilador. Você não precisa instalar o compilador na máquina do cliente para o programa funcionar.
2. O código fonte do programa não fica disponível para o usuário final. Isso confere ao programa uma segurança maior, pois ele não pode ser alterado por alguma pessoa não autorizada.

Um interpretador age de forma semelhante, mas não transforma o código fonte em um código independente. Em suma, ele sempre precisa da linguagem (interpretador) para poder funcionar. Os exemplos mais comuns são as linguagens PHP, ASP e Javascript. As consequências práticas disso são :

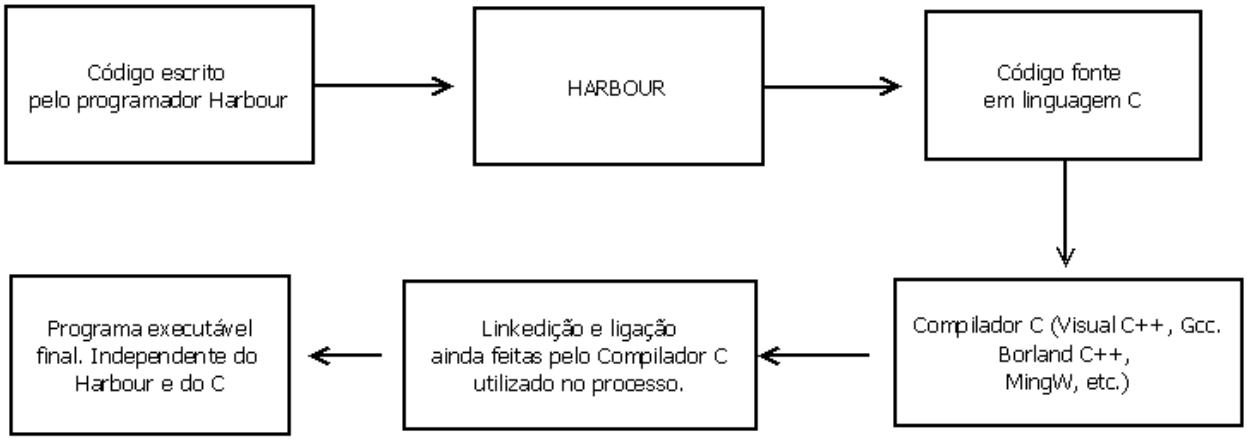
1. O programa final é dependente do interpretador (Você precisa ter o PHP instalado na sua máquina para poder executar programas em PHP, por exemplo)
2. O código fonte do programa fica disponível para o usuário final[4].

Via de regra, um programa compilado é bem mais rápido do que um programa interpretado, e mais seguro também. Ele é mais rápido porque ele é transformado em código de máquina apenas uma vez, durante o ciclo de compilação e linkedição. Já um programa interpretado sempre vai ser lido pela linguagem a qual ele pertence (por exemplo, toda vez que um programa em PHP é executado pelo usuário, ele é lido e checado linha por linha pelo interpretador PHP). Os programas compilados possuem a desvantagem de não serem facilmente portáveis [5] entre plataformas. É mais fácil escrever um programa PHP que funcione em servidores Windows e em Linux do que

[4]Existem programas adicionais que podem obscurecer o código final, mas são etapas adicionais. Via de regra um programa interpretado tem o seu código fonte disponível, se não for protegido por essas ferramentas adicionais

[5]Um programa portável pode funcionar sem problemas em diversos sistemas operacionais, como Windows, Linux, MAC, Android, etc.

Figura 2.3: Processo de geração de um programa escrito em Harbour



escrever um programa em C que funcione nessas duas plataformas citadas. Quando o programa é pequeno a dificuldade praticamente inexiste, mas quando o programa começa a ficar complexo o programador tem que se preocupar mais com a questão da portabilidade.

2.3.1 O Harbour pertence a qual categoria ?

"Escreva uma vez, compile em qualquer lugar"[6] é uma expressão que se encaixa perfeitamente na filosofia da linguagem Harbour. Os executáveis gerados em Harbour podem ser compilados em várias plataformas, dentre elas o Windows, Linux, FreeBSD, MacOS e até MS-DOS. Então o Harbour é um compilador, certo ? Mais ou menos. Na verdade o Harbour não é classificado tecnicamente como um compilador, pois ele depende de um compilador externo para poder gerar o seu programa executável independente. O termo correto para classificar o Harbour seria "tradutor"[7] Se você quer criar um programa executável 100% independente você vai precisar de um compilador C instalado na sua máquina. Isso porque o Harbour gera um outro código fonte que é lido posteriormente pelo compilador C para só depois gerar o código executável. Tecnicamente falando, todo programa executável gerado pelo Harbour é, na verdade, um executável feito em linguagem C. O programador Harbour, sem precisar aprender a linguagem C, gera o resultado final do seu trabalho através dela [8].

Todo esse processo de geração é simplificado para o programador, mas pode ser visualizado através da figura 2.3.

[6]Essa expressão é uma filosofia seguida por diversas linguagens, como Pascal, C++, Ada e C.`https://en.wikipedia.org/wiki/Write_once,_compile_anywhere`

[7]Para alguns autores um programa que faz uma tradução entre linguagens de alto nível é normalmente chamado um tradutor, filtro ou conversor de linguagem.`https://pt.wikipedia.org/wiki/Compilador`(Acessada em Jul-2021)

[8]Apesar de impreciso, usarei o termo "compilar"quando me referir ao processo de geração de programas em Harbour

2.3.2 Instalando o Harbour no seu computador

A instalação do Harbour pode ser feita de duas formas : compilando os fontes do Harbour ou baixando uma versão já compilada por terceiros. Como queremos logo usar o Harbour, iremos adotar o segundo caminho, caso você queira detalhes sobre o processo de geração do Harbour a partir dos fontes basta ver o processo no apêndice G).

Existem diversos locais onde você pode baixar uma versão pré-compilada do Harbour, nós iremos instalar o Harbour de `https://www.hmgforum.com/` porque é uma versão atualizada e usada em produção por inúmeros usuários. As versões pré-compiladas do Harbour acompanham o compilador C utilizado no processo de compilação. O processo de instalação é bem simples, basta ir clicando em next, next, next e finish. Na verdade é uma imprecisão afirmar que o Harbour é um programa instalável. O que o instalador faz é uma simples cópia dos seus arquivos para as pastas selecionadas durante o processo de instalação. O instalador não faz nenhuma alteração no register do Windows ou em qualquer outra configuração do sistema. Ele apenas copia o Harbour e o GCC para determinadas pastas no drive C. Depois disso você ainda vai ter um "trabalho"adicional de incluir os PATHs dos executáveis harbour.exe e gcc.exe na variável PATH do Windows. Faremos tudo isso detalhadamente a seguir.

O nosso roteiro, portanto, segue os seguintes passos :

1. Baixar a versão já compilada do Harbour em `https://www.hmgforum.com`.
2. Executar o instalador.
3. Adicione os paths do harbour.exe e do gcc.exe à veriável de ambiente PATH do Windows.
4. Teste para ver se funcionou.

2.3.3 Baixando o harbour

Usando o seu navegador favorito acesse o site `https://www.hmgforum.com/` e clique no link "Download HMG"[9] conforme a figura 2.4. Esse link o levará a página de downloads onde você deverá baixar a última versão estável.

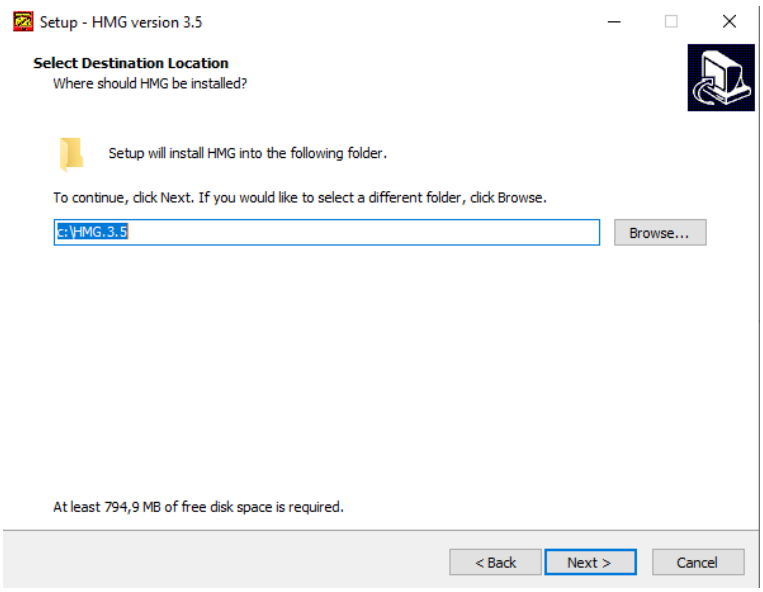
[9]HMG significa "Harbour MiniGUI". Trata-se do harbour acrescido de uma biblioteca desenvolvida por Roberto Lopez, cujo objetivo é permitir a criação de aplicativos "for Windows". Nós não usaremos a biblioteca HMG porque estamos em um momento inicial do nosso aprendizado, de modo que todos os nossos projetos serão em modo texto e compatíveis com outras plataformas (não somente Windows). Nós estamos usando essa distribuição do Harbour porque ela já está compilada e é usada por muitos desenvolvedores em produção.

Figura 2.4: Baixando o Harbour

2.3.4 Executando o instalador

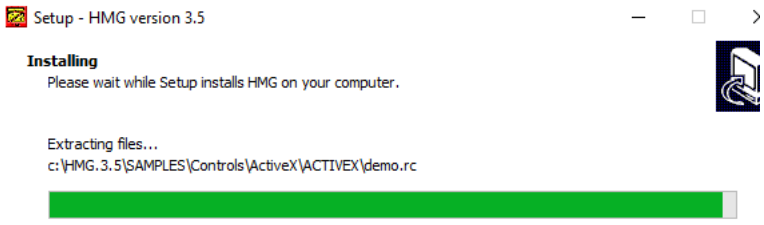
Após o download você deve executar o instalador. A figura 2.5 mostra a escolha da pasta destino (no nosso exemplo a pasta era C:\HMG.3.5). Deixe tudo como está e clique em "Next".

Figura 2.5: Seleção da pasta



Aguarde o processo terminar.

Figura 2.6: Aguarde o término do processo de instalação



2.3.5 Adicione os paths do Harbour e do GCC à variável PATH do Windows

Para que o Harbour funcione corretamente tudo o que você tem que fazer é adicionar os PATHs de onde o Harbour está instalado e também de onde o GCC está instalado. Se você instalou no local padrão, então os PATHs são os seguintes :

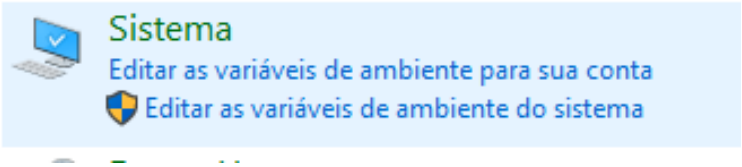
- C:\HMG.3.5\harbour\bin
- C:\HMG.3.5\MINGW\bin

Primeiro acesse o painel de controle do Windows, e clique na caixa de pesquisa no canto superior direito para filtrar uma busca, conforme a figura 2.7. Digite "vari", para filtrar.

Figura 2.7: Buscando as janela de configuração de variáveis do sistema

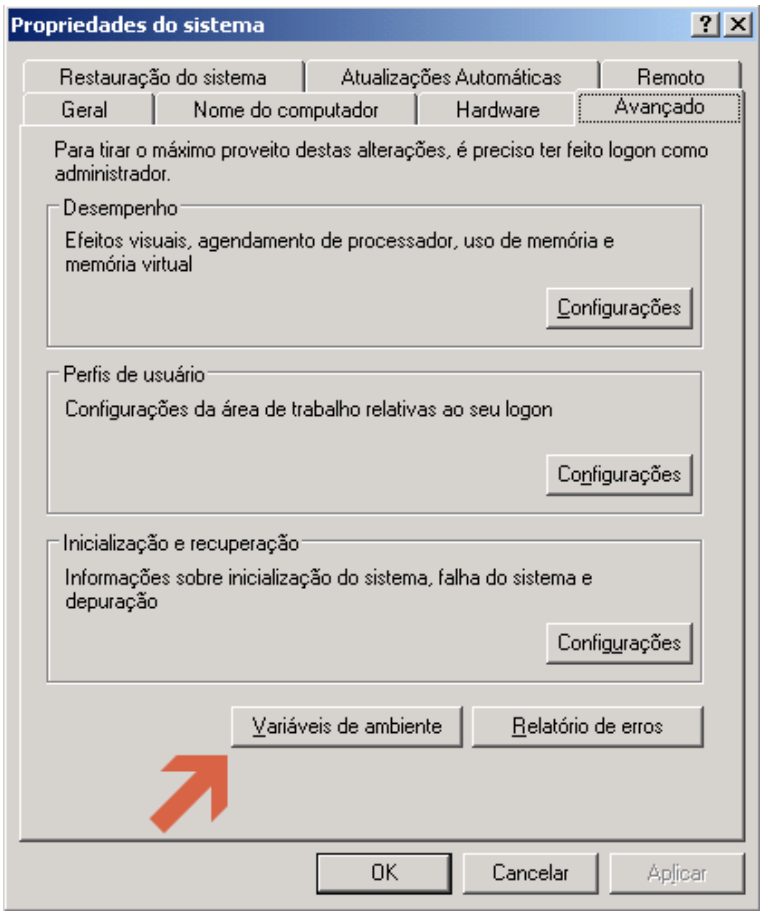
O resultado aparece automaticamente, clique em "Editar as variáveis de ambiente do sistema".

Figura 2.8: Buscando as janela de configuração de variáveis do sistema (parte 2)



Para adicionar essas variáveis de ambiente faça conforme a figura 2.9.

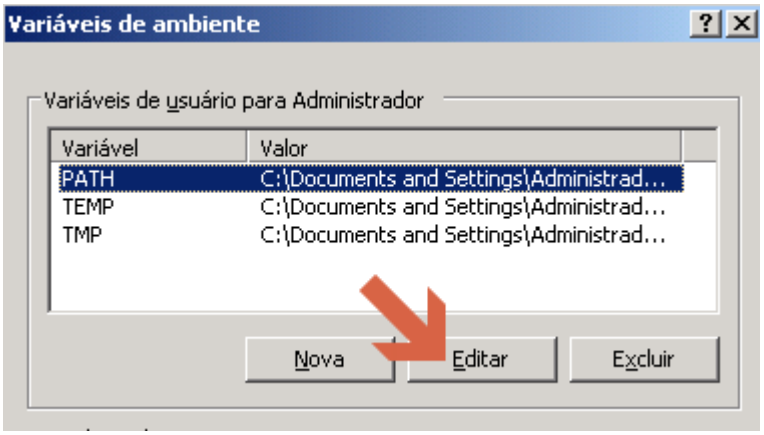
Figura 2.9: Adicionando as variáveis de ambiente



Siga esse procedimento se você é usuário do Windows 7

Após clicar em "Variáveis de ambiente", se você estiver usando o **Windows 7**, a tela é a da figura 2.10. Nesse caso, certifique-se de que a variável PATH está grifada de azul e clique onde a seta está indicando.

Figura 2.10: Adicionando as variáveis de ambiente no Windows 7



Uma janela com o conteúdo existente da variável PATH deve abrir. Vá para o final da linha e acrescente os dados abaixo. Atenção: cuidado para não apagar as variáveis que já estão lá. Vá até o final da linha, **digite um ponto e vírgula no final da linha** e

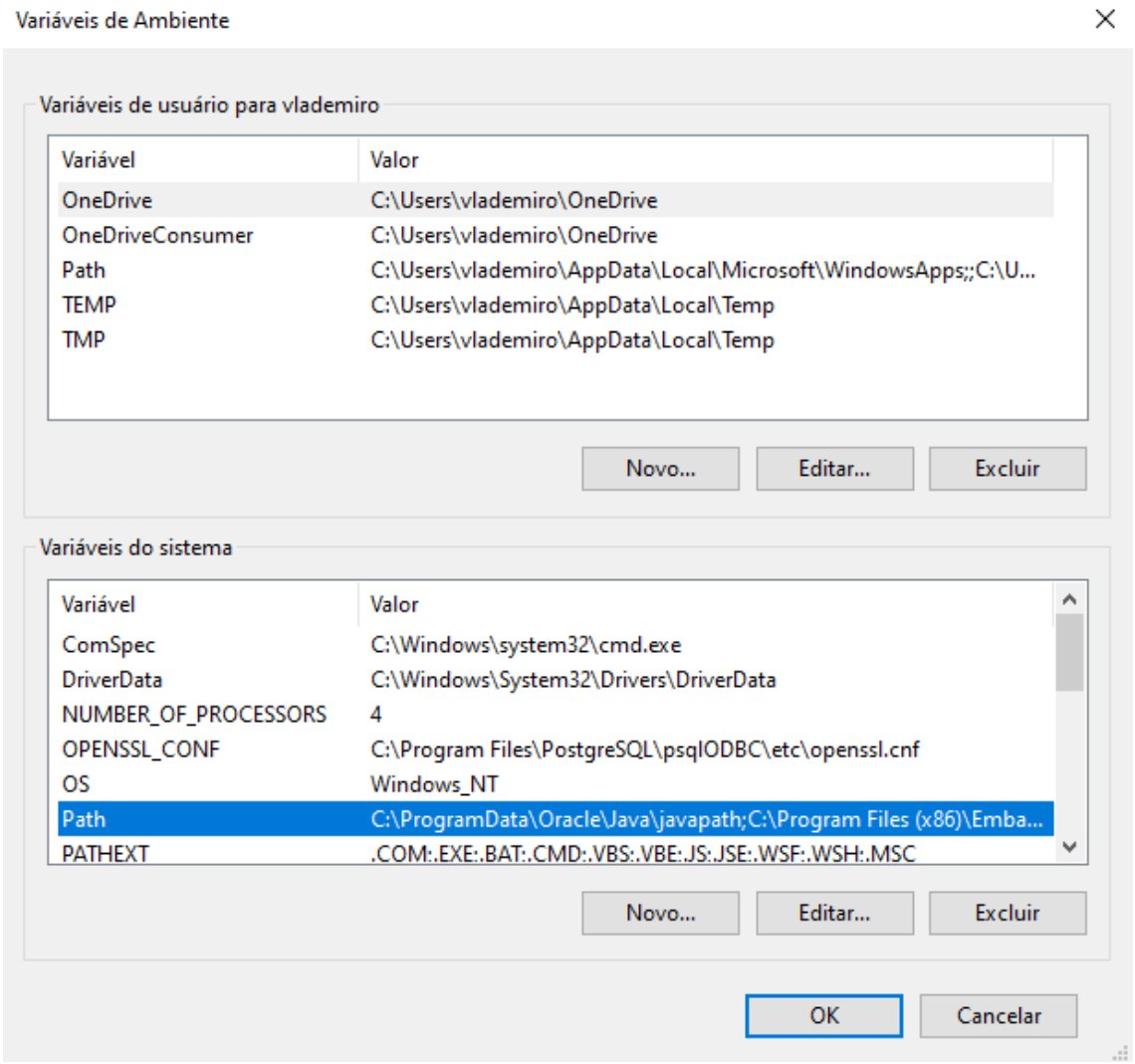
digite o conteúdo baixo :

C:\HMG.3.5\harbour\bin;C:\HMG.3.5\MINGW\bin
Clique em Ok para salvar. Pronto, as variáveis estão configuradas.

Siga esse procedimento se você é usuário do Windows 10

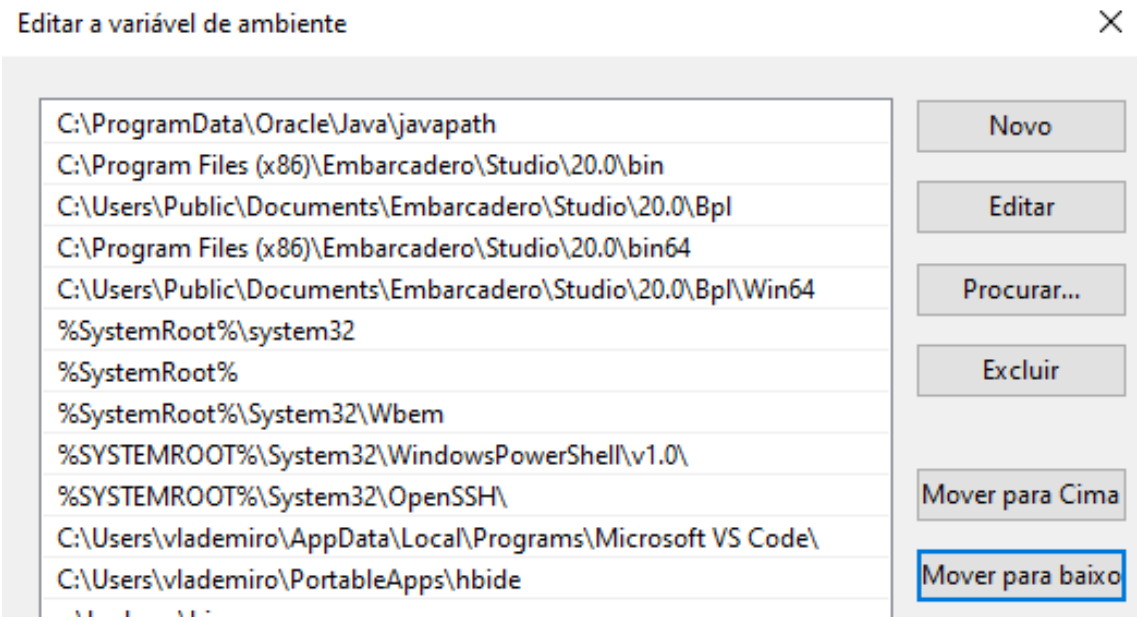
Se você é usuário do **Windows 10**, a tela deve se assemelhar a da figura 2.11. Selecione "Path"para iniciar a edição dessa variável e clique em "Editar...".

Figura 2.11: Alterando a variável de sistema PATH no Windows 10



Uma nova janela surgirá, conforme a figura 2.12.

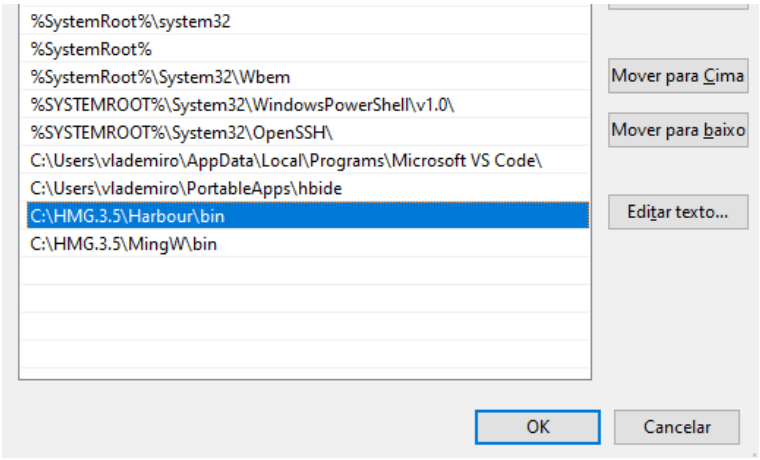
Figura 2.12: Alterando a variável de sistema PATH no Windows 10 - Parte 2



Clique em Novo e digite C:\HMG.3.5\harbour\bin. Clique em para gravar. Clique novamente em Novo e repita o processo para inserir C:\HMG.3.5\MINGW\bin.

O estado final da sua janela deve se parecer com a figura 2.13.

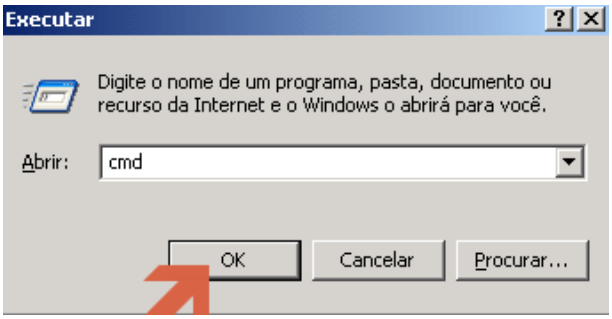
Figura 2.13: Alterando a variável de sistema PATH no Windows 10 - Parte 3



Testando se o Harbour foi instalado corretamente

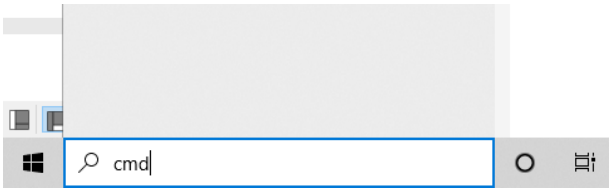
Se você estiver usando o Windows 7, abra o prompt do sistema operacional, conforme a figura 2.14.

Figura 2.14: Abrindo o prompt de comando do windows



Se você é usuário do Windows 10 digite cmd na sua barra de pesquisas e tecle enter, conforme a figura 2.15.

Figura 2.15: Abrindo o prompt de comando do windows no Windows 10



Pronto, temos o prompt de comando aberto. Agora digite “harbour” (sem as aspas). A figura logo abaixo é uma cópia das primeiras linhas que aparecem, mas a listagem é maior.

.:Resultado:.

```
Harbour 3.2.0dev (r2011030937)
Copyright (c) 1999-2020,
https://harbour.github.io/
Syntax:  harbour <file[s][.prg]|@file>[options]
```

Faça a mesma coisa com o compilador da linguagem C: Digite gcc e tecle enter.

.:Resultado:.

```
gcc: fatal error: no input files
compilation terminated.
```

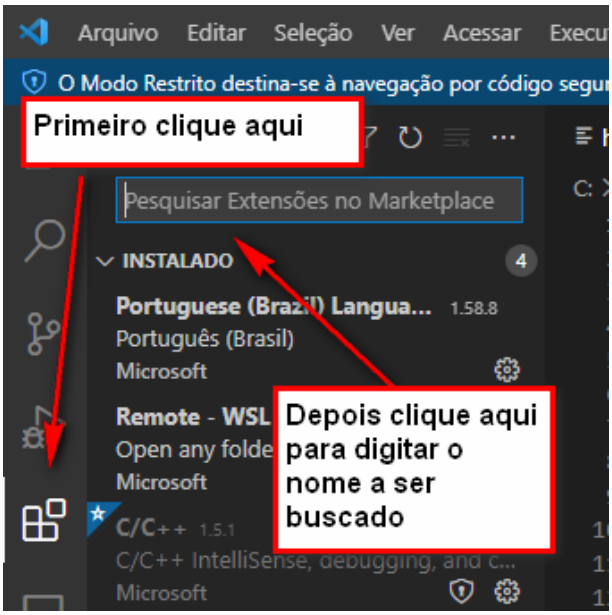
Pronto, o compilador gcc também está instalado.

2.3.6 Instalando um editor de programas

Para a digitação dos códigos você pode usar o Visual Studio Code com uma extensão específica para a linguagem Harbour. O Visual Studio Code pode ser baixado de `https://code.visualstudio.com/`. Após a instalação, você deverá instalar uma extensão para que o código Harbour receba cores e também para antecipar determinados erros de compilação.

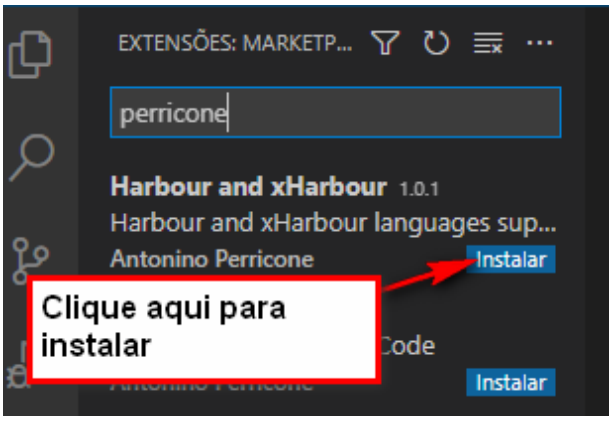
Primeiramente abra o editor e clique em "Extensões", conforme a figura 2.16.

Figura 2.16: Instalando uma extensão para o Harbour



O nome a ser buscado é "Perricone", esse é o sobrenome do autor da extensão do VSCode para Harbour. Faça conforme a figura 2.17.

Figura 2.17: Instalando uma extensão para o Harbour II



Uma dica: você pode adicionar o Path do Visual Studio Code à variável PATH do Windows, da mesma forma que fizemos com o Harbour e com o GCC. Dessa forma você pode "chamar"o editor da linha de comando apenas digitando "code ."(sempre digite o ponto, porque ele significa o diretório corrente). Dessa forma o VS Code irá abrir com uma lista de arquivos no seu lado esquerdo. Assim fica melhor para selecionar determinado arquivo.

Bem, se tudo correu bem até agora você deve ter :

1. O compilador Harbour instalado
2. O compilador C (gcc) instalado
3. O Visual Studio Code com a extensão para Harbour

2.3.7 Dica para uso do prompt de comando

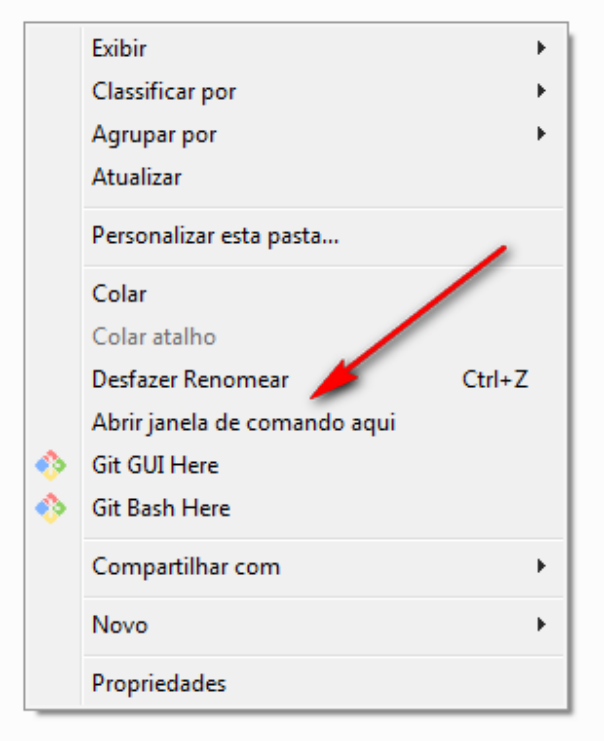
A dica a seguir vai para quem precisa abrir uma janela do Prompt de Comando a partir do Gerenciador de Arquivos do Windows já na pasta corrente.

Se você é usuário do Windows 7 use esse método

Use o Gerenciador do Windows de arquivos para ir até a pasta desejada. Dentro da pasta siga o procedimento abaixo :

1. Certifique-se de não ter nenhum arquivo selecionado;
2. pressione a tecla Shift (fica abaixo da tecla Caps Lock, e é representada por uma seta);
3. ainda sem soltar a tecla Shift, clique com o botão direito do mouse em algum local da pasta (não sobre qualquer arquivo, o clique precisa ser em algum local "em branco").
4. Um menu vai aparecer conforme a figura 2.18.
5. Selecione a seguinte opção : "Abrir janela de comando aqui".

Figura 2.18: Acesso rápido ao Prompt de Comando através do Gerenciador de Arquivos

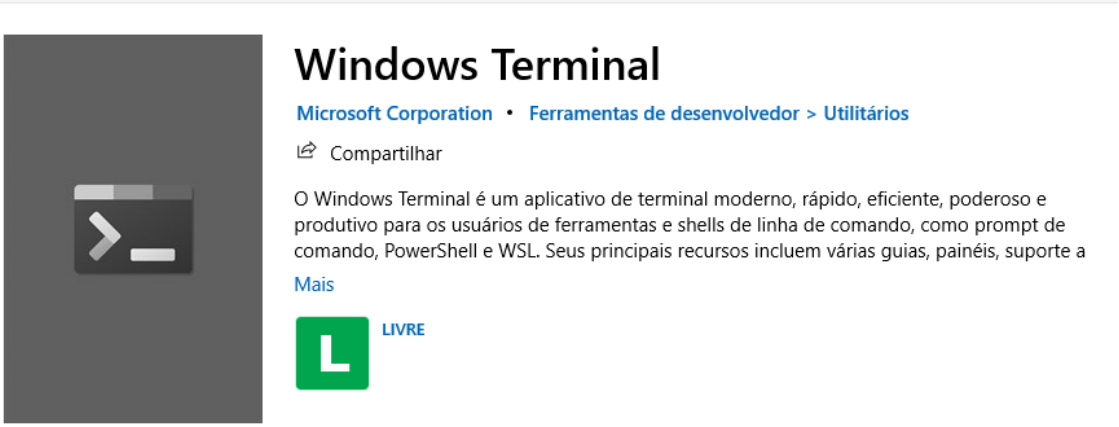


Se você é usuário do Windows 10 use esse método

O procedimento é praticamente o mesmo do Windows 7 (por favor leia a subseção anterior), porém o Prompt padrão é o Powershell.

Muitos usuários não gostam desse novo Prompt, mas saiba que o velho Prompt de Comando ainda existe, ele só não aparece mais no menu. Para que o antigo prompt apareça no menu você deve instalar um aplicativo a partir da loja da Microsoft: o Windows Terminal.

Figura 2.19: Instale o Windows Terminal a partir da Loja da Microsoft

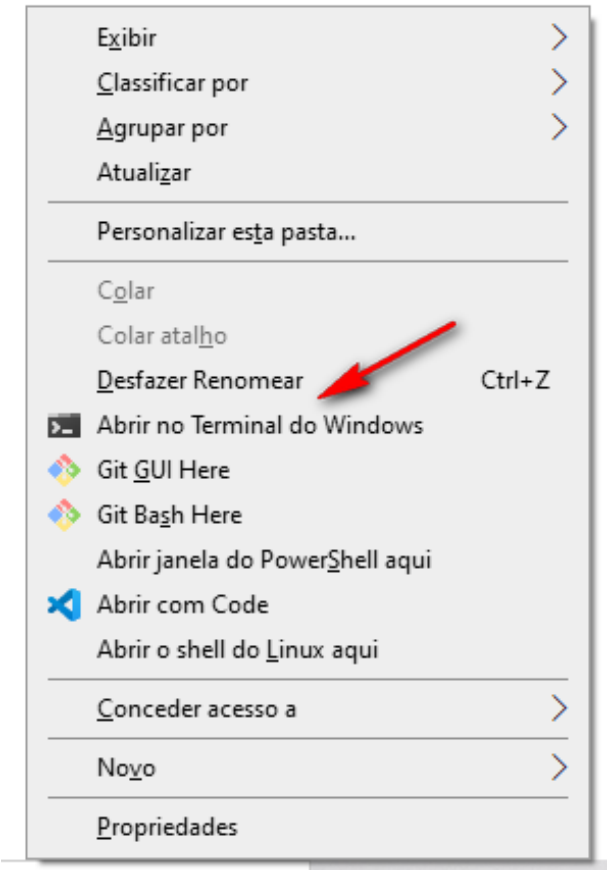


Esse aplicativo é interessante porque ele agrupa em um só aplicativo os quatro terminais mais usados pelo Windows :

1. o Powershell;
2. o Bash (caso você tenha o Subsistema Linux instalado);
3. o Azure
4. o bom e velho Prompt de Comando

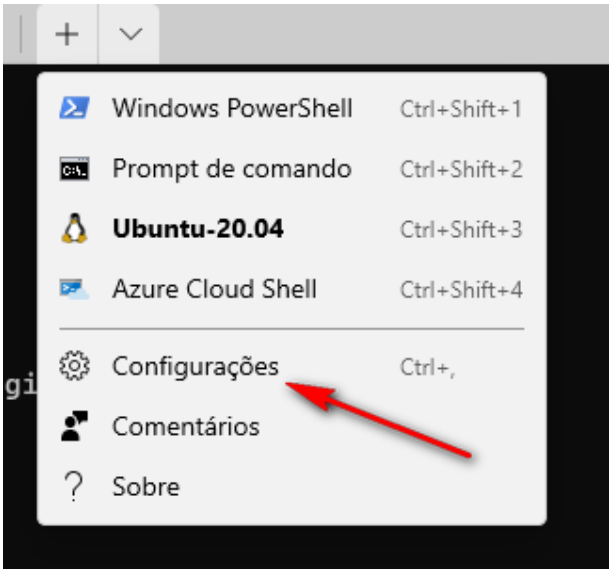
Após a instalação você poderá abrir a pasta corrente no Prompt de Comando (figura 2.20).

Figura 2.20: Opção do Windows Terminal no Menu



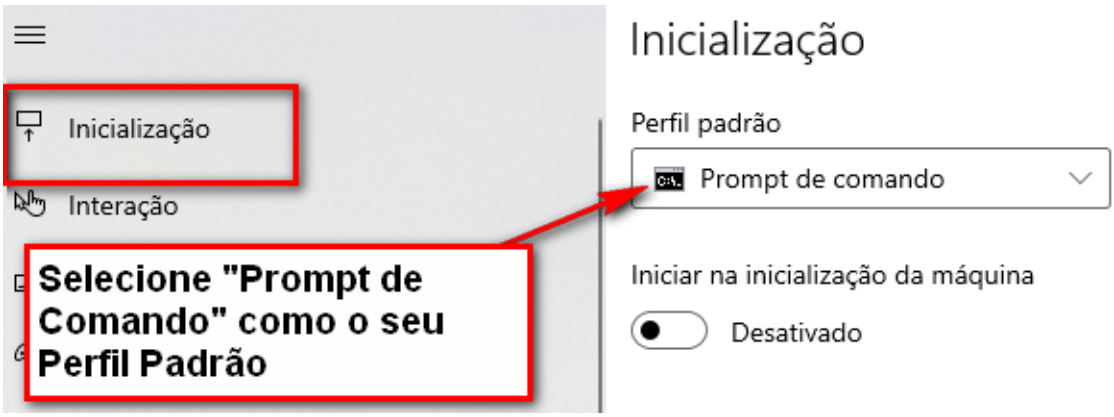
Você poderá configurar esse terminal clicando na parte superior da janela, conforme a figura 2.21.

Figura 2.21: Configuração do Windows Terminal



As opções de configuração são diversas, mas apenas coloque o "Prompt de Comando"como seu perfil padrão, conforme a figura 2.22

Figura 2.22: Configurando o Windows Terminal



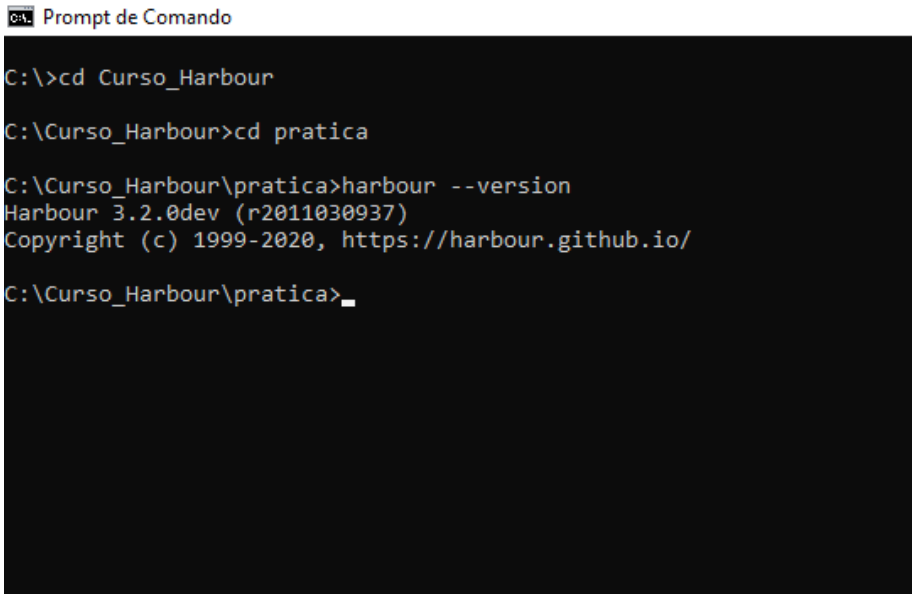
2.3.8 Compilando o código

Agora vamos entender o processo de compilação de um programa em Harbour.

Primeiramente baixe os exemplos desse ebook de `https://drive.google.com/drive/folders/0B0Uz0QRph4mbMzlVMTFCU0VkRUk?resourcekey=0-t7CE_mjXC4wSV8g9TMvM-w&usp=sharing`.

Esse arquivo é do tipo zip, e você pode descompactá-lo na raiz do seu sistema. Dentro desse arquivo tem uma pasta chamada Curso_Harbour, e dentro dela tem duas outras pastas, uma chamada "codigo", com todo o código fonte dos exemplos desse livro, e outra chamada "pratica". Você deve usar a pasta "pratica"para digitar seus testes, note que dentro dessa pasta tem um arquivo chamado "HelloWorld.prg"e outro chamado "hbmk.hbm", não se preocupe em entender o significado deles por enquanto. Acostume-se a usar o prompt de comando para trabalhar nessas pastas. **Um lembrete importante:** não edite os arquivos da pasta "codigo", use-os somente para consulta. Caso sinta necessidade, faça uma cópia.

Figura 2.23: Acostume-se com o Prompt de Comando



Para compilar um programa em Harbour existe um utilitário chamado *hbmk2* que faz

todo o processo descrito no capítulo anterior[10]. Esse programa não faz a compilação, mas gerencia o Harbour e todos os programas envolvidos durante o processo. Digite no prompt de comando *hbmk2* e tecle enter. Algo semelhante com a figura abaixo deve surgir (não mostramos tudo na figura abaixo, apenas as linhas iniciais).

.:Resultado:.

```
Harbour Make (hbmk2) 3.2.0dev (r2015-07-03 09:22)
Copyright (c) 1999-2013, Viktor Szakáts
http://harbour-project.org/
Translation (pt-BR): Vailton Renato <vailtom@gmail.com>
```

Dentro da pasta “codigo” existe um arquivo chamado HelloWorld.prg, agora iremos compilar[11] esse arquivo.

Digite hbmk2 HelloWorld e tecle enter [12]

.:Resultado:.

```
hbmk2: Processando script local: hbmk.hbm
hbmk2: Harbour: Compilando módulos...
Harbour 3.2.0dev (r1507030922)
Copyright (c) 1999-2015, http://harbour-project.org/
Compiling 'HelloWorld.prg'...
Lines 5, Functions/Procedures 1
Generating C source output to '.hbmk\win\mingw\HelloWorld.c'... Done.
hbmk2: Compilando...
hbmk2: Linkando... HelloWorld.exe
```

O processo de compilação é simples e rápido, mas durante esse processo, muitas operações aconteceram. Vamos analisar a saída do comando hbmk2.

- **hbmk2: Processando script local: hbmk.hbm** : O sistema de compilação do Harbour possui inúmeras opções de linha de comando. Não vamos nos aprofundar nessas opções por enquanto. O que queremos salientar é que essas opções podem ficar arquivadas em um arquivo chamado *hbmk.hbm*. Nós já providenciamos esse arquivo com algumas das opções mais usadas, por isso o compilador exibiu essa mensagem.
- **Compiling 'HelloWorld.prg'...** : A fase de compilação do Harbour.
- **Lines 15, Functions/Procedures 1** : Ele analisou o arquivo e encontrou 5 linhas e uma Procedure ou Função (mais na frente veremos o que é isso).
- **Generating C source output to ...** : Ele informa onde gerou o arquivo para o compilador C trabalhar. Não devemos nos preocupar com esse arquivo. O Harbour mesmo o coloca em um lugar isolado, dentro de uma pasta que ele mesmo criou.

[10]Não use o executável harbour.exe para compilar seus programas, como já dissemos, o Harbour é, na verdade, um tradutor, e não um compilador. O executável hbmk2.exe é quem faz todo o processo descrito anteriormente, desde a tradução até o processo de compilação real feito pelo compilador C e a linkedição final

[11]Chamaremos de "compilação"o processo todo, desde a tradução até a linkedição. Na vida real é assim que chamamos a geração de programas que usam compiladores.

[12]Com o passar dos exemplos nós omitiremos detalhes básicos, como teclar enter após a digitação no prompt de comando.

- **hbmk2: Compilando...** : A fase onde o compilador C entra em ação.
- **hbmk2: Linkando... HelloWorld.exe** : A fase final, de linkagem e consequente geração do executável.

2.3.9 Conclusão

Pronto, concluímos um importante processo no entendimento de qualquer linguagem de programação. No futuro, se você for estudar outra linguagem que use um compilador, os processos serão bem semelhantes. O próximo passo é a criação do seu primeiro programa em Harbour.

3 Meu primeiro programa em Harbour

A caminhada é feita como quem cuida de um doente querido, com calma e delicadeza, com paciência e dedicação.

Luiz Carlos Lisboa

Objetivos do capítulo

- Entender a estrutura básica de um programa de computador.
- Criar um programa simples para exibição de mensagens.
- Adquirir o hábito de realizar pequenas alterações nos seus códigos com o intuito de aprender mais.
- Aprender o básico sobre funções.
- Entender as regras de manipulação de caracteres.
- Utilizar comentários nos códigos.
- Entender a lógica por trás das “quebras de linha”.

3.1 De que é feito um programa de computador ?

Todo programa de computador constitui-se de uma sequência lógica de instruções que realizam tarefas específicas. Essas sequências ficam agrupadas dentro de conjuntos ou blocos de instruções chamados de “rotina”. Nós, seres humanos, costumamos usar a expressão “hábito” para expressar uma rotina qualquer de nossas vidas, por exemplo : escovar os dentes após acordar pela manhã, trocar de roupa para ir trabalhar, dirigir um carro, etc.

Dentro de cada rotina nossa existem ações que executamos em uma sequência lógica. Da mesma forma, um programa de computador é composto por um conjunto de rotinas, cada uma delas engloba um bloco de instruções que são executados sequencialmente. Se nós fossemos um computador, a rotina “trocar uma lâmpada” poderia ser descrita assim :

```
ROTINA Trocar uma lâmpada

   1. Buscar uma lâmpada nova.
   2. Pegar uma escada.
   3. Posicionar a escada embaixo da lâmpada a ser trocada.
   4. Subir na escada
   5. Retirar a lâmpada velha
   6. Colocar a lâmpada nova

FINAL DA ROTINA
```

A pequena rotina acima, retirado de [Forbellone e Eberspacher 2005, p. 4], nos mostra a ideia simples por trás de um programa de computador. Essa rotina baseia-se em uma estrutura simples chamada de “sequencial”. No decorrer desse livro, nós estudaremos as outras duas estruturas que compõem a programação estruturada de computadores. Por enquanto, vamos estudar os passos necessários para podermos criar programas simples baseados na estrutura sequencial.

3.2 O mínimo necessário

A linguagem Harbour possui dois tipos de rotinas : os *procedimentos* e as *funções*. Mais na frente, dedicaremos um capítulo inteiro ao estudo dessas rotinas, mas por enquanto, tudo o que você deve saber é que esses blocos de instruções agrupam todos os comandos de um programa de computador. Uma analogia interessante é comparar o desenvolvimento de um programa à leitura de um romance: a grande maioria dos romances está organizada em capítulos. Ao iniciarmos a leitura, começamos pelo capítulo 1 e seguimos em sequencia até o final do livro. Em um programa de computador cada capítulo corresponde a uma rotina, porém a ordem com que essas rotinas são executadas (a ordem de leitura) é determinada por uma rotina especial chamada de “rotina principal” [1].

[1] Em inglês a tradução da palavra “principal” é “main”.

De acordo com [Damas 2013, p. 9] um programa é uma sequência lógica organizada de tal forma que permita resolver um determinado problema. Dessa forma, deve existir um critério, ou regra, que permita definir onde o programa irá começar. Esse "critério", o qual Luis Damas se refere, é a existência da rotina principal. Mas como o computador sabe qual é a rotina principal ?

No caso do Harbour, existe uma rotina (procedimento) onde são colocadas todas as instruções que devem ser executadas inicialmente. Essa rotina [2] chama-se *Main*, e todo bloco a executar fica entre o seu início (PROCEDURE Main) e o comando RETURN, que indica o final da rotina.

Listagem 3.1: O primeiro programa
Fonte: codigos/HelloWorld.prg

```
PROCEDURE Main


RETURN
```

O programa escrito em 3.1 é completamente funcional do ponto de vista sintático, porém ele não executa tarefa alguma. Com isso queremos dizer que o mínimo necessário para se ter um programa sintaticamente correto é a procedure Main[3].

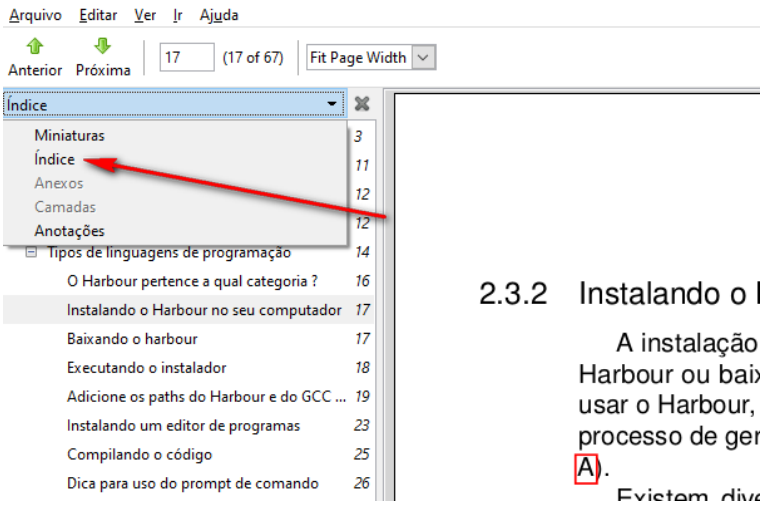
3.2.1 Dica para quem está lendo esse e-book no formato PDF

Eu sou do tempo em que a gente digitava todos os códigos dos exemplos. Digitar faz parte do aprendizado. Mas eu também reconheço que existem pessoas que não querem digitar os códigos. Eu respeito isso. Talvez você já seja um programador experiente, ou talvez queira realizar um teste rápido em apenas um trecho do código. Caso o seu leitor de PDF não permita selecionar o código, saiba que esse problema não é do documento, mas uma limitação do seu leitor. Eu utilizo um leitor chamado Evince. Ele é um software-livre, e tem versões para Windows e Linux. Recomendo esse leitor, caso o seu não permita a seleção dos códigos.

O Evince tem uma característica interessante. No canto esquerdo tem a estrutura do documento com todos os capítulos, seções e subseções. Esse recurso é útil para um deslocamento rápido entre os tópicos. Caso você não consiga ver essa estrutura, basta selecionar o modo de visualização na parte superior, conforme a figura 3.1.

[2] Note que nos exemplos nós usamos a palavra reservada PROCEDURE para iniciar a nossa rotina.
[3] De agora em diante nós chamaremos os procedimentos de "procedures".

Figura 3.1: Selecionando a estrutura do documento



3.3 O primeiro programa

Vamos agora criar o nosso primeiro programa em Harbour. O exemplo a seguir (listagem 6.1) exibe a frase “Hello World” na tela. O símbolo “?” chama-se comando, e é ele quem avalia o conteúdo da expressão especificada e o exibe.

Listagem 3.2: Hello World
Fonte: codigos/HelloWorld1.prg

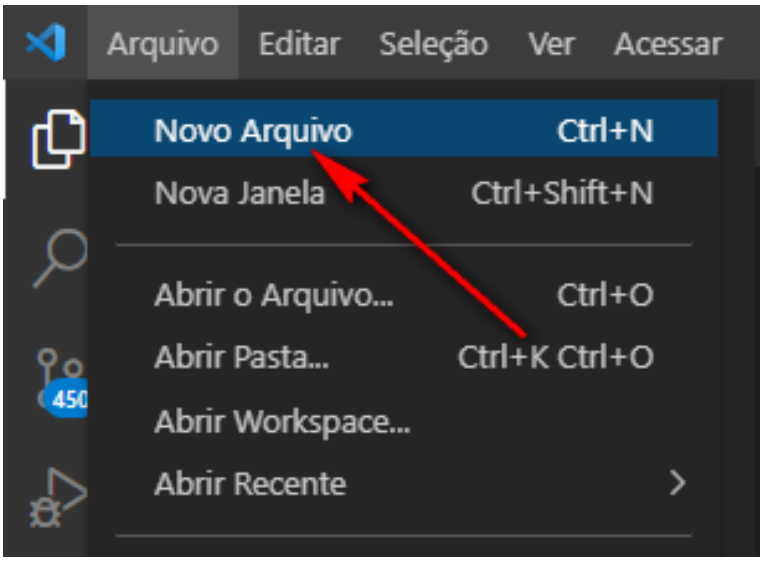
```
PROCEDURE Main

    ? "Hello World"

RETURN
```

Digite o programa usando o seu editor preferiro[4] e compile o programa usando o *hbmk2*.

Abra o seu editor, clique no menu Arquivo, e selecione a opção "Novo Arquivo", conforme a figura 3.2.

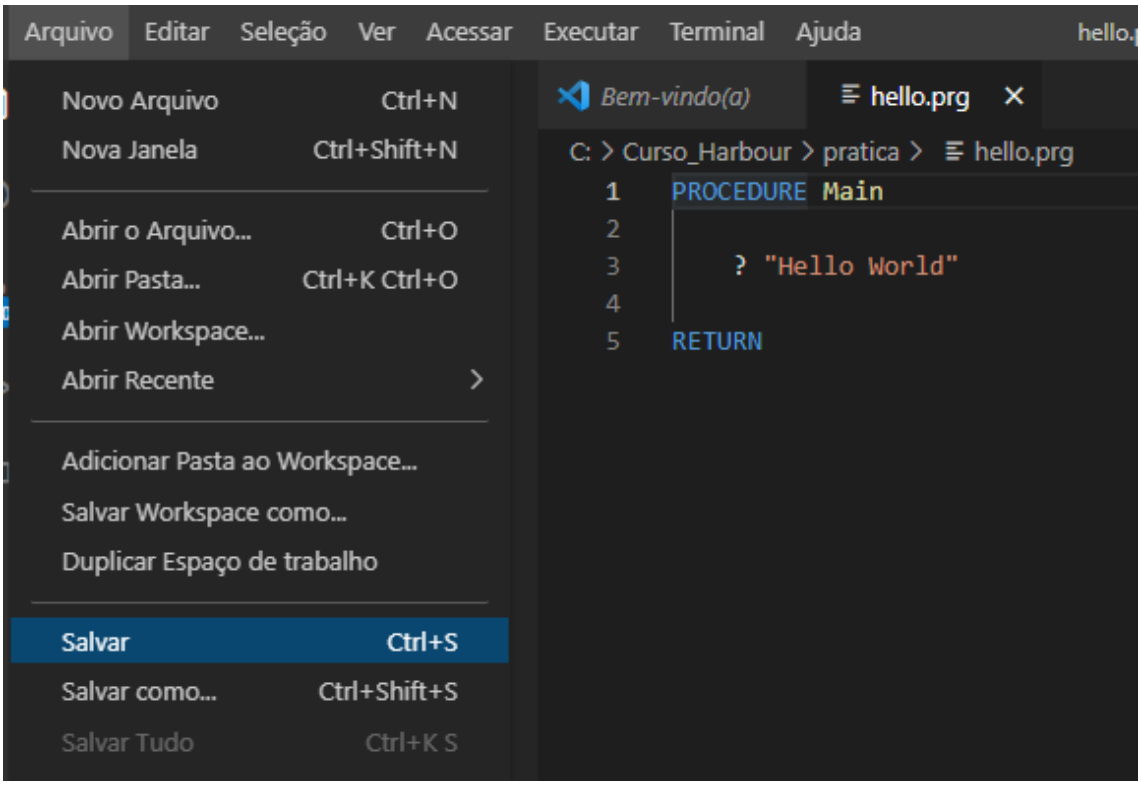
[4]Nas capturas de tela nós usamos o VS Code

Figura 3.2: Novo arquivo pelo Visual Studio Code



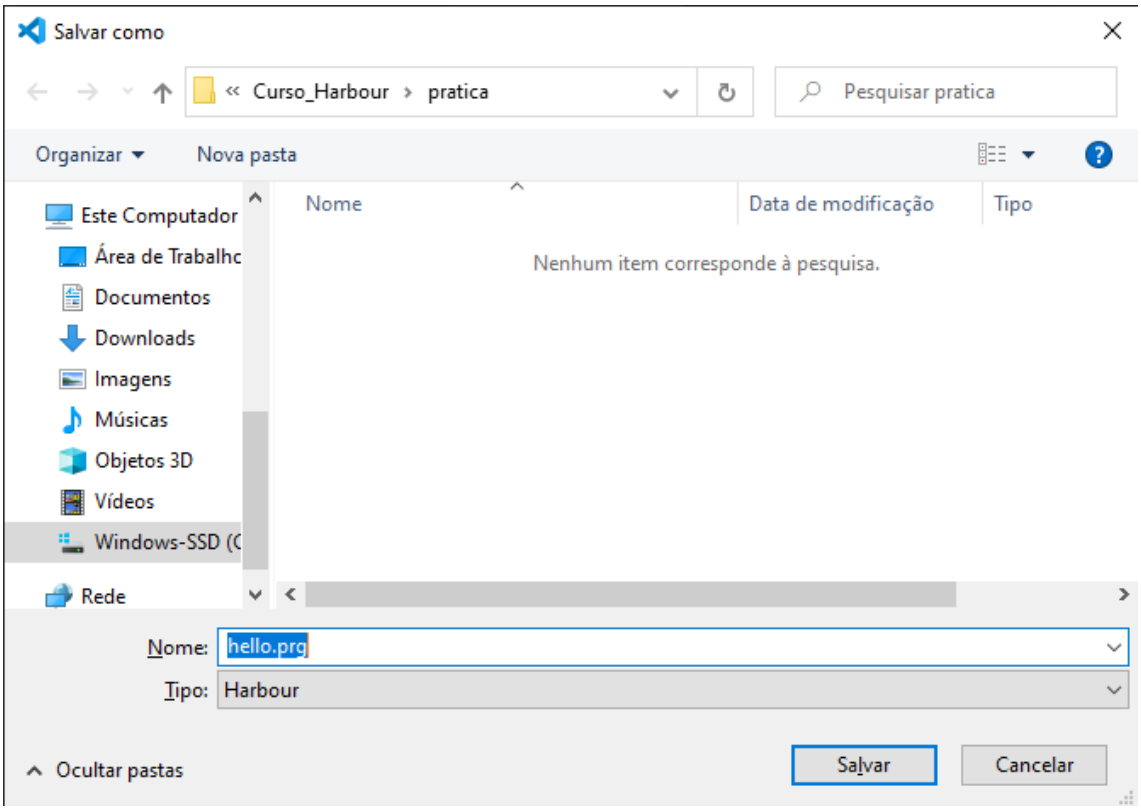
Inicie a digitação do programa da listagem 6.1. Quando concluir a digitação salve o arquivo, conforme a figura 3.3.

Figura 3.3: Salvando o arquivo



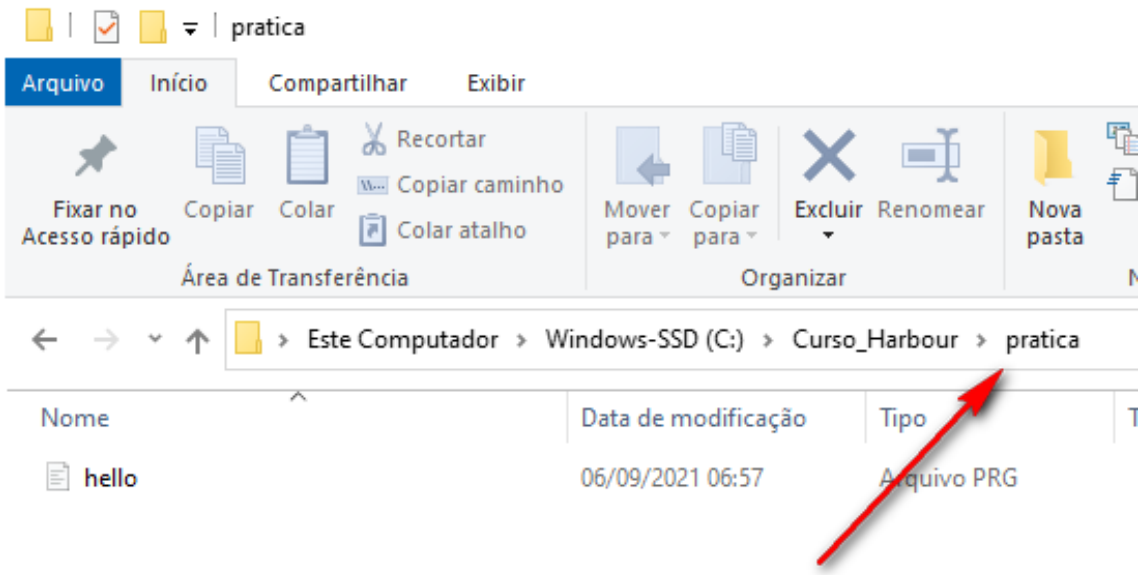
Quando for salvar dê-lhe o nome “hello.prg” e salve-o **na pasta pratica**, conforme a figura 3.4. A extensão “.prg” é a extensão padrão dos códigos de programação escritos em Harbour. Sempre que você criar um arquivo salve-o com a extensão “.prg”.

Figura 3.4: Salvando o arquivo (Parte II)



Agora use o gerenciador de arquivo do Windows e vá até a pasta "pratica"conforme a figura 3.5.

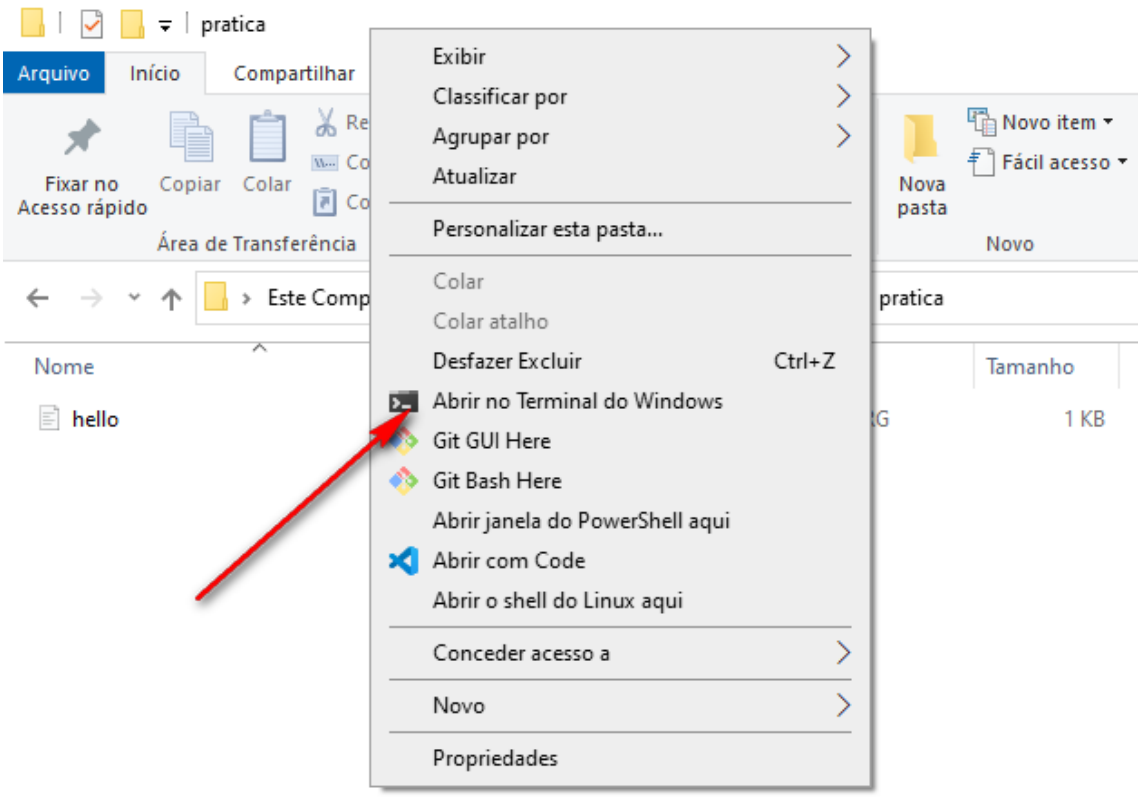
Figura 3.5: Indo para a pasta prática



Clique com o botão direito do mouse em alguma área livre da pasta. Nenhum arquivo deve estar selecionado, clique na área "branca"da pasta "pratica", conforme a

figura 3.6. Essa figura foi retirada de uma máquina com Windows 10, **para usuários do Windows 7, você deve pressionar a tecla Shift enquanto clica com o botão direito do mouse**. Qualquer coisa retorne ao capítulo anterior e reveja o último tópico.

Figura 3.6: Indo para a pasta prática



Se você **não deseja** usar o Explorador de Arquivos, basta entrar no Prompt de comando e usar o comando "cd", conforme a listagem abaixo. O comando "cd"significa "Change Directory"(Muda Diretório), e é usado para mudar de uma pasta (diretório) para outra usando o Prompt de comando.

.:Resultado:.

```
C:\> cd \Curso_Harbour\pratica
C:\Curso_Harbour> cd pratica
```

Digite o comando hbmk2 para gerar o programa. Note que não precisa digitar a extensão do arquivo.

.:Resultado:.

```
C:\Curso_Harbour\pratica> hbmk2 hello
```

Após o processo de compilação você deve executar o programa digitando o seu nome e teclando enter. Conforme a figura abaixo :

.:Resultado:.

```
C:\Curso_Harbour\pratica> hello
```

O programa deve exibir a saída :

.:Resultado:.

```
Hello World
```

Pronto, você gerou o seu primeiro programa em Harbour. Use essa seção como tira-dúvidas pois nós vamos omitir todos esses passos e telas quando formos compilar os outros exemplos. Nos demais exemplos nós apenas mostraremos a listagem para você digitar e uma cópia simples da tela (sem os detalhes do Prompt de Comando) com a respectiva saída.

Apesar de pequeno, esse programa nos diz algumas coisas.

1. Na linha 1 temos o início do programa que é indicado pelo simbolo **PROCEDURE Main**.
2. Na linha 3 temos o símbolo “?” que é o responsável pela exibição de dados na tela do computador.
3. Ainda na linha 3 nós temos a frase “Hello World”. Que é o conteúdo que será exibido na tela.

Faça algumas experiências: tente recompilar o programa da listagem 6.1 mas com algumas pequenas alterações. Os criadores de uma das mais utilizadas linguagens de programação (a linguagem C) afirmam que o caminho para se aprender uma linguagem qualquer é treinar escrevendo códigos. Eles acrescentam que devemos experimentar deixar de fora algumas partes do programa e ver o que acontece [Kernighan e Ritchie 1986, p. 20]. Vamos então seguir essa dica e realizar algumas alterações. Na lista abaixo nós temos cinco dicas de alterações que podemos realizar, apenas tome o cuidado de fazer e testar uma alteração por vez. Não realize duas ou mais alterações para verificar o que mudou pois você poderá não saber o que causou o comportamento. Vamos tentar, então de um por um:

1. Na linha 1 troque a palavra Main por mAin.
2. Na linha 3 troque a frase “Hello World” por outra frase qualquer.
3. Na linha 3 escreva novamente “Hello World”, mas não coloque o último parêntese.
4. Na linha 5 troque RETURN por RETORNO
5. Na linha 5 troque RETURN por RETU

Como é a primeira vez, vamos a seguir comentar os resultados desse teste e acrescentar algumas explicações adicionais.

1. **Teste 1:** Nada aconteceu. Isso porque a linguagem Harbour **não faz distinção entre maiúsculas e minúsculas**. Linguagens assim são chamadas de **Case Insensitive**, sendo portanto a mesma coisa escrever main, Main, mAin ou MAIN.

2. **Teste 2:** O conteúdo que está entre aspas é livre para você escrever o que quiser, e a linguagem Harbour irá exibir **exatamente o que está lá**. Se você escreveu tudo com letras maiúsculas, então o computador irá exibir tudo em maiúsculo, e assim por diante.

3. **Teste 3:** Se você abriu uma aspa mas não fechou (não importa se foi a primeira ou a última) ou se você esqueceu as aspas, então o programa não irá ser gerado. Um erro será reportado.

4. **Teste 4:** Se você trocou RETURN por RETORNO o programa irá exibir um erro e não será gerado. Isso significa que os símbolos que **não estão** entre aspas exigem uma digitação correta deles.

5. **Teste 5:** Se você trocou RETURN por RETU o programa irá funcionar perfeitamente. Isso acontece porque **você não precisa escrever o nome do comando completamente**. Bastam os quatro primeiros dígitos. **Mas não faça isso nos seus programas**. Habitue-se a escrever os nomes completos, pois isso facilitará o seu aprendizado, além de tornar os seus programas mais claros para quem os lê[5]

Note que, se você fez o teste, algumas alterações ficaram sem explicação. Por exemplo, no item cinco, a troca de RETURN por RETU não alterou em nada a geração do programa, mas você provavelmente não iria conseguir explicar a causa sem ajuda externa. Somente com a leitura adicional você ficou sabendo que a causa é uma característica da linguagem Harbour que emula as antigas linguagens Clipper e dBase. O que nós queremos dizer com isso é que: **muito provavelmente você não irá conseguir explicar a causa de todas as suas observações sozinho**. Alguém terá que lhe contar. Mas isso não deve lhe desanimar, pois quando estamos descobrindo uma linguagem nova, muitas perguntas ficarão sem resposta até que você avance alguns capítulos e descubra a causa.

Dica 1

Habitue-se a digitar o nome completo das diversas instruções do Harbour.

[5] Essa característica estranha da linguagem Harbour nasceu do seu compromisso de ser compatível com a linguagem Clipper. O Clipper, assim como o dBase, que nasceu na década de 1980, podia reconhecer um comando apenas pelos quatro primeiros caracteres. Essa característica tornava o texto dos programas (o código fonte) menor e economizava espaço em disco (1 byte significava muito nessa época). Hoje em dia não precisamos mais dessa economia porque os dispositivos de armazenamento de hoje possuem uma capacidade muito superior aos antigos disquetes.

Dica 2

Note que o código da listagem 6.1 possui a parte central "recuada" em relação ao início (*PROCEDURE Main*) e o fim do bloco (*RETURN*). Esse recuo chama-se "indentação"[a] e cada marca de tabulação (obtida com a tecla Tab) representa um nível de indentação. Essa pratica não é nenhuma regra obrigatória, mas ajuda a manter o seu código legível. Tome cuidado com o editor de texto que você utiliza para programar, pois os espaços entre as tabulações podem variar de editor para editor. Fique atento ao tamanho das tabulações no seu editor de texto, as tabulações são úteis para facilitar a marcação das indentações. Geralmente os editores possuem uma opção [b] que permite alterar o tamanho da tabulação. Alguns editores possuem ainda uma opção extra que converte as tabulações automaticamente em espaços, e nós consideramos uma boa ideia habilitar essa opção, caso ela exista. Nos exemplos desse livro as tabulações possuem o tamanho igual a três espaços.

[a] Essa palavra é um neologismo que tem a sua origem na palavra inglesa "indentation". Maiores detalhes em `http://duvidas.dicio.com.br/identacao-ou-indentacao/` e em `https://en.wikipedia.org/wiki/Indentation_(typesetting)` (Acessado em 20-Ago-2016)

[b] Na maioria dos editores essa opção fica no menu *Preferences* , *Preferências* ou *Opções gerais*.

Dica 3

Se você for do tipo que testa tudo (o ideal para quem quer realmente aprender programação), você deve ter percebido que eu menti quando disse que o mínimo necessário para se ter um programa sintaticamente completo é a PROCEDURE Main (Listagem 3.1). Na verdade, um programa é gerado mesmo sem a procedure Main. Mesmo assim você deve se habituar a criar a procedure Main, pois **isso irá garantir que o programa irá iniciar por esse ponto (o Harbour procura por uma procedure com esse nome, e quando acha inicia o programa por ela.)**. Caso você não crie essa procedure, você correrá o risco de não saber por onde o seu programa começou caso ele fique com muitos arquivos. Portanto : **crie sempre uma procedure Main nos seus programas**, mesmo que eles tenham apenas poucas linhas. Isso irá lhe ajudar a criar um bom hábito de programação e lhe poupará dores de cabeça futuras. Programas sem a rotina principal não constituem bons exemplos para o dia a dia. Como nós queremos começar corretamente, vamos sempre criar essa procedure.

A seguir (listagem 3.3) temos uma pequena variação do comando "?", trata-se do comando "??". A diferença é que esse comando não emitirá um *line feed* (quebra de linha) antes de exibir os dados, fazendo com que eles sejam exibidos na linha atual. O efeito é o mesmo obtido com o programa da listagem 6.1.

Listagem 3.3: O comando ??
Fonte: codigos/HelloWorld3.prg

```
PROCEDURE Main()

   ?? "Hello "
   ?? "World"
```

```
                                                                    5
RETU                                                                6
```

.:Resultado:.

```
Hello World
```

Usaremos esses dois comandos para exibir os dados durante esse livro pois é a forma mais simples de se visualizar um valor qualquer. Esses dois comandos admitem múltiplos valores separados por uma vírgula, conforme a listagem 3.4.

Listagem 3.4: O comando ? e ??
Fonte: codigos/oi.prg

```
/*                                                                  1
Comandos ? e  ??                                                    2
*/                                                                  3
PROCEDURE Main                                                      4
                                                                    5
   ? "A festa foi boa."                                             6
   ? "O valor da compra foi de R$" , "4 reais"                      7
                                                                    8
RETURN                                                              9
```

.:Resultado:.

```
A festa foi boa.
O valor da compra foi de R$ 4 reais
```

Dica 4

Habitue-se a criar uma lista de trechos de códigos prontos para poupar tempo na digitação de trechos repetitivos. Esses trechos de códigos são conhecidos como *Snippets* [a]. Um *snippet* é simplesmente um trecho de código que nós salvamos em um arquivo e para para economizar tempo e trabalho na digitação. Um exemplo de um *snippet* :

```
PROCEDURE Main                                                      1
                                                                    2
                                                                    3
RETURN                                                              4
```

Portanto: habitue-se a criar sempre seus próprios trechos de código com passagens repetitivas. Você pode usar o trecho acima para facilitar a digitação.

[a] A palavra *snippet* surgiu como um recurso que a Microsoft disponibilizou nos seus editores de código. Porém, a ideia não é da Microsoft. Ela apenas automatizou um bom hábito que todo programador deve ter.

Prática número 1

Abra o arquivo pratica_hello.prg que está na pasta pratica e complete a digitação de modo a ficar conforme a listagem a seguir :

Listagem 3.5: pratica_hello.prg
Fonte: codigos/pratica_hello.prg

```
PROCEDURE Main

   ? "============================================="
   ? "= PEDIDO DE VENDA N. 12                    ="
   ? "============================================="
   ? " PRODUTO           QTD    VALOR        TOTAL  "
   ? " ============   ======  =======   ==========  "
   ? " Blusa GP Gde       12   150,00      1800,00  "
   ? " Camisa Regat        2    25,00        50,00  "
   ? "                                   ----------  "
   ? "                         TOTAL       1850,00  "
   ? "============================================="

RETURN
```

O resultado deve se parecer com o a tela abaixo :

.:Resultado:.

```
=============================================
= PEDIDO DE VENDA N. 12                     =
=============================================
 PRODUTO           QTD    VALOR        TOTAL
 ============   ======  =======   ==========
 Blusa GP Gde       12   150,00      1800,00
 Camisa Regat        2    25,00        50,00
                                   ----------
                         TOTAL       1850,00
=============================================
```

3.3.1 Classificação dos elementos da linguagem Harbour

Uma linguagem de programação, assim como os idiomas humanos (inglês, português, etc.), possuem uma forma correta de escrita. Por exemplo, se você escrever a frase “Os mininos jogaram bola até mais tarde” você comete um erro, pois a palavra “meninos” não foi escrita corretamente. Nossa capacidade intelectual é muito superior a capacidade de um computador, por isso quem lê a frase citada consegue entender, já que o nosso intelecto consegue contextualizar e deduzir a mensagem. Todavia um computador não possui esse poder, portanto as suas instruções precisam ser escritas corretamente e algumas regras devem ser seguidas.

As regras de sintaxe da linguagem

Uma criança não precisa aprender gramática para poder ler e escrever corretamente. O seu cérebro é tão sofisticado que consegue abstrair determinadas regras de linguagem que são consideradas complexas se colocadas no papel. Quando nós estamos aprendendo uma linguagem de programação algo parecido acontece pois, com o aprendizado que vem da digitação e aplicação dos exemplos, nós conseguimos compreender a maioria das regras gramaticais da linguagem de programação. Por esse motivo nós não iremos listar as "regras gramaticais" da linguagem de programação Harbour, pois tal aprendizado tem se mostrado ineficaz. O que nós vamos aprender nesse momento é uma notação, ou seja, uma forma padronizada para se definir o uso de uma estrutura qualquer da linguagem Harbour. Uma notação não é uma regra de programação, pois ela não pertence a linguagem Harbour. Ela é usada somente para transmitir uma ideia através da documentação. Por exemplo, a notação do comando "?", que você já sabe usar, pode ser expresso conforme o quadro a seguir.

Descrição sintática 1

1. Nome : ?
2. Classificação : comando.
3. Descrição : exibe um ou mais valores no console (tela).
4. Sintaxe

```
? [ <lista de expressões>]
```

Fonte : [Nantucket 1990, p. 4-1]

Vamos comentar o quadro acima. O item um é simplesmente o nome da estrutura da linguagem, o item dois é a classificação do item, no caso do "?" ele é classificado como um comando da linguagem (não se preocupe pois aos poucos nós iremos abordar os outros tipos : declarações, funções, e diretivas de pré-processador). O item três é uma breve descrição. O item quatro da notação com certeza não foi entendido, caso você seja realmente um iniciante na programação. Vamos explicar esses elementos a seguir:

1. Colchete [] : Os colchetes representam, na nossa notação, os elementos opcionais. Tudo o que estiver entre colchetes é opcional. Isso quer dizer que o comando "?" irá funcionar isolado, sem nada. De fato, se você usar somente o comando "?" uma linha em branco será gerada e o programa irá passar para a linha seguinte.
2. Símbolo <> : Os sinais de menor e maior envolvem elementos fornecidos pelo usuário. No caso, o usuário pode fornecer uma lista de expressões.

A notação acima contempla todos os casos, por exemplo :

```
PROCEDURE Main
                                                                    1
                                                                    2
```

```
   ?                                                                    3
   ? "Claudiana"                                                        4
   ? "Roberto" , 12                                                     5
                                                                        6
RETURN                                                                  7
```

Essa notação é útil quando queremos comunicar de forma clara aos outros como um elemento “funciona”, ou algo assim como o seu “manual de uso”. Durante esse livro nós usaremos esses quadros de sintaxe diversas vezes.

3.3.2 Uma pequena pausa para a acentuação correta

Considere o seguinte código abaixo :

Listagem 3.6: Exibindo acentos
Fonte: codigos/oi2.prg

```
                                                                        1
                                                                        2
PROCEDURE Main                                                          3
                                                                        4
   ? "João chegou cedo hoje ", "mas Tainá chegou atrasada."             5
                                                                        6
RETURN                                                                  7
```

Se você digitou, compilou e executou o programa da listagem 3.6 provavelmente você teve problemas com a exibição de palavras acentuadas. Você deve ter notado que a acentuação não apareceu corretamente e no lugar da letra surgiu um símbolo estranho.

Isso aconteceu devido a codificação incompatível entre o editor que você utilizou e a linguagem Harbour. Esse problema pode ser resolvido da seguinte forma:

Listagem 3.7: Exibindo acentos
Fonte: codigos/oi3.prg

```
REQUEST HB_CODEPAGE_UTF8  // <--- Insira essa linha                    1
PROCEDURE Main                                                          2
                                                                        3
   hb_cdpSelect( "UTF8" ) // <---- Insira essa linha                    4
   ? "João chegou cedo hoje ", "mas Tainá chegou atrasada."             5
                                                                        6
RETURN                                                                  7
```

Nós iremos omitir nos códigos de exemplos essas linhas que corrigem a acentuação. Esse problema, contudo, não é da linguagem Harbour. Caso você queira explicações adicionais sobre a acentuação, você pode consultar o apêndice **??**.

Dica 5

Problemas com acentuação estão presentes em todas as linguagens de programação. Isso porque o problema realmente não está na linguagem em si, mas nas diversas formas de codificação de texto. A linguagem Harbour, no seu formato padrão possui uma codificação herdada da linguagem Clipper, mas existem formas de você trocar essa antiga codificação por uma moderna, como o formato UTF-8.
Resolver esse problema é fácil, basta inserir o seguinte comando dentro de MAIN

```
hb_cdpSelect("UTF8")
```

Nas próximas seções desse capítulo nós iremos fazer um pequeno *tour* por algumas características da linguagem Harbour (na verdade de toda linguagem de programação). Procure realizar todas as práticas e não pular nenhum exercício.

3.4 As strings

Até agora nós usamos frases entre aspas para serem exibidas com o comando "?", mas ainda não definimos essas tais frases. Pois bem, um conjunto de caracteres delimitados por aspas recebe o nome de *string*. Sempre que quisermos tratar esse conjunto de caracteres devemos usar aspas simples ou aspas duplas [6] , apenas procure não misturar os dois tipos em uma mesma *string* (evite abrir com um tipo de aspas e fechar com outro tipo, isso impedirá o seu programa de ser gerado).

As vezes é necessário imprimir uma palavra entre aspas ou um apóstrofo dentro de um texto, como por exemplo : "Ivo viu a 'uva' e gostou do preço". Nesse caso, devemos ter cuidado quando formos usar aspas dentro de strings. O seguinte código da listagem 3.8 está errado, porque eu não posso usar uma aspa do mesmo tipo para ser impressa, isso confunde o compilador.

Listagem 3.8: Erro com aspas
Fonte: codigos/ErroAspa.prg

```
PROCEDURE Main                                                  1
                                                                2
    ? "A praia estava "legal""  // Errado                       3
    ? 'Demos um nó em pingo d'água' // Errado                   4
                                                                5
RETURN                                                          6
```

Já o código da listagem 3.9 está correto, porque eu uso uma aspa simples para ser impressa, mas a string toda está envolvida com aspas duplas, e o inverso na linha seguinte :

Listagem 3.9: Uso correto de delimitadores de *string*
Fonte: codigos/Aspa.prg

```
/*                                                              1
Delimitando strings                                             2
```

[6] O Harbour também utiliza colchetes para delimitar *strings*, mas esse recurso é bem menos usado. Prefira delimitar *string* com aspas (simples ou duplas).

```
*/
PROCEDURE Main

    ? "A praia estava 'legal'."
    ? 'A praia estava "legal".'
    ? [Colchetes podem ser usados para delimitar strings!]
    ? [Você deram um "nó em pingo d'água"]
    ? "Mas mesmo assim evite o uso de colchetes."

RETURN
```

Caso a coisa se complique e você precise representar as duas aspas em uma mesma string, basta você usar os colchetes para envolver a string. Mas só use os colchetes nesses casos especiais, isso porque eles são uma forma antiga de representação e alguns editores modernos não irão colorir corretamente o seu código.

.:Resultado:.

```
A praia estava 'legal'.
A praia estava "legal".
Colchetes podem ser usados para delimitar strings!
Você deram um "nó em pingo d'água"
Mas mesmo assim evite o uso de colchetes.
```

Você **não pode** colocar duas strings lado a lado conforme abaixo :

```
PROCEDURE Main

   ? "Olá pessoal." "Hoje vamos sair mais cedo."

RETURN
```

Sempre que for exibir strings com o comando "?" use uma vírgula para realizar a separação. O exemplo a seguir está correto :

```
PROCEDURE Main

   ? "Olá pessoal." ,  "Hoje vamos sair mais cedo."

RETURN
```

.:Resultado:.

```
Olá pessoal. Hoje vamos sair mais cedo.
```

Você também pode usar o sinal de "+" para conseguir o mesmo efeito, mas se você observar atentamente verá que tem uma pequena diferença.

```
PROCEDURE Main

   ? "Olá pessoal." +  "Hoje vamos sair mais cedo."
```

```
                                                                    4
RETURN                                                              5
```

.:Resultado:.

```
Olá pessoal.Hoje vamos sair mais cedo.
```

No primeiro caso (usando vírgula) o Harbour acrescenta um espaço entre as duas strings, e no segundo caso (usando "+") o Harbour não acrescenta o espaço. Nos nossos exemplos envolvendo escrita na mesma linha nós usaremos inicialmente o primeiro formato (usando vírgula), mas conforme formos evoluindo nós veremos que esse formato possui sérias restrições, de modo que o segundo formato é o mais usado. Isso porque o primeiro formato (vírgula) só vai funcionar em conjunto com o comando "?" e o segundo formato ("+") é mais genérico. O segundo formato não será usado inicialmente também porque ele exige do aprendiz um conhecimento de outras estruturas que ele não possui, conforme veremos na próxima seção, quando estudarmos os números.

3.4.1 Quebra de linha

Linguagens como C, C++, Java, PHP, Pascal e Perl necessitam de um ponto e vírgula para informar que a linha acabou. Harbour não necessita desse ponto e vírgula para finalizar a linha, assim como Python e Basic. No caso específico da Linguagem Harbour, o ponto e vírgula é necessário para informar que a linha não acabou e deve ser continuada na linha seguinte.

Listagem 3.10: A linha não acabou
Fonte: codigos/linhaNaoAcabou.prg

```
PROCEDURE Main                                                      1
                                                                    2
    ? "Essa linha está dividida aqui mas " + ;                      3
      "será impressa apenas em uma linha"                           4
                                                                    5
RETURN                                                              6
```

.:Resultado:.

```
Essa linha está dividida aqui mas será impressa apenas em uma linha
```

Uma string não pode ser dividida sem ser finalizada no código. Veja no exemplo acima. Observe que utilizamos o já citado sinal de "+" [7] para poder "unir" a string que foi dividida.

No código a seguir temos um erro na quebra de linha pois a string não foi finalizada corretamente.

Listagem 3.11: A linha não acabou (ERRO)
Fonte: codigos/linhaNaoAcabou.prg

[7] O termo técnico para esse sinal é "operador" (veremos o que é isso mais adiante)

```
PROCEDURE Main

    ? "Essa linha está dividida aqui mas " + ;
      "será impressa apenas em uma linha"

RETURN
```

Outra função para o ponto e vírgula é condensar varias instruções em uma mesma linha, conforme o exemplo abaixo :

```
    ?  12; ? "Olá, pessoal"
```

equivale a

```
    ? 12
    ? "Olá, pessoal"
```

Nós desaconselhamos essa prática porque ela torna os programas difíceis de serem lidos.

Dica 6

Use o ponto e vírgula apenas para dividir uma linha grande demais.

3.5 Números

O seguinte código pode causar um certo desconforto para o iniciante :

```
PROCEDURE Main

   ?  "2" + "2"  // Irá exibir 22 e não 4.

RETURN
```

.:Resultado:.

```
  22
```

Como nós já estudamos as strings, fica fácil de entender o resultado dessa “conta” esquisita. Mas fica a pergunta : como fazemos para realizar cálculos matemáticos (do tipo 2 + 2) ?

A resposta é simples : use um número em vez de uma string. Usar um número é simples, basta retirar as aspas.

```
PROCEDURE Main

   ? 2 + 2

RETURN
```

.:Resultado:.

```
  4
```

Dica 7

Um número não possui aspas o envolvendo.

O que você **não** pode fazer é realizar uma operação entre um número e uma string. O código abaixo está errado :

```
PROCEDURE Main

   ? 2 +  "2"

RETURN
```

.:Resultado:.

```
Error BASE/1081  Argument error: +
Called from MAIN(3)
```

Acima você tem a reprodução da mensagem de erro que o programa gerou quando tentou realizar uma operação matemática entre um número e uma string. Veremos o significado dessa mensagem nos capítulos seguintes.

Outro cuidado que você deve ter quando for usar números é com o separador de decimais. Nunca use uma vírgula como separador, utilize somente o ponto. No exemplo a seguir o programador queria somar dois e trinta e quatro com três e cinquenta e seis.

```
PROCEDURE Main

   ? 2,34 +  3,56

RETURN
```

A soma acima produzirá o estranho resultado abaixo :

.:Resultado:.

```
         2         37         56
```

A vírgula tem uma função especial dentro da linguagem Harbour (e dentro das outras linguagens também), de modo que você **não** deve usar a vírgula como separador decimal. Se você entendeu o funcionamento do comando “?” verá que a vírgula serve para separar os dados que você envia, então o programa interpretou que você estava enviando três valores (o valor do meio é a soma 34 + 3) separados por vírgula.

Para realizar a soma desejada faça assim :

```
PROCEDURE Main

```

```
   ? 2.34 +  3.56

RETURN
```

O programa exibirá :

.:Resultado:.

```
        5.90
```

Dica 8

Usar a vírgula no lugar de um ponto não impede o programa de ser gerado, mas produz um resultado indesejado. Esse tipo de erro classifica-se como “erro de lógica”. Erros de lógica ocorrem quando o programa é gerado, mas o resultado não é o que você esperou que fosse. Mas, se você analisar o erro da soma 2,34 + 3,56 verá que o programa fez exatamente aquilo que foi ordenado, e quem errou, na realidade, foi o programador.
Esse tipo de raciocínio levou ao mito de que : “computador não erra, quem erra é o programador”, mas nem sempre isso é verdade. Muitas vezes o programador não erra, mas o computador gera dados errados. Será que isso é possível ? A quem culpar então ? Diante dessa incerteza, vale a pena ser um programador ? Calma, isso não quer dizer que o computador trabalhe errado. Acontece, em alguns casos que o sistema de informação (Hardware, Rede, Software e Pessoas) trabalha além da capacidade planejada. Isso não ocorre somente com computadores, vamos dar dois exemplos e você entenderá.
Primeiro exemplo: um carro utilitário foi projetado para carregar 4 toneladas de peso, mas ele sempre leva 5 toneladas em média. É lógico que ele irá apresentar problemas bem antes do tempo esperado. A culpa é da montadora ? A culpa é do engenheiro projetista (programador) ? Não. A culpa é do usuário que acabou usando o produto de forma errada.
Segundo exemplo: um médico receitou um medicamento duas vezes por dia, mas o paciente acabou tomando uma dose extra só para garantir. De quem é a culpa ? Do laboratório que desenvolveu o medicamento, do médico ou do paciente ?
Da mesma forma, sistemas de informações tem o seu limite que é definido pelo projetista. Quando um sistema de informação chega ao seu limite ele pode apresentar comportamentos estranhos: um servidor web pode recusar conexões, um programa pode exibir resultados estranhos, etc. Por isso, quando você estiver desenvolvendo seus sistemas é bom pensar no ambiente ideal e nos seus limites. Quando um sistema chega ao seu limite e consegue ser modificado em pouco tempo, sem problemas, para poder aceitar um limite maior, nós dizemos que esse sistema possui escalabilidade [a]. Mas mesmo construindo sistemas escaláveis, o ambiente onde ele será executado pode contribuir para que o sistema trabalhe errado, um exemplo disso é um sistema que funciona em locais com constantes panes elétricas ou oscilações de energia. Tudo isso deve ser observado quando você for implantar o seu futuro sistema em uma organização. O ideal é ter um contrato com clausulas bem definidas, mas isso já é assunto para outro tema.

[a] escalabilidade é uma característica desejável em todo o sistema, em uma rede ou em um processo, que indica sua capacidade de manipular uma porção crescente de trabalho de forma uniforme, ou estar preparado para crescer. Por exemplo, isto pode se referir à capacidade de um sistema em suportar um aumento de carga total quando os recursos (normalmente do hardware) são requeridos. Fonte : `https://pt.wikipedia.org/wiki/Escalabilidade`

3.6 Uma pequena introdução as Funções

O nosso curso todo é fundamentado na prática da programação, porém os primeiros capítulos são um pouco monótonos até que as coisas comecem a fazer sentido e nós possamos desenvolver algo prático. Por isso nós iremos aprender um tipo especial

de instrução que tornará os nossos códigos menos monótonos, trata-se da *função*. Por enquanto aprenderemos apenas o mínimo necessário sobre esse conceito novo, mas nos capítulos posteriores ele será visto detalhadamente. Como estamos iniciando, vamos ficar com o seguinte conceito pouco formal : **uma função é um símbolo que possui parênteses e também um valor pré-determinado**.

A seguir temos um exemplo de uma função. Note que a presença de parênteses após o seu nome é obrigatória para que o Harbour saiba que se trata de uma função.

```
PROCEDURE Main                                                  1
                                                                2
    ? "Olá pessoal!"                                            3
    ? TIME()  //  Imprime a hora                               4
                                                                5
 RETURN                                                         6
```

Observe que na linha 3 do exemplo anterior, o comando “?” imprime “Olá pessoal”, mas na linha 4 o comando “?” não imprime a palavra “TIME()”. Quando o programa chega na linha 4 ele identifica que o símbolo é uma função conhecida e substitui o seu nome pelo valor que ela “retorna”, que é a hora corrente do sistema operacional.

.:Resultado:.

```
Olá pessoal!
12:29:30
```

Prática número 2

Gere o programa acima e execute-o seguidas vezes. Note que o valor de TIME() sempre vem diferente.

Se por acaso a função TIME() tivesse sido digitada entre aspas, então o seu nome é que seria exibido na tela, e não o seu valor, conforme o exemplo a seguir :

```
PROCEDURE Main                                                  1
                                                                2
    ? "TIME()"                                                  3
    ? TIME()  //  Imprime a hora                               4
                                                                5
 RETURN                                                         6
```

.:Resultado:.

```
TIME()
12:29:30
```

Se você quiser usar a função TIME() para mostrar a hora do sistema dentro de uma frase, como por exemplo “Agora são 10:37:45” você **não pode** fazer como os exemplos abaixo :

```
PROCEDURE Main                                                  1
```

```
    ? "TIME()"  // Mostra o nome TIME()
    ? "A hora atual é " TIME() // Gera um erro

 RETURN
```

Para exibir a mensagem você deve fazer assim :

```
PROCEDURE Main

    ? "Agora são ", TIME()  // Use a vírgula do comando ''?''

 RETURN
```

.:Resultado:.

```
Agora são  10:34:45
```

Esse comportamento vale para todas as funções. No próximo capítulo estudaremos essa característica detalhadamente e veremos também que existem formas melhores e mais indicadas de se fazer essa junção. Por enquanto, quando você for exibir uma função junto com uma frase, use o comando “?” e separe-as com uma vírgula.

A seguir, a sintaxe da função TIME().

Descrição sintática 2

1. Nome : TIME()
2. Classificação : função.
3. Descrição : Retorna a hora do sistema.
4. Sintaxe

```
TIME() -> cStringHora
```

Fonte : [Nantucket 1990, p. 5-231]

O símbolo -> representa o retorno de uma função, que é o valor que ela representa (no caso é uma string contendo a hora do sistema). O “c” de cStringHora quer dizer que o valor é um caractere (string) e não um número ou qualquer outro tipo ainda não visto por nós.

Vamos a mais um exemplo : a listagem 3.12 ilustra o uso da função SECONDS(), a “função” dela é “retornar” quantos segundos transcorreram desde a meia-noite.

Listagem 3.12: Funções simples
Fonte: codigos/funcao01.prg

```
/*
```

```
Função                                                                          2
*/                                                                              3
PROCEDURE Main                                                                  4
                                                                                5
                                                                                6
   ? "Desde a meia-note até agora transcorreram ", SECONDS(), " segundos."
                                                                                7
RETURN                                                                          8
```

.:Resultado:.

```
Desde a meia-note até agora transcorreram       62773.81  segundos.
```

Descrição sintática 3

1. Nome : SECONDS()
2. Classificação : função.
3. Descrição : Retorna a quantidade de segundos decorridos desde a meia noite.
4. Sintaxe

```
SECONDS() -> nSegundos
```

Fonte : [Nantucket 1990, p. 5-231]

O “n” de nSegundos quer dizer que o valor é um número. Usaremos essa nomenclatura no capítulo seguinte quando formos estudar as variáveis.

Muito provavelmente você não sabe como estas duas funções realizam o seu trabalho. Você simplesmente as usa e espera que funcionem. Por exemplo: como é que a função TIME() calcula a hora, os minutos e os segundos ? Não sabemos como ela faz essa “mágica”, por isso é útil imaginar uma função como uma uma espécie de “oráculo”: você faz a pergunta (chamando a função) e ela retorna a resposta na forma de um valor [8].

A função TIME() responde a pergunta : “Que horas são ?”. Porém, a maioria das funções exigem uma pergunta mais complexa, do tipo : “Qual é a raiz quadrada de 64 ?”. Essa pergunta é complexa porque ela exige que eu passe algum valor para a função, que no caso seria o 64 para que a suposta função retorne a raiz quadrada.

Pois bem, a função SQRT() serve para calcular a raiz quadrada de um número positivo. Mas, como eu faço para dizer a ela qual é o número que eu desejo saber a raiz quadrada ? Esse valor que eu passo como parte da pergunta chama-se “argumento”

[8] Você verá em muitos textos técnicos a expressão “caixa preta”. Esse termo significa a mesma coisa. Você não sabe o que existe dentro dela (daí o nome preta) mas você sabe quais dados de entrada passar e o que esperar dela. Um exemplo de “caixa preta” é uma simples calculadora de bolso.

(no nosso caso, o 64 é um argumento), e a função SQRT() “funciona” conforme o exemplo abaixo :

```
PROCEDURE Main

   ? SQRT( 64 )

RETURN
```

.:Resultado:.

```
 8
```

Ou seja, o valor que eu desejo calcular é “passado” para a função entre os parênteses (os parênteses servem também para isso, para receber valores que a função necessita para poder trabalhar). Você poderia perguntar : “no caso de TIME() eu não precisei passar um valor, mas no caso de SQRT() eu precisei. Como eu sei quando devo passar algum valor para a função ?”. A resposta é simples : você deve consultar a documentação (a notação da função). No caso de SQRT() a notação é a seguinte :

Descrição sintática 4

1. Nome : SQRT()
2. Classificação : função.
3. Descrição : Retorna a raiz quadrada de um número positivo.
4. Sintaxe

   ```
   SQRT( <nNumero> ) -> nRaiz
   ```

Fonte : [Nantucket 1990, p. 5-223]

Note que a função SQRT exige que você passe um valor. Se o valor fosse opcional então seria [<nNumero>], lembra do papel dos colchetes na nomenclatura ?

3.7 Comentários

Os comentários são notas ou observações que você coloca dentro do seu programa mas que servem apenas para documentar o seu uso trabalho, ou seja, o programa irá ignorar essas linhas de comentários. Eles não são interpretados pelo compilador, sendo ignorados completamente.

O Harbour possui dois tipos de comentários : os de uma linha e os de múltiplas linhas.

Os comentários de uma linha são :

1. “NOTE” : Só pode ser usado antes da linha iniciar. (Formato antigo derivado do Dbase II, evite esse formato.).

2. && : Pode ser usado antes da linha iniciar ou após ela finalizar. (Formato antigo derivado do Dbase II, evite esse formato.).
3. “*” : Só pode ser usado antes da linha iniciar. (Formato antigo derivado do Dbase II. Alguns programadores ainda usam esse comentário, principalmente para documentar o início de uma rotina.).
4. “//”: Da mesma forma, pode ser usado antes da linha iniciar ou após ela finalizar (Inspirados na Linguagem C++, é o formato mais usado para comentários de uma linha).

Os comentários de múltiplas linhas começam com /* e termina com */, eles foram inspirados na Linguagem C. Esse formato é largamente utilizado.

Listagem 3.13: Uso correto de comentários
Fonte: codigos/comentario.prg

```
PROCEDURE Main

   ? "Olá, pessoal" // Útil para explicar a linha corrente.
   // ? "Essa linha TODA será ignorada."
   *  ? "Essa linha também será ignorada."
   && ? "E essa também..."

   ? "O valor : " ,  1200  // Útil para comentar após
                           // o final da linha.

/*

    Note que esse comentário
    possui várias linhas.

*/

RETURN
```

No dia-a-dia, usamos apenas os comentários “//” e os comentários de múltiplas linhas (/* */) mas, caso você tenha que alterar códigos antigos, os comentários “&&” e “*” são encontrados com bastante frequência.

O Harbour (assim como as outras linguagens de programação) não permite a existência de comentários dentro de comentários. Por exemplo : /* Inicio /* Inicio2 */ Fim */.

Listagem 3.14: Uso INCORRETO de comentários
Fonte: codigos/comentarioErrado.prg

```
PROCEDURE Main

    /*
         ERRO !
         Os comentários não podem ser aninhados, ou
         seja, não posso colocar um dentro do outro.
```

```
         /*  Aqui está o erro  */

    */

    ? "Olá pessoal, esse programa não irá compilar."

RETURN
```

Os comentários de múltiplas linhas também podem ser usados dentro de um trecho de código sem problema algum. Por exemplo :

```
    ? 12 /* preço */ + 3 /* imposto */ + 4 // Imprime o custo
```

.:Resultado:.

```
19
```

Dica 9

Os comentários devem ajudar o leitor a compreender o problema abordado. Eles não devem repetir o que o código já informa claramente e também não devem contradizer o código. Eles devem ajudar o leitor a entender o programa. Esse suposto "leitor" pode até ser você mesmo daqui a um ano sem "mecher" no código. Seguem algumas dicas rápidas retiradas de [Kernighan e Pike 2000, p. 25-30] sobre como comentar eficazmente o seu código :

1. Não reporte informações evidentes no código.
2. Quando o código for realmente difícil, como um algoritmo complicado ou uma norma fiscal obscura, procure indicar no código uma fonte externa para auxiliar o leitor (um livro com a respectiva página ou um site). Sempre cite referências que não corram o risco de "sumir" de uma hora para outra (evite blogs ou redes sociais, se não tiver alternativa salve esse conteúdo e tenha cópias de segurança desses arquivos).
3. Não contradiga o código. Quando alterar o seu trabalho atualize também os seus comentários ou apague-os.

3.8 Praticando

Procure agora praticar os seguintes casos a seguir. Em todos eles existe um erro que impede que o programa seja compilado com sucesso, fique atento as mensagens de erro do compilador e tente se familiarizar com elas.

Listagem 3.15: Erro 1
Fonte: codigos/helloErro01.prg

```
/*
* Aprendendo Harbour
```

```
*/
PROCEDURE Main

    ? "Hello" "World"


RETURN
```

Listagem 3.16: Erro 2
Fonte: codigos/helloErro02.prg

```
/*
* Aprendendo Harbour
/*
PROCEDURE Main

    ? "Hello World"


RETURN
```

Listagem 3.17: Erro 3
Fonte: codigos/helloErro03.prg

```
/*
/* Aprendendo Harbour
*/
PROCEDURE Main

    ? "Hello World"


RETURN
```

Listagem 3.18: Erro 4
Fonte: codigos/helloErro04.prg

```
/*
 Aprendendo Harbour
*/
PROCEDURE Main

    ? Hello World


RETURN
```

Listagem 3.19: Erro 5
Fonte: codigos/helloErro05.prg

```
/*
  Aprendendo Harbour
```

```
*/
PROCEDURE Main

    ?  "Hello ";
    ?? "World"



RETURN
```

3.9 Desafio

Escreva um programa que coloque na tela a seguinte saída :

```
LOJAO MONTE CARLO
PEDIDO DE VENDA
=========================================
ITEM            QTD      VAL          TOTAL
------------   ---- --------  ------------
CAMISA GOLA       3    10,00         30,00
LANTERNA FLA      5    50,00        250,00
                                ============
                                      280,00


 DEUS E FIEL.
```

3.10 Exercícios

1. Escreva um programa que imprima uma pergunta : “Que horas ?” e na linha de baixo a resposta. Siga o modelo abaixo :

.:Resultado:.

```
Que horas ?
 10:34:45.
```

Dica : Use a função *TIME()* para imprimir a hora corrente.

2. Escreva um programa que apresente na tela :

.:Resultado:.

```
  linha 1
  linha 2
  linha 3
```

3. Escreva um programa que exiba na tela a *string* : “Tecle algo para iniciar a ‘baixa’ dos documentos”.

4. Escreva um programa que exiba na tela o nome do sistema operacional do computador em que ele está sendo executado. Dica: A função OS() retorna o nome do sistema operacional.

5. Escreva um programa que calcule e mostre a raiz quadrada de 1 , 2, 3, 4, 5, 6, 7, 8, 9 e 10.

4 Constantes e Variáveis

A fonte do saber não está nos livros, ela está na realidade e no pensamento. Os livros são placas de sinalização; o caminho é mais antigo, e ninguém pode fazer por nós a viagem da verdade.

A.-D.Sertillanges - A vida intelectual

Objetivos do capítulo

- Entender o que é uma constante e uma variável bem como a diferença entre elas.
- Criar variáveis de memória de forma adequada.
- Compreender a importância de criar nomes claros para as suas variáveis e constantes.
- Atribuir valores as suas variáveis.
- Diferenciar declaração de variável de inicialização de variável.
- Receber dados do usuário.
- Realizar operações básicas com as variáveis.

4.1 Constantes

Constantes são valores que não sofrem modificação durante a execução do programa. No capítulo anterior nós trabalhamos, mesmo sem saber o nome formal, com dois tipos de constantes : as constantes numéricas e as constantes caracteres (*strings*). Veja novamente dois exemplos dessas constantes :

```
PROCEDURE Main

   ? "Hello World" // ''Hello World'' é uma constante caractere
   ? 2 // 2 é uma constante numérica

RETURN
```

Um programa de computador trabalha com várias constantes durante a sua execução, algumas delas possuem um valor muito especial, como por exemplo 3.1415 ou uma *string* que representa o código de uma cor (por exemplo : “FFFFFF”). Não é uma boa prática de programação simplesmente colocar esse símbolo dentro do seu código e usá-lo. Você pode replicar : “Tudo bem, eu posso criar um comentário para esse número ou string!”, mas e se esse valor se repetir em dez pontos diferentes do seu código ? Será que você vai comentar dez vezes ? E será que compensa comentar dez vezes ? E se houver uma mudança futura na constante, você saberá onde atualizar os comentários ?

Esse problema constuma acontecer mais com números do que com strings. Quando um número (que possui uma importância dentro do problema) é simplesmente colocado dentro do programa ele é chamado de “número mágico”. “Número mágico” é uma expressão irônica para indicar as constantes numéricas que simplesmente aparecem nos programas. Segundo Kernighan e Pike, “todo número pode ser mágico e, se for, deve receber seu próprio nome. Um número bruto no programa fonte não indica a sua importância ou derivação, tornando o programa mais difícil de entender e modificar.” [Kernighan e Pike 2000, p. 21].

Uma forma de evitar o problema com números mágicos é dar-lhes um nome. Eles ainda serão números, mas serão “chamados” por um nome específico. O Harbour resolve esse problema através de um recurso chamado de “constante manifesta”, “constante simbólica” ou ainda “constante de pré-processador”. Esse recurso recebe esse nome porque as constantes recebem um nome especial (um símbolo) e, durante a fase anterior a geração do programa (conhecida também como “fase de processamento”), esse nome é convertido para o seu respectivo valor numérico. Como esse valor é atribuído em tempo de compilação, as constantes de pré-processador estão fora da procedure Main. No exemplo a seguir nós temos uma constante manifesta chamada VOLUME que vale 1000.

```
#define VOLUME 10000
PROCEDURE Main

   ? VOLUME // Vai imprimir 10000

RETURN
```

Essa forma também é usada tradicionalmente por programadores da Linguagem C. Note que, no exemplo acima, caso o volume mude um dia. Eu terei que recompilar o programa mas a alteração do volume dela será feita em apenas um ponto. O valor de uma constante é sempre atribuída em tempo de compilação, por isso a mudança no valor de uma constante requer a recompilação do programa.

Dica 10

A expressão **"tempo de compilação"** refere-se ao momento em a instrução foi definida pelo computador. Nesse caso específico, as constantes manifestas são definidas antes do programa principal ser gerado, ou seja, durante a compilação do programa.
Vamos acompanhar o processo que se esconde durante a definição de uma constante manifesta. Vamos tomar como ilustração a listagem abaixo :

```
#define VOLUME 12
PROCEDURE Main

      ? VOLUME

RETURN
```

Quando você executa o hbmk2 para compilar e gerar o seu programa executável ele transforma a listagem acima em outra (sem que você veja) antes de compilar. A listagem que ele gera é a listagem abaixo :

```

PROCEDURE Main

      ? 12

RETURN
```

Só agora ele irá compilar o programa. Note que ele "retirou" as definições e também substituiu VOLUME por 12. Não se preocupe, pois o seu código fonte permanecerá inalterado, ele não irá alterar o que você fez.

Dica 11

Você deve ter cuidado quando for criar constantes dentro de seu programa. Basicamente são dois os cuidados :

1. Escolha um símbolo que possa substituir esse valor. Evite os “números mágicos” e atribua nomes de fácil assimilação também para as constantes caracteres. Por exemplo, um código de cor pode receber o próprio nome da cor.
2. Decida sabiamente quando você quer que um valor realmente não mude dentro de seu programa. Geralmente códigos padronizados mundialmente, valores de constantes conhecidas (o valor de Pi) e notas de copyright.
3. Siglas de estados ou percentuais de impostos não são bons candidatos porque podem vir a mudar um dia.

Um detalhe importantíssimo é que uma constante manifesta **não é *case insensitive*** como os demais itens da linguagem. Se você criar uma constante com o nome de PI, e quando for usar escrever Pi, então o programa irá reportar um erro.

Dica 12

Use as constantes de pré-processador para evitar números mágicos. Esse recurso não se aplica somente em números. Posso definir strings também [a]. Exemplos :

```
#define PI_VALUE  3.141516
#define COLOR_WHITE "FFFFFF"

PROCEDURE Main

    ... Restante do código ...
```

Não esqueça de definir suas constantes antes da procedure Main, conforme o exemplo acima.

[a] Além de números e strings posso criar constantes manifestas com outros tipos da linguagem Harbour que ainda não foram abordados. No momento certo nós daremos os exemplos correspondentes, o importante agora é entender o significado das constantes manifestas.

Dica 13

Como as constantes são sensíveis a forma de escrita, habitue-se a escrever as constantes apenas com letras maiúsculas. Esse é um bom hábito adotado por todos os programadores de todas as linguagens de programação.

4.1.1 A hora em que você tem que aceitar os números mágicos

Nem todos os números mágicos devem ser convertidos em constantes manifestas. Existe um caso muito especial onde você deve se conformar e manter aquele número

misterioso dentro do seu código. Isso acontece na hora de você aplicar determinadas fórmulas de terceiros dentro do seu código, como por exemplo a fórmula que converte uma temperatura medida em graus Celsius para Fahrenheit (F = C * 1.8 + 32). Você não precisa transformar 1.8 e 32 em constantes manifestas, pois tais valores não serão encontrados isolados dentro do seu código. A mesma coisa também serve para conversões de valores inteiros para valores percentuais, quando durante o processo ocorre uma divisão por 100. Nesse caso, não faz sentido transformar esse 100 em uma constante manifesta. Coordenadas da tela também não devem ser convertidas em números mágicos (veremos o que é isso no capítulo 11). Fique atento, pois no processo de desenvolvimento de programas existem muitas regras de clareza mas sempre tem a maior de todas que é usar o bom senso.

4.1.2 Grupos de constantes

As vezes é importante você ter todas as suas constantes agrupadas em um único arquivo. Isso evita ter que ficar redigitando todas as constantes em vários lugares diferentes. Por exemplo, vamos supor que você criou uma constante para simbolizar o código da cor branca (“FFFFFF”), conforme abaixo :

```
#define BRANCO "FFFFFF"
PROCEDURE Main

   ? BRANCO  // Imprime FFFFFF

RETURN
```

Suponha também que surgiu a necessidade de se criar constantes para as outras cores. Conforme o exemplo a seguir :

```
#define BRANCO "FFFFFF"
#define VERMELHO "FF0000"
#define AZUL "0000FF"
#define AMARELO "FFFF00"
PROCEDURE Main

   ? BRANCO
   ? VERMELHO
   ? AZUL
   ? AMARELO

RETURN
```

No futuro você terá várias constantes de cores. Mas, e se você resolver usar em outro programa ? Você teria que redefinir todas as constantes no novo programa. E se você tiver que criar uma nova constante para incluir uma nova cor ? Você teria que alterar todos os programas.

Para evitar esse problema você pode criar um arquivo de constantes de cores e chamá-lo de “cores.ch”. Veja a seguir o conteúdo de um suposto arquivo chamado de “cores.ch”.

```
#define BRANCO "FFFFFF"
#define VERMELHO "FF0000"
#define AZUL "0000FF"
#define AMARELO "FFFF00"
```

Para usar essas constantes no seu programa basta salvar esse arquivo no mesmo local do seu programa e fazer assim :

```
#include "cores.ch"
PROCEDURE Main

   ? BRANCO
   ? VERMELHO
   ? AZUL
   ? AMARELO

RETURN
```

Pronto, agora você pode usar suas constantes de uma forma mais organizada.

Dica 14

Os arquivos include servem para organizar as suas constantes. Procure agrupar as suas constantes em arquivos includes, por exemplo : cores.ch, matematica.ch, financas.ch, etc. Sempre use a extensão “ch” para criar seus arquivos include. Se por acaso você não tem uma classificação exata para algumas constantes, não perca tempo, crie um arquivo chamado “geral.ch” ou “config.ch” (por exemplo) e coloque as suas constantes nesse arquivo.
O Harbour possui os seus próprios arquivos “include”. Você sempre pode utilizar esses arquivos caso deseje. Eles tem outras funções além de organizar as constantes em grupos, veremos mais adiante essas outras funções.

4.2 Variáveis

Variáveis são locais na memória do computador que recebem uma identificação e armazenam algum dado específico. O valor do dado pode mudar durante a execução do programa. Por exemplo, a variável **nTotalDeItens** pode receber o valor 2,4, 12, etc. Daí o nome “variável”. O criador da linguagem C++ introduz assim o termo :

> basicamente, não podemos fazer nada de interessante com um computador sem armazenar dados na memória [...]. Os “lugares” nos quais armazenamos dados são chamados de *objetos*. Para acessar um objeto necessitamos de um nome. Um objeto com um nome é chamado de variável [Stroustrup 2012, p. 62].

4.2.1 Criação de variáveis de memória

Uma variável deve ser definida antes de ser usada. O Harbour permite que uma variável seja definida (o jargão técnico é “declarada”) de várias formas, a primeira delas

é simplesmente criar um nome válido e definir a ele um valor, conforme a listagem 4.1. Note que a variável fica no lado esquerdo e o seu valor fica no lado direito. Entre a variável e o seu valor existe um sinal (:=) que representa uma atribuição de valor. Mais adiante veremos as outras formas de atribuição, mas habitue-se a usar esse sinal (o nome técnico é “operador”) pois ele confere ao seu código uma clareza maior.

Outro detalhe importante que gostaríamos de chamar a atenção é o seguinte : entre a linha 7 e a linha 8 existe uma linha não numerada. Não se preocupe pois isso significa que a linha 7 é muito grande e a sua continuação foi mostrada embaixo, mas no código fonte ela é apenas uma linha. Essa situação irá se repetir em algumas listagens apresentadas nesse livro.

Listagem 4.1: Criação de variáveis de memória

```
1
/*
2
  Criação de variáveis de memória
3
*/
4
PROCEDURE Main
5
6
7
   cNomeQueVoceEscolher := "Se nós nos julgássemos a nós mesmos, jamais seríam
   ? cNomeQueVoceEscolher
8
9
RETURN
10
```

.:Resultado:.

```
Se nós nos julgássemos a nós mesmos, jamais seríamos condenados.
```

Quando você gerou esse pequeno programa, e o fez na pasta “pratica” então você deve ter recebido do compilador algumas “mensagens estranhas”, conforme abaixo :

.:Resultado:.

```
> hbmk2 var0
hbmk2: Processando script local: hbmk.hbm
hbmk2: Harbour: Compilando módulos...
Harbour 3.2.0dev (r1507030922)
Copyright (c) 1999-2015, http://harbour-project.org/
Compiling 'var0.prg'...
var0.prg(6) Warning W0001  Ambiguous reference 'CNOMEQUEVOCEESCOLHER'
var0.prg(7) Warning W0001  Ambiguous reference 'CNOMEQUEVOCEESCOLHER'
Lines 9, Functions/Procedures 1
Generating C source output to '.hbmk\win\mingw\var0.c'... Done.
hbmk2: Compilando...
hbmk2: Linkando... var0.exe
```

Estamos nos referindo as mensagens :

.:Resultado:.

```
var0.prg(6) Warning W0001  Ambiguous reference 'CNOMEQUEVOCEESCOLHER'
var0.prg(7) Warning W0001  Ambiguous reference 'CNOMEQUEVOCEESCOLHER'
```

Mensagens com a palavra “Warning” não impedem o seu programa de ser gerado, apenas chamam a atenção para alguma prática não recomendada ou um possível erro. Não se preocupe com isso, pois essas mensagens serão explicadas mais adiante ainda nesse capítulo. Por ora pode digitar normalmente os exemplos que virão e ignorar esses avisos estranhos.

4.2.2 Escolhendo nomes apropriados para as suas variáveis

Você não pode escolher arbitrariamente qualquer nome para nomear as suas variáveis de memória. Alguns critérios devem ser obedecidos, conforme a pequena lista logo abaixo :

1. O nome **deve** começar com uma letra ou o caracter “_” (*underscore* [1]).
2. O nome de uma variável **deve** ser formado por uma letra ou um número ou ainda por um *underscore*. Lembre-se apenas de não iniciar o nome da variável com um dígito (0...9), conforme preconiza o ítem 1.
3. Não utilize letras acentuadas ou espaços para compor o nome da sua variável.

O exemplo a seguir (listagem 4.2) mostra alguns nomes válidos para variáveis :

Listagem 4.2: Nomes válidos para variáveis

```
/*
  Nomes válidos
*/
PROCEDURE Main

   nOpc := 100

   _iK := 200  // Válido, porém não aconselhado (Começa com underline).
   nTotal32 := nOpc + _iK
   ? "nOpc = " , nOpc
   ? "_iK = " , _iK
   ? "nTotal32 = " , nTotal32

RETURN
```

.:Resultado:.

```
nOpc =         100
_iK =         200
nTotal32 =         300
```

Dica 15

Apesar do caractere *underscore* ser permitido no início de uma variável essa prática é desaconselhada por vários autores.

[1] Em algumas publicações esse caractere recebe o nome de *underline*

É aconselhável que uma variável NÃO tenha o mesmo nome de uma “palavra reservada” da linguagem Harbour. Palavras reservadas são aquelas que pertencem a própria sintaxe de uma determinada linguagem de programação e o compilador “reserva” para si o direito exclusivo de usá-las. Nós utilizamos aspas para exprimir o termo “palavra reservada” da linguagem Harbour, porque o nome das instruções e comandos do Harbour não são “reservadas”.

Dica 16

Em determinadas linguagens, como a Linguagem C, você não pode nomear uma variável com o mesmo nome de uma palavra reservada. O Clipper (ancestral do Harbour) também não aceita essa prática, de acordo com [Ramalho 1991, p. 68]. O Harbour não reclama do uso de palavras reservadas, mas reforçamos que você **não** deve adotar essa prática.

Listagem 4.3: Uso de palavras reservadas

```
/*
  Evite usar palavras reservadas
*/
PROCEDURE Main

   PRIVATE := 100 // PRIVATE é uma palavra reservada
   REPLACE := 12  // REPLACE também é
   TOTAL := PRIVATE + REPLACE // Total também é
   ? TOTAL

RETURN
```

.:Resultado:.

```
      112
```

Dica 17

[Spence 1994, p. 11] nos informa que o Clipper possui um arquivo (chamado "reserved.ch") na sua pasta include com a lista de palavras reservadas. O Harbour também possui esse arquivo, mas a lista está em branco. Você pode criar a sua própria lista de palavras reservadas no Harbour. Caso deseje fazer isso você precisa seguir esses passos :

1. Edite o arquivo reserved.ch que vem com o Harbour na sua pasta include.
2. Inclua em uma lista (um por linha) as palavras que você quer que sejam reservadas.
3. No seu arquivo de código (.prg) você deve incluir (antes de qualquer função) a linha : *#include "reserved.ch"*.

O Harbour irá verificar, em tempo de compilação, a existência de palavras que estejam nesse arquivo e irá barrar a compilação caso o seu código tenha essas palavras reservadas (por você). Você pode, inclusive, criar a sua própria lista de palavras reservadas independente de serem comandos ou não.

O programa a seguir (Listagem 4.4) exemplifica uma atribuição de nomes inválidos à uma variável de memória.

Listagem 4.4: Nomes inválidos para variáveis

```
/*
  Nomes inválidos
*/
PROCEDURE Main

   9Opc := 100 // Inicializa com um número

RETURN
```

4.2.3 Seja claro ao nomear suas variáveis

Tenha cuidado quando for nomear suas variáveis. Utilize nomes indicativos daquilo que elas armazenam. Como uma variável não pode conter espaços em branco, utilize caracteres *underscore* para separar os nomes, ou então utilize uma combinação de letras maiúsculas e minúsculas [2].

Listagem 4.5: Notações

```
/*
Notações utilizadas
*/
PROCEDURE Main

   Tot_Nota := 1200 // Variável numérica (Notação com underscores)
                    // que representa o total da nota fiscal
```

[2] Essa notação é conhecida como Notação Hungara.

```
   NomCli := "Rob Pike" // Variável caracter (Notação hungara)
                        // que representa o nome do cliente


RETURN
```

Mais adiante estudaremos os tipos de dados, mas já vamos adiantando : procure prefixar o nome de suas variáveis com o tipo de dado a que ela se refere. Por exemplo: **nTot** [3] representa uma variável numérica, por isso ela foi iniciada com a letra "n". Já **cNomCli** representa uma variável caractere (usada para armazenar *strings*), por isso ela foi iniciada com a letra "c". Mais adiante estudaremos com detalhes outros tipos de variáveis e aconselhamos que você siga essa nomenclatura: "n" para variáveis numéricas, "c" para variáveis caracteres (strings), "d" para variáveis do tipo data, "l" para variáveis do tipo lógico, etc. **Mas sempre é bom usar o bom senso** : se você vai criar um código pequeno para exemplificar uma situação, você pode abdicar dessa prática.

[3] Essa notação é chamada por Rick Spence de "Notação Hungara Modificada"

Dica 18

Essa parte é tão importante que vamos repetir : “apesar de ser simples criar um nome para uma variável, você deve priorizar a clareza do seu código. O nome de uma variável deve ser informativo, conciso, memorizável e, se possível, pronunciável” [Kernighan e Pike 2000, p. 3]. Procure usar nomes descritivos, além disso, procure manter esses nomes curtos, conforme a listagem 4.6.

Listagem 4.6: Nomes curtos e concisos para variáveis

```
/*
  NOMES CONSISOS

  Adaptado de (KERNIGHAN, p.3, 2000)
*/
PROCEDURE Main

   /*
   Use comentários "//" para acrescentar informações sobre

   a variável, em vez de deixar o nome da variável grande demais.
   */
   nElem := 1 // Elemento corrente
   nTotElem := 1200 // Total de elementos

RETURN
```

Evite detalhar demais conforme a listagem 4.7.

Listagem 4.7: Nomes longos e detalhados demais

```
/*
  NOMES GRANDES DEMAIS PARA VARI?VEIS. EVITE ISSO!!!

  Adaptado de (KERNIGHAN, p.3, 2000)
*/
PROCEDURE Main

   nNumeroDoElementoCorrente := 1
   nNumeroTotalDeElementos := 1200

RETURN
```

Complementando : escolha nomes auto-explicativos para as suas variáveis, evite comentar variáveis com nomes esquisitos.

4.2.4 Atribuição

Já vimos como atribuir dados a uma variável através do operador := . Agora iremos detalhar essa forma e ver outras formas de atribuição.

O código listado em 4.8 nos mostra a atribuição com o operador "=" e com o comando STORE. Conforme já dissemos, prefira o operador :=.

Listagem 4.8: Outras forma de atribuição

```
/*
Atribuição
*/
PROCEDURE Main

   x = 1200  // Atribui 1200 ao valor x
   STORE 100 TO y, k, z // Atribui 100 ao valor y, k e z

   ? x + y + k + z

RETURN
```

.:Resultado:.

```
      1500
```

A atribuição de variáveis também pode ser feita de forma simultânea, conforme a listagem 4.9.

Listagem 4.9: Atribuição simultânea

```
/*
Atribuição
*/
PROCEDURE Main

   x := y := z := k := 100
   ? x + y + z + k

RETURN
```

.:Resultado:.

```
       400
```

Dica 19

Já foi dito que o Harbour é uma linguagem “Case insensitive”, ou seja, ela não faz distinção entre letras maiúsculas e minúsculas. Essa característica se aplica também as variáveis. Os seguintes nomes são equivalentes : **nNota, NNOTA, nnota, NnOta e NNota**. Note que algumas variações da variável **nNota** são difíceis de se ler, portanto você deve sempre nomear as suas variáveis obedecendo aos padrões já enfatizados nesse capítulo.

4.3 Variáveis: declaração e inicialização

Os operadores de atribuição possuem um triplo papel na linguagem Harbour : elas criam variáveis (caso elas não existam), atribuem valores as variáveis e podem mudar o tipo de dado de uma variável (os tipos de dados serão vistos no próximo capítulo).

1. O ato de criar uma variável chama-se *declaração*. Declarar é criar a variável, que é o mesmo que reservar um espaço na memória para ela.

2. O ato de inicializar uma variável chama-se *inicialização*. Inicializar uma variável é o mesmo que dar-lhe um valor.

Dica 20

Nos capítulos seguintes nós iremos nos aprofundar no estudo das variáveis, mas desde já é importante que você adquira bons hábitos de programação. Por isso iremos trabalhar com uma instrução chamada de *LOCAL*. Você não precisa se preocupar em entender o que isso significa, mas iremos nos acostumar a declarar as variáveis como *LOCAL* e **obrigatoriamente** no início do bloco de código, assim como feito no código anterior (Listagem 4.9).
Nós deixamos claro, no início do livro, que não iremos usar conceitos sem a devida explicação, mas como esse conceito é importante demais resolvemos abrir essa pequena exceção[a]. Nós digitaremos muitos códigos até o final do livro, e em todos usaremos essa palavra ainda não explicada para declarar as nossas variáveis.
Note também que a palavra reservada *LOCAL* não recebe indentação, pois nós a consideramos parte do início do bloco e também que nós colocamos uma linha em branco após a sua declaração para destacá-las do restante do código. Nós também colocamos um espaço em branco após a vírgula que separa os nomes das variáveis, pois isso ajuda na visualização do código.
A existência de linhas e espaços em branco não deixa o seu programa “maior” e nem consome memória porque o compilador simplesmente os ignora. Assim, você deve usar os espaços e as linhas em branco para dividir o seu programa em “mini-pedaços” facilmente visualizáveis.
Podemos sintetizar o que foi dito através da seguinte “equação” :

```
Variáveis agrupadas de acordo com o comentário +
Indentação no início do bloco +
Espaço em branco após as vírgulas +
Linhas em branco dividindo os ``mini-pedaços''  =
-------------------------------------------------
Código mais fácil de se entender.
```

[a] Você só poderá entender completamente esse conceito no final do livro

Dica 21

Procure adquirir o bom hábito de **não** declarar as suas variáveis através de um operador. Use a instrução *LOCAL* para isso.
Você pode, preferencialmente, fazer conforme o exemplo abaixo (declaro e inicializo ao mesmo tempo) :

```
PROCEDURE Main
LOCAL nValor := 12

    ? nValor
    // O restante das instruções ...

RETURN
```

se não souber o valor inicial, então faça conforme o exemplo a seguir (declaro e só depois inicializo) :

```
PROCEDURE Main
LOCAL nValor

    nValor := 12
    ? nValor
    // O restante das instruções ...

RETURN
```

mas nunca faça como o exemplo abaixo (sem a instrução LOCAL)

```
PROCEDURE Main

    nValor := 12
    ? nValor
    // O restante das instruções ...

RETURN
```

Dica 22

Você pode designar o próprio Harbour para monitorar a declaração de variáveis. Isso é muito bom pois evita que você declare uma variável de uma forma não recomendada. Como fazer isso ?
É simples. Lembra do arquivo de configuração de opções de compilação do Harbour chamado de hbmk.hbm ? Ele foi citado no início do livro durante o aprendizado do processo de compilação (capítulo 2). Você pode acrescentar nesse arquivo a opção -w3. Essa opção eleva para três (o nível máximo) de "warnings" (avisos de cuidado) durante o processo de compilação. No nosso diretório de prática nós temos um arquivo hbmk.hbm já com a opção -w3, por isso se você compilar o terceiro exemplo dessa dica, então ele irá exibir um aviso, mas **não deixará de gerar o programa**.
As mensagens geradas durante a compilação do nosso exemplo acima são :

```
Warning W0001  Ambiguous reference 'NVALOR'
```

"Ambiguous reference" (Referência ambigua) quer dizer : "eu não sei exatamente o que você quer dizer com essa atribuição pois a variável não foi inicializada de forma segura." Voltaremos a esse assunto com detalhes no final do livro. Por enquanto, **certifique-se de ter um arquivo hbmk.hbm na sua pasta onde você compila o seu sistema e que dentro desse arquivo esteja definido o nível máximo de alerta (-w3)**.

4.4 Recebendo dados do usuário

No final desse capítulo iremos treinar algumas rotinas básicas com as variáveis que aprendemos, mas boa parte dessas rotinas precisam que você saiba receber dados digitados pelo usuário. O Harbour possui várias maneiras de receber dados digitados pelo usuário, como ainda estamos iniciando iremos aprender uma maneira simples de receber dados numéricos: o comando *INPUT*[4]. O seu uso está ilustrado no código 4.10.

Listagem 4.10: Recebendo dados externos digitados pelo usuário

```
/*
Uso do comando INPUT
*/
PROCEDURE Main
LOCAL nVal1 := 0 // Declara a variável parcela de pagamento
LOCAL nVal2 := 0 // Declara a variável parcela de pagamento

   INPUT "Informe o primeiro valor : " TO nVal1
   INPUT "Informa o segundo valor : " TO nVal2

   ? "A soma dos dois valores é : ", nVal1 + nVal2
```

[4]O comando INPUT recebe também dados caracteres, mas aconselhamos o seu uso para receber apenas dados numéricos.

```
                                                                    13
                                                                    14
RETURN                                                              15
```

Prática número 3

Abra o arquivo basico.prg na pasta “pratica” e salve-o com o nome pratica_input.prg.
Feito isso digite o conteúdo acima.

Quando for executar o programa, após digitar o valor sempre tecle *ENTER*.

.:Resultado:.

```
Informe o primeiro valor : 15
Informa o segundo valor : 30
A soma dos dois valores é :          45
```

Dica 23

Note que, na listagem 4.10 as variáveis foram declaradas em uma linha separada, enquanto que em outras listagens elas foram declaradas em uma única linha. Não existe uma regra fixa com relação a isso, mas aconselhamos você a organizar as variáveis de acordo com o comentário (*//*) que provavelmente você fará após a declaração. Por exemplo, se existem duas variáveis que representam coisas que podem ser comentadas em conjunto, então elas devem ser declaradas em apenas uma linha. A listagem 4.10 ficaria mais clara se as duas linhas de declaração fossem aglutinadas em apenas uma linha. Isso porque as duas variáveis (*nVal1* e *nVal2*) possuem uma forte relação entre si. O ideal seria :

```
LOCAL nVal1 := 0, nVal2 := 0 // Parcelas do pagamento.
```

ou ainda recorrendo a atribuição simultânea de valor :

```
LOCAL nVal1 :=  nVal2 := 0 // Parcelas do pagamento.
```

Note também que nós devemos atribuir um valor as variáveis para informar que elas são números (no caso do exemplo acima, o zero informa que as variáveis devem ser tratadas como numéricas).

Se os dados digitados forem do tipo caractere você deve usar o comando *ACCEPT*. (Veja um exemplo na listagem 6.2). Observe que o *ACCEPT* funciona da mesma forma que o *INPUT*.

Listagem 4.11: Recebendo dados externos digitados pelo usuário (Tipo caractere)

```
                                                                    1
```

```
/*
Uso do comando ACCEPT
*/
PROCEDURE Main
LOCAL cNome // Seu nome

   /* Pede e exibe o nome do usuário */
   ACCEPT "Informe o seu nome : " TO cNome
   ? "O seu nome é : ", cNome

RETURN
```

Prática número 4

Abra o arquivo basico.prg na pasta “pratica” e salve-o com o nome pratica_accept.prg.
Feito isso digite o conteúdo acima.

Após digitar a *string* tecle *ENTER*.

.:Resultado:.

```
Informe o seu nome : Vlademiro Landim Junior
O seu nome é :  Vlademiro Landim Junior
```

Dica 24

Você pode questionar porque temos um comando para receber do usuário dados numéricos e outro para receber dados caracteres. Esses questionamentos são pertinentes, mas não se preocupe pois eles (*ACCEPT* e *INPUT*) serão usados apenas no início do nosso aprendizado. Existem formas mais eficientes para a entrada de dados, mas o seu aprendizado iria tornar o processo inicial de aprendizado da linguagem mais demorado.
Apenas aprenda o "mantra" abaixo e você não terá problemas :

```
Para receber dados do tipo numérico = use INPUT.
Para receber dados do tipo caractere = use ACCEPT.
```

Se mesmo assim você quer saber o porque dessa diferença continue lendo, senão pode pular o trecho a seguir e ir para a seção de exemplos.
Na primeira metade da década de 1980 o dBase II foi lançado. Naquela época não tínhamos os recursos de hardware e de software que temos hoje, tudo era muito simples. Os primeiros comandos de entrada de dados usados pelo dBase foram o *INPUT* e o *ACCEPT*. O *ACCEPT* serve para receber apenas variáveis do tipo caracter. Você não precisa nem declarar a variável, pois se ela não existir o comando irá criá-la para você. O *INPUT* funciona da mesma forma, mas serve também para receber dados numéricos, caracteres, data e lógico. Porém você deve formatar a entrada de dados convenientemente, e é isso torna o seu uso confuso para outros tipos de dados que não sejam os numéricos.
Exemplos de uso do comando INPUT :

```
Usando com variáveis numéricas :

    nVal := 0
    INPUT "Digite o valor : " TO nVal

Usando com variáveis caracteres :

    cNome := " "
    INPUT "Digite o seu nome : " TO "cNome"
    // Note que  variável cNome deve estar entre parênteses
```

Vidal acrescenta que "o comando *INPUT* deve preferivelmente ser utilizado para a entrada de dados numéricos. Para a entrada de dados caracteres use o comando *ACCEPT*" [Vidal 1989, p. 107]. Para usar *INPUT* com dados do tipo data utilize a função CTOD() e com dados do tipo lógico apenas use o dado diretamente (.t. ou .f.)[a].

[a] Veremos os dados lógico e data mais adiante.

4.5 Exemplos

A seguir veremos alguns exemplos com os conceitos aprendidos. Fique atento pois nós usaremos os sinais referentes as quatro operações com números.

Dica 25

Antes de prosseguirmos note que, nas listagens dos códigos, nós acrescentamos um cabeçalho de comentários conforme abaixo :

```
/*
Um pequeno texto com a descrição do que a rotina faz.
Entrada : Dados de entrada.
Saída : Dados de saída.
*/
```

Acostume-se a documentar o seu código dessa forma: uma descrição breve, os dados de entrada e os dados de saída.

4.5.1 Realizando as quatro operações

O exemplo da listagem 6.3 gera um pequeno programa que executa as quatro operações com dois números que o usuário deve digitar. Note que o sinal de multiplicação é um asterisco (“*”) e o sinal de divisão é uma barra utilizada na escrita de datas (“/”) [5].

Listagem 4.12: As quatro operações

```
/*
As quatro operações
Entrada : dois números
Saída : As quatro operações realizadas com esses dois números
*/
PROCEDURE Main
LOCAL nValor1, nValor2 // Valores a serem calculados

   // Recebendo os dados
   ? "Introduza dois números para que eu realize as quatro oper.: "
   INPUT "Introduza o primeiro valor : " TO nValor1
   INPUT "Introduza o segundo valor : " TO nValor2

   // Calculando e exibindo
   ? "Soma.............. : " , nValor1 + nValor2
   ? "Subtraçäo........ : " , nValor1 - nValor2
   ? "Multiplicaçäo.... : " , nValor1 * nValor2
   ? "Divisäo.......... : " , nValor1 / nValor2

RETURN
```

[5] O sinal de uma operação entre variáveis recebe o nome de “operador”. Veremos mais sobre os operadores nos capítulos seguintes.

Prática número 5

Vá para a pasta “pratica” e abra o arquivo pratica_quatro.prg e complete o que falta. O resultado deve se parecer com o quadro abaixo.

.:Resultado:.

```
Introduza dois números para que eu realize as quatro oper.:
Introduza o primeiro valor : 10
Introduza o segundo valor : 20
Adição.......... :          30
Subtração....... :         -10
Multiplicação.... :         200
Divisão......... :           0.50
```

4.5.2 Calculando o antecessor e o sucessor de um número

O exemplo da listagem 4.13 gera um pequeno programa que descobre qual é o antecessor e o sucessor de um número qualquer inserido pelo usuário.

Listagem 4.13: Descobrindo o antecessor e o sucessor

```
/*
Descobre o  antecessor e o sucessor
*/
PROCEDURE Main
LOCAL nValor // Número a ser inserido

   // Recebendo os dados
   ? ""
   ? "**** Descobrindo o antecessor e o sucessor ****"
   ? ""
   INPUT "Introduza o número : " TO nValor

   // Calculando e exibindo
   ? "Antecessor........ : " , nValor - 1
   ? "Sucessor......... : " , nValor + 1

RETURN
```

Prática número 6

Abra o arquivo basico.prg na pasta “pratica” e salve-o com o nome pratica_antsuc.prg.
Feito isso digite o conteúdo acima.

.:Resultado:.

```
**** Descobrindo o antecessor e o sucessor ****

Introduza o número : 10
Antecessor........ :          9
Sucessor.......... :         11
```

Prática número 7

Faça um programa que calcule quantos números existem entre dois números pré-determinados, inclusive esses números. Abra o arquivo pratica_var.prg e complete os itens necessários para resolução desse problema.

Listagem 4.14: Calculando quantos números existem entre dois números

```
/*
Descrição: Calcula quantos números existem entre dois intervalos
           (incluídos os números extremos)
Entrada: Limite inferior (número inicial) e limite superior (número final)
Saída: Quantidade de número (incluídos os extremos)
*/
#define UNIDADE_COMPLEMENTAR 1 // Deve ser adicionada ao resultado final
PROCEDURE Main
LOCAL nIni, nFim // Limite inferior e superior
LOCAL nQtd // Quantidade

   ? "Informa quantos números existem entre dois intervalos"
   ? "(Incluídos os números extremos)"
   INPUT "Informe o número inicial : " TO nIni
   INPUT "Informe o número final : " TO nFim
   nQtd := nFim - nIni + UNIDADE_COMPLEMENTAR

   ? "Entre os números " , nIni, " e " , nFim, " existem ", nQtd, " números"

RETURN
```

.:Resultado:.

```
Informa quantos números existem entre dois intervalos
(Incluídos os números extremos)
Informe o número inicial : 5
Informe o número final : 10
Entre os números          5  e          10  existem
   6  números
```

4.6 Exercícios de fixação

As respostas encontram-se no apêndice H.

1. Escreva um programa que declare 3 variáveis e atribua a elas valores numéricos

através do comando INPUT; depois mais 3 variáveis e atribua a elas valores caracteres através do comando ACCEPT; finalmente imprima na tela os valores.

2. [Horstman 2005, p. 47] Escreva um programa que exibe a mensagem “Oi, meu nome é Hal!”. Então, em uma nova linha, o programa deve imprimir a mensagem “O que você gostaria que eu fizesse ?”. Então é a vez do usuário digitar uma entrada. [...] Finalmente, o programa deve ignorar a entrada do usuário e imprimir uma mensagem “Sinto muito, eu não posso fazer isto.”. Aqui está uma execução típica :

.:Resultado:.

```
Oi, meu nome é Hal!
O que você gostaria que eu fizesse ?
A limpeza do meu quarto.
Sinto muito, eu não posso fazer isto.
```

3. [Horstman 2005, p. 47] Escreva um programa que imprima uma mensagem “Oi, meu nome é Hal!”. Então, em uma nova linha, o programa deve imprimir a mensagem “Qual o seu nome ?” [...] Finalmente, o programa deve imprimir a mensagem “Oi, *nome do usuário*. Prazer em conhecê-lo!” Aqui está uma execução típica :

.:Resultado:.

```
Oi, meu nome é Hal!
Qual é o seu nome ?
Dave
Oi, Dave. Prazer em conhecê-lo.
```

4. Escreva um programa que receba quatro números e exiba a soma desses números.

5. Escreva um programa que receba três notas e exiba a média dessas notas.

6. Escreva um programa que receba três números, três pesos e mostre a média ponderada desses números.

7. Escreva um programa que receba o salário de um funcionário, receba o percentual de aumento (por exemplo 15 se for 15%) e exiba o novo salário do funcionário.

8. Modifique o programa anterior para exibir também o valor do aumento que o funcionário recebeu.

9. Modifique o programa anterior para receber também o percentual do imposto a ser descontado do salário novo do funcionário, e quando for exibir os dados (salário novo e valor do aumento), mostre também o valor do imposto que foi descontado.

10. Escreva um programa que receba um valor a ser investido e o valor da taxa de juros. O programa deve calcular e mostrar o rendimento e o valor total depois do rendimento.

11. Calcule a área de um triangulo. O usuário deve informar a base e a altura e o programa deve retornar a área.

    Dica : $nArea = \frac{nBase * nAltura}{2}$

12. Escreva um programa que receba o ano do nascimento do usuário e retorne a sua idade e quantos anos essa pessoa terá em 2045.

    Dica : YEAR(DATE()) é uma combinação de duas funções que retorna o ano corrente.

13. Escreva um programa que receba uma medida em pés e converta essa medida para polegadas, jardas e milhas.

    Dicas

    - 1 pé = 12 polegadas
    - 1 jarda = 3 pés
    - 1 milha = 1,76 jardas

14. Escreva um programa que receba dois números (nBase e nExpoente) e mostre o valor da potência do primeiro elevado ao segundo.

15. Escreva um programa que receba do usuário o nome e a idade dele . Depois de receber esses dados o programa deve exibir o nome e a idade do usuário convertida em meses. Use o exemplo abaixo como modelo :

    .:Resultado:.

```
  Digite o seu nome : Paulo
  Digite quantos anos você tem : 20

  Seu nome é Paulo e você tem aproximadamente 240 meses de
vida.
```

    Dica : Lembre-se que o sinal de multiplicação é um “*”.

16. Escreva um programa que receba do usuário um valor em horas e exiba esse valor convertido em segundos. Conforme o exemplo abaixo :

    .:Resultado:.

```
  Digite um valor em horas : 3
  3    horas tem       10800      segundos
```

17. Faça um programa que informe o consumo em quilômetros por litro de um veículo. O usuário deve entrar com os seguintes dados : o valor da quilometragem inicial, o valor da quilometragem final e a quantidade de combustível consumida.

4.7 Desafios

4.7.1 Identifique o erro de compilação no programa abaixo.

Listagem 4.15: Erro 1

```
/*
Onde está o erro ?
*/
PROCEDURE Main
LOCAL x, y, x // Número a ser inserido

   x := 5
   y := 10
   x := 20

RETURN
```

4.7.2 Identifique o erro de lógica no programa abaixo.

Um erro de lógica é quando o programa consegue ser compilado mas ele não funciona como o esperado. Esse tipo de erro é muito perigoso pois ele não impede que o programa seja gerado. Dessa forma, os erros de lógica só podem ser descoberto durante a seção de testes ou (pior ainda) pelo cliente durante a execução. Um erro de lógica quase sempre é chamado de *bug*.

Listagem 4.16: Erro de lógica

```
/*
Onde está o erro ?
*/
PROCEDURE Main
LOCAL x, y // Número a ser inserido

   x := 5
   y := 10
   ACCEPT "Informe o primeiro número : " TO x
   ACCEPT "Informe o segundo número : " TO y
   ? "A soma  é ", x + y

RETURN
```

4.7.3 Valor total das moedas

[Horstman 2005, p. 50] Eu tenho 8 moedas de 1 centavo, 4 de 10 centavos e 3 de 25 centavos em minha carteira. Qual o valor total de moedas ? Faça um programa que calcule o valor total para qualquer quantidade de moedas informadas.

Siga o modelo :

.:Resultado:.

```
Informe quantas moedas você tem :

Quantidade de moedas de 1 centavo :  10
Quantidade de moedas de 10 centavos : 3
Quantidade de moedas de 25 centavos : 4

Você tem     1.40 em moedas.
```

4.7.4 O comerciante maluco

Adaptado de [Forbellone e Eberspacher 2005, p. 62]. Um dado comerciante maluco cobra 10% de acréscimo para cada prestação em atraso e depois dá um desconto de 10% sobre esse valor. Faça um programa que solicite o valor da prestação em atraso e apresente o valor final a pagar, assim como o prejuízo do comerciante na operação.

5 Tipos de dados : Valores e Operadores

A sorte favorece a mente preparada.

Louis Pasteur

Objetivos do capítulo

- Entender o que é um operador e um tipo de dado.
- Aprender os tipos de dados básicos da linguagem Harbour.
- Entender os múltiplos papéis do operador de atribuição.
- Compreender os operadores de atribuição +=, -=, /= e *=
- Aprender os operadores unários de pré e pós incremento.
- Saber usar os parênteses para resolver problemas de precedência.
- Entender as várias formas de se inicializar uma data.
- Usar os SETs da linguagem.
- Ter noções básicas do tipo de dado NIL.

5.1 Definições iniciais

De acordo com Tenenbaum,

> um tipo de dado significa um conjunto de valores e uma sequencia de operações sobre esses valores. Este conjunto e essas operações formam uma construção matemática que pode ser implementada usando determinada estrutura de hardware e software [Tenenbaum, Langsam e Augenstein 1995, p. 18]

.

Portanto, o conceito de tipo de dado esta vinculado a dois outros conceitos : variáveis e operadores. O conceito de variável já foi visto, agora nós veremos os tipos de dados que as variáveis do Harbour podem assumir e os seus respectivos operadores.

De acordo com Damas,

> sempre que abrimos a nossa geladeira nos deparamos com uma enorme variedade de recipientes para todo tipo de produtos: sólidos, líquidos, regulares, irregulares etc. Cada um dos recipientes foi moldado de forma a guardar um tipo de bem ou produto bem definido. [...] Os diversos formatos de recipientes para armazenar produtos na nossa geladeira correspondem [...] aos tipos de dados. [Damas 2013, p. 22]

.

Por exemplo, uma variável que recebe o valor 10 é uma variável do tipo numérica. O recipiente a que Luis Damas se refere corresponde ao espaço reservado que conterá esse número. Se a variável for do tipo data, o recipiente será diferente, e assim por diante.

Mas isso não é tudo. Quando definimos uma variável de um tipo qualquer nós estamos definindo também uma série de operações que podem ser realizadas sobre esse tipo. Essas operações são representadas por sinais (“+”, “-”, “*”, etc.) e esses sinais recebem o nome técnico de *operador*. Por exemplo, uma variável numérica possui os operadores de soma, subtração, multiplicação, divisão e alguns outros. Se essa variável for do tipo caractere os operadores envolvidos serão outros, embora alguns se assemelhem na escrita. Podemos exemplificar isso pois nós já vimos, nos capítulos anteriores, um pouco sobre esses operadores :

```
a := 2
b := 3

? a + b  // Exibe 5

c := "2"
d := "3"

? c + d // Exibe  23
```

Embora o mesmo símbolo “+” tenha sido utilizado os resultados são diferentes porque as variáveis são de tipos diferentes. Na realidade, internamente, o operador

“+” que soma números é diferente do operador “+” que concatena caracteres. Quando isso ocorre dizemos que existe uma “sobrecarga” de operadores. Essa decisão dos projetistas de linguagens de programação tem se mostrado sábia, pois para nós fica mais fácil de assimilar. Já pensou se cada tipo de dado tivesse um símbolo diferente ? Por isso é que existe essa tal “sobrecarga” de operadores.

Veja a seguir alguns exemplos de operadores (“+”, “-”, “*” e “/”) que nós já vimos em alguns códigos anteriores. Lembre-se que as variáveis usadas armazenavam números.

```
a := 2
b := 3

? a + b  // Exibe 5
? a - b   // Exibe -1
? a * b   // Exibe 6
? a / b   //  0.6666666666666667
```

Da mesma forma, uma variável que recebe o valor “Aprovado com sucesso” é uma variável do tipo caractere. O tipo caractere possui alguns operadores em comum com o tipo numérico, um deles é o “+” que assume o papel de “concatenar” strings, conforme o exemplo abaixo :

```
a := "Aprovado com "
b := "Sucesso"
? a  + b  // Exibe "Aprovado com sucesso"
```

O conceito de tipo de dado e seus operadores, portanto, são as duas faces de uma mesma moeda. Uma vez definido uma variável como caractere, os tipos de operadores disponíveis são diferentes dos operadores disponíveis para uma variável data, por exemplo. O exemplo da listagem 5.1 serve para ilustrar o erro que acontece quando nós utilizamos operadores incompatíveis com o tipo de dado definido.

Listagem 5.1: Operadores e tipo de dado

```
/*
Tipo de dado e operador
*/
PROCEDURE Main
LOCAL x, y

   x := "Feliz "
   y := "Natal"

   ?  x + y  // Exibirá "Feliz Natal"

   ?  x / y  // Erro de execução (Operador / não pode ser usado em strings)
```

```
                                                                    12
RETURN                                                              13
```

.:Resultado:.

```
Feliz Natal
Error BASE/1084  Argument error: /
Called from MAIN(11)
```

O programa da listagem 5.1 foi executado corretamente até a linha 10, mas na linha 11 ele foi interrompido e uma mensagem de erro foi exibida: “Argument error : /”. O operador de divisão espera dois valores[1] numérico. Mas não foi isso o que aconteceu, os valores que foram passados foram caracteres, daí a mensagem “erro de argumento” (“argument error”).

Dica 26

Você só pode usar os operadores (sinais de operações) se os dados forem de mesmo tipo. Por exemplo: “2” + 2 é uma operação que não pode ser realizada, pois envolve uma string (caractere) e um número.

5.2 Variáveis do tipo numérico

Nós já vimos em códigos anteriores as variáveis do tipo numérico. Vimos também que uma maneira prática de se distinguir um tipo numérico de um tipo caractere é a presença ou ausência de aspas (strings tem aspas, números não). Também abordamos alguns operadores básicos: o “+”, “-”, “*” e “/”. A tabela 5.1 apresenta uma listagem parcial dos operadores matemáticos para tipos numéricos.

Tabela 5.1: Operadores matemáticos

Operação	Operador
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Exponenciação	^
Módulo (Resto)	%

5.2.1 As quatro operações

Como já vimos os quatro operadores básicos no capítulo anterior, apenas iremos apresentar a seguir alguns exemplos simples. A listagem 5.2 gera uma tabuada de somar.

[1] O termo técnico para designar os valores que são passados para uma instrução qualquer é “argumento”.

Listagem 5.2: Operador +.

```
/*
Exemplos de operadores numéricos
*/
PROCEDURE Main
LOCAL nNumero

   ? "Tabuada (Soma)"
   INPUT "Informe um número entre 1 e 9 :" TO nNumero

   ? "Soma"
   ? "----"
   ? nNumero, " + 1 = " , nNumero + 1
   ? nNumero, " + 2 = " , nNumero + 2
   ? nNumero, " + 3 = " , nNumero + 3
   ? nNumero, " + 4 = " , nNumero + 4
   ? nNumero, " + 5 = " , nNumero + 5
   ? nNumero, " + 6 = " , nNumero + 6
   ? nNumero, " + 7 = " , nNumero + 7
   ? nNumero, " + 8 = " , nNumero + 8
   ? nNumero, " + 9 = " , nNumero + 9

RETURN
```

Na execução abaixo o usuário digitou 5.

.:Resultado:.

```
Tabuada (Soma)
Informe um número entre 1 e 9 :5
Soma
----
         5  + 1 =           6
         5  + 2 =           7
         5  + 3 =           8
         5  + 4 =           9
         5  + 5 =          10
         5  + 6 =          11
         5  + 7 =          12
         5  + 8 =          13
         5  + 9 =          14
```

Prática número 8

Digite e execute o exemplo acima. Salve com o nome soma.prg

A listagem 5.3 gera uma tabuada de subtrair. Note que o seu código difere da tabuada de somar pois eu tenho que evitar a geração de números negativos (crianças no início do ensino fundamental ainda não tem esse conceito).

Listagem 5.3: Operador -.

```
/*
```

```
/*
Exemplos de operadores numéricos
*/
PROCEDURE Main
LOCAL nNumero

   ? "Tabuada (Subtração)"
   INPUT "Informe um número entre 1 e 9 :" TO nNumero

   ? "Subtraçäo"
   ? "---------"
   ? nNumero, " - " , nNumero, " = " , nNumero - nNumero
   ? nNumero + 1, " - " , nNumero, " = " , (nNumero + 1) - nNumero
   ? nNumero + 2, " - " , nNumero, " = " , (nNumero + 2) - nNumero
   ? nNumero + 3, " - " , nNumero, " = " , (nNumero + 3) - nNumero
   ? nNumero + 4, " - " , nNumero, " = " , (nNumero + 4) - nNumero
   ? nNumero + 5, " - " , nNumero, " = " , (nNumero + 5) - nNumero
   ? nNumero + 6, " - " , nNumero, " = " , (nNumero + 6) - nNumero
   ? nNumero + 7, " - " , nNumero, " = " , (nNumero + 7) - nNumero
   ? nNumero + 8, " - " , nNumero, " = " , (nNumero + 8) - nNumero
   ? nNumero + 9, " - " , nNumero, " = " , (nNumero + 9) - nNumero

RETURN
```

Na execução abaixo o usuário digitou 4.

.:Resultado:.

```
Tabuada (Subtração)
Informe um número entre 1 e 9 : 4
Subtração
---------
         4  -           4  =           0
         5  -           4  =           1
         6  -           4  =           2
         7  -           4  =           3
         8  -           4  =           4
         9  -           4  =           5
        10  -           4  =           6
        11  -           4  =           7
        12  -           4  =           8
        13  -           4  =           9
```

Prática número 9

Digite e execute o exemplo acima. Salve com o nome subtracao.prg

Prática número 10

Abra o arquivo soma.prg que você acabou de digitar e salve com o nome multiplicacao.prg. Depois altere o que for necessário para reproduzir a tabuada de multiplicar segundo os modelos já gerados.

Prática número 11

Abra o arquivo subracao.prg que você digitou e salve com o nome divisao.prg. Depois altere o que for necessário para reproduzir a tabuada de dividir segundo os modelos já gerados. O resultado deve se parecer com a tela a seguir (no exemplo o usuário digitou 4) :

.:Resultado:.

```
Tabuada (Divisão)
Informe um número entre 1 e 9 : 4
Divisão
-----------
         4  :           4  =          1.00
         8  :           4  =          2.00
        12  :           4  =          3.00
        16  :           4  =          4.00
        20  :           4  =          5.00
        24  :           4  =          6.00
        28  :           4  =          7.00
        32  :           4  =          8.00
        36  :           4  =          9.00
```

Abordaremos agora os operadores “^” e “%”.

5.2.2 O operador %

O operador módulo calcula o resto da divisão de uma expressão numérica por outra. Assim 11 % 2 resulta em 1.

```
a := 11
b := 2
? a % b // Imprime 1
```

Dica 27

Esse operador é útil dentro de códigos para determinar se um número é par ou ímpar. Se o resto de uma divisão por dois for zero então o número é par, caso contrário ele é ímpar. Nós veremos essa técnica quando formos estudar as estruturas de decisão da linguagem Harbour.

5.2.3 O operador de exponenciação (^)

O operador ^serve para efetuar a operação de exponenciação. O primeiro valor corresponde a base e o segundo valor é o expoente. O exemplo abaixo calcula 3^2 e 2^3, respectivamente.

```
a := 3
```

```
? a ^ 2 // Exibe 9

b := 2
? b ^ 3  // Exibe 8
```

Lembre-se que através da exponenciação nós podemos chegar a radiciação, que é o seu oposto. O exemplo abaixo calcula $\sqrt{100}$ e $\sqrt[3]{8}$, respectivamente.

```
a := 100
? a ^ ( 1 / 2 ) // Exibe 10

b := 8
? b ^ ( 1 / 3 )   // Exibe 2
```

5.2.4 Os operadores unários

Os operadores que podem facilmente passar desapercebidos aqui são os operadores unários + e -. Não confunda esses operadores com os operadores matemáticos de adição e subtração. Como o próprio nome sugere, um operador unário não necessita de duas variáveis para atuar, ele atua apenas em uma variável.

```
a := 3
? -a // Imprime -3

b := -1
? -b // Imprime 1
```

5.2.5 A precedência dos operadores numéricos

Outro detalhe a ser levado em conta é a precedência dos operadores. A ordem dos operadores matemáticos é :

1. Os operadores unários (+ positivo ou - negativo)
2. A exponenciação (^)
3. O módulo (%), a multiplicação (*) e a divisão (/).
4. A adição (+) e a subtração (-)

Listagem 5.4: Precedência de operadores matemáticos

```
/*
Tipo de dado e operador
*/
```

```
PROCEDURE Main
LOCAL x, y, z

   x := 2
   y := 3
   z := 4

   ? "Precedência de operadores matemáticos"
   ?
   ? "Dados x = ", x
   ? "      y = ", y
   ? "      z = ", z
   ?
   ? "Exemplos"
   ?
   ?  "x + y * z = " , x + y * z
   ?  "x / y ^ 2 = " , x / y ^ 2
   ?  " -x ^ 3   = " , -x ^ 3

RETURN
```

Prática número 12

Abra o arquivo oper02.prg na pasta “pratica” e complete o que falta.

.:Resultado:.

```
Precedência de operadores matemáticos

Dados x =           2
      y =           3
      z =           4

Exemplos

x + y * z =          14
x / y ^ 2 =           0.22
 -x ^ 3   =          -8.00
```

A ordem de resolução dos cálculos segue logo abaixo :

Dados x = 2, y = 3 e z = 4

2 + $\overbrace{3 * 4}^{1}$ = 14

2 / $\overbrace{3^2}^{1}$ = 0.22

(-2^3) = -8

Caso você deseje alterar a precedência (por exemplo, resolver primeiro a soma para depois resolver a multiplicação), basta usar os parênteses.

Dica 28

Você não precisa decorar a tabela de precedência dos operadores pois ela é muito grande. Ainda iremos ver outros operadores e a tabela final não se restringe a tabela 5.1. Se habitue a utilizar parênteses para ordenar os cálculos, mesmo que você não precise deles. Parênteses são úteis também para poder tornar a expressão mais clara :

```
a := 2
b := 3
z := 4

? a + b * z   // Resulta em 14
? a + ( b * z ) // Também resulta em 14, mas é muito mais claro
```

Se você **não** usar os parênteses você terá que decorar a precedência de todos os operadores.
Não perca tempo decorando precedências. Com a prática você aprenderá a maioria delas. O que você deve fazer é se habituar a usar os parênteses para organizar a forma com que uma expressão é avaliada. Aliás os parênteses são operadores também, mas que são avaliados em primeiro lugar. Essa regra dos parênteses vale para todas as linguagens de programação.

5.2.6 Novas formas de atribuição

Existe uma forma ainda não vista de atribuição. Observe atentamente o código 5.5.

Listagem 5.5: Uma nova forma de atribuição bastante usada.

```
/*
Atribuição de dados a variáveis
*/
PROCEDURE Main
LOCAL x  // Valores numéricos

   ? "Uma forma bastante usada de atribuição"
   x := 10
   ? "x vale ", x

   x := x + 10
   ? "agora vale ", x
   x := x + 10
   ? "agora vale ", x

   x := x * 10
   ? "agora vale ", x

   x := x - 10
```

```
   ? "agora vale ", x

   x := x / 10
   ? "agora vale ", x

RETURN
```

Prática número 13

Abra o arquivo oper06.prg na pasta “pratica” e complete o que falta.

.:Resultado:.

```
Uma forma bastante usada de atribuição
x vale          10
agora vale          20
agora vale          30
agora vale         300
agora vale         290
agora vale          29.00
```

Note que, desde que x tenha sido inicializado, ele pode ser usado para atribuir valores a ele mesmo. Isso é possível porque em uma operação de atribuição, primeiro são feito os cálculos no lado direito da atribuição para, só então, o resultado ser gravado na variável que esta no lado esquerdo da atribuição. Portanto, não existe incoerência lógica nisso. O que é necessário é que a variável x tenha sido previamente inicializada com um valor numérico. O código a seguir irá dar errado porque a variável não foi inicializada.

```
x := x * 2
```

Deveríamos fazer algo como :

```
x := 10
x := x * 2
```

Essas atribuições vistas são tão comuns nas linguagens de programação que o Harbour (e as outras linguagens também) tem um grupo de operadores que facilitam a escrita dessa atribuição. Por exemplo, para fazer x := x + 2 basta digitar x += 2. Alguns exemplos estão ilustrados a seguir :

```
x := 10
x := x * 2

Equivale a
```

```
x := 10
x *= 2
```

A tabela 5.2, retirada de [Vidal 1991, p. 98], resume bem esses operadores e seu modo de funcionamento.

Tabela 5.2: Operadores de atribuição compostos

Operador	Utilização	Operação equivalente
+=	a += b	a := (a + b)
-=	a -= b	a := (a - b)
=	.a *= b.	a := (a * b)
/=.	a /= b	a := (a / b)

Operadores de incremento e decremento

Se você entendeu a utilização dos operadores de atribuição compostos não será difícil para você entender os operadores de incremento e decremento. Eles funcionam da mesma forma que os operadores de atribuição já vistos, mas eles apenas somam ou subtraem uma unidade da variável. O exemplo a seguir irá ilustrar o funcionamento desses operadores.

O operador de incremento funciona assim :

```
Essa operação

    x := 10
    x := x + 1

 Equivale a

    x := 10
    ++x

 e também equivale a

    x := 10
    x++
```

A mesma coisa vale para o operador de decremento.

```
Essa operação

    x := 10
    x := x - 1
```

```
Equivale a

   x := 10
   --x

e também equivale a

   x := 10
   x--
```

Existe uma sutil diferença entre ++x e x++ e também entre - -x e x- -. Utilizados como prefixo (- -x ou ++x) esses operadores alteram o operando antes de efetuar a atribuição. Utilizados como sufixo (x- - ou x++) o operando é alterado e só depois a atribuição é efetuada. A listagem 5.6 exemplifica o que foi dito :

Listagem 5.6: Operadores de incremento e decremento.

```
/*
Incremento e decremento
*/
PROCEDURE Main
LOCAL nNum1 , nNum2  // Valores numéricos


   // Operador de pré-incremento ++
   nNum1 := 0
   nNum2 := ++nNum1
   ? nNum1 // Vale 1
   ? nNum2 // Vale 1

   // Operador de pós-incremento ++
   //inicio
   nNum1 := 0
   nNum2 := nNum1++
   ? nNum1 // Vale 1
   ? nNum2 // Vale 0
   //fim

   // Operador de pré-decremento --
   nNum1 := 1
   nNum2 := --nNum1
   ? nNum1 // Vale 0
   ? nNum2 // Vale 0

   // Operador de pós-decremento --
   //inicio
   nNum1 := 1
   nNum2 := nNum1--
   ? nNum1 // Vale 0
   ? nNum2 // Vale 1
   //fim
```

```
   // Os dois operadores em conjunto
   //inicio
   nNum1 := 1
   nNum2 := 5
   ? nNum1-- * 2 + ( ++nNum2 ) // Mostra 8
   // O cálculo efetuado foi :
   // 1 * 2 + 6
   // Mostra 8
   ? nNum1 // Vale 0
   ? nNum2 // Vale 6
   //fim

RETURN
```

.:Resultado:.

```
1
1
1
0
0
0
0
1
8
0
6
```

Prática número 14

Abra o arquivo oper07.prg na pasta “pratica” e complete o que falta.

Dica 29

Os operadores de atribuição compostos e os operadores de incremento e de decremento são muito usados, não apenas pela linguagem Harbour, mas por muitas outras linguagens de programação. Em todas elas o modo de funcionamento é o mesmo.

5.3 Variáveis do tipo caractere

Já vimos as variáveis do tipo caractere nos capítulos anteriores, vimos que elas armazenam strings (cadeias de caracteres). Essas variáveis não podem realizar somas com números nem outras operações, porque elas somente realizam operações de concatenação, por exemplo : “2” + “2” = “22”.

As variáveis do tipo caractere possuem operadores “+” e “-”. O operador de “+” é largamente usado, mas o operador de “-” é praticamente um desconhecido. Nós já vimos no capítulo anterior alguns exemplos com o operador “+”, nesse nós veremos alguns detalhes adicionais. Esses operadores não realizam as mesmas operações

que eles realizariam se as variáveis fossem numéricas, afinal de contas, não podemos somar nem subtrair strings. Veremos na listagem 5.7 o uso de ambos os operadores. Note que usamos uma função chamada *LEN*, cujo propósito é informar a quantidade de caracteres de uma string. Nós resolvemos mostrar a quantidade de caracteres para que você note que a quantidade de caracteres de uma string não muda, mesmo quando ela tem espaços. O uso de LEN está descrito logo a seguir.

```
PROCEDURE Main
LOCAL cNome := " Algoritmo "

    ? LEN( cNome )  // Imprime o número 11 (Conta com os espaços)

RETURN
```

.:Resultado:.

```
 11
```

A listagem 5.7 a seguir ilustra o uso dos operadores de concatenação “+” e “-”. Ao final de cada string nós imprimimos um “X” maiúsculo para dar a ideia de onde a string termina.

Listagem 5.7: Operadores de concatenação

```
/*
Operadores de strings + e -

Note que todas as strings tem, propositadamente, um
espaço em branco no seu início e outro no seu final.

*/
PROCEDURE Main
LOCAL cPre , cPos
LOCAL cNome

   cPre := "    Harbour     " // String tamanho 16
   cPos := "    Project     " // String tamanho 16

   ? "Exemplo 1 : Concatenando com +"
   ? cPre + cPos
   ?? "X" // Vai ser impresso  após o término da linha acima
   ? "O tamanho da string acima é de " , Len( cPre + cPos )
   ?
   ? "Exemplo 2 : Concatenando com -"
   ? cPre - cPos
   ?? "X" // Vai ser impresso  após o término da linha acima
   ? "O tamanho da string acima é de " , Len( cPre - cPos )

   //inicio
   cNome := " (www.harbour-project.org) "
   ?
   ? "Exemplo 3 : Concatenando três strings com +"
```

```
   ? cPre + cPos + cNome                                                    29
   ?? "X" // Vai ser impresso  após o término da linha acima                30
   ? "O tamanho da string acima é de " , Len( cPre + cPos + cNome)          31
   ?                                                                        32
   ? "Exemplo 4 : Concatenando três strings com -"                          33
   ? cPre - cPos - cNome                                                    34
   ?? "X" // Vai ser impresso  após o término da linha acima                35
   ? "O tamanho da string acima é de " , Len( cPre - cPos - cNome)          36
   //fim                                                                    37
                                                                            38
RETURN                                                                      39
```

Prática número 15

Na pasta pratica existe um arquivo chamado operstr.prg para você completar.

.:Resultado:.

```
Exemplo 1 : Concatenando com +
    Harbour         Project     X
O tamanho da string acima é de          32

Exemplo 2 : Concatenando com -
    Harbour    Project          X
O tamanho da string acima é de          32

Exemplo 3 : Concatenando três strings com +
    Harbour         Project      (www.harbour-project.org) X
O tamanho da string acima é de          59

Exemplo 4 : Concatenando três strings com -
    Harbour    Project (www.harbour-project.org)          X
O tamanho da string acima é de          59
```

Quando o “+” e o “-” são usados com strings, o termo correto é “concatenação” de strings, e não “soma” ou “subtração” de strings. Os operadores “+” e “-” realizam diferentes tipos de concatenações, o operador de “+” é largamente usado com strings, ele simplesmente une as duas strings e não altera os espaços em branco, mas o operador “-” possui uma característica que pode confundir o programador. De acordo com Vidal, “o operador menos realiza a concatenação sem brancos, pois ela remove o espaço em branco do início da string da direita, contudo os espaços em branco que precedem o operador são movidos para o final da cadeia de caracteres” [Vidal 1991, p. 91]. Por esse motivo nós decidimos imprimir o “X” maiúsculo ao final da concatenação, pois ele nos mostra que o tamanho final da string não se alterou.

Dica 30

Evite concatenar strings com o operador menos, existem funções que podem realizar o seu intento de forma mais clara para o programador. Uma concatenação com o operador menos só se justifica nos casos em que a performance é mais importante do que a clareza do código .Por exemplo, no interior de um laço[a] que se repetirá milhares de vezes (um operador é executado mais rapidamente do que uma função).

[a]Veremos o que é um laço nos próximos capítulos.

O operador += também funciona com variáveis caractere, no exemplo a seguir a variável x terminará com o valor “Muito obrigado”.

```
x := "Muito"
x := x + " obrigado"

Equivale a

x := "Muito"
x += " obrigado"
```

5.4 Variáveis do tipo data

As antigas linguagens que compõem o padrão xBase foram criadas para suprir a demanda do mercado por aplicativos comerciais, essa característica levou os seus desenvolvedores a criarem um tipo de dado básico para se trabalhar com datas (e dessa forma facilitar a geração de parcelas, vencimentos, etc.). O Harbour, como sucessor dessas linguagens, possui um tipo de dado primitivo que permite cálculos com datas. O tipo de dado data possuía uma particularidade que o distingue dos demais tipos de dados : ele necessita de uma função para ter o seu valor atribuído a uma variável, essa função chama-se **CTOD()** [2]. Com o Harbour surgiram outras formas de se inicializar um tipo de dado data. Nas próximas seções veremos as formas de inicialização e os cuidados que devemos ter com esse tipo de dado tão útil para o programador.

5.4.1 Inicializando uma variável com a função CTOD()

A forma antiga de se criar uma variável data depende de uma função chamada CTOD().

[2]Abreviação da frase (em inglês) “Character to Date” (“Caracter para Data”). Veremos esses conceitos com detalhes no capítulo sobre funções.

Descrição sintática 5

1. Nome : CTOD
2. Classificação : função.
3. Descrição : converte uma cadeia de caracteres em uma data correspondente.
4. Sintaxe

```
CTOD( <cData> ) -> dData
```

Fonte : [Nantucket 1990, p. 5-42]

Veja um exemplo da criação de uma variável data no código 5.8.

Listagem 5.8: Variável do tipo data

```
/*
Variável Data
*/
PROCEDURE Main


   dvencimento := CTOD( "08/12/2011")
   ? dvencimento

RETURN
```

.:Resultado:.

```
08/12/11
```

Um detalhe importante que com certeza passou desapercebido : a data impressa não foi “oito de dezembro de dois mil e onze”, mas sim “doze de agosto de dois mil e onze”!

Isso acontece porque o Harbour vem pré-configurado para exibir as datas no formato norte-americano (mês/dia/ano). Antes de prosseguirmos com o estudo das datas vamos fazer uma pequena pausa para configurar o formato da data a ser exibida. Como a nossa data obedece ao padrão dia/mês/ano (o padrão britânico), nós iremos configurar a exibição conforme a listagem abaixo usando *SET DATE BRITISH* . Se você quiser exibir o ano com quatro dígitos use *SET CENTURY ON* . Você só precisa usar esses dois comandos **uma vez no início do seu programa**, geralmente no inicio da procedure **Main()**. O exemplo da listagem 5.9 mostra a forma certa de se inicializar uma data.

Listagem 5.9: Variável do tipo data iniciadas corretamente

```
```

```
/*
Variável Data INICIADA COM O SET CORRETO
*/
PROCEDURE Main

   SET DATE BRITISH
   dvencimento := CTOD( "08/12/2011")
   ? dvencimento

RETURN
```

.:Resultado:.

```
08/12/11
```

Não há diferença visível, mas agora a data é “oito de dezembro de dois mil e onze”. Apesar de visualmente iguais, essas datas podem causar problemas com cálculos de prestações a pagar ou contas a receber caso você se esqueça de configurar o SET DATE.

Prática número 16

Reescreva o programa acima para exibir o ano com quatro dígitos. Você só precisa inserir o SET CENTURY com o valor correto no início do programa.

Dica 31

Quando for iniciar o seu programa, logo após a procedure Main, crie uma seção de inicialização e configuração de ambiente. Não se esqueça de configurar os acentos e as datas.

Dica 32

Quando você for usar a função CTOD para inicializar uma data, cuidado com o padrão adotado por SET DATE [a]. Se você gerar uma data inválida nenhum erro será gerado, apenas uma data nula (em branco) irá ocupar a variável. Por exemplo:

```
PROCEDURE Main()

   ? CTOD(''31/12/2015'') // Essa data NÃO é 31 de Dezembro de 2015
                          // porque o SET DATE é AMERICAN (mês/di
                          // e não BRITISH (dia/mês/ano)
                          // ela será exibida assim    /  /

RETURN
```

O que o exemplo acima quer mostrar é : "se você esquecer o SET DATE BRITISH no início do seu programa e declarar erroneamente uma data, conforme o exemplo, a data poderá ser uma data nula. A linguagem Harbour não acusa um erro com datas nulas.
Vamos dar outro exemplo: suponha que você esqueceu o SET DATE BRITISH e quer imprimir 01/02/2004 (primeiro de fevereiro de dois mil e quatro). Como você esqueceu o SET de configuração a data gerada no padrão americano : dois de janeiro de dois mil e quatro (mês/dia/ano).
Só mais um exemplo (eu sei que é chato ficar repetindo): suponha agora que você esqueceu o SET DATE BRITISH e quer imprimir 31/08/2004 (trinta e um de agosto de dois mil e quatro). Como a data padrão obedece ao padrão americano o sistema irá gerar uma data nula, pois no padrão americano 31/08/2004 gera um mês inexistente (mês 31 não existe).
Lembre-se, o SET DATE já possui um valor padrão, ou seja, ele sempre tem um valor. Mesmo que você não declare nenhum SET DATE, mesmo assim ele possui o valor padrão AMERICAN. Veremos esses detalhes mais adiante ainda nesse capítulo quando estudarmos os SETs.

[a] Ainda nesse capítulo estudaremos os SETs da linguagem Harbour. Apenas adiantamos aqui o seu uso por causa das datas

5.4.2 Inicializar usando a função STOD()

Um tipo de dado data já pronto para uso é uma grande vantagem para qualquer linguagem de programação. Mas a sua inicialização com a função CTOD possui dois inconvenientes :

1. Você precisa configurar a data para o padrão correto antes de usar. Na verdade esse não é um inconveniente muito grande, pois muitos sistemas foram desenvolvidos usando essa técnica. Basta não se esquecer de configurar a data apenas uma vez no início do seu programa com SET DATE BRITISH.

2. O segundo incoveniente é mais grave. Suponha que você vai utilizar o seu programa em outro país ou tornar ele um programa internacional. Com várias referências a datas e etc. Nesse caso, você pode ter problemas se ficar usando sempre o padrão dia/mês/ano para atribuir datas.

Pensando nisso surgiu a função STOD() [3]. Graças a ela você pode atribuir um padrão única para escrever as suas datas. Ela funciona assim :

```
PROCEDURE Main                                                                  1
                                                                                2
                                                                                3
   ? STOD(''20020830'') // Essa data é trinta de agosto de dois mil e dois.     4
                                                                                5
RETURN
```

O SET DATE não irá influenciar no valor da data. É claro que se eu quiser exibir a data no formato dia/mês/ano eu terei que usar SET DATE BRITISH, mas ele não tem mais influência sobre a forma com que a data é inicializada. Você deve ter notado que essa função converte uma string representando uma data em um tipo de dado data. O formato da string sempre é ano mês dia tudo junto, sem separadores.

5.4.3 Inicialização sem funções, usando o novo formato 0d

A terceira maneira de se inicializar uma data é a mais nova de todas. Essa maneira utiliza um formato especial criado especialmente para esse fim. A sintaxe dele é :

```
0d20160811  equivale à 11 de Agosto de 2016

0d -> Informa que é um valor do tipo data
2016 -> Ano
08 -> Mês
11 -> Dia
```

Veja um exemplo na listagem 5.10.

Listagem 5.10: Iniciando datas

```
                                                                1
                                                                2
                                                                3
PROCEDURE Main()                                                4
LOCAL dPagamento                                                5
                                                                6
                                                                7
   dPagamento := 0d20161201 // Primeiro de dezembro de 2016     8
   ? dPagamento                                                 9
```

[3] STOD significa “string to data”.

```
                                                                    10
                                                                    11
 RETURN                                                             12
```

.:Resultado:.

```
12/01/16
```

Os formatos de inicialização com CTOD e STOD são bastante úteis para o programador Harbour. Milhares de linhas de código foram escritas usando essas funções, principalmente a primeira, que é a mais antiga. Porém ainda temos um inconveniente: nós dependemos de uma função para criar um tipo de dado. Tudo bem, não é tão grave assim, mas tem um pequeno probleminha : funções podem inicializar datas nulas, o novo formato 0d não permite isso. Por exemplo, a listagem a seguir define uma constante simbólica com um valor data errado.

```
#define HOJE stod("20029999") // Data inválida                      1
PROCEDURE Main                                                      2
                                                                    3
   ? HOJE // Exibir  uma data nula, não um erro.                    4
                                                                    5
RETURN                                                              6
```

O exemplo a seguir cria uma constante simbólica do tipo data, que armazena a data de expiração de uma cópia de demonstração. A data é 15/09/2017, veja como ela independe de SET DATE (sempre será dia 15, mês 9 e ano 2017, não importa a ordem ou formato).

```
#define DATA_LIMITE_COPIA 0d20170915  // Programa expira em          1
PROCEDURE Main                                                      2
                                                                    3
   ? DATA_LIMITE_COPIA                                              4
   SET DATE BRITISH                                                 5
   ? DATA_LIMITE_COPIA                                              6
                                                                    7
RETURN                                                              8
```

.:Resultado:.

```
09/15/17
15/09/17
```

O principal motivo que justifica esse formato novo é que qualquer erro nessa constante irá gerar um erro de compilação, o que é menos ruim do que um erro de execução. Um erro de execução é pior do que um erro de compilação porque ele pode acontecer quando o seu programa já estiver em pleno uso pelos usuários. Quando o formato novo de data é adotado, o Harbour gera um erro e não deixa o programa ser gerado. Conforme o exemplo a seguir :

```
#define DATA_LIMITE_COPIA 0d20173109  // Essa data está errada
PROCEDURE Main

   ? DATA_LIMITE_COPIA
   SET DATE BRITISH
   ? DATA_LIMITE_COPIA

RETURN
```

.:Resultado:.

```
Error E0057  Invalid date constant '0d20173109'
```

O erro acima foi obtido durante a compilação do programa, o que é menos ruim do que se fosse obtido durante a execução do seu programa.

Essa forma alternativa de se iniciar um valor do tipo data ainda não é muito comum nos códigos porque a maioria dos códigos são legados da linguagem Clipper, que não possuía essa forma de inicialização. Porém você deve aprender a conviver com as três formas de inicialização.

O exemplo da listagem 5.11 ilustra as diversas formas de exibição de uma variável data usando a inicialização com CTOD. Procure praticar esse exemplo e modificá-lo, essa é uma etapa importante no aprendizado.

Listagem 5.11: Configurações de data

```

/*
Variável Data exibida corretamente
*/
PROCEDURE Main
LOCAL dVenc // Data de vencimento


   dVenc := CTOD( "12/31/2021") // 31 de dezembro de 2021 (mês/dia/ano)

   ? "Exibindo a data 31 de dezembro de 2021."
   ? "O formato padrão é o americano (mês/dia/ano)"
   ? dVenc

   SET DATE BRITISH // Exibe as datas no formato dia/mês/ano
   ? "Exibe as datas no formato dia/mês/ano"
   ? "O vencimento da última parcela é em " , dVenc

   SET CENTURY ON // Ativa a exibição do ano com 4 dígitos
   ? "Ativa a exibição do ano com 4 dígitos"
   ? "A mesma data acima com o ano com 4 dígitos : ", dVenc

   ? "Outros exemplos com outras datas : "
   ? CTOD("26/08/1970")
   dVenc := CTOD( "26/09/1996")
   ? dVenc
```

```
   dVenc := CTOD( "15/05/1999")
   ? dVenc

   SET CENTURY OFF // Data volta a ter o ano com 2 dígitos
   ? "Data volta a ter o ano com 2 dígitos"
   ? dVenc


RETURN
```

Prática número 17

Abra o arquivo ctod2.prg na pasta “pratica” e complete o que falta.

Prática número 18

Depois, substitua a inicialização com CTOD pela inicialização com STOD. Não se esqueça também de testar com a inicialização 0d. O segredo está na prática.

.:Resultado:.

```
Exibindo a data 31 de dezembro de 2021.
O formato padrão é o americano (mês/dia/ano)
12/31/21
Exibe as datas no formato dia/mês/ano
O vencimento da última parcela é em  31/12/21
Ativa a exibição do ano com 4 dígitos
A mesma data acima com o ano com 4 dígitos :  31/12/2021
Outros exemplos com outras datas :
26/08/1970
26/09/1996
15/05/1999
Data volta a ter o ano com 2 dígitos
15/05/99
```

5.4.4 Os operadores do tipo data

O tipo de dado data possui apenas dois operadores : o “+” e o “-”. Eles servem para somar ou subtrair dias a uma data. O operador “-” serve também para obter o número de dias entre duas datas [Vidal 1991, p. 92]. Por exemplo :

```
SET DATE BRITISH
dVenc := CTOD(''26/09/1970'')
? dVenc + 2 // Resulta em 28/09/70
? dVenc - 2 // Resulta em 24/09/70
```

Quando subtraímos duas datas o valor resultante é o número de dias entre elas. Veja o exemplo ilustrado na listagem 5.12.

Listagem 5.12: Operadores de data

```
/*
Variável Data exibida corretamente
*/
PROCEDURE Main
LOCAL dCompra // Data da compra
LOCAL dVenc // Data de vencimento

   SET DATE BRITISH
   dCompra := CTOD( "01/02/2015" ) // Data da compra
   dVenc := CTOD( "05/02/2015" ) // Data do vencimento

   ? "Data menor (compra) :", dCompra
   ? "Data maior (vencimento) :" , dVenc
   ? "Maior menos a menor (dias entre)"
   ? "Data de vencimento menos a Data da compra"
   ? dVenc - dCompra  // Resulta em 4
   ? "Menor menos a maior (dias entre)"
   ? "Data da compra menos a Data de vencimento"
   ? dCompra - dVenc  // Resulta em -4
   ? "Subtrai dois dias"
   ? "Data de vencimento menos dois dias"
   ? dVenc - 2 // Resulta em 03/02/15
   ? "Soma dois dias (data + número)"
   ? "Data de vencimento mais dois dias"
   ? dVenc + 2 // Resulta em 07/02/15
   ? "Soma dois dias (número + data )  "
   ? "2 + dVenc"
   ? 2 + dVenc // Resulta em 07/02/15 (a mesma coisa)


RETURN
```

Prática número 19

Abra o arquivo ctod4.prg em “pratica” e complete o que falta.

.:Resultado:.

```
Data menor (compra) : 01/02/15
Data maior (vencimento) : 05/02/15
Maior menos a menor (dias entre)
Data de vencimento menos a Data da compra
         4
Menor menos a maior (dias entre)
Data da compra menos a Data de vencimento
```

```
        -4
Subtrai dois dias
Data de vencimento menos dois dias
03/02/15
Soma dois dias (data + número)
Data de vencimento mais dois dias
07/02/15
Soma dois dias (número + data )
2 + dVenc
07/02/15
```

Note que tanto faz **2 + dVenc** ou **dVenc + 2**. O Harbour irá somar dois dias a variável **dVenc**.

Dica 33

Vimos na dica anterior os problemas de se inicializar uma variável data através da função CTOD. Reveja logo abaixo o exemplo :

```
PROCEDURE Main()

   ? CTOD("31/12/2015") // Essa data NÃO é 31 de Dezembro de 2015
                        // porque o SET DATE é AMERICAN (mês/dia/an
                        // e não BRITISH (dia/mês/ano)
                        // ela será exibida assim    /  /

RETURN
```

Esse mesmo exemplo com o padrão de inicialização novo (com 0d) não geraria o erro, pois ele obedece a um único formato.

```
PROCEDURE Main()

   ? 0d20151231     // Essa data É 31 de Dezembro de 2015
                    // COMO o SET DATE é AMERICAN (mês/dia/ano),
                    // ela será exibida assim : 12/31/15

RETURN
```

Mesmo assim, a grande maioria dos programas escritos em Harbour utilizam o padrão de inicialização através da função CTOD. Apesar da grande maioria dos programas usarem o formato de inicialização através da função CTOD, isso não significa que você deva usar também. Tudo irá depender de uma escolha pessoal sua e também, e mais importante, do padrão adotado pela equipe a qual você pertence. Se você trabalha só, inicialize a data com o padrão que você se sentir mais a vontade. Caso você seja membro de uma equipe que mantêm um sistema antigo é melhor você inicializar as datas com CTOD. Caso você queira usar um padrão diferente do da equipe, pergunte aos membros da equipe, principalmente ao chefe da equipe, se eles concordam com essa sua ideia.

5.4.5 Os operadores de atribuição com o tipo de dado data

Até agora vimos alguns exemplos do tipo de dado data em conjunto com o operador de atribuição :=. Todas as formas de atribuição que nós já vimos para os tipos numéricos e caracteres valem também para o tipo data.

```
PROCEDURE Main                                                    1
LOCAL dVenc := dPag := 0d20161211 // Atribuição múltipla          2
                                                                  3
   SET DATE BRITISH                                               4
   ? dVenc, dPag                                                  5
```

```
                                                                                6
RETURN                                                                          7
```

Os operadores += e -= também funcionam para variáveis do tipo data, da mesma forma que os já vistos operadores + e -. O funcionamento dos operadores += e -= segue o mesmo princípio do seu funcionamento com as variáveis numéricas e caracteres. Note, na listagem 5.13 que os operadores *= e /= não funcionam com datas (eles geram um erro de execução).

Listagem 5.13: Operadores += e -=

```
/*                                                                              1
Variável Data e operadores += e -=                                              2
*/                                                                              3
PROCEDURE Main                                                                  4
LOCAL dVenc // Data de vencimento                                               5
   SET DATE BRITISH                                                             6
   dVenc := CTOD( "08/12/2011")                                                 7
   dVenc += 2                                                                   8
   ? dVenc                                                                      9
   dVenc -= 2                                                                   10
   ? dVenc                                                                      11
   dVenc *= 2  // Operador *= e /= não funcionam com datas (ERRO!!)             12
   ? dVenc                                                                      13
                                                                                14
                                                                                15
RETURN                                                                          16
```

.:Resultado:.

```
10/12/11
08/12/11
Error BASE/1083  Argument error: *
Called from MAIN(12)
```

O operador -= pode ser usado também como subtração de datas. Mas o seu uso não é recomendado pois ele muda o tipo de dado da variável de data para numérico, e essa prática não é aconselhada. Veja o exemplo da listagem 5.14.

Listagem 5.14: Operadores -=

```
/*                                                                              1
Variável Data e operadores -=                                                   2
*/                                                                              3
PROCEDURE Main                                                                  4
LOCAL dVenc // Data de vencimento                                               5
LOCAL dPag // Data de pagamento                                                 6
                                                                                7
   SET DATE BRITISH                                                             8
   dVenc := CTOD( "08/12/2011")                                                 9
   dPag := CTOD( "02/12/2011")                                                  10
   dVenc -= dPag                                                                11
                                                                                12
   ? "A diferença entre o vencimento e o pagamento foi de ", dVenc, "dias"
```

```
                                                                    13
RETURN                                                              14
```

.:Resultado:.

```
A diferença entre o vencimento e o pagamento foi de          6 dias
```

Note que a variável dVenc era do tipo data (08 de dezembro de 2001) e depois da aplicação do operador ela virou 6. Isso é fácil de entender se você raciocinar conforme o exemplo a seguir :

```
dVenc := dVenc - dPag // Subtração entre datas retorna os dias entre
```

Dica 34

Se você entendeu as regras dos operadores de data + e - e dos operadores +=, -=, ++ e - - com números, então o entendimento do seu uso com as datas fica muito mais fácil.

Os operadores ++ e - - também funcionam de maneira análoga aos tipos de dados numéricos. Lembre-se que as regras de pós e pré incremento continuam valendo para as datas também.

5.5 Variáveis do tipo Lógico

Os valores do tipo lógico admitem apenas dois valores: falso ou verdadeiro. Eles são úteis em situações que requerem tomada de decisão dentro de um fluxo de informação, por exemplo. Para criar uma variável lógica apenas atribua a ela um valor .t. (para verdadeiro) ou .f. (para falso). Esse tipo de operador é o que demora mais para ser entendido pelo programador iniciante porque ele depende de outras estruturas para que alguns exemplos sejam criados. Tais estruturas ainda não foram abordadas (IF, CASE, operadores lógicos, operadoers comparativos, etc.), por isso vamos nos limitar a alguns poucos exemplos.

```
lPassou := .t. // Resultado do processamento informa que
                  // o aluno passou.
? lPassou
```

Note que o tipo de dado lógico inicia com um ponto e termina com um ponto. A letra “T” e a letra “F” que ficam entre os pontos podem ser maiúsculas ou minuúsculas.

```
.T.  // Verdade
.t.  // Verdade
```

```
.F.  // Falso
.f.  //  Falso
```

As variáveis lógicas possuem seus próprios operadores especiais, de modo que não abordaremos esses operadores agora. Os operadores “+”, “-”, “*” e “/” não se aplicam as variáveis lógicas. As variáveis lógicas tem apenas três operadores : .NOT. , .AND. e .OR. (a sintaxe deles começa com um ponto e termina com um ponto, assim como os valores lógicos .t. e .f.). Por enquanto só podemos entender completamente o operador .NOT., pois os outros dois dependem do entendimento dos operadores de comparação que nós ainda não vimos.

O operador .NOT. atua sobre uma única expressão. Se a expressão for verdadeira irá transformá-la em falsa e vice-versa. Por exemplo :

```
a := .t.
? .NOT. a // Falso
? !a // Falso
```

O operador .NOT. é um operador unário, ou seja, ele atua diretamente sobre o valor e não necessita de outra variável. O seu significado é simples, ele inverte o valor lógico de uma variável. Veja mais exemplos a seguir :

```
.NOT. .F.  // Verdade
.NOT. .f.  // Verdade

.NOT. .T.  // Falso
.NOT. .t.  //  Falso
```

O Harbour possui uma sintaxe alternativa do operador .NOT. chamada simplesmente de “!”. Assim, temos :

```
! .F.  // Verdade
! .f.  // Verdade

! .T.  // Falso
! .t.  //  Falso
```

Veremos o uso prático de valores lógicos nos próximos capítulos.

Dica 35

Uma confissão: lembro-me de quando estava aprendendo a programar e o professor apresentou as variáveis e os operadores lógicos. Hoje eu sei que eles são simples, mas naquela época eu fiquei sem entender o uso deles até que fui apresentado aos operadores de comparação e as estruturas de decisão. Por isso, se você está se sentindo desconfortável com esses operadores, não se preocupe pois o seu completo entendimento só se dará na sua aplicação prática nos capítulos seguintes.

5.6 De novo os operadores de atribuição

Nós já estudamos, nos capítulos anteriores, os operadores de atribuição, pois sem eles não poderíamos inicializar as variáveis. Mas vamos rever o que foi visto e acrescentar alguns poucos detalhes. Vimos que existem três formas de se atribuir um valor a uma variável.

1. Usando o comando *STORE*
2. Utilizando o operador *=*
3. Através do operador *:=* **(prefira esse)**

Nós vimos também que os operadores de atribuição possuem um triplo papel na linguagem Harbour : elas criam variáveis (caso elas não existam), atribuem valores as variáveis e podem mudar o tipo de dado de uma variável. Como ainda não tínhamos visto os tipos básicos de dados não poderíamos abordar esse terceiro papel que os operadores de atribuição possuem. Agora nós veremos os operadores de atribuição sendo usados para mudar o tipo de dado de uma variável, mas vamos logo adiantando que essa prática não é aconselhada, embora ela seja bastante vista nos códigos de programas.

Portanto, da mesma forma que você **não** deve usar um operador para declarar uma variável, você também não deve usá-lo para mudar o tipo de dado de uma variável. Procure usar o operador de atribuição apenas para atribuir valor a uma variável, essa deve ser a sua única função.

O exemplo da a seguir ilustra o uso do operador de atribuição para mudar o tipo de dado de uma variável. Veja como é simples.

```
PROCEDURE Main                                                  1
LOCAL xValor                                                    2
                                                                3
    xValor := 12                                                4
    ? xValor                                                    5
    xValor := "Cliente inadimplente"                            6
    ? xValor                                                    7
    SET DATE BRITISH                                            8
    xValor := DATE()                                            9
    ? xValor                                                    10
    xValor := .f.                                               11
```

```
    ? xValor                                                    12
    // O restante das instruções ...                            13
                                                                14
RETURN                                                          15
```

.:Resultado:.

```
12
Cliente inadimplente
21/08/16
.F.
```

Se por acaso você for “quebrar essa regra” e reaproveitar uma variável (esse momento pode chegar um dia) então coloque nela o prefixo “x” (como foi feito nesses exemplos acima) no lugar de um prefixo de tipo de dado. Um prefixo “x” indica que o tipo de dado da variável irá mudar no decorrer da rotina.

Dica 36

Use os operadores de atribuição preferencialmente para atribuir valores a uma variável (inicialização). Evite a prática de declarar uma variável com o operador de atribuição e também evite mudar uma variável já atribuída de um tipo de dado para outro. Se tiver de “quebrar essas regras” tenha cuidado redobrado (evite essa prática em rotinas extensas, por exemplo).

Encerramos nesse ponto a seção que trata dos quatro principais tipos de dados do Harbour : numérico, caractere, lógico e data. Vimos também a parte que faltava para fechar o assunto sobre os operadores de atribuição. O próximo capítulo tratará de um assunto que requer o conhecimento desses quatro tipos de dados, e que seria impossível de ser dado sem um prévio conhecimento do operador lógico. Trata-se dos operadores relacionais.

5.7 Operadores especiais

O Harbour possui também outros símbolos que são classificados como “operadores especiais”. Não iremos nos deter nesses “operadores” pois eles dependem de alguns conceitos que não foram vistos ainda. Mas iremos fazer uma breve lista :

1. Função ou agrupamento () : Já vimos o uso de parênteses para agrupar expressões e para a escrita da função Main(). Ele também é usado como uma sintaxe alternativa para o operador & (ainda não visto).
2. Colchetes [] : Também já vimos o seu uso como um delimitador de *strings*, mas ele também é usado para especificar um elemento de uma matriz (veremos o que é matriz em um capítulo a parte).
3. Chaves : Definição literal de uma matriz ou de um bloco de código. Será visto nos próximos capítulos.
4. Identificador de Alias -> : Não será visto nesse documento por fugir do seu escopo.

5. Passagem de parâmetro por referência : Será visto no tópico especial sobre funções.

6. Macro & : Será visto mais adiante.

5.8 O estranho valor NIL

O Harbour possui um valor chamado NIL [4]. Quando você cria uma variável e não atribui valor a ela, ela tem um tipo de dado indefinido e um valor indefinido. Esse valor é chamado de *NIL*.

Listagem 5.15: O valor NIL.

```
/*
O valor NIL
*/
PROCEDURE Main
LOCAL x

   ? "O valor de x é  : ", x    // O valor de x é  :  NIL

RETURN
```

Existem duas formas de uma variável receber o valor NIL.

1. Atribuindo o valor NIL a variável. Por exemplo : x := NIL

2. Inicializando-a sem atribuir valor. Por exemplo LOCAL x

O valor NIL não pode ser usado em nenhuma outra operação. Ele apenas pode ser atribuído a uma variável (operador :=) e avaliado posteriormente (operador ==). Nos próximos capítulos veremos alguns exemplos práticos envolvendo esse valor.

5.9 Um tipo especial de variável : *SETs*

A linguagem Harbour possui um tipo especial de variável. Esse tipo especial não obedece as regras até agora vistas, por exemplo, a atribuição não é através de operadores, nem podemos realizar quaisquer operações sobre elas. Essas variáveis especiais são conhecidas como *SETs* e possuem uma característica peculiar : quando o programa inicia elas são criadas automaticamente, de modo que a única coisa que você pode fazer é mudar o estado de um *SET* para um outro valor pré-determinado pela linguagem. Para ilustrar o funcionamento de um *SET* nós iremos usar um exemplo já visto : o *SET DATE*.

```
SET DATE BRITISH
dVenc := CTOD(''26/09/1970'')
? dVenc // Exibe 26/09/70
```

[4] *NIL* é uma palavra derivada do latim cujo significado é “nada”

De acordo com Spence, os *SETs* " são variáveis que afetam a maneira como os comandos operam. Um *SET* pode estar ligado ou desligado, pode, também, ser estabelecido um valor lógico" [Spence 1991, p. 59]. Complementando a definição de Spence, podemos acrescentar que o valor de um determinado *SET* pode ser também uma expressão ou um literal (Por exemplo : SET DECIMALS TO 4 ou SET DELIMITERS TO "[]"). Não se preocupe se você não entendeu ainda o significado dessas variáveis, o que você deve fixar no momento é :

1. Você não pode criar um *SET*. Todas essas variáveis especiais são criadas pelo programa automaticamente.
2. Não existem *SETs* sem valores. Todos eles possuem valores criados durante a inicialização do programa ou seja, um valor padrão[5]. Por exemplo, no caso de *SET DATE* (que altera a forma como a data é vista) o valor padrão é *AMERICAN*, que mostra a data no formato mês, dia e ano.
3. Você não pode atribuir qualquer valor a um *SET* porque ele possui uma lista de valores válidos. Cada *SET* possui a sua lista de valores válidos.

Outro exemplo de *SET*, que já foi visto, é o *SET CENTURY*, que determina se o ano de uma data deve ser exibido com dois ou com quatro dígitos. Esse tipo de *SET* permite apenas dois valores: "ligado" ou "desligado", conforme abaixo :

```
SET CENTURY ON  // Exibe o ano com quatro dígito nas datas
SET CENTURY OFF // Exibe o ano com dois dígito nas datas
```

Os valores *ON* e *OFF* são representações de valores lógicos.

Dica 37

Procure colocar todos os seus *SETs* juntos e na seção inicial do programa. Isso torna o programa mais fácil de ser lido. Geralmente essa seção é logo abaixo das declarações *LOCAL* da função *MAIN*. Procure também destacar essa seção colocando uma linha antes e uma linha depois da seção. Por exemplo :

```
FUNCTION Main()
LOCAL x,y,x

   // Configuração do ambiente de trabalho
   SET DATE BRITISH  // Data no formato dia/mês/ano
   SET CENTURY ON  // Ano exibido com quatro dígitos
   SET DECIMALS TO 4 // Os números decimais são exibidos com 4 casas
   //

   ... Continua...

RETURN NIL
```

[5]O termo técnico para "valor padrão" é valor *default*.

5.10 Exercícios de fixação

As respostas estão no apêndice H.

Vamos agora praticar um pouco mais resolvendo os seguintes exercícios de fixação do conteúdo. O assunto sobre os operadores ainda não acabou. No próximo capítulo veremos os operadores lógicos e as estruturas de decisão da linguagem. Mas antes, resolva os seguintes exercícios.

1. Calcule a área de um círculo. O usuário deve informar o valor do raio.

   Dica : $nArea = PI * nRaio^2$.

   Lembrete : Não esqueça de criar a constante PI (Onde PI = 3.1415) .

2. Escreva um programa que receba um valor numérico e mostre :

   - O valor do número ao quadrado.
   - O valor do número ao ao cubo.
   - O valor da raiz quadrada do número.
   - O valor da raiz cúbica do número.

   Nota : suponha que o usuário só irá informar números positivos.

   Dica : lembre-se que $\sqrt[2]{100} = 100^{\frac{1}{2}}$

3. Construa um programa para calcular o volume de uma esfera de raio R, em que R é um dado fornecido pelo usuário.

   Dica : o volume V de uma esfera é dado por $V = \frac{4*PI*Raio^3}{3}$.

   Veja se seus valores “batem” com a execução do programa a seguir.

   .:Resultado:.

```
Cálculo do volume de uma esfera
Informe o valor do raio da esfera 3
O volume da esfera é :       17832.47
```

4. Construa programas para reproduzir as equações abaixo. Considere que o lado esquerdo da equação seja a variável que queremos saber o valor. As variáveis do lado direito devem ser informadas pelo usuário. Siga o exemplo no modelo abaixo :

   IMPORTANTE: Estamos pressupondo que o usuário é um ser humano perfeito e que não irá inserir letras nem irá forçar uma divisão por zero.

   - $z = \frac{1}{x+y}$
     Modelo :

```
PROCEDURE Main
LOCAL x,y,z

     INPUT "Insira o valor de x : " TO x
```

```
     INPUT "Insira o valor de y : " TO y
     z = ( 1 / ( x + y ) )
     ? "O valor de z é : " , z


RETURN
```

.:Resultado:.

```
Insira o valor de x : 10
Insira o valor de y : 20
O valor de z é :           0.03
```

Agora faça o mesmo com as fórmulas seguintes.

- $x = \frac{y-3}{z}$
  Veja se seus valores “batem” com a execução do programa a seguir.

.:Resultado:.

```
Informe o valor de y : 1
Informe o valor de z : 2
        -1.00
```

- $k = \frac{x^3+z/5}{y^2+8}$
  Veja se seus valores “batem” com a execução do programa a seguir.

.:Resultado:.

```
Informe o valor de x : 1
Informe o valor de y : 3
Informe o valor de z : 4
         0.11
```

- $y = \frac{x^3 z}{r} - \frac{4n}{h}$
  Veja se seus valores “batem” com a execução do programa a seguir.

.:Resultado:.

```
Informe o valor de x : 2
Informe o valor de z : 3
Informe o valor de r : 4
Informe o valor de h : 5
Informe o valor de n : 6
      -114.00
```

- $t = \frac{y}{2} + \frac{3k^2}{4n}$
  Veja se seus valores “batem” com a execução do programa a seguir.

.:Resultado:.

```
Informe o valor de y : 1
```

```
Informe o valor de k : 2
Informe o valor de n : 3
        27.50
```

5. Crie um programa que leia dois valores para as variáveis cA e cB, e efetue a troca dos valores de forma que o valor de cA passe a possuir o valor de cB e o valor de cB passe a possuir o valor de cA. Apresente os valores trocados.

   Dica : Utilize uma variável auxiliar cAux.

6. São dadas três variáveis nA,nB e nC. Escreva um programa para trocar seus valores da maneira a seguir:

   - nB recebe o valor de nA
   - nC recebe o valor de nB
   - nA recebe o valor de nC

   Dica : Utilize uma variável auxiliar nAux.

7. Leia uma temperatura em graus Fahrenheit e apresentá-la convertida em graus Celsius. A fórmula de conversão é $nC = (nF - 32) * (5/9)$, onde nC é o valor em Celsius e nF é o valor em Fahrenheit.

8. Uma oficina mecânica precisa de um programa que calcule os custos de reparo de um motor a diesel padrão NVA-456. O custo é calculado com a fórmula $nCusto = \frac{nMecanicos}{0.4} * 1000$. O programa deve ler um valor para o número de mecânicos (nMecanicos) e apresentar o valor de custo (nCusto). Os componentes da equação do custo estão listados abaixo :

   - nCusto = Preço de custo de reparo do motor.
   - nMecanicos = Número de mecânicos envolvidos.
   - 0.4 = Constante universal de elasticidade do cabeçote.
   - 1000 = Constante para conversão em valor monetário.

9. Escreva um programa que receba um valor (par ou ímpar) e imprima na tela os 3 próximos sequenciais pares ou ímpares.

   Por exemplo

   .:Resultado:.

```
Informe o número : 4
     6,     8 e    10.
```

   outro exemplo (com um valor ímpar)

   .:Resultado:.

```
Informe o número : 5
    7,    9 e    11.
```

10. No final do capítulo anterior nós mostramos um exemplo de um programa que calcula a quantidade de números entre dois valores quaisquer (incluindo esses valores). A listagem está reproduzida a seguir :

codigos/pratica_var.prg

```
                                                                                1
/*                                                                              2
Descrição: Calcula quantos números existem entre dois intervalos                3
           (incluídos os números extremos)                                      4
Entrada: Limite inferior (número inicial) e limite superior (número final)      5
Saída: Quantidade de número (incluídos os extremos)                             6
*/                                                                              7
#define UNIDADE_COMPLEMENTAR 1 // Deve ser adicionada ao resultado final        8
PROCEDURE Main                                                                  9
LOCAL nIni, nFim // Limite inferior e superior                                  10
LOCAL nQtd // Quantidade                                                        11
                                                                                12
   ? "Informa quantos números existem entre dois intervalos"                    13
   ? "(Incluídos os números extremos)"                                          14
   INPUT "Informe o número inicial : " TO nIni                                  15
   INPUT "Informe o número final : " TO nFim                                    16
   nQtd := nFim - nIni + UNIDADE_COMPLEMENTAR                                   17
                                                                                18
   ? "Entre os números " , nIni, " e " , nFim, " existem ", nQtd, " número      19
                                                                                20
RETURN
```

Modifique esse programa (salve-o como excluidos.prg) para que ele passe a calcular a quantidade entre dois valores quaisquer, excluíndo esses valores limites.

6 Algoritmos e Estruturas de controle

O homem é um animal utilizador de ferramentas... Sem ferramentas ele não é nada, com ferramentas ele é tudo.

Thomas Carlyle

Objetivos do capítulo

- Entender o que é uma estrutura de controle.
- O que é um algoritmo.
- Aprender a representar os códigos já vistos na forma de algoritmos.
- Aprender o básico sobre estruturas de decisão.
- Aprender o básico sobre as estruturas de repetição.

6.1 Estruturas de controle

Você pode até programar sem usar as estruturas de controle, mas dificilmente realizará algo útil. Esse capítulo é um importante divisor de águas, pois ele nos trás conceitos essenciais para quem quer realmente programar. Ele não é um capítulo difícil e os códigos, a partir dele, começarão a ficar mais interessantes. A partir desse ponto nós iremos começar a desenvolver algoritmos simples mas que constituem a base de qualquer programa de computador.

Até agora os programas que foram desenvolvidos obedecem a uma estrutura chamada sequencial (ou linear), basicamente resumida em entrada, processamento e saída. Mas a linguagem Harbour possui mais dois tipos de estruturas que permitem ao computador tomar decisões baseado nos valores de variáveis, e também executar um processamento de centenas de linhas [1]. Essas estruturas foram desenvolvidas na década de 1960 por Brian e Jacopini. O trabalho desses dois pesquisadores demonstrou que **todo e qualquer** programa de computador pode ser escrito em termos de somente três estruturas de controle : sequência, decisão e repetição [Deitel e Deitel 2001, p. 101].

```
                                  ⎧ Sequência
Estruturas de controle            ⎨ Decisão
                                  ⎩ Repetição
```

Nós já estudamos as estruturas de sequência. Nesse capítulo faremos uma pequena revisão das estruturas de sequência e veremos as estruturas de decisão e de repetição. Nós usaremos uma ferramenta nova, chamada “algoritmo”, para poder entender melhor cada uma dessas estruturas. No próximo capítulo começaremos a abordar essas estruturas através da linguagem Harbour, esse capítulo é uma introdução puramente teórica.

6.2 Algoritmos : Pseudo-códigos e Fluxogramas

O conceito central da programação e da ciência da computação é o de algoritmo [Guimarães e Lages 1994, p. 2]. Programar é construir algoritmos.

Até agora nós apresentamos os aspectos básicos de toda linguagem de programação utilizando a linguagem Harbour. Procuramos também apresentar algumas técnicas que produzem um código “limpo” e fácil de ser compartilhado e lido (até por você mesmo). Esses conceitos são importantes, mas você irá precisar de mais para poder criar programas capazes de resolver eficientemente um problema qualquer. É nesse ponto que entram os algoritmos. Se você nunca ouviu falar de algoritmos deve estar pensando em algum tipo de software, mas não é nada disso. Você só vai precisar de papel e caneta para isso, embora já existam bons softwares que ajudam na confecção de um algoritmo. Da mesma forma que um engenheiro faz um projeto antes de “ir construindo” o imóvel, um bom programador deve fazer um algoritmo antes de ir programando. Os problemas que nós resolvemos até agora são simples demais para recorrermos a algoritmos, mas a partir desse capítulo eles irão se tornar mais complexos e mais interessantes. Segundo Guimarães e Lages :

> A formulação de um algoritmo geralmente consiste em um texto contendo comandos (instruções) que devem ser executados numa ordem prescrita.

[1] Sem você ter que digitar essas linhas uma a uma, Deitel chama isso de *transferência de controle*

> Esse texto é uma representação concreta do algoritmo e [...] é expandido somente no espaço da folha de papel [Guimarães e Lages 1994, p. 2]

Nas próximas sub-seções nós abordaremos duas ferramentas simples que são usadas na confecção de algoritmos : o pseudo-código e o fluxograma.

6.2.1 Pseudo-código

O pseudo-código[2] como o próprio nome sugere é uma versão na nossa língua da linguagem de programação. A ideia é permitir que com um conjunto básico de primitivas seja possível ao projetista pensar **apenas** no problema, e não na linguagem de programação. Apenas tome cuidado para não ficar muito distante da linguagem.

6.2.2 Fluxograma

O fluxograma é um desenho de um processo que nos auxilia a ter uma visão geral do problema. Você pode usar fluxogramas em seus projetos futuros, desde que seja apenas para documentar pequenos trechos de código estruturado (uma função ou o interior de um método [3]) ou para tentar reproduzir um processo de trabalho em alto nível, como uma linha de montagem, a compra de mercadorias para revenda ou qualquer outro processo organizacional. Segundo Yourdon [Yourdon 1992, p. 275], grande parte das críticas em relação aos fluxogramas deve-se à má utilização nas seguintes áreas :

1. Como ferramenta erroneamente usada para modelar grandes porções de código. A lógica de um fluxograma é procedural e sequencial.
2. O fluxograma foi desenvolvido antes da criação das linguagens estruturadas, e nada impede que o programador (ou analista) crie modelos complexos e desestruturados de fluxogramas.

Nós usaremos basicamente apenas quatro componentes :

1. Um quadro retangular com as bordas arredondadas que representa o início e o fim do fluxograma.
2. Um quadro retangular que representa uma instrução qualquer ou um conjunto de instruções.
3. Um quadro em forma de losango que representa uma decisão.
4. Setas que interligam esses quadros

Se nós fossemos representar através de fluxogramas os códigos que nós desenvolvemos até agora só teríamos uma sequencia de retangulos interligados conforme a figura 6.1.

Nos próximos capítulos nós apresentaremos os comandos do Harbour para controle de fluxo com suas respectivas representações em fluxograma.

[2] Pseudo é uma palavra de origem grega que significa falso.

[3] Nos capítulos seguintes nós iremos criar as nossas próprias funções, por isso precisaremos aprender a usar algoritmos. Os métodos não serão abordados nesse livro.

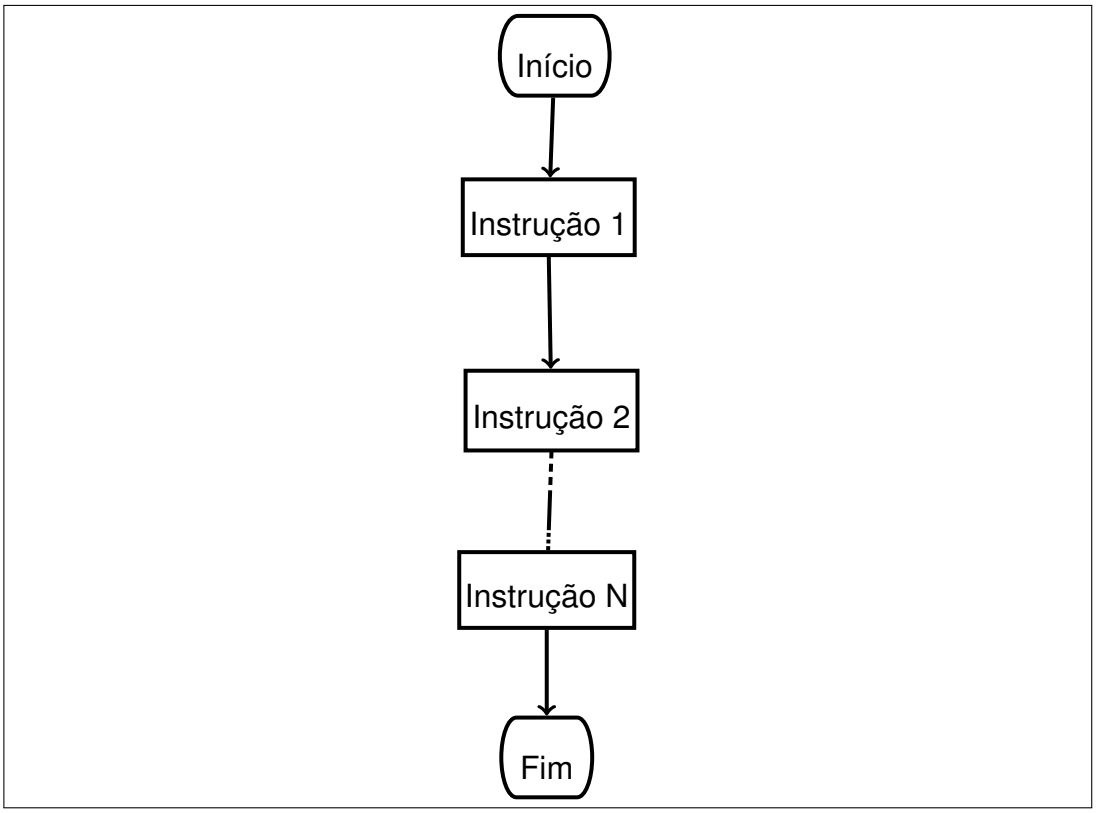


Figura 6.1: Uma estrutura sequencial

6.3 O que é uma estrutura sequencial

Algumas verdades são tão óbvias que fica difícil para o iniciante percebê-las. Quando nós afirmamos que “nós já estudamos as estruturas de sequência” você pode estar se perguntando : o que exatamente isso significa ? Seriam as variáveis ? Seriam os tipos de dados ? Por acaso seriam os operadores ?

A resposta é simples : não, não se trata disso. Quando nós usamos a palavra estrutura nós não estamos falando de uma instrução da linguagem Harbour, mas sim **da forma com a qual nós “estruturamos” um raciocínio para resolver um problema**.

Todos os programas que nós desenvolvemos até agora obedecem a um padrão bem claro. Ele inicia na parte superior do código, logo após a PROCEDURE Main, e vai “descendo” na direção de RETURN. Nesse meio termo existe um problema a ser resolvido. Essa “lista de instruções” define a estrutura que nós usamos até agora para programar : a estrutura sequencial.

No capítulo 3 desse livro nós fomos apresentados a estrutura sequencial. Vamos reproduzir novamente a rotina de troca de lâmpada, vista no início do capítulo 3.

```
ROTINA Trocar uma lâmpada

   1. Pegar uma escada.
   2. Posicionar a escada embaixo da lâmpada.
   3. Buscar uma lâmpada nova
```

```
   4. Subir na escada
   5. Retirar a lâmpada velha
   6. Colocar a lâmpada nova

FINAL DA ROTINA
```

A partir de agora iremos formalizar a exibição desse raciocínio, conforme a listagem intitulada “Algoritmo 1”.

Algoritmo 1: Estrutura de sequência

Entrada: Escada e lâmpada
Saída: Lâmpada trocada

```
1 início
2     Pegar uma escada
3     Posicionar a escada embaixo da lâmpada.
4     Buscar uma lâmpada nova
5     Subir na escada
6     Retirar a lâmpada velha
7     Colocar a lâmpada nova
8 fim
```

O Algoritmo 1 possui os seguintes elementos :

1. Título descritivo
2. Dados de entrada (mais ou menos como os ingredientes de uma receita).
3. Operações de saída (algo como o produto final).

Esse tipo de raciocínio é considerado o mais simples porque ele obedece a uma “sequência” bem definida de instruções. Esse tipo de estrutura aparece em todos os problemas computacionais. Por exemplo, o cálculo da área de um retângulo obedece a regras bem definidas. Não importa o tamanho do triângulo nem o seu tipo, a fórmula permanece a mesma.

Esse mesmo algoritmo seria representado assim na forma de fluxograma (figura 6.2) :

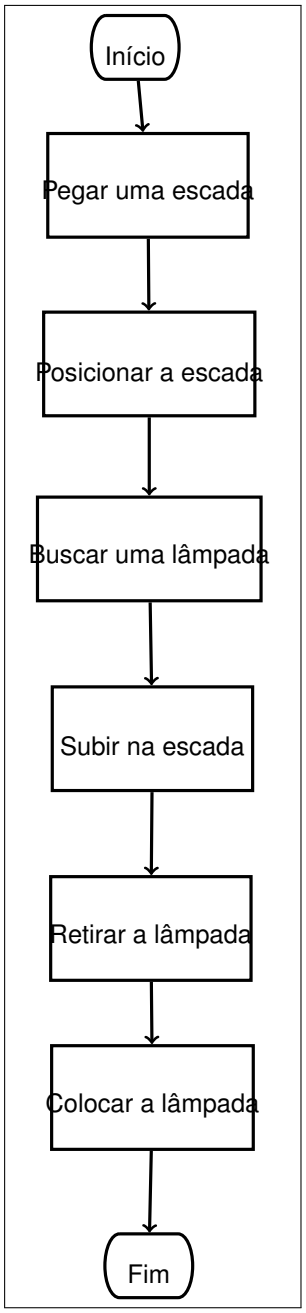


Figura 6.2: Troca de lâmpada usando a estrutura sequêncial.

Esse exemplo é puramente ilustrativo e didático, pois não faz sentido representar estruturas puramente sequenciais com fluxogramas. A estrutura sempre será uma sequência monótona de quadros retangulares “empilhados”.

6.4 Alguns algoritmos básicos

Acabamos de aprender um conceito novo: o algoritmo. Vimos que o existem algoritmos escritos em pseudo-códigos e também em fluxogramas. Vimos também que os fluxogramas não são usados para representar estruturas sequenciais, pois isso iria redundar em uma pilha de quadros. Iremos agora ver a representação de alguns

códigos já visto nos capítulos anteriores no formato de pseudo-código, pois isso nos ajudará na confecção dos futuros pseudo-códigos.

6.4.1 Hello World

Você com certeza se lembra desse código a seguir :

Listagem 6.1: Hello World

```
PROCEDURE Main

    ? "Hello World"

RETURN
```

A sua representação em pseudo-código é :

Algoritmo 2: Hello World

```
1 início
2 |  escreva "Hello World"
3 fim
```

Praticamente a mesma coisa.

6.4.2 Recebendo dados

Para receber dados nós vimos o comando INPUT (usado para receber números) e o comando ACCEPT (usado para receber strings). Abaixo temos um exemplo com o comando ACCEPT.

Listagem 6.2: Recebendo dados externos digitados pelo usuário

```

/*
Uso do comando ACCEPT
*/
PROCEDURE Main
LOCAL cNome // Seu nome

   /* Pede e exibe o nome do usuário */
   ACCEPT "Informe o seu nome : " TO cNome
   ? "O seu nome é : ", cNome

RETURN
```

No pseudo-código temos apenas um comando para receber dados: o comando “leia”. Como a função do pseudo-código é abstrair do problema as características da linguagem de programação, então não tem sentido termos dois comandos. O algoritmo seguinte usa o comando “leia”, que pode receber qualquer tipo de dado. Note também que não precisamos seguir as regras de nomenclatura para variáveis, inclusive podemos criar nossas variáveis acentuadas. Lembre-se: pseudo-códigos podem ser escritos até mesmo em folhas de guardanapo, não tem muitas regras para serem seguidas. Você pode inclusive estar bem distante de um computador, basta ter a

ideia e anotar em algum lugar. Note também que a variável não precisou ser declarada, embora alguns autores façam isso.

Algoritmo 3: Recebendo dados

```
1 início
2 |  leia Usuário
3 |  escreva "Seu nome é ", Usuário
4 fim
```

6.4.3 Cálculo

Você com certeza deve se recordar desse exemplo a seguir :

Listagem 6.3: As quatro operações

```
/*
As quatro operações
Entrada : dois números
Saída : As quatro operações realizadas com esses dois números
*/
PROCEDURE Main
LOCAL nValor1, nValor2 // Valores a serem calculados

   // Recebendo os dados
   ? "Introduza dois números para que eu realize as quatro oper.: "
   INPUT "Introduza o primeiro valor : " TO nValor1
   INPUT "Introduza o segundo valor : " TO nValor2

   // Calculando e exibindo
   ? "Soma.............. : " , nValor1 + nValor2
   ? "Subtraçäo........ : " , nValor1 - nValor2
   ? "Multiplicaçäo.... : " , nValor1 * nValor2
   ? "Divisäo.......... : " , nValor1 / nValor2

RETURN
```

O seu equivalente em pseudo-código :

Algoritmo 4: As quatro operações

```
1 início
2 |  leia Valor1
3 |  leia Valor2
4 |  escreva "Soma : ", Valor1 + Valor2
5 |  escreva "Subtração : ", Valor1 - Valor2
6 |  escreva "Multiplicação : ", Valor1 * Valor2
7 |  escreva "Divisão : ", Valor1 / Valor2
8 fim
```

Note que os sinais (operadores) são os mesmos. Na nossa cultura, nós representamos o operador de divisão por “:” e o de multiplicação por um “x” ou um ponto. Faça da forma que quiser, mas nos nossos exemplos nós iremos manter o formato dos operadores do Harbour.

6.4.4 Atribuição

O operador do Harbour para atribuição recomendado é o ":=". Lembra do código abaixo ?

```
PROCEDURE Main
LOCAL nValor := 12

    ? nValor
    // O restante das instruções ...

RETURN
```

Em pseudo-código ele ficaria assim :

Algoritmo 5: Atribuição

```
1 início
2 |   valor ← 12
3 |   escreva valor
4 fim
```

ou assim :

Algoritmo 6: Atribuição

```
1 início
2 |   valor := 12
3 |   escreva valor
4 fim
```

Você decide. Como você deve ter percebido, a liberdade na escrita de pseudo-códigos é muito grande. Com apenas uma seção de capítulo nós resumimos aproximadamente os três capítulos anteriores em linguagem Harbour (e em qualquer linguagem de programação). Isso acontece porque o pseudo-código serve para estruturar o seu raciocínio sem os muitos empecilhos que a linguagem traria. Programar é uma atividade criativa, e o processo criativo necessita dessa liberdade. Apenas lembre-se de escrever o seu código corretamente (seguindo as regrinhas da linguagem e as boas práticas) na hora de transcrever o código do pseudo-código para a linguagem Harbour.

6.5 Exercícios de aprendizagem

Nesses exercícios, nós apresentaremos o algoritmo em pseudo-código e você deve converter esse algoritmo em um programa escrito em Harbour. Siga o exemplo dos exercício resolvido.

1. **[Exercício resolvido]** Um professor tem uma turma de cinco alunos com as suas respectivas notas. Esse professor deseja calcular e imprimir a média da turma ao lado de cada nota do aluno. O algoritmo do programa está escrito em pseudo-código. A sua tarefa é converter esse algoritmo em um programa em Harbour. (Tente resolver esse exercício antes, só depois vá comparar com a resolução dele).

Algoritmo 7: Comparativo nota x média da turma

Entrada: As notas dos alunos
Saída: A impressão de cada nota com a média da turma ao lado

```
1  início
2      leia nota1
3      leia nota2
4      leia nota3
5      leia nota4
6      leia nota5
7      média ← ( nota1 + nota2 + nota3 + nota4 + nota5 ) / 5
8      escreva "Aluno 1 : ", nota1, "Média : ", média
9      escreva "Aluno 2 : ", nota2, "Média : ", média
10     escreva "Aluno 3 : ", nota3, "Média : ", média
11     escreva "Aluno 4 : ", nota4, "Média : ", média
12     escreva "Aluno 5 : ", nota5, "Média : ", média
13 fim
```

Resposta

Você é livre para realizar alguns acréscimos durante a conversão do algoritmo em código, mas vale abrir mão das boas práticas adotadas :

- Declare as variáveis como LOCAL antes de usá-las.
- Ao criar nomes para as variáveis não use acentos. Note que no algoritmo os acentos podem aparecer, mas eles não podem ser usados no seu código para nomear suas variáveis.
- Use as regras de nomenclatura para nomear as variáveis.
- Transforme os números mágicos em constantes simbólicas.

Note que nós transcrevemos a descrição do algoritmo, as entradas e as saídas para o comentário no topo da rotina Harbour.

Listagem 6.4: Comparativo nota x média da turma

```
/*
Comparativo nota x média da turma
Entrada : As notas dos alunos
Saída :  A impressão de cada nota com a média da turma ao lado
*/
#define TOTAL_DE_ALUNOS 5
PROCEDURE Main
LOCAL nNota1, nNota2, nNota3, nNota4, nNota5 // Notas dos alunos
LOCAL nMedia // Media

   INPUT "Informe a nota do aluno 1 : " TO nNota1
   INPUT "Informe a nota do aluno 2 : " TO nNota2
   INPUT "Informe a nota do aluno 3 : " TO nNota3
   INPUT "Informe a nota do aluno 4 : " TO nNota4
   INPUT "Informe a nota do aluno 5 : " TO nNota5

```

```
                                                                    17
   nMedia := ( nNota1 + nNota2 + nNota3 + nNota4 ) / TOTAL_DE_ALUNOS
                                                                    18
   ? "Nota do aluno 1 : ", nNota1, "Média da turma : " , nMedia     19
   ? "Nota do aluno 2 : ", nNota2, "Média da turma : " , nMedia     20
   ? "Nota do aluno 3 : ", nNota3, "Média da turma : " , nMedia     21
   ? "Nota do aluno 4 : ", nNota4, "Média da turma : " , nMedia     22
   ? "Nota do aluno 5 : ", nNota5, "Média da turma : " , nMedia     23
                                                                    24
RETURN                                                              25
```

2. O algoritmo a seguir converte uma temperatura de Celsius para Fahrenheit. Transforme esse algoritmo em um programa. Importante: não precisa eliminar os números mágicos 9, 160 e 5. Essa é uma exceção a regra dos números mágicos pois esses números compõem uma fórmula.

Algoritmo 8: Cálcula e exibe uma temperatura Celsius em Fahrenheit.

Entrada: Temperatura em Celsius
Saída: Temperatura em Fahrenheit

```
1  início
2  |  leia Celsius
3  |  Fahrenheit ← ( 9 * Celsius + 160 ) / 5
4  |  escreva Fahrenheit
5  fim
```

3. O algoritmo a seguir calcula o salário líquido de um professor. Transforme esse algoritmo em um programa.

Algoritmo 9: Cálcula e exibe o salário bruto e o salário líquido de um professor

Entrada: Horas trabalhadas, valor da hora e percentual de desconto
Saída: Salário bruto e o salário líquido

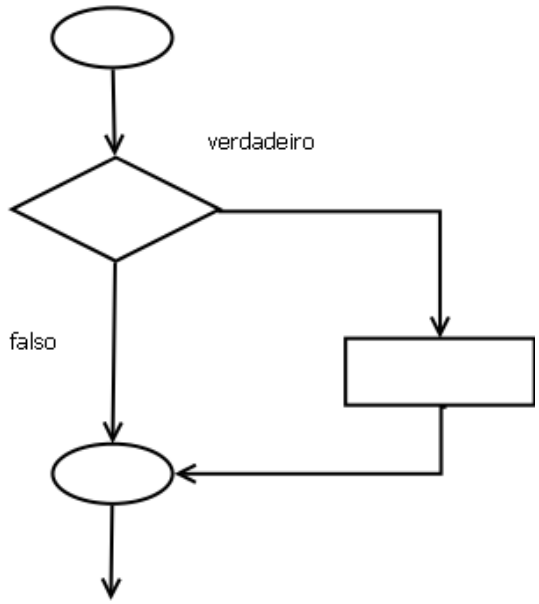
```
1  início
2  |  leia HorasTrabalhadas
3  |  leia ValorDaHora
4  |  leia PercDesconto
5  |  SalBruto ← HorasTrabalhadas * ValorDaHora
6  |  TotalDeDesconto ← ( PercDesconto / 100 ) * SalBruto
7  |  SalLiq ← SalBruto - TotalDeDesconto
8  |  escreva SalBruto
9  |  escreva SalLiq
10 fim
```

6.6 Estruturas de decisão

As estruturas de decisão (também chamadas de “estruturas de seleção”) não possuem a linearidade encontrada nas estruturas sequenciais. De acordo com Forbellone e Eberspächer, estruturas de decisão são aquelas que “permitem a escolha de um grupo de ações (bloco) a ser executado quando determinadas **condi-**

ções, representadas por expressões lógicas ou relacionais são ou não satisfeitas" [Forbellone e Eberspacher 2005, p. 33]. Uma representação em fluxograma de uma estrutura de decisão pode ser vista na figura 6.3. O losango representa o ponto onde a decisão é tomada.

Figura 6.3: Estrutura de decisão (Seleção única)



Uma estrutura de decisão permite que determinados blocos das rotinas sejam executados apenas se o resultado de uma decisão for considerada verdadeira ou falsa (lembra do tipo de dado lógico, visto no capítulo anterior ?). Para ilustrar isso, vamos acrescentar mais algumas linhas à nossa rotina (algoritmo 10) de trocar a lâmpada (exemplo baseado em [Forbellone e Eberspacher 2005, p. 4]).

Algoritmo 10: Troca de lâmpada usando uma estrutura de decisão.

```
   Entrada: Escada e lâmpada
   Saída: Lampada trocada
1  início
2      Pegar uma escada.
3      Posicionar a escada embaixo da lâmpada.
4      Buscar uma lâmpada nova
5      Subir na escada
6      Retirar a lâmpada velha
7      Colocar a lâmpada nova
8      Descer da escada
9      Acionar o interruptor
10     se lâmpada não acender então
11         Buscar outra lâmpada
12         Subir na escada
13         Retirar a lâmpada com defeito
14         Colocar a lâmpada nova
15         Descer da escada
16         Acionar o interruptor
17 fim
```

Se você prestar bem atenção a essa rotina, verá que ela ainda tem uma falha na sua lógica. Por exemplo: e se a segunda lâmpada estiver com defeito ? Não temos uma ação para tomar nesse caso. Mas não vamos nos preocupar com isso agora, pois essa rotina só ficará completamente finalizada quando estudarmos as estruturas de repetição. Por enquanto vamos nos concentrar na linha 10 do nosso algoritmo. Note que esse passo requer uma decisão da pessoa que está trocando a lâmpada. Se a lâmpada não acender, as linhas seguintes são executados , caso contrário a rotina se encerra. Note também que esses passos estão indentados. Esse recuo é para indicar que esses passos pertencem a um bloco que pode ou não ser executado, e essa decisão é tomada na linha 10 através da instrução *SE*.

Esse mesmo algoritmo está representado na forma de fluxograma (figura 6.4). Note que nós agrupamos vários passos dentro de apenas um retângulo para poder melhorar a visualização do fluxograma. Esse agrupamento de vários passos em apenas um retângulo não é uma prática muito usada, normalmente os fluxogramas possuem os passos bem definidos nos seus retângulos.

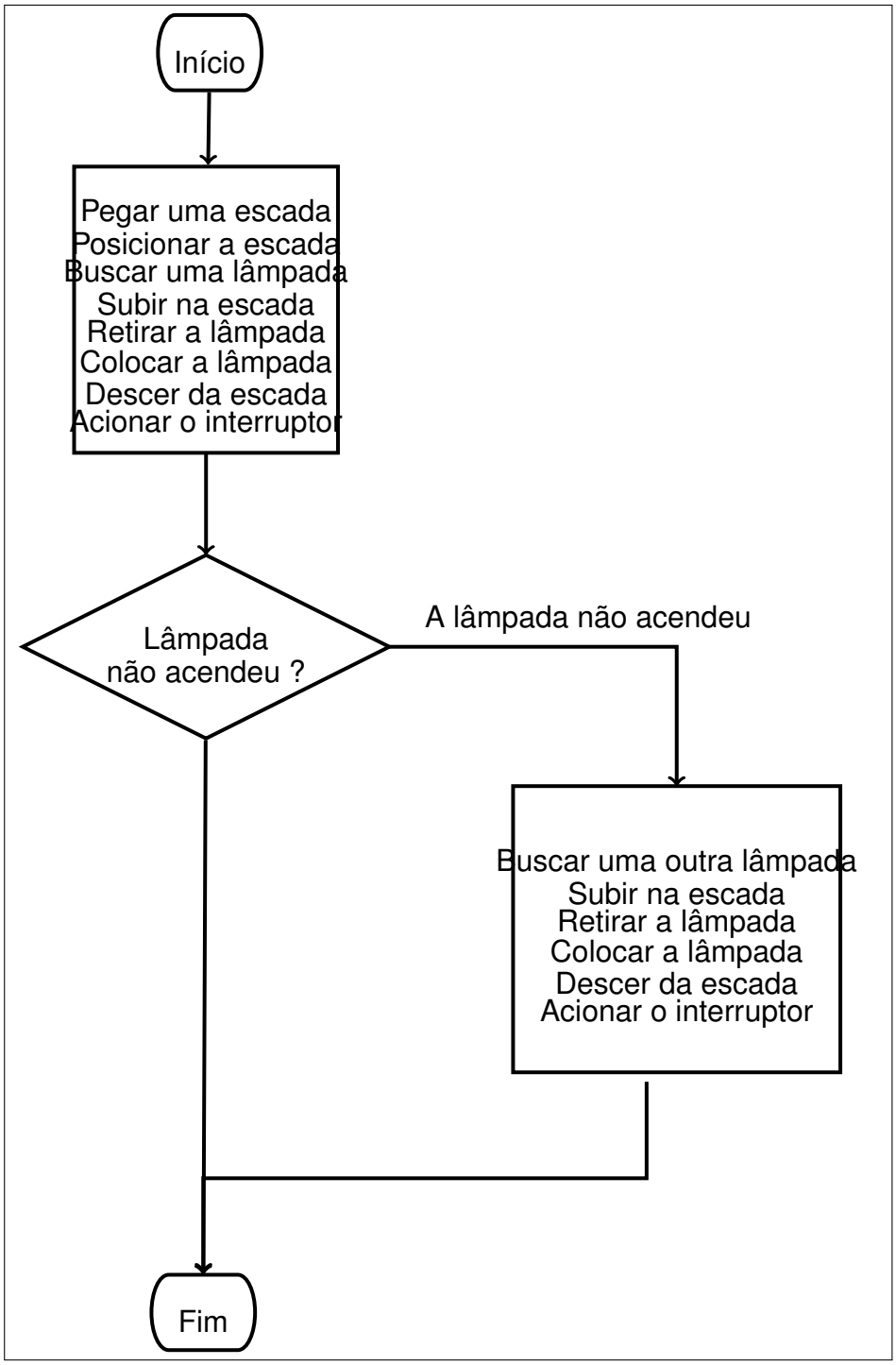


Figura 6.4: Troca de lâmpada usando a estrutura de decisão.

Dica 38

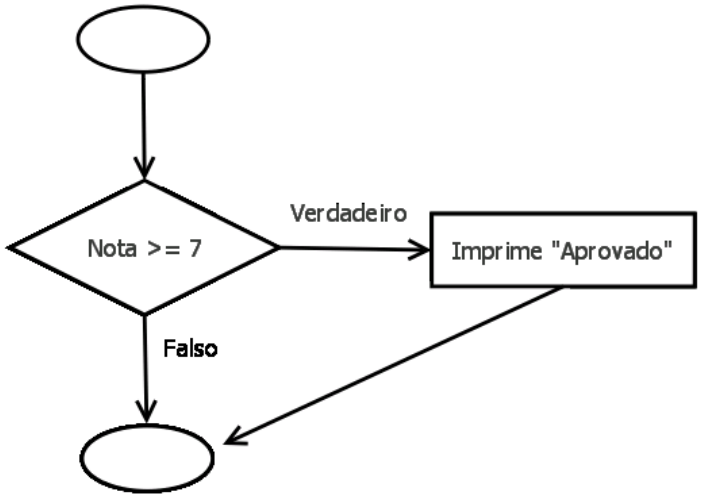
Os fluxogramas e os pseudo-códigos possuem algumas regras específicas para a sua criação, mas você não é obrigado a seguir essas regras e pode quebrá-las de vez em quando (como nós fizemos no algoritmo 6.4, agrupando várias instruções em um retângulo). É melhor quebrar essas regras de vez em quando do que não criar algoritmo algum antes de programar as suas rotinas. Não seja um “escravo” das regras, mas também não abra mão totalmente delas. Use o bom senso que vem com a experiência.

Dica 39

Quando você estiver bem familiarizado com algumas rotinas, você pode abrir mão dos algoritmos. Mas sempre que surgir uma situação nova, é melhor criar um algoritmo (principalmente um pseudo-código) antes.

A figura 6.5 nos mostra mais outro exemplo de decisão, semelhante ao anterior. Nesse algoritmo, se o aluno tirar uma nota maior ou igual a sete, a rotina exibirá a palavra “Aprovado”.

Figura 6.5: Representação de uma estrutura de seleção em um fluxograma



Note que uma estrutura de decisão deriva de uma estrutura de sequência. O caminho principal é o chamado “caminho esperado”, que é a sequência principal. O algoritmo a seguir é a representação em pseudo-código do fluxograma 6.5.

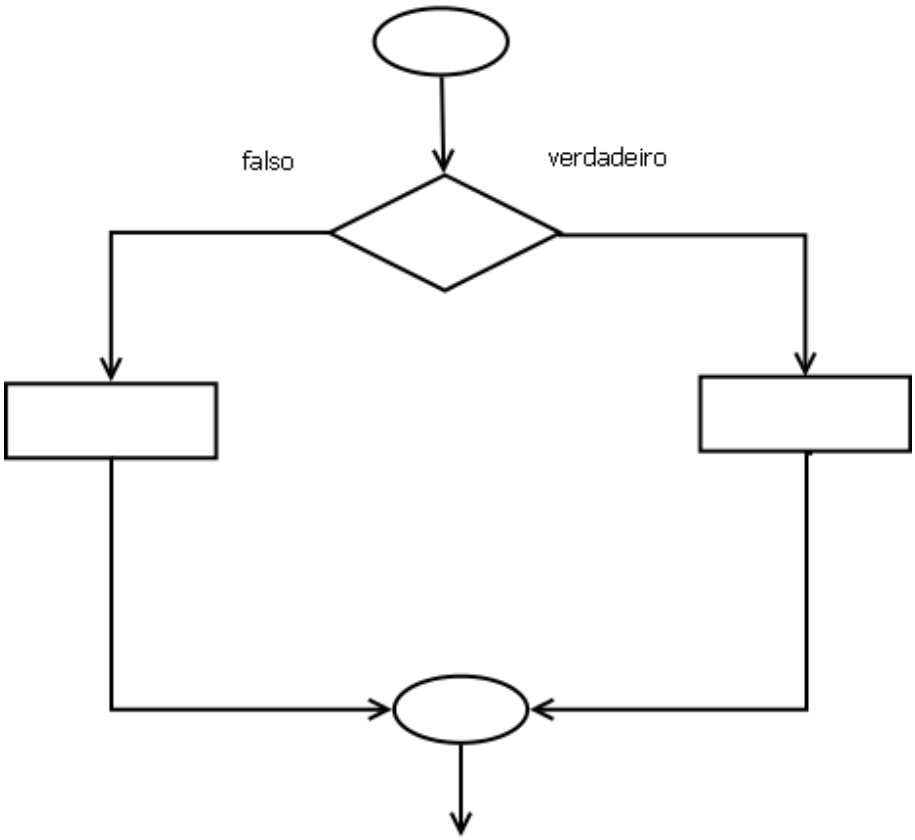
Algoritmo 11: Lê o valor da nota e escreva aprovado caso ele tenha sido aprovado.

Entrada: Nota do aluno
Saída: escreva “aprovado” caso ele tenha sido aprovado

```
1 início
2 |  leia nota
3 |  se nota >= 7 então
4 |  |  escreva "Aprovado"
5 fim
```

Existem casos, em que dois fluxos distintos de código são gerados a partir de apenas uma decisão, por exemplo: “Se o aluno tirar uma nota maior ou igual a sete, imprima aprovado, caso contrário imprima reprovado”. Veja a seguir um exemplo genérico dessa representação em um fluxograma.

Figura 6.6: Estrutura de seleção SE (Seleção dupla)



A seguir temos a representação de um algoritmo em pseudo-código que possui dois blocos que dependem de uma decisão (o aluno foi ou não aprovado ?). A aplicação desses exemplos em Harbour só será feito no próximo capítulo.

Algoritmo 12: Lê o valor da nota e escreva aprovado ou recuperação.

Entrada: Nota do aluno
Saída: escreva “aprovado” ou “recuperação”

```
1 início
2 |  leia nota
3 |  se nota >= 7 então
4 |  |  escreva "Aprovado"
5 |  senão
6 |  |  escreva "Recuperação"
7 fim
```

6.7 Estruturas de repetição

Nós acrescentamos um elemento até então desconhecido nas nossas rotinas : a seleção, que permite a tomada de decisão. Mas como nós deixamos claro, a rotina anterior não consegue prever um caso excepcional (mas possível) da segunda lâmpada não funcionar. A rotina a seguir resolve esse problema, mas note que o problema não é resolvido de forma satisfatória.

```
ROTINA Trocar uma lâmpada

   1. Pegar uma escada.
   2. Posicionar a escada embaixo da lâmpada.
   3. Buscar uma lâmpada nova
   4. Subir na escada
   5. Retirar a lâmpada velha
   6. Colocar a lâmpada nova
   7. Descer da escada
   8. Acionar o interruptor
   9. Se a lâmpada não acender, então
       10. Buscar outra lâmpada
       11. Subir na escada
       12. Retirar a lâmpada com defeito
       13. Colocar a lâmpada nova
       14. Descer da escada
       15. Acionar o interruptor
       16. Se a lâmpada não acender, então
           17. Buscar outra lâmpada
           18. Subir na escada
           19. Retirar a lâmpada com defeito
           20. Colocar a lâmpada nova
           21. Descer da escada
           22. Acionar o interruptor
           23. Se a lâmpada não acender, então
               24. Buscar outra lâmpada
               25. Subir na escada
```

```
                26. Retirar a lâmpada com defeito
                27. Colocar a lâmpada nova
                28. Descer da escada
                29. Acionar o interruptor

                ... Segue a verificação

FINAL DA ROTINA
```

Até quando será feito o teste da lâmpada ? Não sabemos. Essa solução exige uma quantidade extra de códigos que tornaria a rotina impraticável. Quantas lâmpadas eu tenho para testar, caso todas falhem ? Também não sabemos. A rotina acima não consegue responder a essas perguntas. Ela, sem dúvida, oferece uma solução para o problema da falha na segunda lâmpada, na terceira, na quarta, etc. Mas a solução não é satisfatória porque eu passei a digitar uma quantidade muito grande de código repetido. Códigos repetidos são um grande problema na programação de computadores, muito tempo, dinheiro e esforço foram despendidos na solução desse problema.

A rotina a seguir resolve o problema, note que alguns passos sofreram alteração na ordem.

```
ROTINA Trocar uma lâmpada

   1. Acionar o interruptor
   2. Se a lâmpada não acender, então
       3. Pegar uma escada
       4. Posicionar a escada embaixo da lâmpada.
       5. Repita
          6. Buscar uma lâmpada nova
          7. Subir na escada
          8. Retirar a lâmpada velha
          9. Colocar a lâmpada nova
          10. Descer da escada
          11. Acionar o interruptor
       12. Até a lâmpada acender OU
            Não tiver lâmpada no estoque.

FINAL DA ROTINA
```

Os passos de 5 até o 12 contém um bloco de repetição. Esse bloco será repetido até a lâmpada acender ou enquanto tiver lâmpada no estoque. Por isso que os passos 6 até 11 sofrem indentação também, para indicar que estão no interior de um outro bloco.

Nós deixaremos a implementação dos pseudo-códigos e dos respectivos fluxogramas para quando nós formos estudar as declarações em Harbour que realizam essa tarefa.

6.8 Conclusão

Nós vimos três tipos de blocos: um bloco principal (sequencial), um bloco de decisão e um bloco de repetição[4]. Essas, portanto, são as três estruturas que compõem a base da programação estruturada de computadores. Essas estruturas, criadas na década de 1960, ainda hoje são essenciais para a construção de programas de computadores. Na década de 1980 surgiu um outro tipo de estrutura que deu origem a uma nova forma de programar: a programação orientada a objetos. Nesse livro nós não iremos abordar o paradigma orientado a objeto em profundidade, mas queremos apenas ressaltar de antemão que tudo o que vimos nesse livro (as três estruturas da programação estruturada) são essenciais para a programação de computadores de um modo geral, e que a programação orientada ao objeto **não** pode ser compreendida satisfatoriamente sem o paradigma[5] estruturado.

[4] No apêndice D você encontrará um resumo dessas estruturas na forma de fluxogramas.

[5] A palavra paradigma é bastante usado na computação, mas ela pertence ao campo da filosofia e da teoria do conhecimento. Paradigmas são normas que determinam como nós devemos pensar e agir. Você está aprendendo a pensar estruturadamente através dessas três estruturas abordadas (sequência, decisão e repetição).

7 Estruturas de decisão e Operadores de comparação

Programação é compreensão.

Kristen Nygaard

Objetivos do capítulo

- Entender o que é uma estrutura de decisão.
- Saber diferenciar uma estrutura de decisão de uma estrutura sequencial.
- Saber usar os operadores de comparação.
- Criar estruturas de decisão usando o IF.
- O CASE e o SWITCH.

7.1 Estruturas de decisão

Mizrahi define assim as estruturas de decisão :

> permitem determinar qual a ação a ser tomada com base no resultado de uma expressão condicional. Isso significa que podemos selecionar entre ações alternativas dependendo de critérios desenvolvidos no decorrer da execução do programa [Mizrahi 2004, p. 70]. (as estruturas de decisão)

Vamos dividir a explicação da autora citada em pequenos blocos para que nosso entendimento possa ficar mais claro. As estruturas de decisão :

1. “permitem determinar qual a ação a ser tomada” : em resumo, permitem realizar “escolhas”.
2. “com base no resultado de uma expressão condicional”: ainda não vimos o que é uma expressão condicional, mas o resultado dessa tal expressão é um tipo de dado lógico: verdadeiro ou falso (estudamos um pouco sobre eles no capítulo sobre tipos de dados).

O primeiro item provavelmente deve ter ficado claro para você. A “ação a ser tomada” é determinada pelas estruturas de decisão vistas no capítulo anterior. O segundo item talvez tenha lhe causado um certo desconforto. Afinal de contas, o que é uma “expressão condicional” ? O conceito é bem simples: lembra do capítulo anterior, quando nós desenvolvemos um algoritmo para determinar se o aluno passou ou não ? Lá nesse algoritmo tinha a seguinte condição : “Se a nota for maior ou igual a sete então o aluno passou”. Bem, isso é uma expressão condicional. São expressões que podem ser lidas em forma de pergunta, e a resposta a essa pergunta é sempre sim ou não (verdadeiro ou falso). Vamos refazer a expressão mentalmente em forma de pergunta : “A nota é maior ou igual a sete ?”. Se a resposta for “sim” então executa uma ação, se for “não”, executa outra ação.

Mas, ainda fica a pergunta : “como esses valores lógicos (verdadeiro ou falso) são gerados ?”. A resposta é : através dos operadores relacionais que nós estudaremos agora. Esse conceito novo será estudado nesse capítulo. Mas vamos dar um exemplo bem básico que lhe ajudará a entender o que é esse tal operador. Tomemos a expressão condicional : “Se a nota for maior ou igual a sete”. Pois bem, esse “maior ou igual” é um operador relacional. Ele estabelece “relações” do tipo igual, maior ou igual, diferente, menor ou igual, etc. Esse exemplo está ilustrado na figura 7.1, onde temos a relação entre a expressão condicional, o tipo de dado e o operador relacional.

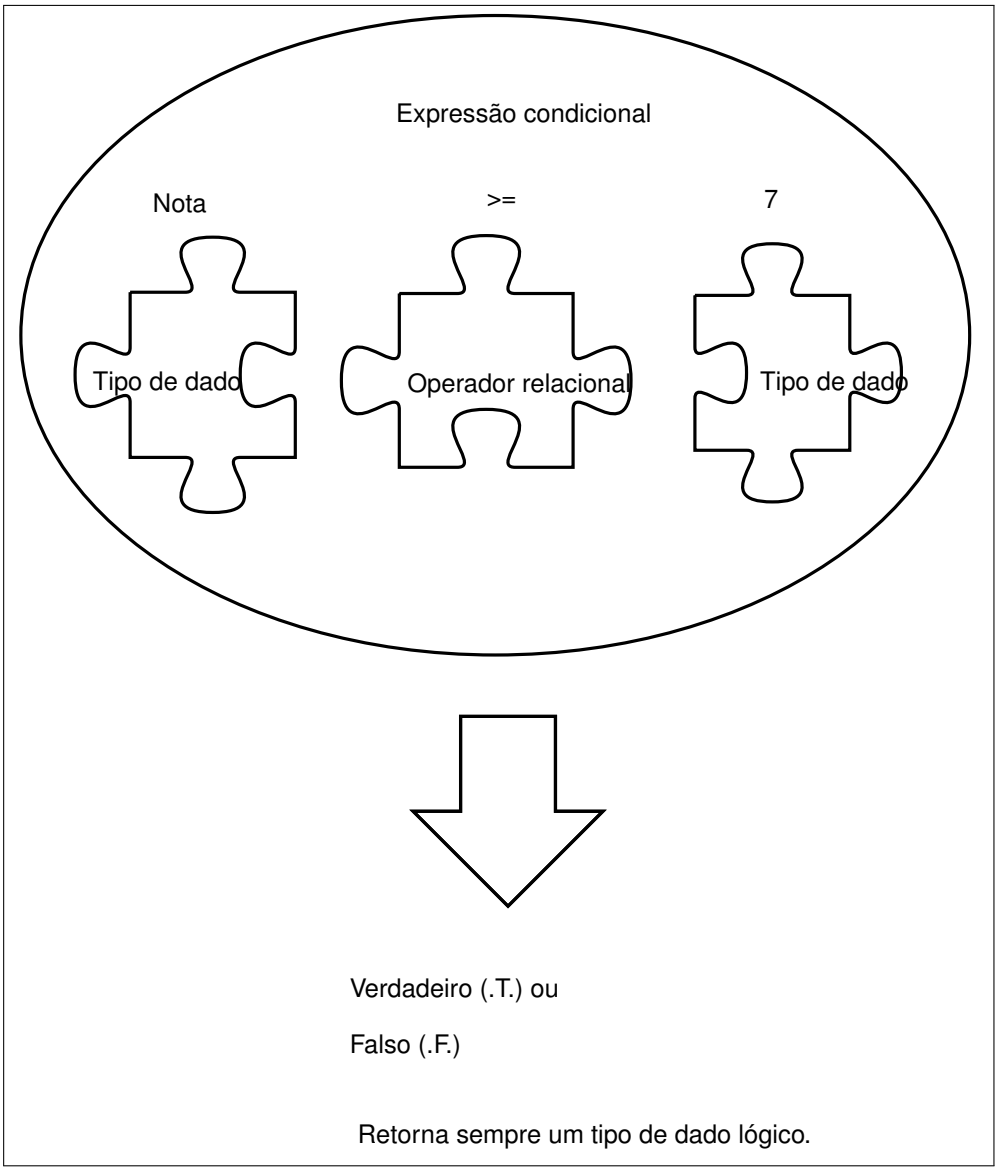


Figura 7.1: Tipos de dados, operadores de comparação e expressão condicional

O tipo de dado lógico, os operadores relacionais e as estruturas de seleção são três componentes estreitamente relacionados. Se você souber apenas um ou dois desses componentes o assunto ficará sem sentido e você com a sensação de que não entendeu completamente uma estrutura de decisão. Já sabemos o que é um tipo de dado lógico (vimos isso no capítulo sobre tipos de dados), agora vamos partir para os operadores relacionais e em seguida para os comandos de decisão.

7.2 Operadores relacionais

Os operadores relacionais são binários que geram resultados lógicos (verdadeiro ou falso) a partir do relacionamento (comparação) de duas expressões [Vidal 1991, p. 93].

A expressão da direita é comparada com a da esquerda, resultando um valor verdadeiro (.t.) ou falso (.f.). Você só pode comparar valores que sejam do mesmo tipo de dado. Uma relação do tipo 2 > “Ana” irá resultar em um erro pois não faz sentido

comparar tipos diferentes. Os operadores relacionais do Harbour estão listados na tabela 7.1.

Tabela 7.1: Operadores relacionais

Operação	Operador
Menor que	<
Maior que	>
Igual a	=
Duplo igual a	==
Diferente de	<>
Diferente de	#
Diferente de	!=
Menor ou igual a	<=
Maior ou igual a	>=
Comparação de strings (Está contido)	$

Agora vamos ver como funcionam os operadores relacionais através de exemplos. Na realidade todos eles apresentam um comportamento semelhante : produzem um valor lógico como resultado. Vamos ilustrar isso com o operador “maior que” (>) na listagem 7.1.

Listagem 7.1: Exemplo com um operador relacional maior que

```
/*
Operador relacional de >
*/
PROCEDURE Main
LOCAL nNum1, nNum2

   INPUT "Informe o número 1 : " TO nNum1
   INPUT "Informe o número 2 : " TO nNum2
   ?
   ? "A variável nNum1 é maior do que nNum2 ? " , nNum1 > nNum2
   ? "A variável nNum2 é maior do que nNum1 ? " , nNum2 > nNum1

RETURN
```

Vamos supor que o usuário digite 2 e 3 :

.:Resultado:.

```
Informe o número 1 : 2
Informe o número 2 : 3

A variável nNum1 é maior do que nNum2 ?  .F.
A variável nNum2 é maior do que nNum1 ?  .T.
```

Vamos analisar o código que foi executado. As linhas 7 e 8 já são nossas velhas conhecidas. A linha 9 apenas dá um espaço para melhorar a visualização. A novidade está nas linhas 10 e 11 : foi realizado, na linha 10 uma pergunta, apenas para ilustrar, e a comparação foi feita, retornando um valor falso (.F.). Você poderia, sempre em forma

de pergunta, ler assim a relação da listagem : “A variável nNum1 é maior do que a variável nNum2 ?”. No exemplo foi digitado 2 para nNum1 e 3 para nNum2, resultando em um valor lógico falso (.F.). Na linha 11 a pergunta seria : “A variável nNum2 é maior do que a variável nNum1 ?”. Nesse caso a resposta seria uma valor lógico verdadeiro (.T.).

Note que ainda não usamos variáveis lógicas, apenas valores lógicos. O exemplo a seguir (listagem 7.2) ilustra como podemos armazenar o resultado de uma comparação (um valor lógico) em uma variável.

Listagem 7.2: Exemplo de um operador relacional “maior que” e variável lógica

```
/*
Operador relacional de > e variável lógica
*/
PROCEDURE Main
LOCAL nNum1, nNum2 // Valores
LOCAL lResultado  // Resultado lógico

   INPUT "Informe o número 1 : " TO nNum1
   INPUT "Informe o número 2 : " TO nNum2

   lResultado := ( nNum1 > nNum2 )

   ? "A variável nNum1 é maior do que nNum2 ? " , lResultado

RETURN
```

.:Resultado:.

```
Informe o número 1 : 3
Informe o número 2 : 2
A variável nNum1 é maior do que nNum2 ?  .T.
```

Se você entendeu as duas listagens (7.1 e 7.2) então você entenderá o funcionamento dos demais operadores relacionais. Se você ainda não entendeu, não se preocupe, pois na seção seguinte apresentaremos o comando de decisão *IF* seguido de exemplos com os outros operadores que ainda faltam. Ao final do capítulo, você terá praticado com todos os operadores.

7.3 Estruturas de decisão

De acordo com Swan,

> utilizam os operadores relacionais de igualdade e lógicos que comparam operandos de diversas maneiras. Todos esses operadores tem o mesmo objetivo - produzir um valor representando verdadeiro ou falso que possa ser usado para direcionar o *fluxo* de um programa [Swan 1994, p. 93]. (as estruturas de decisão)

O Harbour dispõe das seguintes estruturas de decisão (ou de controle de fluxo) :

1. IF ... ENDIF

2. DO CASE ... ENDCASE
3. SWITCH ... END

7.4 IF

A primeira estrutura de decisão que nós iremos estudar, chama-se *IF*[1]. Essa estrutura equivale a estrutura SE que nós vimos no capítulo anterior e é responsável pela criação de um bloco de código que será executado somente se a comparação realizada pelos operadores for verdadeira. Ele pode ser ilustrado na listagem 7.3 logo a seguir :

Listagem 7.3: Estrutura de decisão IF

```
/*
Exemplo com o comando IF
*/
PROCEDURE Main
LOCAL nNum1, nNum2 // Valores

   INPUT "Informe o número 1 : " TO nNum1
   INPUT "Informe o número 2 : " TO nNum2

   IF ( nNum1 > nNum2 )
      ? "A variável nNum1 é maior do que nNum2."
   ENDIF

   ? "Final da rotina"

RETURN
```

Até agora, as instruções que nós estudamos ocupam somente uma linha, mas uma estrutura de decisão é diferente de tudo o que já vimos até agora, porque ela ocupa no mínimo três linhas e pode conter, dentro dela outras instruções. No nosso exemplo ela ocupou três linhas, iniciando na linha 10 e encerrando na linha 12. A linha 11 só é executada se o resultado da comparação (nNum1 > nNum2) retornar um valor lógico verdadeiro.

Vamos exibir dois casos diferentes do programa da listagem 7.3. No primeiro deles, o usuário digitou 10 e 20 para nNum1 e nNum2, respectivamente. O IF recebeu o resultado da comparação e, como não retornou verdadeiro, não executou o interior do seu bloco.

O usuário digitou 10 e 20.

.:Resultado:.

```
Informe o número 1 : 10
Informe o número 2 : 20
Final da rotina
```

Agora, nesse caso, o interior do bloco IF foi executado, porque a comparação retornou verdadeira.

[1]O significado de IF é “Se”.

O usuário digitou 20 e 10.

.:Resultado:.

```
Informe o número 1 : 20
Informe o número 2 : 10
A variável nNum1 é maior do que nNum2.
Final da rotina
```

Um ponto importante a destacar é o recuo (indentação) que é feito no interior do bloco de código do IF (linha 11). Esses recuos não são obrigatórios, mas você deve se habituar a fazê-los para que o seu código fique claro.

Dica 40

Da mesma forma que você deve indentar o interior do bloco principal, você deve aplicar recuos nos blocos internos de todas as suas estruturas de controle (o comando IF é uma estrutura de controle de decisão). Nos exemplos a seguir você verá que as estruturas de controle podem ficar uma dentro da outra[a], habitue-se a aplicar indentações nelas também, quantos níveis forem necessários.

[a]Quando uma estrutura fica uma dentro da outra, o termo correto é estruturas “aninhadas”

O IF possui ainda uma segunda parte que nós ainda não vimos. Ela cria um segundo bloco que será executado se a comparação retornar falsa. Veja um exemplo na listagem 7.4 logo a seguir.

Listagem 7.4: Comando de decisão IF ... ELSE

```
/*
Exemplo com o comando IF
*/
PROCEDURE Main
LOCAL nNum1, nNum2 // Valores

   INPUT "Informe o número 1 : " TO nNum1
   INPUT "Informe o número 2 : " TO nNum2

   IF ( nNum1 > nNum2 )
      ? "A variável nNum1 é maior do que nNum2."
   ELSE
      ? "A variável nNum1 é menor ou igual a nNum2."
   ENDIF

   ? "Final da rotina"

RETURN
```

O usuário digitou 2 e 3.

.:Resultado:.

```
Informe o número 1 : 2
Informe o número 2 : 3
A variável nNum1 é menor ou igual a nNum2.
```

```
Final da rotina
```

Esse segundo bloco inicia-se com a cláusula[2] ELSE (que traduzido significa “senão”) e vai até o final do comando.

Existem ainda casos onde uma mesma declaração IF pode levar a vários blocos. Acompanhe o desenvolvimento do pequeno problema a seguir (listagem 7.5). Observe que nós introduziremos um outro operador chamado de >= (“maior ou igual”), mas o seu princípio de funcionamento é semelhante ao do operador > (“maior”), que já foi visto.

Listagem 7.5: Estrutura de seleção IF

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota

   // Recebe o valor da nota
   nNota := 0
   INPUT "Informe a nota do aluno : " TO nNota

   // Decide se foi aprovado ou não de acordo com a média
   IF ( nNota >= 7 )
      ? "Aprovado"
   ENDIF


RETURN
```

Essa listagem está representada no apêndice A. Caso queira vê-la, vá para o apêndice A e busque a representação da listagem 7.5.

.:Resultado:.

```
Informe a nota do aluno : 8
Aprovado
```

O código listado em 7.5 está correto, mas se o aluno for reprovado ele não exibe mensagem alguma. Essa situação é resolvida usando um ELSE que nós já vimos. Veja como fica na listagem 7.6.

Listagem 7.6: Estrutura de seleção IF ... ELSE ... ENDIF

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota

   // Recebe o valor da nota
   nNota := 0
```

[2] O termo cláusula pode ser entendido aqui como uma parte da estrutura principal, e que não tem existência independente. Nesse caso o ELSE não pode ser usado sem um IF que a anteceda.

```
   INPUT "Informe a nota do aluno : " TO nNota

   // Decide se foi aprovado ou não de acordo com a média
   IF ( nNota >= 7 )
      ? "Aprovado"
   ELSE
      ? "Reprovado"
   ENDIF

RETURN
```

.:Resultado:.

```
Informe a nota do aluno : 4
Reprovado
```

Atenção, imagine agora a seguinte mudança : o diretor da escola lhe pede para incluir uma mudança no código: se o aluno tirar menos de 7 e mais de 3 ele vai para recuperação e não é reprovado imediatamente. Esse problema pode ser resolvido de duas formas diferentes : inserindo uma nova estrutura IF dentro do ELSE da primeira estrutura ou usando uma cláusula ELSEIF. As duas respostas resolvem o problema.

A listagem 7.7 mostra a primeira forma, que é inserindo uma estrutura dentro da outra (observe a indentação).

Listagem 7.7: Estrutura de seleção IF ... ELSE ... ENDIF com uma estrutura dentro de outra.

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota

   // Recebe o valor da nota
   nNota := 0
   INPUT "Informe a nota do aluno : " TO nNota

   // Decide se foi aprovado ou não de acordo com a média
   IF ( nNota >= 7 )
      ? "Aprovado"
   ELSE
      IF ( nNota >= 5 )
         ? "Recuperação"
      ELSE
         ? "Reprovado"
      ENDIF
   ENDIF


RETURN
```

A listagem 7.8 aborda a segunda forma, que é com a inclusão de um ELSEIF (que faz parte do comando IF principal e não deve sofrer indentação).

Listagem 7.8: Estrutura de seleção IF ...ELSEIF ... ELSE ... ENDIF

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota

   // Recebe o valor da nota
   nNota := 0
   INPUT "Informe a nota do aluno : " TO nNota

   // Decide se foi aprovado ou não de acordo com a média
   IF ( nNota >= 7 )
      ? "Aprovado"
   ELSEIF ( nNota >= 5 )
      ? "Recuperação"
   ELSE
      ? "Reprovado"
   ENDIF


RETURN
```

As duas listagens (7.7 e 7.8) produzem o mesmo resultado abaixo.

.:Resultado:.

```
Informe a nota do aluno : 5
Recuperação
```

Dica 41

O código da listagem 7.8 é mais claro do que o código da listagem 7.7. Quando você se deparar com uma situação que envolve múltiplos controles de fluxo e apenas uma variável prefira a estrutura IF ... ELSEIF ... ELSE ... ENDIF. Múltiplos níveis de indentação tornam o código mais difícil de ler.

Para finalizar veja no quadro a seguir a sintaxe completa do IF.

Descrição sintática 6

1. Nome : IF
2. Classificação : declaração.
3. Descrição : Executa um dentre vários blocos de instruções.
4. Sintaxe

```
IF <lcondição1>
<instruções>...
[ELSEIF <lcondição2>]
<instruções>...
[ELSE]
<instruções>...
END[IF]
```

Fonte : [Nantucket 1990, p. 2-20]

Iremos, nas próximas subseções, ver como os tipos de dados abordados implementam os seus respectivos operadores relacionais. Usaremos as estruturas de seleção (IF) para que os exemplos fiquem mais interessantes.

7.5 O tipo numérico e os seus operadores relacionais

7.5.1 O operador de igualdade aplicado aos tipos numéricos.

O Harbour possui dois operadores de igualdade : o “=” e o “==”. O exemplo da listagem demonstra uma operação de igualdade usando o operador “==”.

Listagem 7.9: Igualdade

```
/*
Operador ==
*/
PROCEDURE Main
LOCAL nTabuada1, nTabuada2 , nResp

   INPUT "Informe um número : " TO nTabuada1
   INPUT "Informe um outro número : " TO nTabuada2
   ?
   ? "Quanto é " , nTabuada1 , " vezes ", nTabuada2, " ? "
   ?
   INPUT "Resposta : " TO nResp
   IF (( nTabuada1 * nTabuada2 ) == nResp )
      ? "Certa a resposta."
   ENDIF

   ? "Final da rotina"

```

```
RETURN                                                                    19
```

.:Resultado:.

```
Informe um número : 10
Informe um outro número : 2

Quanto é          10  vezes           2  ?

Resposta : 20
Certa a resposta.
Final da rotina
```

Se você usasse o operador "=" no lugar do operador "==" o resultado seria exatamente o mesmo. No entanto, nós preferimos usar nos nossos exemplos o operador "==" pois ele serve somente para comparações, enquanto que o operador "=" pode servir para realizar atribuições e comparações também.

Prática número 20

Altere o programa da listagem 7.9

Dica 42

O operador "=" é um operador "sobrecarregado". Isso significa que ele está sendo usado para duas coisas diferentes. Ele tanto serve para comparar quanto serve para atribuir. Por isso nós aconselhamos que você evite o seu uso.

```
Para comparar use o operador == (Duplo igual)
Para atribuir use o operador := (Atribuição)
```

Outra razão, mais importante ainda, será vista mais adiante, nesse capítulo, quando estudarmos detalhadamente a comparação entre strings.

7.5.2 Os operadores "maior que" e "menor que" (e seus derivados) aplicados ao tipo de dado numérico.

O funcionamento dos operadores "maior que" e "menor que" (e seus derivados) é intuitivo e não difere do que aprendemos nas aulas de matemática. Veremos a seguir alguns exemplos. A listagem 7.10 ilustra o uso do operador menor ou igual em um programa que calcula se o valor informado é isento ou não de imposto de renda.

Listagem 7.10: Operador menor ou igual

```
/*                                                                        1
Exemplos usando o operador "menor que" e "menor ou igual a"              2
Tabela de classificação de imposto de renda.                              3
Fonte :                                                                   4
http://www.calcule.net/calculos.trabalhistas/tabela.imposto.de.renda.php  5
                                                                          6
Entrada : Base de cálculo                                                 7
Saída : Informa se o valor é isento de imposto ou não.                    8
```

```
*/
#define VALOR_MAXIMO_ISENTO 1903.98
PROCEDURE Main
LOCAL nSalario

   INPUT "Informe o valor do salário : " TO nSalario

   IF nSalario <= VALOR_MAXIMO_ISENTO
      ? "Isento."
   ELSE
      ? "Você tem imposto a pagar."
   ENDIF

RETURN
```

Nessa execução o usuário digitou um valor de 4700.

.:Resultado:.

```
Informe o valor do salário : 4700
Você tem imposto a pagar.
```

Nessa execução foi digitado um valor de 1500.

.:Resultado:.

```
Informe o valor do salário : 1500
Isento.
```

Note que o uso do operador “menor que” é muito semelhante. Vamos alterar o programa da listagem 7.10 para que ele passe a usar esse operador. Veja o resultado na listagem 7.11.

Listagem 7.11: Operador menor que

```
/*
Exemplos usando o operador "menor que" e "menor ou igual a"
Tabela de classificação de imposto de renda.
Fonte :
http://www.calcule.net/calculos.trabalhistas/tabela.imposto.de.renda.php

Entrada : Base de cálculo
Saída : Informa se o valor é isento de imposto ou não.
*/
#define VALOR_MAXIMO_ISENTO 1903.98
PROCEDURE Main
LOCAL nSalario

   INPUT "Informe o valor do salário : " TO nSalario

   IF nSalario < (VALOR_MAXIMO_ISENTO + 1)
            // Acima eu aumentei uma unidade
            // para usar o operador <
```

```
      ? "Isento."
   ELSE
      ? "Você tem imposto a pagar."
   ENDIF

RETURN
```

O resultado final é o mesmo.

Já vimos exemplos com os operadores “maior que” e “maior ou igual a”. Veja a seguir (listagem 7.12) mais um exemplo usando o operador “maior que”.

Listagem 7.12: Operador maior que

```
/*
Exemplos usando o operador "maior que" e "maior ou igual a"
Taxação extra sobre conta de luz.

Entrada : O consumo de energia
Saída : Informa se o valor é taxado ou não.
*/
#define VALOR_MAXIMO_ISENTO 2400
PROCEDURE Main
LOCAL nConsumo

   INPUT "Informe o valor do consumo de energia : " TO nConsumo

   IF nConsumo > VALOR_MAXIMO_ISENTO
      ? "Vai pagar taxa adicional."
   ELSE
      ? "Isento de taxas adicionais."
   ENDIF

RETURN
```

Nessa simulação o usuário digitou 1200:

.:Resultado:.

```
Informe o valor do consumo de energia : 1200
Isento de taxas adicionais.
```

Prática número 21

Leia as listagens 7.10 e 7.11 e veja o que foi feito para que o operador passasse de “menor ou igual a” para “menor que”. Use a mesma lógica para alterar o programa da listagem 7.12 para que ele passe a usar o operador “maior ou igual a”.

7.5.3 Os operadores de diferença aplicados ao tipo de dado numérico.

Apenas um sinal é necessário para expressar a diferença entre expressões, mas o Harbour possui três operadores que significam a mesma coisa. Isso se deve a herança

que ele recebeu do antigo dBase (década de 1980) e do Clipper (década de 1990). A seguir temos uma lista desses operadores (todos eles significam a mesma coisa) :

- “<>” : O formato mais antigo, e ainda o mais usado. O seu formato é interessante pois é como se ele fosse a combinação de um operador “menor que” e um operador “maior que”. Fica fácil de aprender, pois quando um valor é “menor que” ou “maior que” outro valor, então ele é diferente.
- “#” : Esse formato surgiu com a linguagem Clipper. Não é muito usado atualmente, mas ele também é fácil de assimilar pois se parece muito com o operador de diferença que nós já conhecemos das aulas de matemática.
- “!=” : Esse formato foi inspirado na linguagem C, C++, PHP e derivados. É encontrado em códigos, mas não tanto quanto o formato “<>”.

O primeiro formato, que é o mais antigo, e ainda se encontra presente na maioria das listagens atuais. Veremos a seguir (listagem 7.13) um exemplo com o formato <> : .

Listagem 7.13: Operador de diferença

```
/*
Operador de diferença

Entrada : A resposta a pergunta
Saída : Informa se o valor é correto ou não.
*/
#define RESPOSTA 10
PROCEDURE Main
LOCAL nResposta

   INPUT "Informe quanto é 5 + 5 : " TO nResposta

   IF nResposta <> RESPOSTA
      ? "Resposta está errada."
   ELSE
      ? "Certa a resposta."
   ENDIF

RETURN
```

.:Resultado:.

```
Informe quanto é 5 + 5 : 10
Certa a resposta.
```

Prática número 22

Altere o programa da listagem 7.13 para que ele passe a utilizar os outros dois operadores. Crie mais dois programas para isso, um para cada operador. Faça ligeira modificações nos dois programas que você for criar.

7.6 O tipo data e os seus operadores relacionais

O tipo data possui os mesmos operadores que o tipo numérico. Eles funcionam de forma semelhante também. Vamos dar três exemplos que abrangem os casos já vistos.

7.6.1 O operador de igualdade e o tipo de dado data

O exemplo a seguir (listagem 7.14) serve para ilustrar o uso do operador de igualdade aplicado a um tipo de dado data. Note que ele usa a função DATE() para automatizar a entrada da data de hoje. A função serve para lembrar o dia limite para o envio de um relatório importante.

Listagem 7.14: Operador de igualdade com datas

```
/*
Operador de igualdade aplicado a um tipo de dado data

O programa emite um aviso se for o último dia para o envio de dados
Entrada : A data de hoje gerado por DATE()
Saída : Se hoje for o dia, então emite um aviso.
*/
#define DATA_LIMITE 0d20160901
PROCEDURE Main

   SET DATE BRITISH
   ? "Hoje é " , DATE()
   ? "A data limite é ", DATA_LIMITE
   // Note que essa rotina não tem dados de entrada
   // gerados pelo usuário
   // pois o dia de hoje é informado por DATE()
   IF DATE() == DATA_LIMITE
      ? "Lembrete especial: hoje é o último dia para enviar"
      ?? " os dados do balanço patrimonial."
   ENDIF

RETURN
```

Quando essa rotina é executada ela só irá executar o aviso se a data for realmente igual a data limite especificada.

.:Resultado:.

```
Hoje é  11/09/16
A data limite é  01/09/16
```

Veja que o prazo venceu e o aviso não foi gerado. Isso porque no dia em que o programa foi executado (11/09/2016) a data correspondente era superior a data limite, mas não igual. Resolveremos esse problema na próxima seção.

7.6.2 Os operadores “maior que” e “menor que” (e seus derivados) aplicados ao tipo de dado data.

Vamos supor que o seu cliente solicitou uma pequena modificação no programa da listagem anterior (listagem 7.14). Ele quer ser avisado se passou do prazo de envio dos dados. Esse problema é resolvido com o operador “maior” e uma simples comparação entre datas, conforme a listagem 7.15.

Listagem 7.15: Os operadores “maior que” e “menor que” (e seus derivados) com datas

```
/*
Operador de igualdade aplicado a um tipo de dado data

O programa emite um aviso se for o último dia para o envio de dados
Entrada : A data de hoje gerado por DATE()
Saída : Se hoje for o dia, então emite um aviso.
*/
#define DATA_LIMITE 0d20160901
PROCEDURE Main

   SET DATE BRITISH
   ? "Hoje é " , DATE()
   ? "A data limite é ", DATA_LIMITE
   // Note que essa rotina não tem dados de entrada
   // gerados pelo usuário
   // pois o dia de hoje é informado por DATE()
   IF DATE() > DATA_LIMITE
      ? "Urgente!!! Passou do prazo de envio dos dados"
   ELSEIF DATE() == DATA_LIMITE
      ? "Lembrete especial: hoje é o último dia para enviar"
      ?? " os dados do balanço patrimonial."
   ENDIF

RETURN
```

.:Resultado:.

```
Hoje é  11/09/16
A data limite é  01/09/16
Urgente!!! Passou do prazo de envio dos dados
```

O programa da listagem anterior funciona perfeitamente, mas ele com certeza não atende totalmente as necessidades do cliente. Isso porque o aviso é gerado apenas no dia limite para o envio dos dados e também quando o prazo já venceu. Nesse caso, o ideal é criar um outro tipo de aviso, que nos avise quando o dia limite estiver chegando perto. O programa da listagem 7.16 resolve esse problema, note que foi feito uma pequena alteração no programa anterior. Vamos supor que a data de hoje é 11/09/2016 e a data limite é 15/09/2016.

Listagem 7.16: Os operadores “maior que” e “menor que” (e seus derivados) com datas

```
/*
Operador de igualdade aplicado a um tipo de dado data
```

```
                                                                                3
                                                                                4
Entrada : A data de hoje gerado por DATE()                                      5
Saída : Se hoje for o dia ou for menor, então emite um aviso.                   6
*/                                                                              7
#define DATA_LIMITE 0d20160915                                                  8
PROCEDURE Main                                                                  9
                                                                                10
   SET DATE BRITISH                                                             11
   ? "Hoje é " , DATE()                                                         12
   ? "A data limite é ", DATA_LIMITE                                            13
   // Note que essa rotina não tem dados de entrada                             14
   // gerados pelo usuário                                                      15
   // pois o dia de hoje é informado por DATE()                                 16
   IF DATE() > DATA_LIMITE                                                      17
      ? "Urgente!!! Passou do prazo de envio dos dados"                         18
   ELSEIF DATE() < DATA_LIMITE                                                  19
      ? "Está chegando perto do dia limite"                                     20
   ELSEIF DATE() == DATA_LIMITE                                                 21
      ? "Lembrete especial: hoje é o último dia para enviar"                    22
      ?? " os dados do balanço patrimonial."                                    23
   ENDIF                                                                        24
                                                                                25
RETURN                                                                          26
```

.:Resultado:.

```
Hoje é  11/09/16
A data limite é  15/09/16
Está chegando perto do dia limite
```

Mas a novela das datas ainda não acabou. Existe algo de errado no programa acima. Na verdade existe um erro de lógica nele. Vamos supor que hoje é 11/09/2016 e a data limite é 15/10/2017 (mais de um ano depois). Sempre que essa rotina for executada o aviso será gerado. Isso fará com que o aviso perca a eficácia, pois nós não queremos ser avisados com muitos dias de antecedência. A listagem 7.17 a seguir, resolve o problema gerando o aviso quando faltarem menos de dez dias para o prazo se vencer.

Listagem 7.17: Os operadores “maior que” e “menor que” (e seus derivados) com datas

```
/*                                                                              1
Operador de igualdade aplicado a um tipo de dado data                           2
                                                                                3
                                                                                4
Entrada : A data de hoje gerado por DATE()                                      5
Saída : Se hoje for o dia ou for menor, então emite um aviso.                   6
*/                                                                              7
#define DATA_LIMITE 0d20160915                                                  8
#define DIAS_AVISO 10 // Emitir o aviso com 10 dias ou menos                    9
PROCEDURE Main                                                                  10
                                                                                11
   SET DATE BRITISH                                                             12
```

```
   ? "Hoje é " , DATE()
   ? "A data limite é ", DATA_LIMITE
   // Note que essa rotina não tem dados de entrada
   // gerados pelo usuário
   // pois o dia de hoje é informado por DATE()
   IF DATE() > DATA_LIMITE
      ? "Urgente!!! Passou do prazo de envio dos dados"
   ELSEIF DATE() < DATA_LIMITE
      IF ( DATA_LIMITE - DATE() ) < 10
         ? "Está chegando perto do dia limite"
      ENDIF
   ELSEIF DATE() == DATA_LIMITE
      ? "Lembrete especial: hoje é o último dia para enviar"
      ?? " os dados do balanço patrimonial."
   ENDIF

RETURN
```

Teste com a data de hoje igual a 11/09/2016 e o limite igual a 15/09/2016

.:Resultado:.

```
Hoje é  11/09/16
A data limite é  15/09/16
Está chegando perto do dia limite
```

Prática número 23

Faça uma alteração no programa da listagem 7.17 para que ele informe quantos dias faltam para o prazo se vencer. Siga o modelo abaixo :

```
Hoje é  11/09/16
A data limite é  15/09/16
Está chegando perto do dia limite
Faltam           4  dias
```

7.6.3 Os operadores de diferença aplicados ao tipo de dado data.

Os operadores de diferença que nós vimos com os números também se aplicam ao tipo de dado data. Vamos supor que uma operação muito importante só possa ser realizada no dia 15/07/2017, nem antes nem depois. Vamos supor também que o seu cliente deseja que o sistema emita um aviso caso alguém tente executar essa operação em um outro dia. A listagem 7.18 nos dá uma solução para esse problema.

Listagem 7.18: Os operadores de diferença aplicados ao tipo data

```
/*
Operador de diferença aplicado a um tipo de dado data
*/
#define DATA_DA_OPERACAO 0d20170715
PROCEDURE Main

   SET DATE BRITISH
```

```
   ? "Hoje é " , DATE()
   ? "A data da operaçäo é ", DATA_DA_OPERACAO
   // Note que essa rotina não tem dados de entrada
   // gerados pelo usuário
   // pois o dia de hoje é informado por DATE()
   IF DATE() <> DATA_DA_OPERACAO

   ? "Aviso importante!! Essa operação não pode ser realizada hoje."
   ELSE
      ? "Rotina especial sendo executada agora"
      // Segue o hipotético código a ser executado
      ? "Operação finalizada"
   ENDIF

RETURN
```

.:Resultado:.

```
Hoje é  11/09/16
A data da operação é  15/07/17
Aviso importante!! Essa operação não pode ser realizada hoje.
```

7.7 O tipo caractere e os seus operadores relacionais

7.7.1 Os operadores “maior que” e “menor que” (e seus derivados) com caracteres.

Comparar caracteres requer alguns cuidados adicionais, vamos abordar a seguir esses cuidados.

Os operadores estudados até agora foram aplicados a data e a números, mas faz sentido dizer que “Ana” <= “Antônio” ? Sim, faz sentido. Esse sentido talvez não esteja muito claro, por isso iremos fazer uma pequena pausa para entender como as variáveis caracteres são armazenadas na memória do computador. Não é complicado, pois as comparações entre strings se referem a ordem alfabética das letras de cada string. Uma dica é fazer de conta que você está consultando um dicionário, as palavras que vem antes, tipo : “Ana”, “Beatriz” e “Carla” são “menores” do que as palavras que vem depois, tipo : “Sara”, “Teresa” e “Zênia”. O programa da listagem 7.19 serve para ilustrar o que foi dito.

Listagem 7.19: Comparação entre strings

```
/*
Comparação entre caracteres
*/
PROCEDURE Main
LOCAL cNome1, cNome2 // Nomes

   ? "O programa seguinte coloca dois nomes em ordem alfabética"

   ACCEPT "Digite um nome : " TO cNome1
```

```
   ACCEPT "Digite outro nome : " TO cNome2

   IF ( cNome1 > cNome2 )
      ? cNome2
      ? cNome1
   ELSE
      ? cNome1
      ? cNome2
   ENDIF

RETURN
```

Vamos supor que o usuário digitou primeiro “Ana” e depois “Antonio”

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
Digite um nome : Ana
Digite outro nome : Antonio
Ana
Antonio
```

Como a linguagem Harbour faz isso ? Temos duas explicações para isso: uma curta e uma longa. Vamos ficar com a explicação mais curta, mas caso você deseje entender um pouco mais sobre codificação de caracteres (você vai precisar entender em algum momento futuro) você pode consultar os apêndices e **??**. Bem, mas vamos a explicação rápida: o Harbour consulta internamente uma tabela que ele possui[3]. Essa tabela está exibida grosseiramente a seguir :

Tabela 7.2: Alguns caracteres da tabela ASCII

Código	Caractere
65	A
66	B
67	C
68	D

Essa tabela está detalhada no apêndice **??**.

É lógico que a tabela não contém somente esses caracteres, ele possui todos os caracteres do nosso alfabeto, mais alguns não imprimíveis. Mas, vamos ao que importa: quando você pede para o programa verificar quem é “maior”, o que o Harbour faz é consultar o código numérico equivalente na sua tabela.

[3] Isso vale para as outras linguagens de programação também.

Dica 43

A tabela padrão do Harbour chama-se tabela ASCII extendida. Ela não é a única, pois depois dela surgiram outras tabelas. Mas se nenhuma for definida, a tabela que irá vigorar é a tabela ASCII extendida.
ASCII (do inglês American Standard Code for Information Interchange; "Código Padrão Americano para o Intercâmbio de Informação") é um código binário que codifica um conjunto de 128 sinais: 95 sinais gráficos (letras do alfabeto latino, sinais de pontuação e sinais matemáticos) e 33 sinais de controle, utilizando portanto apenas 7 bits para representar todos os seus símbolos[a]. A tabela ASCII extendida acrescenta mais caracteres a tabela ASCII, totalizando no final 256 caracteres.
Para nossos interesses imediatos, a letra A vale 65 e cada letra do alfabeto segue essa ordem. Assim, B = 66, C= 67, e assim por diante. Por isso você não precisa decorar os códigos, pois a sequência sempre obedece a ordem alfabética.

[a]Fonte : `https://pt.wikipedia.org/wiki/ASCII`. Acessado em 17-Set-2016.

Por exemplo, quem é maior : ANA ou ALFREDO ? O Harbour irá comparar o primeiro caractere de cada string, no caso o “A”. Como eles são iguais a 65, então ele pesquisa o subsequente. No nosso caso, a letra “N” (código 78 na tabela ASCII) de “ANA” ocupa uma posição superior a letra “L” (código 76) de “ALFREDO”. Nós iremos usar o programa gerado anteriormente na listagem 7.19 para aplicar o exemplo :

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
Digite um nome : ANA
Digite outro nome : ALFREDO
ALFREDO
ANA
```

IMPORTANTE : Você pode estar se perguntando : “Por que eu devo compreender esse conceito de tabela de caracteres ? Não bastava dizer que as comparações obedecem a ordem alfabética ?”. Bem, em termos. Dizer que a comparação obedece a ordem alfabética ajuda no aprendizado, mas ela não é totalmente certa. Vamos executar mais uma vez o programa e comparar agora “ANA” com “alfredo” (letras minúsculas).

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
Digite um nome : ANA
Digite outro nome : alfredo
ANA
alfredo
```

Note que agora houve uma inversão. “ANA” agora é “menor” do que “alfredo”. É por isso que essa explicação de que as comparações obedecem a ordem alfabética são incompletas e necessitam de um acréscimo. Para entender completamente você precisa entender que existe uma tabela (ASCII) e que nessa tabela existem codificações também para letras minúsculas. A letra “A” vale 65, mas a letra “a” vale 97. Os códigos para letras maiúsculas se iniciam em 65 (com o “A”) e terminam em 90 (com o “Z”),

mas quando as letras são minúsculas o código inicia-se em 97 (com o “a”) e vai até 122 (com o “z”). Lembrando sempre que essa codificação obedece ao padrão ASCII, que será o único que trabalharemos nesse livro, mas existem outras codificações. Mas fique certo de uma coisa : **independente da codificação, a lógica de funcionamento é a mesma para todas**.

Dica 44

Se você estiver comparando somente letras maiúsculas (ou somente letras minúculas) nas strings a serem comparadas, então vale a pena usar a lógica da ordem alfabética. Mas se existirem letras maiúsculas e minúsculas misturadas então você deve converter tudo para maiúsculas ou tudo para minúsculas e depois efetuar a comparação. Isso é feito através de uma função chamada UPPER (que converte uma string para maiúsculas) ou de uma outra função chamada LOWER (que converte uma string para minúsculas). O exemplo a seguir altera o nosso programa da listagem 7.19 para que ele passe a ignorar as letras minúsculas.

Listagem 7.20: Comparação entre strings

```
/*
Comparação entre caracteres
*/
PROCEDURE Main
LOCAL cNome1, cNome2 // Nomes

   ? "O programa seguinte coloca dois nomes em ordem alfabética"

   ACCEPT "Digite um nome : " TO cNome1
   ACCEPT "Digite outro nome : " TO cNome2

   IF ( UPPER(cNome1) > UPPER(cNome2) )
      ? cNome2
      ? cNome1
   ELSE
      ? cNome1
      ? cNome2
   ENDIF

RETURN
```

Nesse caso nós usamos a função UPPER para forçar a comparação apenas com letras maiúsculas, mesmo se o usuário digitar com letras minúsculas. Se você trocar a função UPPER pela função LOWER o resultado será o mesmo.

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
Digite um nome : ANA
Digite outro nome : alfredo
alfredo
ANA
```

Um espaço em branco no início também é um caractere

Cuidado quando for comparar strings com espaços em branco no início ou no final. O código ASCII do espaço em branco é 32, logo qualquer palavra que se inicie com um espaço em branco será “menor” do que qualquer outra que se inicie com uma letra. O programa a seguir é uma adaptação do programa anterior, ele propositadamente incluiu uma string com espaço em branco no início. Veja a listagem 7.21 e o respectivo resultado.

Listagem 7.21: Comparação entre strings

```
/*
Comparação entre caracteres
*/
PROCEDURE Main
LOCAL cNome1, cNome2 // Nomes

   ? "O programa seguinte coloca dois nomes em ordem alfabética"

   cNome1 := " PEDRO"
   cNome2 := "ANA"

   IF ( cNome1 > cNome2 )
      ? cNome2
      ? cNome1
   ELSE
      ? cNome1
      ? cNome2
   ENDIF

RETURN
```

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
 PEDRO
ANA
```

Esse problema pode ser evitado se você retirar os espaços em branco do início da string. O Harbour possui uma função chamada LTRIM que remove os espaços em branco a esquerda de uma string. O programa a seguir (listagem 7.22) é uma adaptação do programa anterior. Note que ele removeu os espaços em branco na hora da comparação.

Listagem 7.22: Comparação entre strings

```
/*
Comparação entre caracteres
*/
PROCEDURE Main
LOCAL cNome1, cNome2 // Nomes

   ? "O programa seguinte coloca dois nomes em ordem alfabética"

```

```
   cNome1 := " PEDRO"                                                    9
   cNome2 := "ANA"                                                       10
                                                                         11
   IF ( LTRIM(cNome1) > LTRIM(cNome2) )                                  12
      ? cNome2                                                           13
      ? cNome1                                                           14
   ELSE                                                                  15
      ? cNome1                                                           16
      ? cNome2                                                           17
   ENDIF                                                                 18
                                                                         19
RETURN                                                                   20
```

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
ANA
 PEDRO
```

Note que o programa retirou o espaço em branco apenas para comparar, mas a variável cNome1 continuou com o valor “ Pedro” (com o espaço a esquerda).

Prática número 24

Modifique o programa da listagem 7.22 para que ele exiba o nome PEDRO sem o espaço em branco a esquerda. Veja o modelo :

.:Resultado:.

```
O programa seguinte coloca dois nomes em ordem alfabética
ANA
PEDRO
```

Dica: existe mais de uma forma de se fazer isso. A mais usada é usar a função LTRIM para modificar o conteúdo da variável. Mas fique a vontade para resolver do seu jeito. O importante mesmo é praticar.

7.7.2 O operador de igualdade e o tipo de dado caractere

Dica 45

Quando for comparar caracteres strings, sempre use o operador “==” em vez do operador “=”.

A igualdade entre caracteres obedece a algumas regras adicionais. Nessa seção nós não iremos nos aprofundar muito nessas regras, pois uma abordagem completa iria requerer o conhecimento dos operadores lógicos de negação que nós ainda não vimos. Vamos, portanto para uma abordagem resumida.

Um espaço em branco no final também é um caractere

Bem, se um espaço em branco é um caractere válido, então devemos ter cuidado quando formos comparar strings com caracteres em branco no final delas. A lógica é a

mesma.

Listagem 7.23: Comparação entre strings

```
/*
Comparação entre caracteres
*/
#define CHAVE "harbour123"
PROCEDURE Main
LOCAL cChave // Palavra chave

   ACCEPT "Informe a palavra chave : " TO cChave

   IF ( cChave  ==  CHAVE )
      ? "Acesso concedido"
   ELSE
      ? "Acesso negado"
   ENDIF

RETURN
```

.:Resultado:.

```
Informe a palavra chave : harbour123
Acesso concedido
```

Faça agora o seguinte teste: execute de novo o programa acima, quando você terminar de digitar a palavra chave NÃO TECLE ENTER, mas em vez disso pressione a barra de espaços uma vez. Agora sim, tecle ENTER.

.:Resultado:.

```
Informe a palavra chave : harbour123
Acesso negado
```

Você já deve ter entendido. O espaço adicional que você inseriu também foi interpretado como um caractere, o que fez com que as strings sejam consideradas diferentes.

7.7.3 Os operadores de diferença aplicados ao tipo de dado caracteres.

Tudo o que se aplica aos operadores de igualdade também podem ser aplicados aos operadores de diferença. Veja a seguir o mesmo exemplo da senha utilizando um operador de diferença.

Listagem 7.24: Comparação entre strings

```
/*
Comparação entre caracteres
*/
#define CHAVE "harbour123"
PROCEDURE Main
LOCAL cChave // Palavra chave

   ACCEPT "Informe a palavra chave : " TO cChave
```

```
                                                                    9
   IF ( cChave  !=  CHAVE ) // poderia ser também <> ou #           10
      ? "Acesso negado"                                             11
   ELSE                                                             12
      ? "Acesso concedido"                                          13
   ENDIF                                                            14
                                                                    15
RETURN                                                              16
```

Aqui temos um exemplo com o acesso concedido.

.:Resultado:.

```
Informe a palavra chave : harbour123
Acesso concedido
```

E agora com acesso negado.

.:Resultado:.

```
Informe a palavra chave : 908789
Acesso negado
```

Lembra quando nós dissemos para não usar o operador de igualdade "=" para comparar strings ? Pois bem, a mesma coisa se aplica aos operadores de diferença.

Se você prosseguir com os testes, verá que resultados estranhos irão surgir. Por exemplo:

.:Resultado:.

```
Informe a palavra chave : harbour123456789
Acesso concedido
```

O que houve ? A palavra chave é "harbour123" e o sistema concedeu acesso quando eu digitei "harbour123456789". O Harbour possui algum bug ? Não, não é nada disso.

Vamos primeiro dar a resposta, mas a explicação detalhada está na próxima subseção. A resposta é simples: não use o operador de diferença, use a negação do operador "==". O código a seguir (listagem 7.25) explica com exemplos.

Listagem 7.25: Comparação entre strings

```
/*                                                                  1
Comparação entre caracteres                                         2
*/                                                                  3
#define CHAVE "harbour123"                                          4
PROCEDURE Main                                                      5
LOCAL cChave // Palavra chave                                       6
                                                                    7
   ACCEPT "Informe a palavra chave : " TO cChave                    8
                                                                    9
   IF !( cChave  ==  CHAVE ) // Negação de ==                       10
      ? "Acesso negado"                                             11
   ELSE                                                             12
      ? "Acesso concedido"                                          13
   ENDIF                                                            14
```

```
                                                                    15
RETURN                                                              16
```

Agora sim.

.:Resultado:.

```
Informe a palavra chave : harbour123456789
Acesso nagado
```

Veja na seção a seguir a explicação detalhada desse “mistério” das comparações entre strings.

Aspectos confusos da comparação entre strings

A comparação entre strings é um aspecto confuso da linguagem Harbour, devido ao seu compromisso de otimização de buscas no seu banco de dados interno[4]. É de extrema importância que essa parte fique completamente entendida.

Por padrão temos o seguinte comportamento : **“Se a string da esquerda conter a string da direita então elas são iguais”**.

Abaixo temos duas strings que são consideradas iguais :

```
? "Vlademiro" = "Vlad"  // A comparação retorna .t.
```

Agora, se nós preenchermos com espaços em branco a string da direita (de modo que elas duas tenham o mesmo tamanho), então elas são consideradas diferentes

```
? "Vlademiro" = "Vlad     "  // A comparação retorna .f.
```

Esse comportamento estranho tem a sua explicação em um das características da linguagem Harbour : a existência do seu próprio banco de dados. Segundo Spence : “na prática nós necessitamos de comparações não exatas. [...] Ao procurar os registros em uma base de dados que comecem com S (digamos, procurando por Skorasczewski), queremos comparações não exatas” [Spence 1991, p. 62].

Essa característica também vale para a negação da igualdade quando elas envolvem strings. Os exemplos abaixo podem tornar o código de qualquer um confuso, pois a nossa mente não trabalha bem com negações lógicas.

```
? "Vlademiro" != "Vlad"       // A comparação retorna .f.
? "Vlademiro" != "Vlad     "  // A comparação retorna .t.
```

Isso pode dar um “nó” na cabeça de qualquer um.

[4]O banco de dados internos do Harbour, os arquivos DBF não serão abordados nesse trabalho.

Dica 46

Cuidado quando for comparar variáveis do tipo caractere. Com certeza você percebeu que as regras de igualdade não se aplicam a elas da mesma forma que se aplicam as outras variáveis. Quando for comparar strings use o operador "==" (Duplo igual). Ele evita as comparações esquisitas que você acabou de ver.

```
? "Vlademiro" == "Vlad"  // (.f.)
? "Vlademiro" == "Vlad     "  //  (.f.)
? "Vlademiro" == "Vlademiro " // (.f.) A string da direita
                                        //    tem um espaço a ma
? "Vlademiro" == "Vlademiro" // Só retorna verdadeiro nesse caso
```

Para verificar a diferença use o operador de negação combinado com o operador de duplo igual, conforme o exemplo a seguir :

```
? .NOT. ( "Vlademiro" == "Vlad" ) // Retorna .t. (diferentes)

ou

? !( "Vlademiro" == "Vlad" )
```

Deixe para usar o operador "=" quando você estiver buscando valores aproximados no banco de dados do Harbour [a].
Use o operador duplo igual ("==") para comparar strings entre si e a combinação dos operadores duplo igual e de negação para negar uma comparação. **O ideal mesmo é você usar essa dica para todos os tipos de variáveis e não usar o operador "=".**

[a] O banco de dados do Harbour não segue o padrão SQL. O Harbour pode acessar os bancos de dados relacionais baseados em SQL, mas não diretamente. Esses tópicos estão além do escopo desse livro.

Dica 47

Em resumo:

1. Use o operador == para comparar igualdades.
2. Use a negação do operador == para comparar diferenças.
3. Lembre-se de que um espaço em branco no início ou no final da string já é suficiente para que ela seja diferente de uma string semelhante mas sem os espaços em branco iniciais ou finais.
4. Para eliminar os espaços iniciais de uma string use a função LTRIM, para eliminar os espaços finais de uma string use a função RTRIM. Para eliminar os espaços iniciais e finais use a função **ALLTRIM**.

7.7.4 Um operador que só compara caracteres : $

Outro operador que merece destaque é o operador “$” (Está contido). Ele só funciona com variáveis do tipo caractere e serve para informar se uma cadeia de string está contida na outra.

```
// Leia assim : ``Natal'' está contido em ``Feliz Natal'' ?
? "Natal" $ "Feliz Natal"  // Retorna .t.

? "natal" $ "Natal"  // Retorna .f. ("natal" é diferente de "Natal"
```

Lembre-se: o conteúdo de strings é comparado levando-se em consideração as diferenças entre maiúsculas e minúsculas ("Natal"é diferente de "natal").

Agora, atenção! Vamos supor que você deseja verificar se uma string digitada está contida em outra. Por exemplo: você tem uma lista de estados onde um imposto será aplicado a uma taxa de 2% e deseja usar o operador “está contido” para fazer a verificação. Esses estados são Ceará, Maranhão e Piauí, e a string é composta pela sigla desses estados “CEMAPI”. Conforme a listagem 7.26.

Listagem 7.26: Comparação entre strings

```
/*
Comparação entre caracteres
*/
#define ESTADOS_2PORCENTO "CEMAPI"
PROCEDURE Main
LOCAL cSigla // Sigla

   ACCEPT "Informe a sigla do estado : " TO cSigla

   cSigla := UPPER( cSigla ) // Se o usuário digitou em minúsculas
                             // então converte para maiúscula.

   IF cSigla $ ESTADOS_2PORCENTO
```

```
      ? "Alíquota de 2 por cento"
   ELSE
      ? "Alíquota normal"
   ENDIF



RETURN
```

.:Resultado:.

```
Informe a sigla do estado : CE
Alíquota de 2 por cento
```

Até aí tudo bem, mas se o usuário digitar o estado de Amapá, o programa irá atribuir erroneamente uma alíquota de 2% também, isso é um erro de lógica que pode causar problemas graves no futuro.

.:Resultado:.

```
Informe a sigla do estado : AP
Alíquota de 2 por cento
```

Isso aconteceu porque a string “AP” está contida em “CEMAPI”. Por isso é uma boa prática você usar algum separador qualquer para delimitar as siglas (pode ser um ponto e vírgula, um underline ou dois pontos, por exemplo). A listagem a seguir resolve o problema utilizando dois pontos como separador:

Listagem 7.27: Comparação entre strings

```
/*
Comparação entre caracteres
*/
#define ESTADOS_2PORCENTO "CE:MA:PI"
PROCEDURE Main
LOCAL cSigla // Sigla

   ACCEPT "Informe a sigla do estado : " TO cSigla

   cSigla := UPPER( cSigla ) // Se o usuário digitou em minúsculas
                             // então converte para maiúscula.
   IF ":" $ cSigla
      ? "Formato incorreto"
   ELSE
      IF cSigla $ ESTADOS_2PORCENTO
         ?  "Alíquota de 2 por cento"
      ELSE
         ? "Alíquota normal"
      ENDIF
   ENDIF
```

```
RETURN                                                                    23
```

Agora sim.

.:Resultado:.

```
Informe a sigla do estado : AP
Alíquota normal
```

Sim, eu sei... Se o usuário digitar "E:"o programa vai aceitar. Nesse caso, você deve verificar se a sigla que o usuário digitou é uma sigla válida. A melhor forma de se verificar isso é através de uma função. Como o assunto ainda não foi abordado esse caso não foi contemplado.

7.8 Resumo dos operadores relacionais

A listagem 7.28 a seguir nos trás um pequeno resumo do que foi visto em matéria de operadores nesse capítulo. Procure digitar essa listagem e depois efetuar algumas modificações por conta própria.

Listagem 7.28: Operadores relacionais

```
/*                                                                        1
Operadores relacionais: um resumo.                                        2
*/                                                                        3
PROCEDURE Main                                                            4
LOCAL x, y // Valores numéricos                                           5
LOCAL dData1, dData2 // Datas                                             6
LOCAL cString1, cString2 // Caracteres                                    7
                                                                          8
   SET DATE BRIT // Data dd/mm/aa                                         9
   SET CENTURY ON // Ano com 4 dígitos                                    10
                                                                          11
   ? "Comparando variáveis numéricas"                                     12
   ?                                                                      13
   x := 2                                                                 14
   y := 3                                                                 15
   ? 'x := 2'                                                             16
   ? 'y := 3'                                                             17
   ? "x > y : " , x > y // Retorna falso (.f.)                            18
   ? "y = 3 : " , y = 3 // Retorna verdadeiro (.t.)                       19
   ? "y <> x : ", y <> x // Retorna verdadeiro (.t.)                      20
   ?                                                                      21
   ? "Comparando datas"                                                   22
   ?                                                                      23
   dData1 := CTOD("01/09/2015")                                           24
   dData2 := CTOD("02/09/2015")                                           25
   ? 'dData1 := CTOD("01/09/2015")'                                       26
   ? 'dData2 := CTOD("02/09/2015")'                                       27
   ? "dData1 >= dData2 : ", dData1 >= dData2 // Retorna falso (.f.)       28
                                                                          29
   ? "( dData1 + 1 ) = dData2 : ", ( dData1 + 1 ) = dData2 // Retorna verdadei
```

```
   ?                                                                    30
   ? "Comparando strings"                                               31
   ?                                                                    32
   cString1 := "Vlademiro"                                              33
   cString2 := "Vlad"                                                   34
                                                                        35
   ? 'cString1 := "Vlademiro"'                                          36
   ? 'cString2 := "Vlad"'                                               37
                                                                        38
   ?  " cString1 == cString2 : " , cString1 == cString2 // .f.         39
                                                                        40
   /* Cuidado !! */                                                    41
   /* Leia com atenção o tópico sobre comparação de strings */         42
                                                                        43
   ?  "Cuidado !! com a comparaçäo abaixo. Ela é confusa por causa do =."
   ?  " cString1 = cString2 : " , cString1 = cString2  // .t.          44
   ?  "Use sempre o operador == no lugar de ="                          45
                                                                        46
RETURN                                                                  47
```

.:Resultado:.

```
Comparando variáveis numéricas

x := 2
y := 3
x > y :  .F.
y = 3 :  .T.
y <> x :  .T.

Comparando datas

dData1 := CTOD("01/09/2015")
dData2 := CTOD("02/09/2015")
dData1 >= dData2 :  .F.
( dData1 + 1 ) = dData2 :  .T.

Comparando strings

cString1 := "Vlademiro"
cString2 := "Vlad"
 cString1 == cString2 :  .F.
Cuidado !! com a comparação abaixo. Ela é confusa por causa do =.
 cString1 = cString2 :  .T.
Use sempre o operador == no lugar de =
```

7.9 Exercícios de fixação

1. [Ascencio e Campos 2014, p. 45] Faça um programa que receba o número de horas trabalhadas e o valor do salário mínimo, calcule e mostre o salário a receber,

seguindo essas regras :

a) a hora trabalhada vale a metade do salário mínimo.
b) o salário bruto equivale ao número de horas trabalhadas multiplicado pelo valor da hora trabalhada.
c) o imposto equivale a 3% do salário bruto.
d) o salário a receber equivale ao salário bruto menos o imposto

2. [Ascencio e Campos 2014, p. 64] Faça um programa que receba três números e mostre-os em ordem crescente. Suponha que o usuário digitará três números diferentes.

3. Altere o programa anterior para que ele mesmo não permita que o usuário digite pelo menos dois números iguais.

4. [Ascencio e Campos 2014, p. 78] Faça um programa para resolver equações de segundo grau.

a) O usuário deve informar os valores de *a*, *b* e *c*.
b) O valor de *a* deve ser diferente de zero.
c) O programa deve informar o valor de delta.
d) O programa deve informar o valor das duas raízes.

5. [Forbellone e Eberspacher 2005, p. 46] Faça um programa que leia o ano de nascimento de uma pessoa, calcule e mostre sua idade e verifique se ela já tem idade para votar (16 anos ou mais) e para conseguir a carteira de habilitação (18 anos ou mais).

6. [Forbellone e Eberspacher 2005, p. 47] O IMC (Índice de massa corporal) indica a condição de peso de uma pessoa adulta. A fórmula do IMC é igual ao peso dividido pelo quadrado da altura. Elabore um programa que leia o peso e a altura de um adulto e mostre a sua condição de acordo com a tabela abaixo :

a) abaixo de 18.5 : abaixo do peso
b) entre 18.5 : peso normal
c) entre 25 e 30 : acima do peso
d) acima de 30 : obeso

8 Expressões lógicas complexas e mais estruturas de decisão

O ótimo é inimigo do bom.

Voltaire

Objetivos do capítulo

- Aprender a usar expressões lógicas complexas.
- Aprender a usar os operadores E e OU.
- Entender o conceito de “Ou exclusivo”.
- Compreender o que é uma avaliação de curto-circuito e os cuidados que você deve ter com ela.
- Ser apresentado a estrutura de seleção CASE ... ENDCASE e a sua semelhança com a estrutura IF.
- A estrutura de seleção SWITCH e os cuidados que devem ser tomados com ela.

8.1 Expressões lógicas complexas

No capítulo anterior nós estudamos os operadores relacionais e as suas operações simples de comparação entre dois valores de mesmo tipo. Tais valores são constantes ou variáveis, e o resultado obtido da relação será sempre um valor lógico. Acontece, porém que existem casos onde a expressão a ser analisada como sendo falsa ou verdadeira é composta de várias expressões simples. Por exemplo : “Se chover amanhã e relampejar, eu fico em casa.” Essas expressões são chamadas de expressões lógicas complexas. Assim como, em um idioma humano, uma conjunção pode unir duas orações simples em uma oração complexa, nas linguagens de programação duas ou mais expressões lógicas podem ser combinadas de modo a ter no final um único valor. A maioria das linguagens de programação possuem apenas dois conectivos de expressões lógicas simples: o “E” e o “OU”.

Os operadores lógicos atuam nas variáveis lógicas, nas comparações (vistas na seção anterior) e na negação de expressões. Os operadores estão descritos na tabela 8.1.

Tabela 8.1: Os operadores lógicos

Operação	Operador
Fornece .T. se todos os operandos forem verdadeiros	.AND.
Fornece .T. se um dos operandos forem verdadeiros	.OR.
Fornece verdadeiro se o seu operando for falso	.NOT.
Outra notação para o operador .NOT.	!

Nós já estudamos o operador unário de negação (.NOT. ou !). Agora estudaremos os operadores E e OU.

8.1.1 O operador E

O operador E é usado para realizar uma operação chamada de “conjunção”. Nesse caso, para que a expressão complexa seja considerada verdadeira, então todas as expressões componentes também devem ser verdadeiras. **Se existir uma expressão falsa, então o todo será falso**. A tabela 8.2 lhe ajudará a entender melhor o operador lógico “E”.

Tabela 8.2: Operação de conjunção

A	B	A e B
.F.	.F.	.F.
.F.	.T.	.F.
.T.	.F.	.F.
.T.	.T.	.T.

Quando nós dizemos : “Se chover amanhã e relampejar, eu fico em casa”, nós estamos usando o operador “E” para conectar duas expressões lógicas. O operador “E” sempre devolve um valor verdadeiro se e somente se todas as expressões lógicas forem verdadeiras. Por exemplo, a expressão anterior poderia ser avaliada dessa forma :

```
Choveu ? SIM ou NÃO ?
Resposta: SIM.

Relampejou ? SIM ou NÃO ?
Resposta: NÃO.

Logo, eu posso sair. (A conclusão)
```

O seguinte programa (listagem 8.1) ilustra o exemplo acima.

Listagem 8.1: Expressão lógica

```
/*
Operadores lógicos complexos.
*/
PROCEDURE Main
LOCAL cResp1, cResp2

   ACCEPT "Está chovendo ? (S/N) " TO cResp1
   cResp1 := UPPER( cResp1 ) // Converte a letra para maiúscula caso.

   ACCEPT "Está relampejando ? (S/N) " TO cResp2
   cResp2 := UPPER( cResp2 ) // Converte a letra para maiúscula caso.

   IF ( ( cResp1 == "S" ) .AND. ( cResp2 == "S" ) )
      ? "Fique em casa."
   ELSE
      ? "Pode sair."
   ENDIF

RETURN
```

O usuário respondeu “sim” à primeira pergunta e “não” à segunda pergunta.

.:Resultado:.

```
Está chovendo ? (S/N) s
Está relampejando ? (S/N) n
Pode sair.
```

As expressões complexas podem ter mais de um componente. No exemplo a seguir temos três expressões componentes. O programa a seguir avalia as três notas de um aluno, caso pelo menos uma seja inferior a 3 então o aluno está reprovado sem direito a recuperação.

Listagem 8.2: Expressão lógica
Fonte: codigos/logica01.prg

```
/*
Operadores lógicos complexos.
*/
PROCEDURE Main
```

```
LOCAL nNota1, nNota2, nNota3
LOCAL nMedia


   ? "Para que o aluno passe a média tem que ser maior do que 6."

   ? "Porém, se tiver alguma nota menor do que 3 o aluno está reprovado."
   INPUT "Informe o valor da primeira nota :" TO nNota1
   INPUT "Informe o valor da segunda nota :" TO nNota2
   INPUT "Informe o valor da terceira nota :" TO nNota3

   IF ( nNota1 >= 3 .AND. nNota2 >= 3 .AND. nNota3 >= 3 )
      nMedia := ( nNota1 + nNota2 + nNota3 ) / 3
      IF nMedia >= 6
         ? "Aluno aprovado"
      ELSE

   ? "Aluno de recuperação. Aguardando o resultado da próxima prova"
      ENDIF
   ELSE
      ? "Aluno reprovado sem direito a recuperação."
   ENDIF

RETURN
```

Um suposto aluno tirou 10, 10 e 2,5. Apesar do esforço ele foi reprovado.

.:Resultado:.

```
Para que o aluno passe a média tem que ser maior do que 6.
Porém, se tiver alguma nota menor do que 3 o aluno está reprovado.
Informe o valor da primeira nota :10
Informe o valor da segunda nota :10
Informe o valor da terceira nota :2.5
Aluno reprovado sem direito a recuperação.
```

8.1.2 O operador OU

O operador OU é usado para realizar uma operação chamada de “disjunção não exclusiva”. Nesse caso, para que a expressão complexa seja considerada verdadeira, então basta que, no mínimo, uma das expressões componentes seja verdadeira. **Se existir uma expressão verdadeira, então o todo será verdadeiro**. A tabela 8.3 lhe ajudará a entender melhor o operador lógico “OU”.

Tabela 8.3: Operação de disjunção não-exclusiva

A	B	A ou B
.F.	.F.	.F.
.F.	.T.	.T.
.T.	.F.	.T.
.T.	.T.	.T.

Quando nós dizemos : “Se chover amanhã ou relampejar, eu fico em casa”, nós estamos usando o operador “OU” para conectar duas expressões lógicas. O operador “OU” sempre devolve um valor verdadeiro se no mínimo uma das expressões lógicas for verdadeira. Nesse caso, as possibilidades se ficar em casa aumentam, pois basta relampejar ou chover para que eu fique em casa.

Lembre-se: O operador lógico OU é representado por .OR.

Prática número 25

Faça uma alteração no programa da listagem 8.1 para que ele passe a usar o “OU” (.OR.) ao invés de “E” (.AND.) .
Faça o mesmo com o programa da listagem 8.2.

Cuidado com o “OU exclusivo”.

Se você prestou bastante atenção, verá que nós usamos o termo “disjunção não-exclusiva” para nomear expressões com “OU”. Isso significa que existem dois tipos de disjunções (usos do “OU”). Uma disjunção não-exclusiva, que nós já vimos, e uma disjunção exclusiva, também conhecida por “OU exclusivo”. O Harbour e a maioria das linguagens de programação **não** tem esse tal “OU exclusivo”, portanto não se preocupe com ele. Mas é bom você saber o que isso significa. A palavra “OU” pode ter dois significados no nosso dia-a-dia. O significado não-exclusivo, que é o que nós usamos no Harbour e na maioria das linguagens, e o significado exclusivo. Um exemplo de “ou exclusivo”: quando alguém lhe pede para escolher uma de duas opções, como o personagem Morpheus, do filme Matrix, mostrando duas pílulas para o Neo. Nesse caso o Neo teve que fazer uma escolha. Ou seja, com o “OU exclusivo” você **precisa** escolher uma das opções. Isso é um exemplo de OU exclusivo, ou seja, uma das opções tem que necessariamente ser a verdadeira.

Você pode estar pensando: “para que complicar me ensinando algo que o Harbour não tem ?”. A resposta é : “você precisa entender certos conceitos de lógica, mesmo que o Harbour não utilize tais conceitos.” O “OU exclusivo” pode aparecer em algum algoritmo ou em alguma outra linguagem futura que você venha a aprender [1]. Quando isso acontecer (não é “se”, mas “quando”) você estará preparado.

8.1.3 Avaliação de “curto-circuito”

Os operadores lógicos permitem concatenar várias comparações efetuadas entre operadores relacionais. Por exemplo :

```
a := 10
b := 2
c := 3
? b > a .AND. c > b // Falso
```

No exemplo acima, temos duas expressões: “b > a” e “c > b” conectadas por um operador .AND. . Note que o valor de b não é maior do que o valor de a, nesse caso

[1] C++ e Visual Basic possuem o operador “OU exclusivo”. No Visual Basic ele chama-se XOR.

avaliar “c > b” seria uma perda de tempo e de recursos. Lembre-se que o operador .AND. requer que todas as expressões sejam verdadeiras, mas se logo a primeira é falsa, porque eu deveria avaliar as demais ? Portanto, o programa não irá mais avaliar “c > b” pois seria uma perda de tempo. Esse tipo de avaliação é chamada de avaliação de curto circuito [Spence 1991, p. 66].

Avaliações de curto circuito são comuns em todas as linguagens de programação. É uma forma de economizar processamento e recursos, porém você deve ter cuidado quando for testar as suas expressões. Procure testar todos os casos de verdadeiro e falso em expressões que contém .AND., pois se as expressões seguintes tiverem algum erro de sintaxe, elas não serão avaliadas caso a primeira expressão seja falsa. Vamos alterar o programa das três notas, que nós já vimos para ilustrar isso. O exemplo da listagem 8.3 é uma versão modificada do programa das três notas. Ele possui um erro de sintaxe na linha 15: em vez de digitar “nNota3 >= 3” eu digitei “nNota3 >= e”. Isso irá gerar um erro de sintaxe, pois o programa irá buscar o valor da variável “e” que não existe.

Listagem 8.3: Expressão lógica

```
/*                                                                          1
Operadores lógicos complexos.                                               2
*/                                                                          3
PROCEDURE Main                                                              4
LOCAL nNota1, nNota2, nNota3                                                5
LOCAL nMedia                                                                6
                                                                            7
                                                                            8
   ? "Para que o aluno passe a média tem que ser maior do que 6."           9
                                                                            10
   ? "Porém, se tiver alguma nota menor do que 3 o aluno está reprovado."
   INPUT "Informe o valor da primeira nota :" TO nNota1                     11
   INPUT "Informe o valor da segunda nota :" TO nNota2                      12
   INPUT "Informe o valor da terceira nota :" TO nNota3                     13
                                                                            14
   IF ( nNota1 >= 3 .AND. nNota2 >= 3 .AND. nNota3 >= e )                   15
      nMedia := ( nNota1 + nNota2 + nNota3 ) / 3                            16
      IF nMedia >= 6                                                        17
         ? "Aluno aprovado"                                                 18
      ELSE                                                                  19
                                                                            20
   ? "Aluno de recuperaçäo. Aguardando o resultado da próxima prova"
      ENDIF                                                                 21
   ELSE                                                                     22
      ? "Aluno reprovado sem direito a recuperaçäo."                        23
   ENDIF                                                                    24
                                                                            25
RETURN                                                                      26
```

Porém, quando eu executo com os valores abaixo o programa não gera um erro de sintaxe: um suposto aluno tirou 2, 10 e 10. Mas o programa “rodou” perfeitamente.

.:Resultado:.

```
Para que o aluno passe a média tem que ser maior do que 6.
```

```
Porém, se tiver alguma nota menor do que 3 o aluno está reprovado.
Informe o valor da primeira nota :2
Informe o valor da segunda nota :10
Informe o valor da terceira nota :10
Aluno reprovado sem direito a recuperação.
```

Se você observou atentamente, deve ter notado que a primeira expressão é falsa, portanto o programa não avaliou as outras.

O segredo é testar todos os casos.

Vamos agora testar com 7, 8 e 3.

.:Resultado:.

```
Para que o aluno passe a média tem que ser maior do que 6.
Porém, se tiver alguma nota menor do que 3 o aluno está reprovado.
Informe o valor da primeira nota :7
Informe o valor da segunda nota :8
Informe o valor da terceira nota :3
Error BASE/1003  Variable does not exist: E
Called from MAIN(15)
```

Agora sim, o erro apareceu. Moral da história: “em expressões complexas com o operador lógico .AND. procure testar todos os casos possíveis.”

8.2 A estrutura de seleção DO CASE ... ENDCASE

Nós já estudamos a estrutura de decisão IF, mas o Harbour possui mais duas estruturas que podem ser usadas: o CASE e o SWITCH. Se você entendeu o funcionamento do IF, o entendimento dessas outras duas não será problema para você. Vamos iniciar com o CASE.

A estrutura de seleção DO CASE ... ENDCASE é uma variante da estrutura IF ... ELSEIF ... ELSE ... ENDIF. Vamos continuar aquele exemplo, do capítulo anterior, que determina se o aluno passou de ano ou não : suponha agora que o operador se enganou na hora de introduzir a nota do aluno e digitou 11 em vez de 1. O programa não pode aceitar valores inválidos: o menor valor possível é zero e o maior valor possível é dez.

As duas listagens a seguir resolvem o problema. A primeira utiliza IF ... ELSEIF ... ELSE ... ENDIF e a segunda utiliza DO CASE ... ENDCASE. Note que as duas estruturas são equivalentes.

Listagem 8.4: Estrutura de seleção IF ... ELSEIF ... ELSE ... ENDIF

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota

   // Recebe o valor da nota
   nNota := 0
   INPUT "Informe a nota do aluno : " TO nNota

```

```
   // Decide se foi aprovado ou não de acordo com a média
   IF ( nNota > 10 ) .OR. ( nNota < 0 )
      ? "Valor inválido"
   ELSEIF ( nNota >= 7 )
      ? "Aprovado"
   ELSEIF ( nNota >= 5 )
      ? "Recuperaçäo"
   ELSEIF ( nNota >= 0 )
      ? "Reprovado"
   ELSE
      ? "Valor inválido"
   ENDIF


RETURN
```

Agora usando DO CASE ... ENDCASE na listagem 8.5.

Listagem 8.5: Estrutura de seleção DO CASE ... ENDCASE
Fonte: codigos/if06.prg

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota

   // Recebe o valor da nota
   nNota := 0
   INPUT "Informe a nota do aluno : " TO nNota

   // Decide se foi aprovado ou não de acordo com a média
   DO CASE
   CASE ( nNota > 10 ) .OR. ( nNota < 0 )
      ? "Valor inválido"
   CASE ( nNota >= 7 )
      ? "Aprovado"
   CASE ( nNota >= 5 )
      ? "Recuperaçäo"
   CASE ( nNota >= 0 )
      ? "Reprovado"
   OTHERWISE
      ? "Valor inválido"
   ENDCASE


RETURN
```

Essa listagem está representada no apêndice A. Caso queira vê-la, vá para o apêndice A e busque a representação da listagem 8.5.

O resultado não muda. Tanto faz usar IF ... ELSEIF ... ELSE ... ENDIF ou usar DO CASE ... ENDCASE.

.:Resultado:.

```
Informe a nota do aluno : 9.5
Aprovado
```

Dica 48

Ao testar múltiplas condições, uma declaração CASE é preferível a uma longa sequencia de IF/ELSEIFs. [...] A sentença OTHERWISE serve como a última condição; cuida de qualquer situação não prevista por um CASE [Spence 1991, p. 70].

Descrição sintática 7

1. Nome : DO CASE
2. Classificação : declaração.
3. Descrição : Executa um dentre vários blocos de instruções.
4. Sintaxe

```
DO CASE
CASE <lcondição1>
     <instruções>...
[CASE <lcondição2>]
     <instruções>...
[OTHERWISE]
     <instruções>...
END[CASE]
```

Fonte : [Nantucket 1990, p. 2-7]

Existe mais um problema que pode ocorrer. Imagine agora que o usuário em vez de digitar a nota 6 digite um Y e passe um caractere para ser avaliado, quando a variável foi definida como numérica. Isso é um erro que faz com que o programa pare no meio da execução (erros de execução). Vamos abordar essa situação quando estivermos estudando o controle de erros do Harbour.

8.3 SWITCH

A estrutura de seleção SWITCH pode ser exemplificada na listagem 8.6.

Listagem 8.6: Um exemplo de SWITCH

```
/*
Switch
*/
PROCEDURE Main
LOCAL nOpc

```

```
   ? "Uso de SWITCH "                                                    7
   INPUT "Informe a sua opção ( 1 , 2 ou 3)  : " TO nOpc                 8
                                                                         9
   SWITCH nOpc                                                           10
   CASE 1                                                                11
      ? "Escolheu 1"                                                     12
   CASE 2                                                                13
      ? "Escolheu 2"                                                     14
      EXIT                                                               15
   CASE 3                                                                16
      ? "Escolheu 3"                                                     17
   OTHERWISE                                                             18
      ? "Opção inválida"                                                 19
   END                                                                   20
   ? "Terminou!"                                                         21
                                                                         22
RETURN                                                                   23
```

Essa estrutura é uma das implementações novas do Harbour, ela foi inspirada na estrutura de seleção switch da linguagem C. Apesar de simples, essa estrutura possui algumas particularidades que listaremos a seguir :

1. A estrutura irá analisar se o valor da variável é igual aos valores informados nos respectivos CASEs, quando o valor for igual então o processamento irá “entrar” no CASE correspondente.

2. Cuidado, uma vez que o processamento entre no CASE correspondente, você deve informar para ele sair com a instrução EXIT. Se você não digitar EXIT então o processamento irá “entrar” nos CASEs subsequentes, mesmo que o valor da variável não seja igual ao valor do CASE.

A listagem 8.6 ilustra isso. Se o usuário digitar 1, então o programa irá entrar nos CASEs 1 e 2 até que encontra um EXIT para sair. Caso o usuário digite 2, então o programa irá entrar no CASE 2, encontrar um EXIT e sair. Se, por sua vez, o usuário digitar 3 o programa irá entrar no CASE 3 e na cláusula OTHERWISE. Esse comportamento estranho é verificado na Linguagem C e em todas as linguagens derivadas dela, como C++, PHP e Java. Inclusive, existe um termo para esse comportamento : “fall through”, que significa aproximadamente “despencar”. Kernighan e Pike [Kernighan e Pike 2000, p. 18] nos recomendam para não criarmos *fall through* dentro de um SWITCH. Apenas um caso de *fall through* é tido como claro, que é quando vários cases tem um código idêntico, conforme abaixo :

```
SWITCH nOpc
CASE 1
CASE 2
CASE 3
   ...
   EXIT
CASE 4
```

```
      ...
      EXIT
   END
```

A estrutura de seleção SWITCH não aceita expressões em sua condição, por exemplo, o código da listagem 8.7 **está errado** pois contém uma expressão nas avaliações de condição :

Listagem 8.7: Um exemplo de SWITCH COM ERRO

```
/*
Switch
*/
PROCEDURE Main
LOCAL nAltura

   ? "SIMULAÇÃO DE ERRO COM SWITCH "
   INPUT "Informe a sua altura (Ex: 1.70)  : " TO nAltura

   SWITCH nAltura
   CASE nAltura > 2.0
      ? "ALTO"
   CASE nAltura > 1.60 .AND. nAltura < 2.0
      ? "ESTATURA MEDIANA"
   CASE nAltura <= 1.6
      ? "BAIXA ESTATURA"
   END

RETURN
```

Com tantas limitações, qual a vantagem de se usar SWITCH ? Não bastaria um DO CASE ... END CASE ou um IF .. ELSEIF ... ELSE ... ENDIF ?

Três motivos justificam o uso do SWITCH :

1. Ele é muito mais rápido [2] do que as outras estruturas de seleção. Se você for colocar uma estrutura de seleção dentro de um laço que se repetirá milhares de vezes, a performance obtida com um SWITCH pode justificar o seu uso.

2. O SWITCH é uma estrutura que existe em outras linguagens, como já dissemos, por isso ela não é nenhuma característica “esquisita” da linguagem Harbour. Muitos programadores usam SWITCH em seus códigos em C, C++, etc.

3. O SWITCH obriga a análise de apenas uma variável. Isso torna a programação mais segura quando se quer analisar apenas um valor.

Dica 49

Não esqueça do EXIT dentro de um caso de um SWITCH. Só omita o EXIT (*fall through*) quando múltiplos casos devam ser tratados da mesma maneira.

[2] Maiores detalhes em `http://www.kresin.ru/en/hrbfaq_3.html` [Kresin 2016].

Descrição sintática 8

SWITCH : Executa um dentre vários blocos de instruções.

```
SWITCH <xVariavel>
CASE <valor1>
         <instruções>...
         [EXIT]
[CASE <valor2>]
         <instruções>...
         [EXIT]
[OTHERWISE]
         <instruções>...
END
```

9 Estruturas de repetição

E se um dia ou uma noite um demônio se esgueirasse em tua mais solitária solidão e te dissesse: ”Esta vida, assim como tu vives agora e como a viveste, terás de vivê-la ainda uma vez e ainda inúmeras vezes: e não haverá nela nada de novo, cada dor e cada prazer e cada pensamento e suspiro e tudo o que há de indivisivelmente pequeno e de grande em tua vida há de te retornar, e tudo na mesma ordem e sequência - e do mesmo modo esta aranha e este luar entre as árvores, e do mesmo modo este instante e eu próprio. A eterna ampulheta da existência será sempre virada outra vez, e tu com ela, poeirinha da poeira!”

Friedrich Nietzsche

Objetivos do capítulo

- Entender o problema que uma estrutura de repetição resolve.
- Representar graficamente uma estrutura de repetição.
- Classificar os tipos de estruturas de repetição.
- As estruturas de repetição do Harbour.

Estrutura de repetição (ou *loops*) são estruturas que repetem uma mesma sequência de código um dado número de vezes. O número de vezes com que essa sequência é repetida pode ser fixa (conhecida antecipadamente) ou pode ser variável (baseada em uma condição de saída). O Harbour possui esses dois tipos de estruturas de repetição. Segundo Swan,

> "os loops permitem programar processos repetitivos, economizando espaço em código (você tem que digitar o comando repetitivo apenas uma vez não importando quantas vezes ele seja necessário) e executando inúmeras tarefas.[...] Sem loops a programação de computador como a conhecemos não existiria [Swan 1994, p. 107].

A criação das estruturas de repetição marca uma nova fase na programação de computadores. Antigamente, muito antigamente (década de 1950 e 1960), os programadores usavam um comando chamado GOTO para "poder chegar" a um determinado trecho do seu código. Isso levava a códigos confusos e mal estruturados. O problema decorria basicamente da necessidade de voltar para um trecho anterior do código. Por exemplo : vamos supor que eu queira imprimir a sequencia de números de 1 até 3. O código está representado abaixo :

```
PROCEDURE Main

   ? 1
   ? 2
   ? 3

RETURN
```

Agora, vamos supor que alguém lhe peça para imprimir a sequencia de números de 1 até 1000! Não é uma solução inteligente usar o mesmo método da listagem anterior. Mas como fazer isso se eu só tenho as estruturas de sequencia e as de decisão ? Foi aí que alguém resolveu inventar um comando chamado GOTO (GOTO significa "Go to" ou em português "Vá para"). Esse comando permite que a sequência de leitura do código seja quebrado para um ponto anterior (ou posterior) do código. [1]. Nós vamos ilustrar o GOTO usando o algoritmo da listagem 13.

[1] **Esse tipo de desvio não existe no Harbour**. O Harbour possui um comando com esse nome, mas ele serve para outros fins

Algoritmo 13: Exibe o valor de 1 até 1000.

```
1  início
2      x ← 0
3      :ContaDeNovo
4      x ← x + 1
5      escreva x
6      se x > 1000 então
7          Vá para :Fora
8      fim
9      Vá para :ContaDeNovo
10     :Fora
11     escreva "Fim"
12 fim
```

A listagem acima é apenas para ilustrar o uso do GOTO (Vá para), ela possui as seguintes características adicionais : uma marcação chamada "etiqueta" que se inicia com o símbolo ":" e um comando "Vá para" que me manda para a etiqueta em questão. Essas duas "novidades" **não existem no Harbour**, é so para ilustrar o uso do GOTO. Vamos comentar esse programa hipotético :

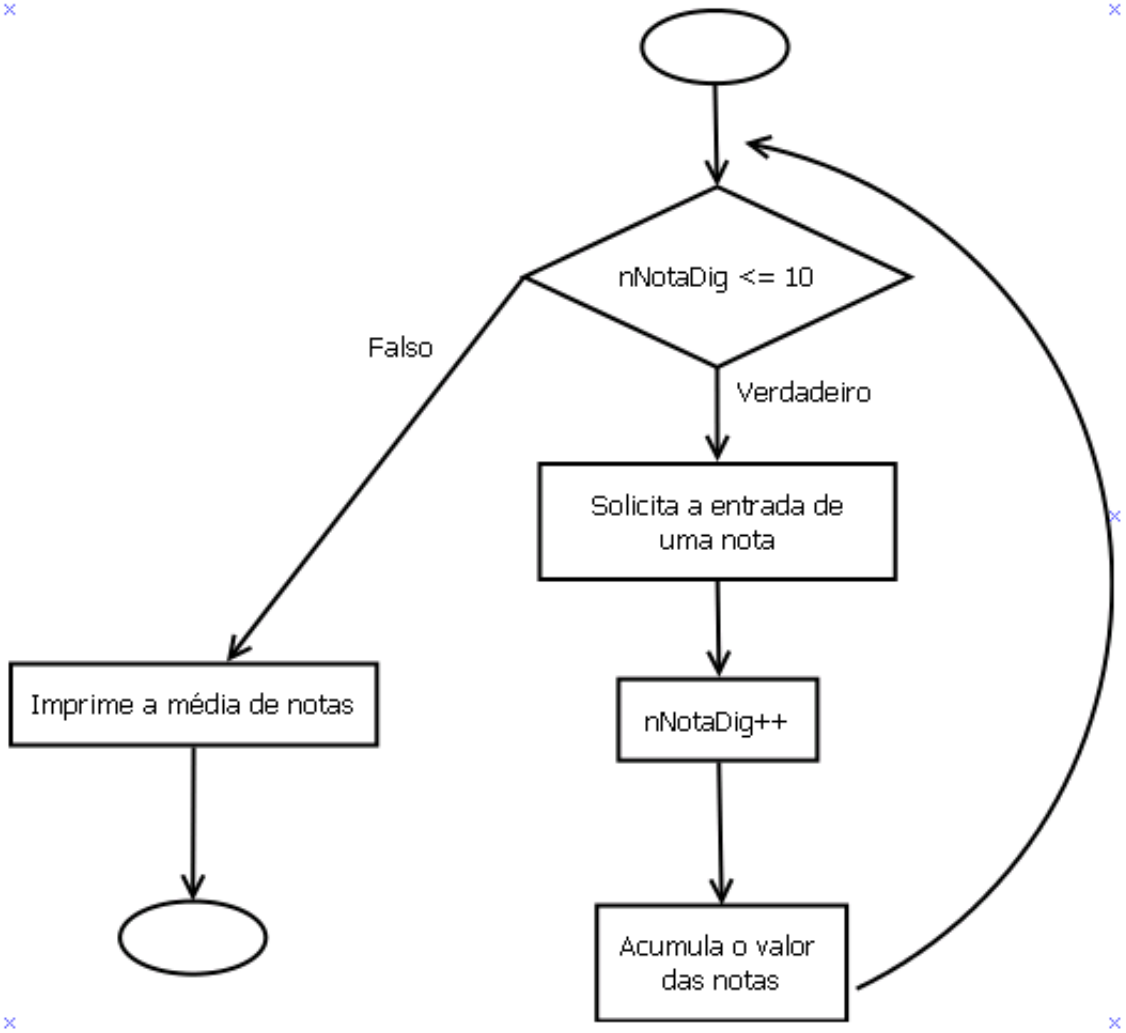
1. Primeiro declara a variável e inicia com valor zero.
2. Cria uma "etiqueta" chamada ":ContaDeNovo" para marcar um ponto de repetição.
3. Depois incrementa a variável.
4. Se o valor da variável for maior do que 1000, então executa um comando "Vá para" que o manda para a etiqueta ":Fora" (linha 10) do nosso hipotético programa e o encerra. Se a variável não for maior do que 1000 então continua e encontra a um comando "Vá para" (linha 9) que desvia a sequência para a etiqueta ":ContaDeNovo".

Dessa forma, o programa poderia retornar para a linha 3 e incrementar x até que ele chegue ao valor desejado. Tudo bem se você não entendeu completamente a listagem hipotética, o fato é que tais recursos existiam nas antigas linguagens de programação e eram a única forma de desviar a sequência para um ponto anterior. Ninguém questionava tal comando, até que os códigos ficaram confusos e difíceis de manter. Um grupo de programadores insistia que não tinha nada de errado com o GOTO e que a culpa era dos programadores despreparados que escreviam códigos ruins. Outro grupo afirmava que o GOTO levava a uma programação mal estruturada, pois "abria uma porta" para que códigos ruins fossem escritos, afinal de contas, errar é humano. Essa disputa foi definitivamente encerrada somente na década de 1980, até lá todas as linguagens tinham um comando chamado GOTO. Atualmente apenas as linguagens C e Pascal possuem esse comando, não porque elas precisem dele, mas porque elas nasceram na década de 1970, em meio a essa controvérsia.

O problema do GOTO foi definitivamente resolvido através das estruturas de repetição da linguagem. A figura 9.1 mostra como um fluxograma representa uma estrutura de repetição. Nós resolvemos "recortar" apenas a parte em que a estrutura de repetição é implementada, deixando de fora a inicialização de nNotaDig, que é feito antes do laço. Note que o fluxograma não possui nenhuma notação para estruturas de repetição,

mas ela pode facilmente ser notada porque existe um bloco de código associado a uma estrutura de decisão. No final desse bloco de código uma seta retorna para a estrutura de decisão para verificar a condição de permanência na estrutura. Esse é o grande defeito do fluxograma pois, como ele foi criado na época do GOTO, a ausência de uma notação específica para estruturas de repetição abre uma brecha para um código confuso. Existe uma outra notação gráfica chamada de Diagrama N-S que resolve esse problema, mas nós não usaremos esse diagrama nesse livro.

Figura 9.1: Fluxograma de uma estrutura de repetição



Dica 50

Antes de escrever o seu código faça um pseudo-código, depois de criar o código em Harbour tente representar o seu código com um fluxograma. O pseudo-código é mais importante do que um fluxograma, portanto habitue-se a criar um pseudo-código antes de criar o código na linguagem de programação. Com o passar do tempo você pode abandonar a criação de fluxogramas, pois eles são mais úteis na etapa de aprendizado.

9.1 Anatomia das estruturas de repetição

As estruturas de repetição tem os seguintes componentes:

- Um bloco que irá se repetir
- Uma condição (ou teste) que fará com que a repetição pare

9.2 Classificação das estruturas de repetição

De acordo com a condição de saída nós temos três tipos de blocos:

- Repetição com teste no início
- Repetição com teste no final
- Repetição com um número fixo de vezes

9.2.1 Repetição com teste no início

Vamos resolver agora o problema da exibição dos números de 1 até 1000 usando uma estrutura de repetição com teste no início.

Algoritmo 14: Estrutura de repetição com teste no início.

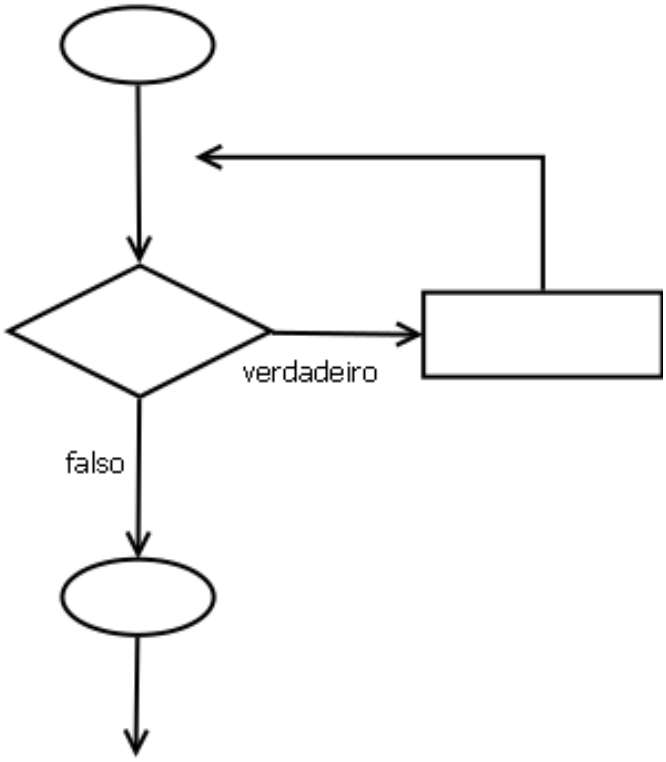
```
   Entrada: x
   Saída: Exibe o valor de 1 até 1000
1  início
2  |  x ← 1
3  |  enquanto  x <= 1000 faça
4  |  |  escreva x
5  |  |  x ← x + 1
6  |  fim
7  |  escreva "Fim"
8  fim
```

A estrutura de repetição **enquanto**[2] é aquela em que o bloco do laço (linhas 4 e 5 do algoritmo 14) se repete enquanto é cumprida uma determinada condição (x <= 1000). Se a condição for avaliada falsa o programa prossegue para a linha após o laço (linha 7).

A representação em fluxograma dessa estrutura está representada na imagem 9.2.

[2] Em inglês *while*

Figura 9.2: Estrutura de repetição WHILE



DO WHILE : A implementação do laço *enquanto*

A estrutura DO WHILE (Faça enquanto) é a mais básica de todas, nela o programa entra em *loop* e fica repetindo enquanto for verdade a condição de permanência. Conforme a listagem 9.1, que implementa o algoritmo 14.

Listagem 9.1: DO WHILE
Fonte: codigos/loop01.prg

```
/*
Estruturas de controle de fluxo
*/
PROCEDURE Main
LOCAL nNota // Nota

   nNota := 1 // Recebe o valor da nota
   DO WHILE ( nNota <= 1000 ) // Condição de permanência
      // O programa ficará aqui enquanto
      // nNota for menor ou igual a 1000
      ? nNota++
   ENDDO
   ? "Fim : ", nNota  // Exibe 1001

RETURN
```

O quadro abaixo exibe o final da execução do programa da listagem 9.1.

.:Resultado:.

```
       993
       994
       995
       996
       997
       998
       999
      1000
Fim :        1001
```

Dica 51

Cuidado com a condição de saída do laço. No caso da listagem 9.1 essa condição é satisfeita graças ao incremento da variável **nNota** na linha 16. Sem esse incremento o programa nunca iria finalizar normalmente (o programa nunca iria sair do *loop*).

Dica 52

Cuidado com a condição de entrada no laço. No caso da listagem 9.1 essa condição é satisfeita graças ao valor de **nNota** que é menor ou igual a 1000. Se a variável fosse maior do que 1000 o *loop* nunca seria executado. Em determinadas situações isso é o certo, mas em outras situações não, tudo irá depender do problema a ser resolvido. O que queremos dizer é que os laços exigem que você tenha um controle sobre as condições de entrada e saída dele.

9.2.2 Repetição controlada por um contador

O programa da listagem 9.1 foi desenvolvido usando uma técnica conhecida como "repetição controlada por contador". Nela, nós sabemos antecipadamente quantas iterações [3] o programa irá realizar. Naquele caso ele realizou 1000 iterações. Vamos agora apresentar um outro exemplo dessa mesma técnica. O programa da listagem 9.2 recebe a nota de 10 alunos e exibe a média da turma no final.

Listagem 9.2: DO WHILE (Digitação de notas)
Fonte: codigos/loop02.prg

```
/*
Estruturas de controle de fluxo
Adaptada de (DEITEL; DEITEL, 2001, p.111)

Programa para receber a nota de 10 alunos e exibir a média da turma
Entrada : 10 noras
Saída : A média
*/
#define QTD_ALUNOS 10 // Quantidade de iterações
```

[3] Iteração significa : "repetição de alguma coisa ou de algum processo". No nosso contexto o termo iteração pode ser interpretado como : "o número de vezes que o interior do laço é repetido". Não confunda iteração com interação. Interação significa "ação mútua" ou "influência recíproca entre dois elementos".

```
PROCEDURE Main
LOCAL nTotNotas ,; // Soma das notas
      nNotaDig ,; // Número de notas digitadas
      nNota ,; // O valor de uma nota
      nMedia // Médias de notas

   // Inicialização
   nTotNotas := 0
   nNotaDig := 1 // Condição de permanência no laço

   // Processamento
   DO WHILE ( nNotaDig <= QTD_ALUNOS ) // Repete QTD_ALUNOS vezes
      INPUT "Forneça a nota do aluno : " TO nNota
      nNotaDig++
      nTotNotas += nNota
   ENDDO

   // Fase de término
   nMedia := nTotNotas / QTD_ALUNOS
   ? "A média geral é ", nMedia  // Exibe a média

RETURN
```

.:Resultado:.

```
Forneça a nota do aluno : 10
Forneça a nota do aluno : 9
Forneça a nota do aluno : 6
Forneça a nota do aluno : 7
Forneça a nota do aluno : 4
Forneça a nota do aluno : 6
Forneça a nota do aluno : 9
Forneça a nota do aluno : 10
Forneça a nota do aluno : 7
Forneça a nota do aluno : 9
A média geral é           7.70
```

9.2.3 Repetição controlada por um sentinela ou Repetição indefinida

Uma repetição controlada por um sentinela (ou repetição indefinida) permite que o laço seja repetido um número indefinido de vezes, dessa forma o usuário é quem define a condição de saida do laço. Ele vale tanto para a digitação de uma nota quanto para a digitação de um número muito grande de notas. Essa técnica envolve uma variável numérica conhecida como sentinela, cujo valor é usado para armazenar a nota do aluno corrente, mas também serve para sair do laço quando o valor for igual a um número pré-determinado. Segundo Deitel, “o valor da sentinela deve ser escolhido de forma que não possa ser confundido com um valor aceitável fornecido como entrada” [Deitel e Deitel 2001, p. 114].

Listagem 9.3: DO WHILE (Digitação de notas com sentinela)
Fonte: codigos/loop03.prg

```
/*
Estruturas de controle de fluxo
Programa para cálculo da média da turma com repetição controlada por sentinela
Adaptada de (DEITEL; DEITEL, 2001, p.117)
Programa para receber a nota de N alunos e exibir a média da turma
Entrada : N notas
Saída : A média
*/
PROCEDURE Main
LOCAL nTotNotas ,; // Soma das notas
      nNotaDig ,; // Número de notas digitadas
      nNota ,; // O valor de uma nota
      nMedia // Médias de notas


   // Inicialização
   nTotNotas := 0
   nNotaDig := 0
   nNota := 0
   ? "Forneça a nota ou tecle -1 para finalizar"
   INPUT "Forneça a nota : " TO nNota

   // Processamento
   DO WHILE ( nNota <> -1 ) // Repete enquanto não for -1
      nTotNotas += nNota
      nNotaDig++
      INPUT "Forneça a nota : " TO nNota
   ENDDO

   // Fase de término
   IF ( nNotaDig <> 0)
      nMedia := nTotNotas / nNotaDig
      ? "A média geral é ", nMedia  // Exibe a média
   ELSE
      ? "Nenhuma nota foi digitada"
   ENDIF

RETURN
```

.:Resultado:.

```
Forneça a nota ou tecle -1 para finalizar
Forneça a nota : 10
Forneça a nota : 8
Forneça a nota : 9
Forneça a nota : -1
A média geral é          9.00
```

Dica 53

Quando for criar uma repetição indefinida (baseada em sentinela) procure escolher um valor de saída fora do intervalo válido, dessa forma o usuário não corre o risco de digitar esse valor pensando que ele é um valor válido. Alguns exemplos de valores de sentinela : -1, 9999, -9999, etc.

Dica 54

Boas práticas para se criar uma repetição indefinida :

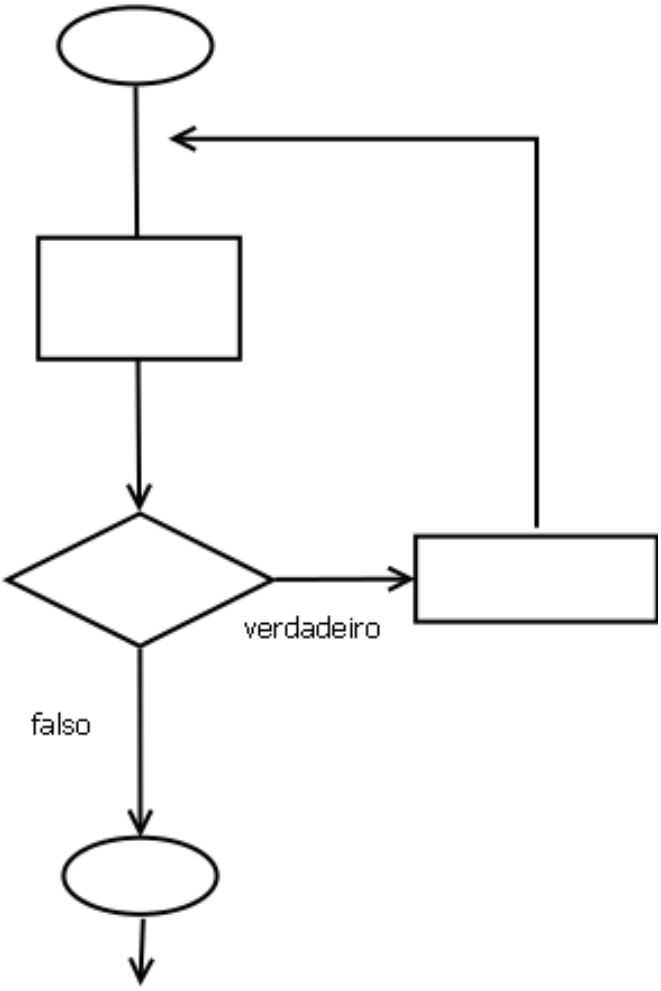
1. O primeiro valor deverá ser lido antes da entrada do laço.
2. O próximo valor deverá ser lido antes de ser avaliado.

Na nossa listagem 9.3 esses valores são lidos pelo comando *INPUT*.

9.2.4 Repetição com teste no final

No exemplo a seguir temos um algoritmo com o teste no final. Isso significa dizer que o laço é executado pelo menos uma vez, pois o teste de saída é feito no final do laço. A representação em fluxograma desse tipo de laço é :

Figura 9.3: Estrutura de repetição REPITA ATÉ



Note que, de forma implícita, temos um losango (que equivale a uma estrutura de decisão). Esse losango representa a condição de saída do laço.

O seguinte algoritmo nos mostra como fazer uma xícara de chá.

Algoritmo 15: Fazer um chá

```
1  início
2      Pegar a chaleira
3      Colocar água
4      Acender o fogo
5      Colocar a chaleira no fogo
6      repita
7          Esperar
8      até ferver a água;
9      Pegar saquinho de chá
10     Introduzí-lo na chaleira
11     repita
12         Esperar
13     até chá ficar pronto;
14 fim
```

Note o raciocínio por trás do algoritmo: eu sempre vou ter que esperar um tempo

determinado enquanto a água não ferve e enquanto o chá não fica pronto. Não existe a possibilidade de eu não esperar (não entrar no laço).

O equivalente, em pseudo-código, do problema da troca de lâmpada, abordado no capítulo sobre algoritmos está representado abaixo :

Algoritmo 16: Um exemplo de uma estrutura de repetição (troca de lâmpada)

```
1  início
2      Acionar o interruptor
3      se lâmpada não acender então
4          Pegar uma escada
5          Posicionar a escada embaixo da lâmpada
6          repita
7              Buscar uma lâmpada nova
8              Subir na escada
9              Retirar a lâmpada velha
10             Colocar a lâmpada nova
11             Descer da escada
12             Acionar o interruptor
13         até lâmpada acender OU não tiver lâmpada no estoque;
14 fim
```

Agora vamos ver outro exemplo: é muito frequente a realização de validações de entrada de dados nas aplicações. O exemplo a seguir usa o laço com teste no final para ver se o número é maior do que zero.

Algoritmo 17: Valida o número

Entrada: número
Saída: Verifica se o número é maior do que zero

```
1  início
2      número ← 0
3      repita
4          leia número
5          se (número <= 0) então
6              escreva "O valor deve ser maior do que zero"
7      até número > 0;
8  fim
```

Esse algoritmo corresponde ao fluxograma a seguir :

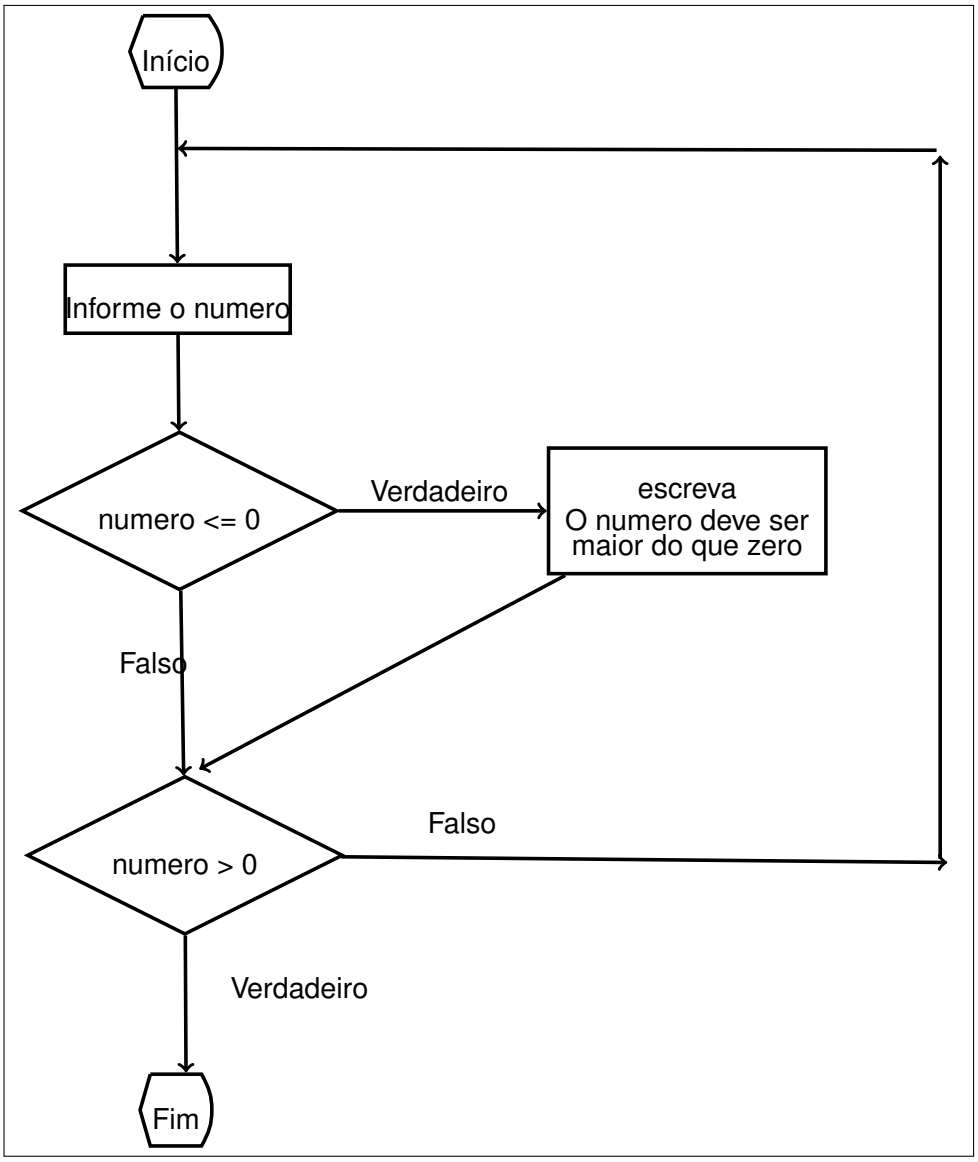


Figura 9.4: Validação de número

Dica 55

Os pseudo-códigos são mais importantes do que os fluxogramas. O fluxograma é importante porque ele nos dá um contato visual mais amplo com a rotina, mas ele pode mais complicar do que ajudar. Prefira usar o pseudo-código.

O laço do tipo "REPITA ATÉ" [4] é outro tipo especial de laço *WHILE*[5]. O Harbour **não tem** um laço desse tipo, mas ele pode ser facilmente obtido através do exemplo abaixo :

```
lCondicaoDeSaida := .f.
DO WHILE .T.  // Sempre é verdade (Sempre entra no laço)
```

[4] Esse laço é um velho conhecido dos programadores da linguagem PASCAL. O laço chama-se *REPEAT ... UNTIL*. Os programadores C tem também a sua própria versão no laço *do ... while*.

[5] Note que o laço *FOR* é o primeiro tipo especial de laço *WHILE*.

```
        ... Processamento
        IF lCondicaoDeSaida  // Condição de saída
            EXIT  // Sai
        ENDIF
    ENDDO
```

Esse laço difere dos demais porque ele **sempre** será executado **pelo menos** uma vez. Isso porque a sua condição de saída não está no início do laço, mas no seu final. No exemplo acima em algum ponto do processamento a variável **lCondicaoDeSaida** deve receber o valor verdadeiro para que o processamento saia do laço.

Dica 56

Em laços do tipo **REPITA ATÉ** você deve ter cuidado somente com a condição de saída do laço. A condição de entrada dele sempre é verdade.

Dica 57

Se você tem certeza de que o processamento do laço será executado no mínimo uma vez use a construção **REPITA ATÉ**.

9.2.5 Um pouco de prática

O problema do fatorial

Ao produto dos números naturais começando em n e decrescendo até 1 denominamos de fatorial de n e representamos por n!. Segundo tal definição, o fatorial de 5 é representado por 5! e lê-se 5 fatorial[6].

5! é igual a 5 . 4 . 3 . 2 . 1 que é igual a 120, assim como 4! é igual a 4 . 3 . 2 . 1 que é igual a 24, como 3! é igual a 3 . 2 . 1 que é igual a 6 e que 2! é igual a 2 . 1 que é igual a 2.

Por definição tanto 0!, quanto 1! são iguais a 1.

Para um fatorial genérico temos:

n! = n . (n - 1)! = n . (n - 1) . (n - 2)! = n . (n - 1) . (n - 2) . (n - 3) . ... . 1!

O programa da listagem 9.4 implementa um fatorial.

Listagem 9.4: Implementando um fatorial

```
/*
Programa que calcula o valor de N! (fatorial de N)
*/
PROCEDURE Main
LOCAL nFat, i , n

    INPUT "Informe o número : " TO n
    nFat := 1
    i := 2
    DO WHILE i <= n
        nFat := nFat * i
```

[6]Fonte: http://www.matematicadidatica.com.br/Fatorial.aspx - On line: 17-Out-2016

```
        i := i + 1
    ENDDO
   ? "O seu fatorial é : "
    ?? nFat

RETURN
```

Essa listagem está representada no apêndice A. Caso queira vê-la, vá para o apêndice A e busque a representação da listagem 9.4.

Prática número 26

Abra o arquivo pratica_loop01.prg na pasta “pratica” e implemente um fatorial conforme o código da listagem 9.4.

O problema do juro composto

O regime de juros compostos é o mais comum no sistema financeiro e portanto, o mais útil para cálculos de problemas do dia-a-dia. Os juros gerados a cada período são incorporados ao principal para o cálculo dos juros do período seguinte[7].

Chamamos de capitalização o momento em que os juros são incorporados ao principal.

Após três meses de capitalização, temos:

1º mês: M =P.(1 + i)

2º mês: o principal é igual ao montante do mês anterior: M = P x (1 + i) x (1 + i)

3º mês: o principal é igual ao montante do mês anterior: M = P x (1 + i) x (1 + i) x (1 + i)

Simplificando, obtemos a fórmula:

$M = P.(1 + i)^n$

Importante: a taxa i tem que ser expressa na mesma medida de tempo de n, ou seja, taxa de juros ao mês para n meses.

O código a seguir implementa um cálculo de juros compostos com períodos trimestrais.

Listagem 9.5: Juros compostos

```

/*
Cálculo de juros compostos
*/
PROCEDURE Main
LOCAL nCapital
LOCAL nMontante
LOCAL n
LOCAL nRendimento
LOCAL nTaxa
LOCAL nTrimestre
LOCAL x

    INPUT "Informe o capital : " TO nCapital
```

[7]Fonte: http://www.somatematica.com.br/emedio/finan3.php - On line: 17-Out-2016

```
    INPUT "Informe a taxa : " TO nTaxa
    INPUT "Informe os anos: " TO x
   nMontante := nCapital
   nTrimestre := 1
   n := 4 * x
    DO WHILE .T.
       /* Cálculo do rendimento trimestral */
        nRendimento := nTaxa * nMontante
        /* Cálculo do montante trimestral */
        nMontante := nCapital * ( 1 + nTaxa ) ^ nTrimestre
        /* Rendimento e montantes trimestrais */
        ? "Rendimento = " , nRendimento, " Montante = " , nMontante
        IF nTrimestre == n
            EXIT
        ENDIF
        // Modifique o valor do trimestre
        nTrimestre := nTrimestre + 1
    ENDDO
RETURN
```

Prática número 27

Abra o arquivo pratica_loop02.prg na pasta “pratica” e implemente o problema com juros compostos conforme o código da listagem 9.5.

9.3 Um tipo especial de laço controlado por contador

Os laços controlados por contador são muito usados por programadores de diversas linguagens. De tão usado, esse laço ganhou uma estrutura especial chamada de *FOR*. Quando a estrutura *FOR* começa a ser executada, a variável **nContador** é declarada e inicializada ao mesmo tempo. Abaixo segue um exemplo simples dessa estrutura :

```
FOR x := 1 TO 10
      ? x
NEXT
```

A variável contador (no caso o **x**) é automaticamente inicializada quando o laço é iniciado. Nos nossos códigos de exemplo essa variável será inicializada fora do laço através de *LOCAL*, conforme já frisamos nos capítulos anteriores. Ainda não podemos revelar a razão de inicializar com *LOCAL*, apenas podemos adiantar que é uma boa prática de programação e que, além disso, o seu código fica mais claro.

Listagem 9.6: Comparação entre os laços FOR e WHILE

```
/*
Estruturas de controle de fluxo
Comparação entre os laços FOR e WHILE
*/
```

```
PROCEDURE Main
LOCAL nContador

   // Processamento while
   ? "Processando com o laço WHILE"
   nContador := 1 // Incialização obrigatória
   DO WHILE ( nContador <= 10 )
      ? nContador
      ++nContador
   ENDDO
   ? "Após o laço WHILE o valor de nContador agora é " , nContador


   // Processamento
   ? "Processando com o laço FOR "
   FOR nContador := 1 TO 10
       ? nContador
   NEXT
   ? "Após o laço FOR o valor de nContador agora é " , nContador

RETURN
```

.:Resultado:.

```
Processando com o laço WHILE
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10
Após o laço WHILE o valor de nContador agora é          11
Processando com o laço FOR
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10
Após o laço FOR o valor de nContador agora é          11
```

A estrutura *FOR* ainda possui uma interessante variação, observe a seguir :

```
FOR x := 1 TO 10 STEP 2
      ? x
NEXT
```

A cláusula *STEP* da estrutura *FOR* determina a quantidade a ser mudada para cada iteração do loop. Esse valor pode ser positivo ou negativo. Se essa cláusula não for especificada o valor do incremento é de um para cada iteração.

Dica 58

Quando a situação exigir uma quantidade determinada de iterações dê preferência ao laço *FOR*.

Dica 59

Deitel e Deitel nos advertem : “evite colocar expressões cujos valores não mudam dentro de laços” [Deitel e Deitel 2001, p. 137]. Essa advertência tem o seguinte sentido : evite realizar cálculos cujo valor será sempre o mesmo dentro de laços. Por exemplo, se você colocar a expressão (12 * 43000) dentro de um laço que se repetirá mil vezes então essa expressão será avaliada mil vezes sem necessidade! Os autores acrescentam que “os modernos compiladores otimizados colocarão automaticamente tais expressões fora dos laços no código gerado em linguagem de máquina [...] porém, ainda é melhor escrever um bom código desde o início.”

O Harbour aceita um tipo especial de comentário na estrutura *FOR*. Esse comentário, caso você queira usá-lo, se restringe a **apenas uma palavra** após o *NEXT* da estrutura *FOR*. Veja um exemplo abaixo :

```
FOR x := 1 TO 10
      ? x
NEXT x
```

Se fosse...

```
NEXT final do processamento de x
```

...geraria um erro, por causa do espaço (tem que ser apenas uma palavra).

Essa forma de comentário foi inspirado nas primeiras versões de uma linguagem chamada BASIC e é usado por alguns programadores, embora não tenha se popularizado. A função desse comentário é indicar, no final do laço, qual foi a variável que foi incrementada no início do mesmo laço. Isso é útil quando o laço é grande demais e ocupa mais de uma tela de computador, ficando difícil para o programador saber qual é a variável que pertence ao inicio do seu respectivo laço. Por exemplo :

```
FOR x := 1 TO 10

      ... Várias linhas de código
```

```
        FOR y := 1 TO 100

              ... Várias linhas de código


        NEXT y

        ? x

NEXT x
```

Dica 60

Você pode se deparar com laços que são tão grandes que não cabem no espaço de uma tela no seu computador. Dessa forma, habitue-se a informar no final do laço qual foi a variável que iniciou esse laço ou algum comentário adicional. Utilize comentários (//) para informar a condição no final do laço. Por exemplo :

```
DO WHILE nNotas <= 100

     // Vários comandos

 ENDDO // Final do processamento de Notas  (nNotas)


 Ou


 FOR nNotas := 1 TO 100

       // Vários comandos

 NEXT  // Final do processamento de Notas (nNotas)
```

Se o laço for grande você deve comentar o seu final. Prefira usar comentários no final de cada laço a usar esses comentários especiais já citados. Por exemplo :

```
FOR x := 1 TO 10

    ... Muitas linhas de código

NEXT  // x

ou

NEXT // Um comentário sucinto

no lugar de

NEXT x  // <=== Evite esse formato
```

Dica 61

Não altere o conteúdo de uma variável contadora de um laço *FOR*. O código abaixo funciona mas pode confundir o programador, além de ser difícil de ser lido.

```
FOR x := 1 TO 100
      x := x + 1
NEXT
```

Dica 62

Kernighan nos adverte sobre a importância da consistência de um idioma:

> Assim como os idiomas naturais, as linguagens de programação tem idiomas, maneiras convencionais pelas quais os programadores experientes escrevem um código comum. Um aprendizado central de qualquer linguagem é o desenvolvimento da familiaridade com os seus idiomas. Um dos idiomas mais comuns é a forma de um loop [Kernighan e Pike 2000, p. 13]

A listagem 9.7 exibe um programa sintaticamente correto, mas confuso devido a inconsistência do idioma (*loop* fora do padrão).

Listagem 9.7: Programa confuso devido a forma não usual do uso dos seus laços.

```
/*
Consistência
*/
PROCEDURE Main
LOCAL nContador

   ? "Processando com o laço WHILE "
   nContador := 1
   DO ;
   WHILE ;
   nContador < 100
       ? nContador
       nContador += 15
   ENDDO
   ? "O valor após a saída é " , nContador

   FOR ;
   nContador ;
   := 1 ;
   TO;
   100
       ? nContador
   NEXT


RETURN
```

9.4 Os comandos especiais : EXIT e LOOP

O Harbour dispõe de comandos que só funcionam no interior de estruturas de repetição, são eles : EXIT e LOOP. Esses dois comandos funcionam somente no interior dos laços WHILE e FOR.

O comando EXIT força uma saída prematura do laço, e o comando LOOP desvia a

sequencia para o topo do laço.

Listagem 9.8: Saída prematura com EXIT.
Fonte: codigos/loop09.prg

```
/*
Desafio : Comando EXIT
*/
PROCEDURE Main
LOCAL nContador
LOCAL cResposta

   ? "Processando com o laço WHILE "
   nContador := 1
   DO WHILE nContador < 100
       ? nContador
       ACCEPT "Deseja contar mais um ? (S/N) " TO cResposta

   IF !( cResposta $ "Ss" )  //  cResposta não está contido em "Ss"
          EXIT
       ENDIF
       nContador += 1
   ENDDO
   ? "O valor após a saída é " , nContador

   ? "Processando com o laço FOR "
   FOR nContador := 1 TO 100
       ? nContador
       ACCEPT "Deseja contar mais um ? (S/N) " TO cResposta
       IF !( cResposta $ "Ss" ) // cResposta não está contido em "Ss"
          EXIT
       ENDIF
   NEXT
   ? "O valor após a saída é " , nContador

RETURN
```

Uma possível execução desse programa poderia exibir a tela a seguir:

.:Resultado:.

```
Processando com o laço WHILE
         1
Deseja contar mais um ? (S/N) s
         2
Deseja contar mais um ? (S/N) s
         3
Deseja contar mais um ? (S/N) s
         4
Deseja contar mais um ? (S/N) n
O valor após a saída é           4
Processando com o laço FOR
         1
```

```
Deseja contar mais um ? (S/N) s
         2
Deseja contar mais um ? (S/N) s
         3
Deseja contar mais um ? (S/N) s
         4
Deseja contar mais um ? (S/N) s
         5
Deseja contar mais um ? (S/N) n
O valor após a saída é           5
```

Listagem 9.9: Desvio para o topo do laço com LOOP.
Fonte: codigos/loop10.prg

```
/*
Desafio : Comando LOOP
*/
PROCEDURE Main
LOCAL cResposta
LOCAL nContador

   ? "Processando com o laço WHILE "
   DO WHILE .T.
      ACCEPT "Confirma a operaçäo ? (S/N) " TO cResposta

   IF !( cResposta $ "SsNn" ) // Leia cResposta não está contido em "SsNn"
         ? "Digite 'S' para sim ou 'N' para näo"
         LOOP
      ELSE
         EXIT
      ENDIF
   ENDDO

   ? "Processando com o laço FOR "
   FOR nContador := 1 TO 5
       ? nContador

   ACCEPT "Deseja prosseguir para a etapa seguinte (S/N) " TO cResposta
       IF ( cResposta $ "nN" ) // Leia cResposta está contido em "Nn"
         ? "Pulando sem executar o restante da operaçäo " , nContador
         LOOP
       ENDIF
       ? "RESTANTE DA OPERAÇÄO " , nContador
   NEXT
   ? "O valor após a saída é " , nContador

RETURN
```

Uma possível execução desse programa poderia exibir a tela a seguir:

.:Resultado:.

```
Processando com o laço WHILE
```

```
Confirma a operação ? (S/N) s
Processando com o laço FOR
         1
Deseja prosseguir para a etapa seguinte (S/N) n
Pulando sem executar o restante da operação          1
         2
Deseja prosseguir para a etapa seguinte (S/N) s
RESTANTE DA OPERAÇÃO          2
         3
Deseja prosseguir para a etapa seguinte (S/N) s
RESTANTE DA OPERAÇÃO          3
         4
Deseja prosseguir para a etapa seguinte (S/N) s
RESTANTE DA OPERAÇÃO          4
         5
Deseja prosseguir para a etapa seguinte (S/N) n
Pulando sem executar o restante da operação          5
O valor após a saída é           6
```

Dica 63

Os comandos EXIT e LOOP são usados frequentemente por programadores Harbour, mas você deve evitar o uso desses comandos caso você tenha a opção de colocar a condição de permanência no topo do laço (lembra-se da dica sobre idiomas ?) . Quando for usá-los não esqueça de usar a indentação e, se possível, colocar comentários adicionais.

Os comandos EXIT e LOOP possuem uma sutil diferença dentro de um laço [Papas e Murray 1994, p. 203]. O EXIT causa o final da execução do laço. Em contraste, o LOOP faz com que todos os comandos seguintes a ele sejam ignorados, **mas não evita o incremento da variável de controle de laço**. A listagem 9.10 ilustra essa situação.

Listagem 9.10: Diferenças entre EXIT e LOOP.
Fonte: codigos/loop13.prg

```
/*
Diferenças entre loop e continue
*/
PROCEDURE Main
LOCAL nCont
LOCAL cResp

   ? "LOOP"
   FOR nCont := 1 TO 10
      ? "O valor de nCont é " , nCont
      IF nCont == 5
         ? "Vou para o topo do laço, mas nCont será incrementada."
         LOOP
      ENDIF
   NEXT
   ? "O valor fora do laço é " , nCont // nCont vale 11
```

```
   ? "EXIT"
   FOR nCont := 1 TO 10
      ? "O valor de nCont é " , nCont
      IF nCont == 5
         ? "VOU SAIR IMEDIATAMENTE SEM INCREMENTAR nCont"
         EXIT
      ENDIF
   NEXT
   ? "O valor fora do laço é " , nCont // nCont vale 5

RETURN
```

Uma possível execução desse programa poderia exibir a tela a seguir:

.:Resultado:.

```
LOOP
O valor de nCont é           1
O valor de nCont é           2
O valor de nCont é           3
O valor de nCont é           4
O valor de nCont é           5
Vou para o topo do laço, mas nCont será incrementada.
O valor de nCont é           6
O valor de nCont é           7
O valor de nCont é           8
O valor de nCont é           9
O valor de nCont é          10
O valor fora do laço é          11
EXIT
O valor de nCont é           1
O valor de nCont é           2
O valor de nCont é           3
O valor de nCont é           4
O valor de nCont é           5
VOU SAIR IMEDIATAMENTE SEM INCREMENTAR nCont
O valor fora do laço é           5
```

Esse comportamento se explica da seguinte forma : o incremento de uma variável se dá no topo do laço (FOR), e não na base dele (NEXT). O topo do laço (FOR) tem três funções : inicializar a variável, atribuir valor de incremento a variável e verificar se a condição ainda é satisfeita. Se a condição não for satisfeita o laço é abandonado e a variável sempre terá um valor superior ao limite superior do laço FOR. O NEXT funciona como uma espécie de LOOP, ele só faz mandar para o topo do laço.

9.5 FOR EACH

O laço *FOR EACH* será visto parcialmente aqui porque ele depende do entendimento de estruturas de dados ainda não vistas. Esse laço é usado para percorrer todos os itens de :

1. Uma string
2. Um array
3. Um hash

Como ainda não vimos o que é array nem o que é um hash deixaremos os respectivos exemplos para quando abordarmos as estruturas de dados complexas da linguagem Harbour. Por enquanto vamos demonstrar o uso desse poderoso laço com as strings, através da listagem 9.11.

Listagem 9.11: Um exemplo de FOR EACH com strings
Fonte: codigos/loop11.prg

```
/*
For each
*/
PROCEDURE Main
LOCAL cCliente
LOCAL cEl

   ? "Processando com o laço FOREACH "
   ? "Ele pegará a string e processará letra por letra"
   ACCEPT "Informe seu nome : " TO cCliente
   FOR EACH cEl IN cCliente
      ? cEl
   NEXT


RETURN
```

.:Resultado:.

```
Processando com o laço FOREACH
Ele pegará a string e processará letra por letra
Informe seu nome : Harbour
H
a
r
b
o
u
r
```

A listagem 9.12 exibe a string de “trás para frente” com a cláusula *DESCEND*.

Listagem 9.12: Um exemplo de FOR EACH com strings (Reverso)
Fonte: codigos/loop15.prg

```
/*
For each
*/
PROCEDURE Main
LOCAL cNome
```

```
LOCAL cEl

   ? "Processando com o laço FOREACH "
   ? "Ele pegará a string e processará letra por letra"
   ? "mas irá exibir ao contrário"
   ACCEPT "Informe seu nome : " TO cNome
   FOR EACH cEl IN cNome DESCEND
      ? cEl
   NEXT


RETURN
```

.:Resultado:.

```
Processando com o laço FOREACH
Ele pegará a string e processará letra por letra
mas irá exibir ao contrário
Informe seu nome : Harbour
r
u
o
b
r
a
H
```

O laço FOR EACH também aceita um total de até 255 variáveis no seu processamento. É muito mais do que qualquer programador necessita. Confira na listagem 9.13 um exemplo com apenas duas variáveis.

Listagem 9.13: Um exemplo de FOR EACH com strings (Reverso)
Fonte: codigos/loop16.prg

```
/*
For each
*/
PROCEDURE Main
LOCAL cCliente , cSobrenome
LOCAL cEl , cEl2

   ? "Processando com o laço FOREACH "
   ? "Ele pegará a string e processará letra por letra"
   ACCEPT "Informe seu nome : " TO cCliente
   ACCEPT "Informe o seu sobrenome : " TO cSobreNome
   FOR EACH cEl, cEl2 IN cCliente , cSobreNome
      ? cEl, cEl2
   NEXT


RETURN
```

.:Resultado:.

```
Processando com o laço FOREACH
Ele pegará a string e processará letra por letra
Informe seu nome : Paulo César
Informe o seu sobrenome : Toledo
P T
a o
u l
l e
o d
  o
```

Note que o número de vezes que o FOR EACH irá iterar é determinado pela menor string. No nosso exemplo a menor string contém “Toledo”.

Dica 64

Sempre que você precisar processar o interior de uma string use o laço FOR EACH ou uma função de manipulação de strings. No capítulo referente a funções, você verá que o Harbour possui inúmeras funções prontas para o processamento de strings, a correta combinação desses dois recursos (o laço FOR EACH e as funções de manipulação de strings) tornarão os seus programas mais eficientes e mais fáceis de serem lidos.

9.6 Conclusão

Encerramos uma importante etapa no nosso aprendizado. As estruturas de seleção e de repetição constituem a base de qualquer linguagem de programação.

9.7 Desafios

9.7.1 Compile, execute e descubra

O programa da listagem 9.14 possui um erro de lógica. Veja se descobre qual é.

Listagem 9.14: Onde está o erro de lógica ?

```
/*
Desafio : Erro de lógica
*/

PROCEDURE Main
LOCAL nContador

   ? "Processando com o laço WHILE "
   nContador := 1
   DO WHILE nContador <> 100
       ? nContador
       nContador += 15
   ENDDO
   ? "O valor após a saída é " , nContador
```

```
                                                                    15
RETURN                                                              16
```

O programa da listagem 9.15 possui um erro de lógica. Veja se descobre qual é.

Listagem 9.15: Onde está o erro de lógica ?

```
/*                                                                  1
Desafio : Erro de logica                                            2
*/                                                                  3
PROCEDURE Main                                                      4
LOCAL cResposta                                                     5
                                                                    6
   ? "Processando com o laço WHILE "                                7
   ACCEPT "Confirma a operaçäo ? (S/N) " TO cResposta               8
   DO WHILE .T.                                                     9
                                                                    10
   IF !( cResposta $ "SsNn" ) // Leia cResposta não está contido em "SsNn"
         ? "Digite 'S' para sim ou 'N' para não"                    11
         LOOP                                                       12
      ELSE                                                          13
         EXIT                                                       14
      ENDIF                                                         15
   ENDDO                                                            16
                                                                    17
                                                                    18
RETURN                                                              19
```

9.7.2 Compile, execute e MELHORE

O programa da listagem 9.16 (adaptado de [Papas e Murray 1994, p. 204]) nos apresenta um joguinho no qual o usuário precisa adivinhar um número. Esse programa está funcionando sem erros de lógica. O desafio agora é melhorar esse programa. Eis aqui algumas dicas :

1. Quando o usuário informar o palpite, o programa deve informar se o mesmo está acima do valor ou abaixo do valor sorteado.
2. Se o valor que o usuário informar diferir do valor sorteado em menos ou mais de três unidades exiba a mensagem adicional : “Está quente!”.
3. Pense em outras formas de melhorar o programa. Por exemplo : os dois intervalor (10 e 40) podem ser informados por um segundo jogador. E por aí vai. O mais importante é “mecher” no código.

Listagem 9.16: Jogo da adivinhação

```
/*                                                                  1
Exemplos de loop e continue                                         2
Adaptado de PAPAS and MURRAY (1994)                                 3
                                                                    4
Esse código sorteia um número com a função HB_RAND                  5
```

```
                                                                    6
HB_RandomInt( 10 , 40 ) // Sorteia um número entre 10 e 40          7
                                                                    8
*/                                                                  9
PROCEDURE Main                                                      10
LOCAL nRand, nPalpite, nTentativa                                   11
LOCAL lSorte                                                        12
                                                                    13
   nRand := HB_RandomInt( 10 , 40 ) // Realiza o sorteio            14
   lSorte := .f. // Inicia lSorte com falso                         15
   nTentativa := 0                                                  16
                                                                    17
   ? "Tente adivinhar um número entre 10 e 40"                      18
   DO WHILE .NOT. lSorte // Faça enquanto não tiver sorte           19
      ++nTentativa                                                  20
      INPUT "Tente adivinhar meu número : " TO nPalpite             21
      IF nPalpite == nRand                                          22
         lSorte := .t.                                              23
      ELSE                                                          24
         LOOP                                                       25
      ENDIF                                                         26
      ? "Acertou! Você precisou de " , nTentativa , " tentativas"   27
   ENDDO                                                            28
                                                                    29
RETURN                                                              30
```

9.8 Exercícios de fixação

1. [Forbellone e Eberspacher 2005, p. 65] Construa um programa que leia um conjunto de dados contendo altura e sexo (“M” para masculino e “F” para feminino) de 10 pessoas e, depois, calcule e escreva :

   a) a maior e a menor altura do grupo

   b) a média de altura das mulheres

   c) o número de homens e a diferença porcentual entre eles e as mulheres

2. [Forbellone e Eberspacher 2005, p. 66] Anacleto tem 1,50 metro e cresce 2 centímetros por ano, enquanto Felisberto tem 1,10 metro e cresce 3 centímetros por ano. Construa um programa que calcule e imprima quantos anos serão necessários para que Felisberto seja maior do que Anacleto.

3. Adaptado de [Deitel e Deitel 2001, p. 173] Identifique e corrija os erros em cada um dos comandos (Dica: desenvolva pequenos programinhas de teste com eles para ajudar a descobrir os erros.) :

   a)
   ```
   nProduto := 10
   c := 1
   DO WHILE ( c <= 5 )
       nProduto *= c
   ```

```
ENDDO
++c
```

b)

```
x := 1
DO WHILE ( x <= 10 )
    x := x - 1
ENDDO
```

c) O código seguinte deve imprimir os valores de 1 a 10.

```
x := 1
DO WHILE ( x < 10 )
    ? x++
ENDDO
```

10 Funções

Depois de escalar uma grande montanha se descobre que existem muitas outras montanhas para escalar.

Nelson Mandela

Objetivos do capítulo

- Entender o que é uma função.
- Aprender a usar as funções.
- Aprender e usar algumas das principais funções do Harbour.

Uma função é um conjunto de instruções desenhadas para cumprir determinada tarefa e agrupadas em uma unidade com um nome para referí-las. [Mizrahi 2004, p. 94]. Nós usamos algumas funções até agora, por exemplo :

1. DATE() : Retorna a data de hoje, no formato data.
2. TIME() : Retorna uma string com a hora, minuto e segundos.
3. SQRT() : Retorna a raiz quadrada de um número.
4. UPPER() : Converte uma string para maiúscula.

Lembre-se : as funções são usadas somente para consultas. Eu não posso atribuir valores a uma função. A única coisa que eu posso é passar para elas argumentos e esperar um retorno. Por exemplo: a função SQRT exige que eu passe um número positivo e me retorna a raiz quadrada desse número.

O objetivo desse capítulo é apresentar a você algumas funções da linguagem Harbour. Todas as funções em Harbour (isso vale para as outras linguagens também) foram construídas em um momento posterior a criação da linguagem, disponibilizadas como código livre para download ou vendidas em pacotes extras. Outros programadores as desenvolveram, as testaram e, por fim, elas acabaram sendo distribuídas juntamente com a linguagem. De acordo com o criador da linguagem C, uma função não faz parte essencial de uma linguagem de programação [Kernighan e Ritchie 1986, p. 24]. A implicação prática dessa afirmação é : as funções podem ser criadas por outras pessoas (inclusive por você) e adicionadas ao código que você está trabalhando. Esse capítulo irá lhe apresentar algumas funções já existentes no Harbour, em um capítulo posterior nós veremos como criar as nossas próprias funções.

10.1 A anatomia de uma função

Uma função é composta basicamente por :

1. Um nome
2. Um par de parenteses
3. Uma lista de valores de entrada (separados por vírgulas)
4. Um retorno (um tipo de dado)

Nem todas as funções apresentam os quatro ítens enumerados acima, você verá que, no mínimo, uma função é composta por um nome e um par de parênteses. Além disso, não basta ter os elementos corretos, além disso, eles precisam estar em uma ordem sintática correta. Por exemplo, a seguinte construção sintática está errada :

```
TIME() := "10:30:45"
```

Uma função portanto, não é uma variável. Uma função somente pode ser executada[1] e o seu valor de retorno (um tipo de dado) pode ser armazenado em uma variável se você desejar. Os exemplos abaixo estão corretos :

1. Apenas exibo a hora atual com o comando "?".

```
? TIME()
```

2. A hora foi armazenada em uma variável para uso posterior (ela não foi exibida).

```
cInstante := TIME()
```

3. Não está errado, mas não surte efeito algum. O dado retornado não está sendo armazenado e nem exibido.

```
TIME()
```

Apenas digitar TIME() não irá surtir efeito algum (nem causa erro de execução). A função TIME() só faz retornar um valor (a hora, minuto e segundo correntes) mas não o exibe. Você precisa de um comando extra para exibir esse valor ou uma variável para guardá-la.

Em muitos casos você necessita passar um “valor” para uma função (o nome técnico para esse “valor” é argumento [2]). Quase sempre uma função retorna um valor que resulta do tipo de processamento que ela executa (esse outro “valor” recebe o nome de “valor de retorno da função” [3]).

Observe o código da listagem 10.1 a seguir, por exemplo :

Listagem 10.1: Exemplo de funções com parâmetros.
Fonte: codigos/funcao03.prg

```
/*
Exemplo de funções com parâmetros.
*/
PROCEDURE Main

   ? Int( 12.456 ) // Retorna 12
   ? Year( Date() ) // Retorna o ano da data que foi passada

RETURN
```

O exemplo abaixo exemplifica a execução do código da listagem 10.1 em algum dia do ano de 2016.

.:Resultado:.

```
        12
 2016
```

[1] Também usamos a expressão “chamar” para indicar que uma função foi executada. Por exemplo: “a função TIME() foi chamada na linha 176 do código de alteração de notas fiscais”.

[2] Você verá em outros textos técnicos a palavra “parâmetro”. Existe uma sutil diferença entre parâmetro e argumento, nos capítulos seguintes nós iremos esclarecer o significado dessas expressões.

[3] É comum, em ambientes de trabalho e fóruns, expressões como “a função *CalculaSalario* **retornou** um valor muito alto” ou “A função *SomaProduto* **retornou** zero.”.

Nesse exemplo, a função INT recebe como argumento um valor (no exemplo esse valor é 12.456, através da variável x) e retorna a sua parte inteira (12).

Note que uma função pode receber como argumento uma outra função no lugar de um valor constante. Na referida listagem 10.1 a função YEAR(), recebe uma data gerada pela função DATE(). Como isso é possível ?

Cada função aceita determinado tipo de parâmetro. Por exemplo a função YEAR() recebe um tipo de dado data como entrada. Se você passar um número ou um caractere para essa função um erro de execução será gerado.

Da mesma forma, cada função retorna um valor de um tipo particular de dado. A função DATE() não recebe entradas mas sempre retorna um valor do tipo data. Por isso a construção YEAR(DATE()) é válida, já a construção YEAR(“José”) é inválido e gera um erro de execução.

10.2 Funções por categoria

A única forma de se aprender sobre funções é praticando através da criação de códigos. Portanto, a próxima seção possui vários códigos para digitação. É essencial que você não se limite apenas a digitar e executar os códigos, mas que procure realizar algumas modificações.

As funções foram agrupadas por tipo [4], conforme a lista abaixo :

1. Funções de manipulação de *strings*
2. Funções de data e hora
3. Funções matemáticas
4. Função de conversão

Essa lista não é exaustiva. Existem inúmeras categorias de funções, por exemplo: acesso a banco de dados, arquivos, financeiras, estatísticas, etc. O objetivo desse capítulo é apenas apresentar-lhe as funções através de exemplos.

10.2.1 Funções de manipulação de *strings*

1. ALLTRIM(<cTexto>)

   - Descrição : Remove os espaços em branco em torno da string.
   - Entrada : <cTexto> - Uma string qualquer.
   - Saída : A mesma string sem os espaços em branco da direita e da esquerda.

Veja um exemplo de uso da função ALLTRIM na listagem 10.2

Listagem 10.2: ALLTRIM.
Fonte: codigos/alltrim.prg

```
/*                                                                  1
Uso de alltrim                                                      2
```

[4] Essa classificação foi adaptada de [Ramalho 1991] e as descrições foram adaptadas de [Nantucket 1990]

```
*/
PROCEDURE Main
LOCAL cNome , cSobreNome

   cNome := "     José          "
   cSobreNome := "       Quintas      "

   ? "Concatenando o nome e o sobrenome sem ALLTRIM"
   ? cNome + " " + cSobreNome

   ? "Agora concatenando com ALLTRIM"
   cNome := AllTrim( cNome )
   cSobreNome := AllTrim( cSobreNome )
   ? cNome + " " + cSobreNome

RETURN
```

.:Resultado:.

```
Concatenando o nome e o sobrenome sem ALLTRIM
     José                 Quintas
Agora concatenando com ALLTRIM
José Quintas
```

2. LEN(<cTexto>)

   - Descrição : Retorna o tamanho do texto ou do array[5] especificado.
   - Entrada : <cTexto> - Uma string qualquer ou um array.
   - Saída : Um valor numérico inteiro representando o tamanho da string ou do array.

3. UPPER(<cTexto>)

   - Descrição : Retorna o texto fornecido totalmente em letras maiúsculas.
   - Entrada : <cTexto> - Uma string qualquer.
   - Saída : A string escrita totalmente em letras maiúsculas.

4. LOWER(<cTexto>)

   - Descrição : Retorna o texto fornecido totalmente em letras minúsculas.
   - Entrada : <cTexto> - Uma string qualquer.
   - Saída : A string escrita totalmente em letras minúsculas.

   Na listagem a seguir (listagem 10.3) temos um exemplo de uso das funções LEN, UPPER e LOWER.

[5] Ainda veremos esse conceito em um capítulo a parte, os exemplos abordados nesse capítulo abrangem somente strings

Listagem 10.3: LEN, UPPER e LOWER.
Fonte: codigos/funcao05.prg

```
/*
Exemplo de funções com parâmetros.
*/
PROCEDURE Main
LOCAL cNome

   ACCEPT "Digite o seu nome : " TO cNome
   ? "O seu nome possui " , Len( cNome ) , " letras"
   ? "O seu nome com letras maiúsculas se escreve assim : " ,;
                                                  Upper( cNome )
   ? "O seu nome com letras minúsculas se escreve assim : " ,;
                                                  Lower( cNome )

RETURN
```

.:Resultado:.

```
Digite o seu nome : Eolo
O seu nome possui           4  letras
O seu nome com letras maiúsculas se escreve assim :  EOLO
O seu nome com letras minúsculas se escreve assim :  eolo
```

5. LEFT(<cTexto> , <nQtd>)

   - Descrição : Retorna <nQtd> caracteres a partir do primeiro caractere da esquerda.
   - Entrada :
     - <cTexto> - Uma string qualquer.
     - <nQtd> - Quantidade de caracteres que devem ser retornados a partir da esquerda.
   - Saída : A string de entrada escrita somente com os <nQtd> caracteres iniciais.

6. RIGHT(<cTexto> , <nQtd>)

   - Descrição : Retorna <nQtd> caracteres a partir do último caractere da esquerda.
   - Entrada :
     - <cTexto> - Uma string qualquer.
     - <nQtd> - Quantidade de caracteres que devem ser retornados a partir da direita.
   - Saída : A string de entrada escrita somente com os <nQtd> caracteres finais.

7. SUBSTR(<cTexto> , <nInicio> [, <nTamanho>]) [6]

[6] O nome SUBSTR significa “substring”, ou seja, um subconjunto da *string* original.

- Descrição : Retorna os “n” caracteres a partir da posição <nInicio>. O tamanho da string de retorno é dada por [nTamanho]. Se [nTamanho] não for especificado, então retorna todos os caracteres de <nInicio> até o final da *string*.
- Entrada :
  - <cTexto> - Uma string qualquer.
  - <nInicio> - A posição inicial onde a *substring* inicia.
  - [nTamanho] - O tamanho da *substring*.
- Saída : <cTextoRetorno> - A string de entrada escrita somente com os <nQtd> caracteres iniciais.

A função SUBSTR, como já vimos, possui um parâmetro opcional. Esse parâmetro, se for omitido recebe da própria função um valor que é dado por padrão. Chamamos esse valor padrão de valor *default*. A listagem 10.4 exemplifica o uso das funções LEFT, RIGHT e SUBSTR.

Listagem 10.4: LEFT, RIGHT e SUBSTR.
Fonte: codigos/funcao06.prg

```
PROCEDURE Main
LOCAL cNome

   hb_cdpSelect("UTF8")

   ACCEPT "Digite um nome (mánimo 9 letras) : " TO cNome
   ? "Os três primeiros caracteres de seu nome : " ,;
                                        Left( cNome , 3 )
   ? "Os três últimos caracteres de seu nome : " ,;
                                        Right( cNome , 3 )
   ? "Um trecho iniciando na 3a letra e "
   ?? "finalizando na 5a letra : " , SubStr( cNome , 3 , 2 )

   ? "Um trecho iniciando na 3a letra até o final : " ,;
                                     SubStr( cNome , 3 )
RETURN
```

.:Resultado:.

```
Digite um nome (mínimo 9 letras) : Roberto Borges
Os três primeiros caracteres de seu nome :  Rob
Os três últimos caracteres de seu nome :  ges
Um trecho iniciando na 3a letra e finalizando na 5a letra :  be
Um trecho iniciando na 3a letra até o final :  berto Borges
```

8. REPLICATE(<cTexto> , <nQtd>)

   - Descrição : Gera cópias sequenciais da *string* <cTexto>. A quantidade de cópias é definida por <nQtd>.
   - Entrada :

  - <cTexto> - Uma string qualquer.
  - <nQtd> - Quantidade de cópias.
- Saída : A string de entrada <cTexto> replicada <nQtd> vezes.

O nome de uma função geralmente diz muito sobre ela, no caso específico de REPLICATE (Listagem 10.5) ela cria varias cópias (replicações) de uma mesma *string*.

Listagem 10.5: A função REPLICATE.
Fonte: codigos/funcao04.prg

```
/*
Exemplo de funções com dois parâmetros.
*/
PROCEDURE Main

   ? Replicate( "A" , 10 ) // Retorna 12 vezes a string "A"
   ? Replicate( "Oi ", 20 )  // Retorna 20 vezes a string "Oi "

RETURN
```

.:Resultado:.

```
AAAAAAAAAA
Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi Oi
```

10.2.2 Funções de manipulação de datas

Já vimos como trabalhar com as funções DATE , TIME e YEAR nos capítulos anteriores, nessa seção veremos mais algumas funções de manipulação de datas com os respectivos exemplos.

1. MONTH(<dData>)
   - Descrição : Retorna um número representando o mês.
   - Entrada : <dData> - Uma data válida.
   - Saída : O número do mês (1 = Janeiro , ... , 12 = Dezembro).
2. CMONTH(<dData>)
   - Descrição : Retorna o nome representando o mês.
   - Entrada : <dData> - Uma data válida.
   - Saída : O nome do mês.
3. DOW(<dData>)
   - Descrição : Retorna um número representando o dia da semana.
   - Entrada : <dData> - Uma data válida.
   - Saída : O número do mês (1 = Janeiro , ... , 12 = Dezembro).

4. CDOW(<dData>)

    - Descrição : Retorna o nome do dia da semana.
    - Entrada : <dData> - Uma data válida.
    - Saída : O nome do dia da semana (Segunda-feira, Terça-feira, etc.).

A listagem 10.6 mostra alguns exemplos com funções de manipulação de data. **Uma novidade aqui** é o uso das instruções **REQUEST HB_LANG_PT** e da função **hb_langSelect("PT")**. Essas instruções ativam o suporte do Harbour a lingua portuguesa, dessa forma o nome do dia da semana e o nome do mês aparecerão em português. Caso você omita essas instruções, os nomes aparecerão em inglês.

Listagem 10.6: Exemplo de funções que manipulam datas.
Fonte: codigos/funcao07.prg

```
/*
Exemplo de funções
*/
REQUEST HB_LANG_PT // Disponibiliza suporte a lingua portuguesa

PROCEDURE Main
LOCAL dData AS DATE

   hb_cdpSelect("UTF8") // Acentuação em UTF-8
   hb_langSelect( "PT" ) // Suporte a lingua portuguesa nas mensagens
   SET DATE BRITISH // Data no padrão dia/mês/ano

   dData := CToD( "26/08/2016" )
   ? "O número do mês correspondente é : " , Month( dData )
   ? "O nome do mês é " , CMonth( dData )
   ? "O número correspondente ao dia da semana é : " , DoW( dData )
   ? "O dia da semana é " , CDow( dData )
   ? "O dia do mês correspondente é " , Day( dData )
   ? "O ano é " , Year( dData )

RETURN
```

.:Resultado:.

```
O número do mês correspondente é :     8
O nome do mês é  Agosto
O número correspondente ao dia da semana é :     6
O dia da semana é  Sexta-feira
O dia do mês correspondente é   26
O ano é   2016
```

10.2.3 Funções matemáticas

1. ABS(<nNum>)

    - Descrição : Retorna o valor absoluto de um número.

- Entrada : <nNum> - Um número válido.
- Saída : O valor absoluto de <nNum>.

A listagem 10.7 ilustra o uso da função ABS. Observe também que nós usamos um laço WHILE na forma “REPITA ATÉ” para controlar a entrada de dados válidos.

Listagem 10.7: Exemplo da funções ABS.
Fonte: codigos/funcao08.prg

```
/*
Exemplo de funções
*/
PROCEDURE Main
LOCAL nNum

   DO WHILE .T.   // "REPITA ATÉ" (Sempre entra no laço)
      INPUT "Digite um número negativo : " TO nNum
      IF nNum < 0  // A condição de saída está aqui.
         EXIT
      ENDIF
   ENDDO
   ? "O valor absoluto de ", nNum , " é " , Abs( nNum )

RETURN
```

.:Resultado:.

```
Digite um número negativo : -10
O valor absoluto de         -10  é          10
```

2. MAX(<nNum1> , <nNum2>)

   - Descrição : Retorna o maior valor entre duas expressões.
   - Entrada : <nNum1> e <nNum2> - Dois números (ou expressões numéricas) para serem comparados.
   - Saída : O maior valor dentre <nNum1> e <nNum2>.

O código da listagem 10.8 exemplifica o uso da função MAX. O Harbour também possui a função MIN, que faz o contrário de MAX[7].

Listagem 10.8: Exemplo das funções MAX e MIN.
Fonte: codigos/funcao09.prg

```
/*
Exemplo de funções
*/
PROCEDURE Main
LOCAL nNum1, nNum2

```

[7] Veja como o nome de uma função é importante. Nem precisaremos detalhar o uso da função MIN. Quando você for criar as suas próprias funções procure nomes intuitivos para elas.

```
   INPUT "Digite um número qualquer : " TO nNum1
   INPUT "Digite agora outro número : " TO nNum2
   ? "O maior número é ", Max( nNum1, nNum2 )
   ? "O menor número é ", Min( nNum1, nNum2 )


RETURN
```

.:Resultado:.

```
Digite um número qualquer : 100
Digite agora outro número : 90
O maior número é         100
O menor número é          90
```

3. ROUND(<nNum> , <nCasasDecimais>)

   - Descrição : Arredonda um valor.
   - Entrada :
     - <nNum> - O valor a ser arredondado.
     - <nCasasDecimais> - A quantidade de casas decimais para o arredondamento
   - Saída : O valor de entrada arredondado.

A listagem 10.9 ajuda a entender o funcionamento da função ROUND. Mas se você digitar um valor do tipo 150.567 talvez você não compreenda o sentido dela completamente. Experimente digitar o seguinte valor : 153.4989 e note que quando a função ROUND tem um argumento negativo ela “arredonda” até a parte inteira da casa das unidades. Se o valor do argumento fosse -2 então ela “arredondaria” até o valor da casa das centenas.

Listagem 10.9: Exemplo da função ROUND.
Fonte: codigos/funcao10.prg

```
/*
Exemplo de funções
*/
PROCEDURE Main
LOCAL nNum

   DO WHILE .T.
      INPUT "Digite um número decimal qualquer  : " TO nNum
      IF INT( nNum ) != nNum
         EXIT
      ENDIF
   ENDDO
   ? "ROUND( nNum , 3 ) ==> ", Round( nNum , 3 )
   ? "ROUND( nNum , 2 ) ==> ", Round( nNum , 2 )
   ? "ROUND( nNum , 1 ) ==> ", Round( nNum , 1 )
   ? "ROUND( nNum , 0 ) ==> ", Round( nNum , 0 )
```

```
   ? "ROUND( nNum , -1) ==> ", Round( nNum , -1 )

RETURN
```

.:Resultado:.

```
Digite um número decimal qualquer  : 45.9801
ROUND( nNum , 3 ) ==>          45.980
ROUND( nNum , 2 ) ==>          45.98
ROUND( nNum , 1 ) ==>          46.0
ROUND( nNum , 0 ) ==>          46
ROUND( nNum , -1) ==>          50
```

Agora que já compreendemos como funciona o arredondamento na linguagem Harbour, vamos fazer uma pequena pausa para abordarmos alguns problemas decorrentes de arredondamentos e de diferenças. Esse problema está presente em outras linguagens também. Leia com atenção as três dicas a seguir.

As regras de arredondamento

As regras de arredondamento obedecem às normas abaixo :

- Se o valor da casa a ser arredondada for igual ou superior a cinco, então arredonda para a unidade imediatamente superior. Por ex: ROUND(1.55 , 1) vai dar 1.6;
- Se o valor da casa a ser arredondada for inferior a cinco, então não altera a casa decimal imediatamente superior. Por ex: ROUND(1.49 , 1) vai dar 1.4.

Cuidado quando for trabalhar com números e arredondamentos. Para ilustrar, pegue o exemplo da listagem 10.9 e entre com o valor 1.5, depois execute de novo e entre com o valor 1.49. Os valores serão exibidos conforme abaixo :

Digite 1.5 O programa retorna :

.:Resultado:.

```
ROUND( nNum , 3 ) ==>          1.500
ROUND( nNum , 2 ) ==>          1.50
ROUND( nNum , 1 ) ==>          1.5
ROUND( nNum , 0 ) ==>          2   <==========
ROUND( nNum , -1) ==>          0
```

Depois digite 1.49 E o programa retorna :

.:Resultado:.

```
ROUND( nNum , 3 ) ==>          1.490
ROUND( nNum , 2 ) ==>          1.49
ROUND( nNum , 1 ) ==>          1.5
ROUND( nNum , 0 ) ==>          1   <===========
ROUND( nNum , -1) ==>          0
```

Note que esses exemplos obedecem a norma matemática citada no início desse bloco. Porém algum usuário poderia questionar porque de 1.5 passou para 2 no primeiro caso (grifado com seta) e de 1.5 passou para 1 no segundo caso (grifado com seta). É fácil ver que o problema não está no arredondamento, mas sim na exibição dos números. No segundo caso, você não está arredondando 1.5 , mas sim 1.49 !

Soma e depois arredonda ou arredonda e depois soma ? Faz diferença ?

Sim, faz toda a diferença. Sistemas de controle financeiro ou sistemas que trabalham com medidas bem exatas requerem um controle total sobre as operações que envolvem arredondamento. O programa da listagem 10.10 ilustra esse problema :

Listagem 10.10: Soma e arredonda ou Arredonda e soma ?
Fonte: codigos/round01.prg

```
PROCEDURE Main
LOCAL nNum1, nNum2, nNum3, nNum4

   nNum1 := ( 100 / 3 )
   nNum2 := ( 1000 / 33 )
   nNum3 := ( 500 / 1.5 )
   nNum4 := ( 101 / 3 )

   ? "Somo e depois arredondo"
   ? Round( nNum1 + nNum2 + nNum3 + nNum4 , 2 )
   ? "Arredondo e depois somo"
   ? Round( nNum1 , 2 ) + ;
     Round( nNum2 , 2 ) + ;
     Round( nNum3 , 2 ) + ;
     Round( nNum4 , 2 )

RETURN
```

.:Resultado:.

```
Somo e depois arredondo
       430.64
Arredondo e depois somo
       430.63
```

A solução imediata para esse problema é adotar um desses dois padrões nos seus programas, nunca os dois padrões [8] .

Problema clássico de arredondamento

Problemas com cálculos matemáticos não são exclusivos da linguagem Harbour. Todas as linguagens possuem essa limitação, que deve ser conhecida pelo programador.

[8] Essa questão tem levado a muitas soluções (todas boas) nos fóruns, você pode visitar `http://pctoledo.com.br/forum/viewtopic.php?f=4&t=16568` **ou** `http://www.pctoledo.com.br/forum/viewtopic.php?f=4&t=16603&p=100122&hilit=round#p100122` para maiores informações.

A seguinte listagem 10.11 foi adaptada para o Harbour de um fonte escrito em C++ (O fonte original apresenta o mesmo comportamento e foi criado para ilustrá-lo).

Listagem 10.11: Problemas com somas e diferenças.
Fonte: codigos/arredondamento.prg

```
/*
Erros de arredondamento
Adaptado de Horstman, p.55
*/
PROCEDURE Main
LOCAL nPreOri := 300000000000000 // Preço original
LOCAL nPreDes := nPreOri - 0.05 // Preço com desconto
LOCAL nDesconto := nPreOri - nPreDes // Desconto calculado

   ? "O valor esperado do desconto  :  0.05"
   ? "O valor calculado do desconto : ", nDesconto

RETURN
```

.:Resultado:.

```
O valor esperado do desconto  : 0.05
O valor calculado do desconto :          0.06
```

Portanto, tenha cuidado com somas gigantescas e pequenas diferenças advindas desses valores.

10.2.4 Funções de conversão

Em todos os nossos exemplos temos usados comandos simples, porém poderosos, para a exibição de dados. Basicamente usamos sempre o comando “?”, algumas poucas vezes o “??”. Com esse comando nós podemos exibir vários dados de tipos diferentes em apenas uma linha sem problemas, basta separá-los com vírgulas.

```
   ?  "O valor esperado é " , 5
```

.:Resultado:.

```
O valor esperado é 5
```

```
   ?  "Hoje é " , DATE()
```

.:Resultado:.

```
Hoje é 13/08/2016
```

Porém a linguagem Harbour não admite a concatenação direta de tipos de dados diferentes. A facilidade na hora de exibir na mesma linha se deve a versatilidade do comando "?" e não a característica da linguagem em si. Essa característica está presente em outras linguagens também, com algumas poucas exceções. O que acontece, então quando nós queremos concatenar um tipo de dado com outro ? A resposta é : transforme tudo em tipo de dado caractere para poder efetuar a concatenação [9]. Essa transformação é feita com o auxílio de funções de conversão de dados. Nas próximas subseções nós veremos o uso das mais usadas.

Convertendo um tipo caractere para numérico

Use a função VAL para converter um tipo de dado caractere para um tipo numérico. A sintaxe da função VAL está descrita a seguir :

VAL(<cString>)

- Descrição : Converte um texto numérico em caractere
- Entrada : <cString> - Valor caractere (Ex : "1200.00").
- Saída : O mesmo valor convertido para o tipo numérico.

A listagem 10.12 nos mostra uma conversão de um tipo caractere para um tipo numérico. Na linha 19 tentamos fazer essa mesma conversão sem a função VAL e obtemos um erro de execução.

Listagem 10.12: Conversão de caractere para numérico.
Fonte: codigos/convert01.prg

```
1
2  PROCEDURE Main
3  LOCAL cNota
4  LOCAL nNota
5
6     cNota := "1200.00"  // Uma variável caractere
7     nNota := 300
8
9     ? nNota + Val( cNota )
10    ? "Deu certo, mas a próxima vai gerar um erro de execução..."
11    ? nNota + cNota
12
13 RETURN
```

.:Resultado:.

```
      1500.00
Deu certo, mas a próxima vai gerar um erro de execução...
Error BASE/1081  Argument error: +
Called from MAIN(19)
```

[9] Inclusive, o termo "concatenação", nós já vimos, é um termo restrito a *strings*.

Convertendo um tipo numérico para caractere

Use a função STR para converter um tipo de dado numérico para um tipo caractere. A sintaxe da função STR está descrita a seguir :

STR(<nNumero> [, <nTamanho> [, <nCasasDecimais>]])

- Descrição : Converte um texto caractere em numérico
- Entrada :
  - <nNumero> - Valor numérico (Ex : 1200.00).
  - [nTamanho] - Tamanho da *string*
  - [nDecimais] - Casas decimais
- Saída : O mesmo valor convertido para o tipo caractere.

A listagem 10.13 nos mostra uma conversão de um tipo numérico para um tipo caractere.

Listagem 10.13: Conversão de numérico para caractere.
Fonte: codigos/convert02.prg

```
/*
Exemplo de funções
*/
PROCEDURE Main
LOCAL nTotal

   nTotal := 1300.456

   ? "O total da venda foi de " + Str( nTotal )
   ? "O total da venda foi de " + Str( nTotal , 10 )
   ? "O total da venda foi de " + Str( nTotal , 5 , 2)
   ? "O total da venda foi de " + Str( nTotal , 3 , 2)
   ? "O total da venda foi de " + HB_ValToStr( nTotal )
   ? "Deu certo até agora, mas a próxima vai gerar um erro."
   ? "O total da venda foi de " + nTotal

RETURN
```

.:Resultado:.

```
O total da venda foi de       1300.456
O total da venda foi de       1300
O total da venda foi de *****
O total da venda foi de ***
O total da venda foi de       1300.456
Deu certo até agora, mas a próxima vai gerar um erro.
Error BASE/1081  Argument error: +
Called from MAIN(21)
```

Note que a função STR tem dois parâmetros opcionais, todos os dois dizem respeito ao tamanho da string resultante. Note também, que se os parâmetros possuírem um valor inferior ao tamanho da string resultante STR exibirá asteriscos no lugar do número, como se houvesse um “estouro” no tamanho. Esse comportamento não é nenhum erro da linguagem. É mais ou menos como se a função dissesse : “Eu não posso exibir a expressão com o espaço que você está me dando, então eu vou exibir asteriscos para lhe avisar que o valor existe, mas ele é superior ao espaço que foi reservado para ele”. As regras do espaço de exibição obedecem a duas regrinhas simples :

1. O primeiro valor numérico (<nTamanho>) é o valor espaço total (incluindo os decimais (<nCasasDecimais>)). Ex: 10000.00 possui <nTamanho> igual a sete e <nCasasDecimais> igual a 2. Note que o ponto de separação de decimal não entra na contagem.
2. O segundo valor numérico (<nCasasDecimais>) é o valor espaço reservado para os decimais

Dica 65

Tenha cuidado ao reservar espaço para um número exibido pela função STR. Se você reservar um espaço muito pequeno então asteriscos serão exibidos. A seguir temos alguns exemplos de números e os valores dos dois parâmetros opcionais de STR.

- 1000.00
  - O tamanho total é seis, sendo dois para as casas decimais.
  - O valor mínimo do tamanho resulta em STR(1000.00 , 6 , 2).
- 20000
  - O tamanho total é cinco, sem casas decimais.
  - O valor mínimo do tamanho resulta em STR(20000 , 5)
  - se você usar STR(20000 , 5 , 1) o seu programa irá exibir asteriscos pois a parte inteira agora só tem quatro de espaços disponíveis.

Dica 66

Para converter de um valor numérico para um valor caractere, sem se preocupar em calcular o espaço para a exibição, use a combinação de STR com ALLTRIM.

```
nVal := 10000.00
? ``O valor final é `` + ALLTRIM( STR( nVal ) )
```

.:Resultado:.

```
O valor final é 10000.00
```

Você deve ter notado também, na listagem 10.13, a existência de uma função chamada hb_ValTOStr. Essa função é uma implementação nova do Harbour e converte qualquer tipo de dado para caractere. É tentador usar essa função para todos os casos, porém é interessante que você aprenda as outras funções que convertem um tipo de dado específico para o tipo caractere, pois essas funções são usadas com muita frequência.

A seguir um exemplo [10] usando hb_ValToStr para converter diversos tipos de dados para caractere.

```
Set( _SET_DATEFORMAT, "yyyy-mm-dd" )                                    1
? hb_ValToStr( 4 ) == "         4"                                      2
? hb_ValToStr( 4.0 / 2 ) == "          2.00"                            3
? hb_ValToStr( "String" ) == "String"                                   4
? hb_ValToStr( hb_SToD( "20010101" ) ) == "2001-01-01"                  5
? hb_ValToStr( NIL ) == "NIL"                                           6
? hb_ValToStr( .F. ) == ".F."                                           7
? hb_ValToStr( .T. ) == ".T."                                           8
```

STRZERO(<nNumero> [, <nTamanho> [, <nCasasDecimais>]])

Essa função funciona da mesma maneira que a função STR, com a diferença de substituir os espaços em branco iniciais por zeros.

A listagem 10.14 nos mostra uma conversão de um tipo numérico para um tipo caractere usando a função STRZERO.

Listagem 10.14: Conversão de numérico para caractere com STRZERO.
Fonte: codigos/convert03.prg

```
/*                                                                      1
Exemplo de funções                                                      2
*/                                                                      3
PROCEDURE Main                                                          4
LOCAL nTotal                                                            5
                                                                        6
   nTotal := 1300.456                                                   7
                                                                        8
   ? "O total da venda foi de " + StrZero( nTotal )                     9
   ? "O total da venda foi de " + StrZero( nTotal , 10 , 2 )            10
                                                                        11
RETURN                                                                  12
```

.:Resultado:.

```
O total da venda foi de 0000001300.456
O total da venda foi de 0001300.46
```

Convertendo um tipo data para caractere

DTOC(<dData>)

[10] Retirado de http://www.gtxbase.org/index.php/en/harbour-project-en/43-hrg-es-es/hrg-en-api/strings/299-hb-en-hb-valtostr em 13-Ago-2016

- Descrição : Converte uma data para caractere.
- Entrada : <dData> - Uma data válida.
- Saída : O mesmo valor convertido para o tipo caractere.

Não confunda com nossa conhecida função CTOD(<cData>), que faz o oposto.
A listagem 10.13 nos mostra uma conversão de um tipo data para um tipo caractere.

Listagem 10.15: Conversão de data para caractere.
Fonte: codigos/convert04.prg

```
/*
Conversão de data para caractere
*/
PROCEDURE Main
LOCAL cData

   SET DATE BRITISH
   cData := DToC( Date() )

   ? "A data de hoje é " + cData

RETURN
```

Tabela resumo das funções de conversão

Tabela 10.1: Tabela de conversao

De/Para	Caractere	Data	Lógico	Numérico
CARACTERE		CTOD(<c >)	<c >$"Tt"	Val(<c >)
DATA	DTOC(<d >)	–	–	–
LÓGICO	IIF(<l >, "Verdadeiro", "Falso")	–	–	–
NUMÉRICO	STR(<n >)	–	–	–

10.3 Conclusão

Abordamos nesse capítulo algumas das principais funções usadas pelo Harbour, mas elas não são as únicas. Uma excelente documentação está sendo desenvolvida e pode ser acessada on-line em `https://harbour.github.io/doc/` e em `https://harbour.github.io/`.

Dica 67

O Harbour possui muitas funções, mas é praticamente impossível decorar a chamada de todas elas. Uma ajuda pode ser encontrada no aplicativo hbmk2, que é usado para gerar executáveis. Caso você tenha se esquecido da grafia de uma função em particular pode digitar *hbmk2 -find <busca>* para poder pesquisar, Por exemplo, se você quer saber todas as funções que tem "valto" no seu nome poderá fazer :

.:Resultado:.

```
> hbmk2 -find valto
xhb.hbc (instalado):
   ValToArray()
   ValToBlock()
   ValToCharacter()
   ValToDate()
   ValToHash()
   ValToLogical()
   ValToNumber()
   ValToObject()
   ValToPrg()
   ValToPrgExp()
   ValToType()
Núcleo Harbour (instalado):
   hb_ValToExp()
   hb_ValToStr()
```

Note que a listagem agrupa os resultados por bibliotecas (xhb e harbour) e ainda informa se essa biblioteca está instalada.
Para listar todas as funções do harbour faça : *hbmk2 -find **.
Se você está familiarizado com a linha de comando você deve saber que a saída pode ser direcionada para um arquivo cujo nome será dado por você. Por exemplo : *hbmk2 -find * > minhasaida.txt*.

11 Interface modo texto

> Mantenha-o simples: tão simples quanto possível, mas não mais simples do que isto.
>
> Albert Einstein

Objetivos do capítulo

- Entender como funciona o sistema de coordenadas padrão do Harbour
- Aprender a usar os elementos básicos da interface com o usuário.
- O sistema de cores do Harbour.
- Menus com @ PROMPT.
- O sistema de exibição @ ... SAY ... GET
- Máscaras de exibição de dados.
- Salvando e restaurando telas
- O comando @ ... GET ... LISTBOX

11.1 A interface padrão do Harbour

"Piratas do vale do silício" é um filme que todo desenvolvedor deveria assistir. O filme relata como foi a trajetória pessoal e profissional de Steve Jobs e Bill Gates e, além disso, também aborda as idas e vindas da indústria do software nas décadas de 1970, 1980 e 1990. Uma lição importante que o filme nos deixa é : "ser um programador excepcional é importante, mas você não pode se iludir achando que apenas isso é o que importa para vencer o jogo (nem sempre limpo) das grandes corporações ." O filme não mostra um Bill Gates programador de primeiríssima linha, mas logo no início o retrata como um bom jogador de pocker (um bom blefador). Também existem incontáveis referências ao trabalho duro e ao esforço acima da média, como aquela cena em que Gates passa a noite trabalhando no escritório e é encontrado dormindo sob a sua mesa de trabalho. Steve Jobs é retratado como um visionário homem de negócios, sempre obcecado pela perfeição naquilo que faz, um patrão grosseiro e até mesmo um pouco sádico, como na cena em que ele humilha um candidato a uma vaga na sua empresa. Um dos objetivos de Jobs, segundo o filme, é deixar a sua marca no Universo. Jobs também não é retratado como um programador excepcional, embora ele tenha conhecimentos em programação. Uma cena que chama a atenção é quando Gates descobre o protótipo de uma interface gráfica e um estranho aparelhinho chamado "mouse". Ele diz, quase em tom profético, mais ou menos assim : "precisamos disso, senão estamos fora do jogo". Naquela época a interface que predominava era um modelo conhecido como "modo texto", que consistia em uma tela semelhante ao "Prompt de comando" do windows. No filme (e na realidade também), Gates entendeu que, se permanecesse apenas com o MS-DOS [1] e sua parceria com a IBM, logo iria ser "engolido" por algum gigante do software que lançasse uma interface gráfica.

A linguagem Harbour possui um conjunto padrão de comandos que garantem uma eficiente interface de entrada de dados, porém essa interface padrão só funciona em modo texto [2] . Ainda hoje você pode encontrar incontáveis aplicativos nesse formato, mas o usuário final não gosta mais de aplicativos assim, pois eles são considerados esteticamente feios e transmitem uma ideia de antiguidade e atraso tecnológico[3]. Você pode até argumentar que essa interface é segura, permite acesso a todas os avanços tecnológicos atuais [4] e que grandes cadeias de lojas e bancos utilizam tais aplicativos, mas não vai adiantar muito. Cedo ou tarde o "filho do dono" vai assumir a direção, ou algum "consultor organizacional" será contratado, e essa inocente mudança pode selar o futuro dos seus aplicativos "modo texto", e você, assim como Gates, correrá o risco de estar fora do jogo, apesar de ter um excelente produto. O Harbour também possui uma série de bibliotecas gráficas que garantem um eficiente modo de se comunicar com o usuário usando o Windows, o Linux e o Mac; contudo esse capítulo não abordará essas bibliotecas. Nós vamos continuar programando em modo texto não porque ele é melhor para o usuário, mas porque ele é melhor para você aprender a programar.

[1] Dentro da História da computação é considerado por alguns como sendo o produto que decidiu o destino da então minúscula Microsoft. o MS-DOS possui nativamente uma interface de linha de comandos através do seu interpretador de comandos, command.com.

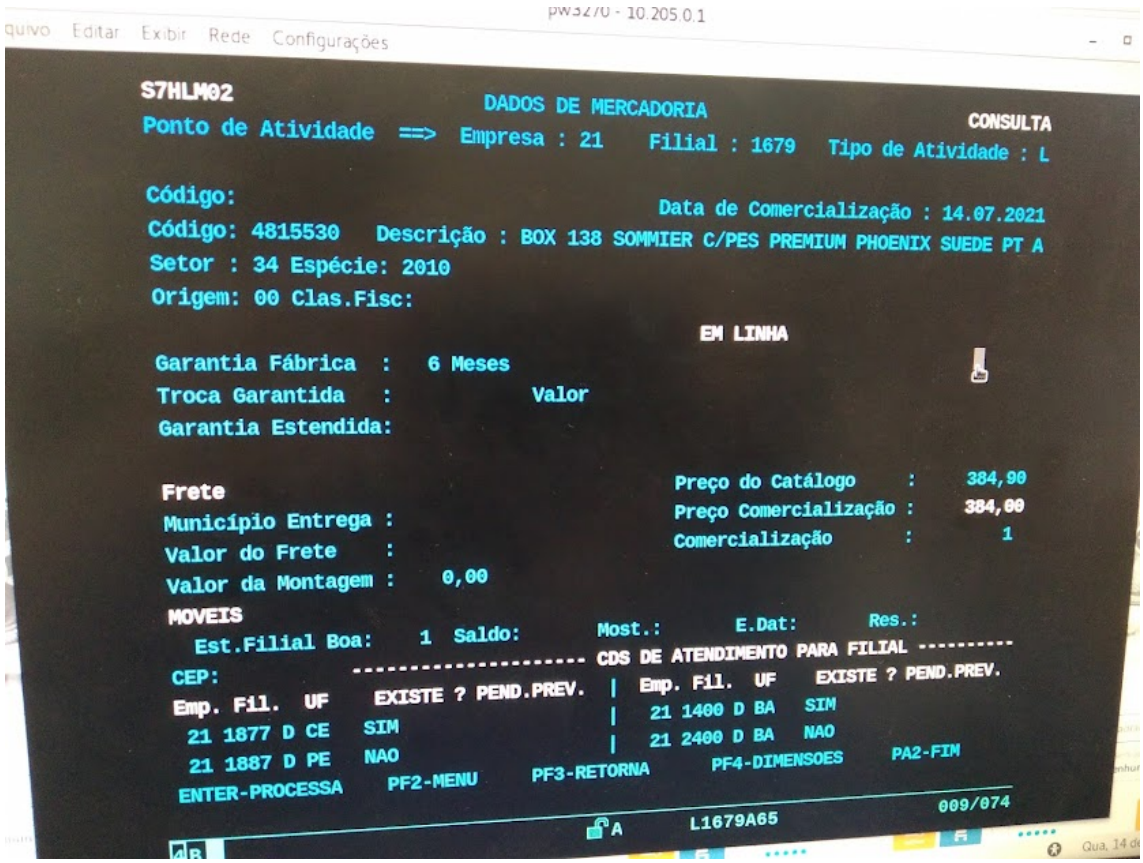
[2] Alguns programadores não acham que isso seja um problema. Sinta-se a vontade para discordar do que esta lendo, essa seção de introdução reflete apenas a minha opinião pessoal.

[3] Há quem discorde dessa afirmação, e eu sou um dos que discordam, mas muita gente pensa assim, e no final das contas eles pagam pelo software

[4] Por exemplo : acesso a banco de dados relacionais, thread, orientação a objeto, xml e outros formatos de dados

Na verdade, os conceitos de "melhor"ou "pior"para o usuário final é bem relativo. Os bancos, grandes cadeias de supermercados e lojas de departamentos ainda usam interfaces modo texto (figura 11.1), mas é inegável que tais interfaces estão restritas a determinadas aplicações.

Figura 11.1: Foto de uma interface modo-texto, usada por uma grande loja de departamentos com dezenas de filiais.



O objetivo desse capítulo é lhe apresentar os outros elementos da interface texto que é o padrão do Harbour. Esses elementos são bem simples e de fácil assimilação, de modo que você não deve demorar muito neles. Caso você deseje dar manutenção em algum aplicativo legado da era Clipper ou você ache viável desenvolver uma solução “modo-texto”, então é importante que você aprenda bem essa interface, caso contrário, não precisa se aprofundar muito[5].

11.1.1 Em que consiste um “Prompt de comando”

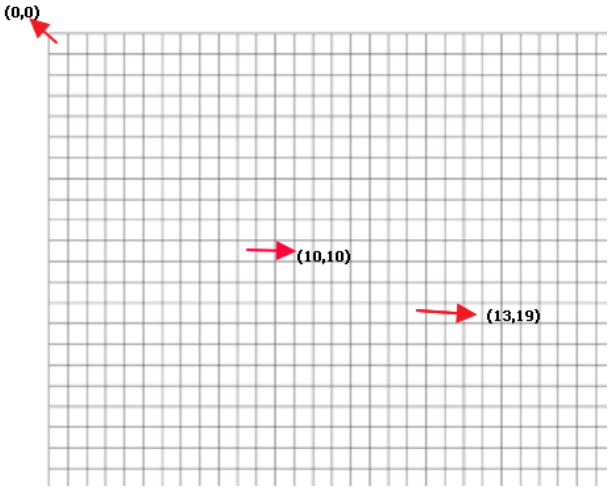
Antigamente, os computadores não tinham monitores. A saída de dados ocorria em uma espécie de impressora semelhante a um teletipo. Quando os monitores surgiram o objetivo inicial era simular uma “impressora” dessas. Dessa forma, você digitava comandos, teclava ENTER e o resultado saia na linha subsequente e assim por

[5]Essa introdução possui algumas opiniões bem pessoais, não é meu objetivo ofender ou diminuir alguém. Muito pelo contrário, respeito muitos os colegas que acreditam e apostam nessa interface, é apenas uma questão de opinião pessoal. Eu, particularmente acho até que o modo texto é uma forma produtiva para entrada de dados, pois evita o uso do teclado e do mouse ao mesmo tempo.

diante. Com o passar dos anos esse modelo sofreu modificações, mas a sua essência permaneceu a mesma.

Toda interface de linha de comando também possui um sistema de coordenadas, semelhante a um plano cartesiano. Porém essas coordenadas iniciam-se na parte superior esquerda do monitor e vão aumentando a medida que nos deslocamos para a direita e para baixo. Imagine o seu monitor como uma folha quadriculada usada para construir gráficos. Cada quadradinho possui a sua própria coordenada: o "quadrado" do canto superior esquerdo possui a coordenada (0,0), o "quadrado" que possui as coordenadas (10,20) está na linha 10 e na coluna 20, e assim por diante. A figura 11.2 ilustra esse raciocínio.

Figura 11.2: Sistema de coordenadas



Note que a numeração das coordenadas começam com zero, portanto a linha dez equivale a décima primeira linha, a coluna doze equivale a décima terceira coluna, e assim por diante: sempre adicionando um ao valor real da coordenada. É bom lembrar que não existem coordenadas negativas.

Dica 68

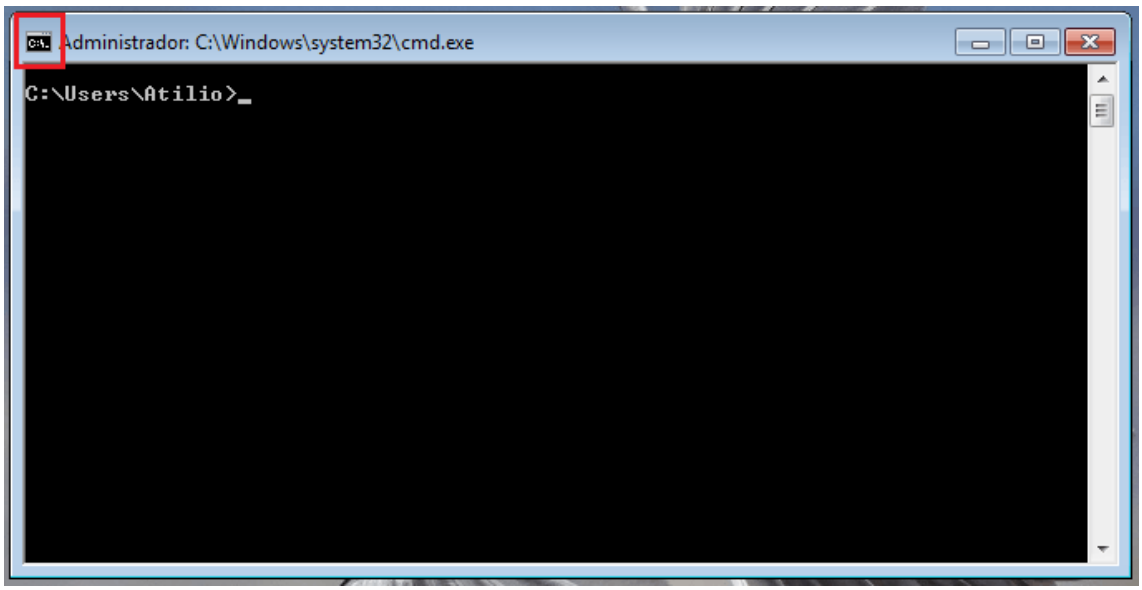
Você talvez já tenha ouvido falar na palavra "pixel". Pois bem, esse nosso sistema de coordenadas **não tem nada a ver com isso**. Não cometa o erro de confundir o "quadradinho" com o pixel. Pixel [...] é o menor elemento num dispositivo de exibição (como por exemplo um monitor), ao qual é possível atribuir-se uma cor. De uma forma mais simples, um pixel é o menor ponto que forma uma imagem digital, sendo que o conjunto de milhares de pixels formam a imagem inteira. (Fonte : `https://pt.wikipedia.org/wiki/Pixel` On-line: 25-Set-2016).

Bem, as coordenadas começam na coordenada (0,0), mas onde elas terminam ? Antigamente (década de 1980) os monitores possuíam apenas 25 linhas por 80 colunas, então o sistema de coordenadas terminava na linha 24 e na coluna 79. Mas agora os modernos prompts de comandos permitem uma quantidade maior de "quadradinhos". Para demonstrar isso realize a seguinte experiência[6] :

[6]Retirada de `https://terminaldeinformacao.com/2014/03/04/editando-o-prompt-de-comando-do-windows-cmd/` (On-line:25-Set-2016)

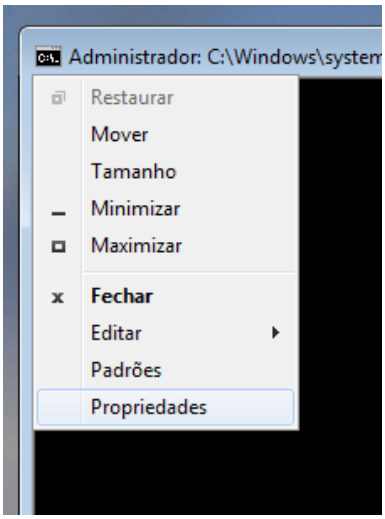
Primeiramente abra o Prompt de Comando, ou procurando o aplicativo, ou executando o comando cmd (pelo Executar – Windows + R). Após a tela ser aberta, clique no ícone do prompt, conforme imagem abaixo:

Figura 11.3: Prompt de comando



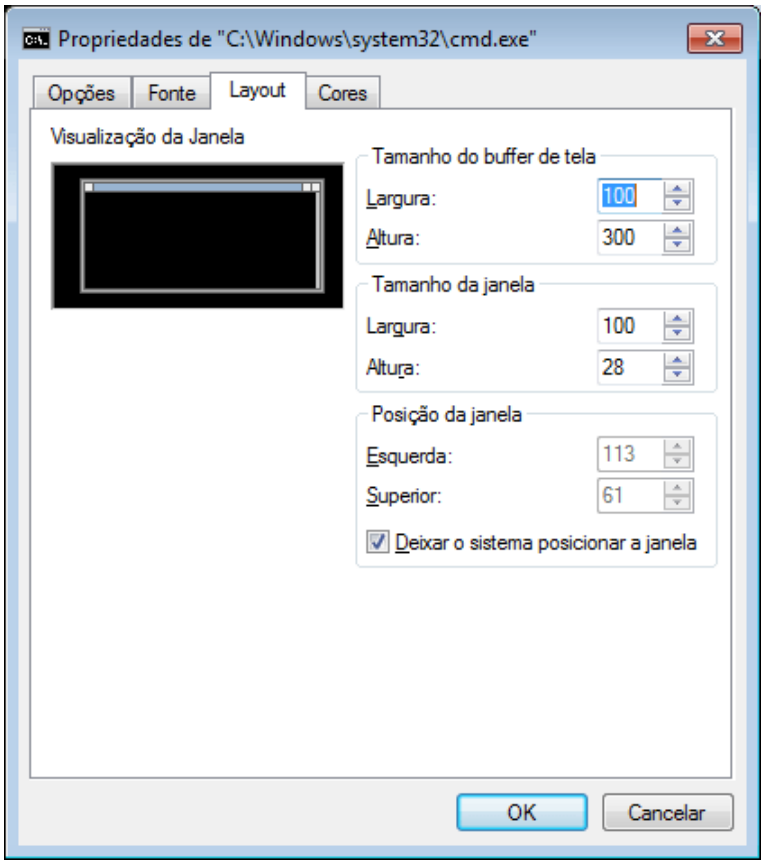
Depois, clique na opção Propriedades:

Figura 11.4: Propriedades do Prompt de comando



Na terceira aba você poderá alterar o tamanho da janela informando a quantidade máxima de “quadradinhos”.

Figura 11.5: Configurando as coordenadas



Note que na figura 11.5 nós temos um grupo de opções chamado de “tamanho da janela”. É dentro desse grupo que você pode informar as coordenadas largura (coluna) e altura (linha). Nessa figura o sistema de coordenadas do Harbour irá até a coordenada (27, 99). Mas isso não quer dizer que você não poderá usar coordenadas superiores a estas, por exemplo, se você exibir um caractere na coordenada (60,200) o Harbour irá posicionar, mas você não conseguirá ver o caractere, já que o prompt (na figura 11.5) termina em (27,99).

11.2 Trabalhando com o sistema de coordenadas padrão do Harbour

Até agora os comandos de entrada e saída que nós trabalhamos utilizam coordenadas. Eles emulam o antigo teletipo que nós citamos no início desse capítulo. Seria mais ou menos como se o seu monitor fosse uma impressora, dessa forma, você usa o comando “?” e o computador exibe o conteúdo e passa para a linha de baixo. Da mesma forma os comandos INPUT e ACCEPT passam para a linha de baixo após teclarmos ENTER. A partir de agora nós iremos lhe apresentar os demais comandos da interface padrão do Harbour, mas antes disso certifique-se de que entendeu o sistema de coordenadas abordado na seção anterior. É bom reforçar que quando nós falamos de coordenadas nós não estamos nos referindo a um ponto (pixel), nós estamos nos referindo a um “quadradinho” da suposta folha quadriculada. Cada quadrado recebe apenas um caractere ASCII, dessa forma, se eu mando imprimir a letra “A” na posição (1,2) eu estou me referindo ao quadrado inteiro que irá conter a letra “A”.

Os comandos que se utilizam de coordenadas iniciam-se com um @. Assim temos os seguintes comandos :

1. @ ... BOX
2. @ ... CLEAR
3. @ ... GET
4. @ ... PROMPT
5. @ ... SAY
6. @ ... TO

Nas seções subsequentes nós abordaremos esses comandos e apresentaremos alguns relacionados a exibição de informações na tela, como os comandos que permitem salvar e restaurar telas.

11.3 Desenhando um retângulo na tela

Existem situações onde você precisa desenhar uma figura retangular na tela. Apenas um retângulo na tela, nada muito sofisticado. O Harbour dispõe de dois comandos para a exibição de figuras retangulares na tela: o @ ... BOX e o @ ... TO. O @ ... BOX é mais complicado mas ele permite uma variação maior nas bordas do retângulo, já o @ ... TO é bem mais simples. Se você quiser pular o comando @ ... BOX e ir direto para o @ ... TO pode ficar a vontade, pois será ele (@ ... TO) que nós usaremos nos exemplos posteriores.

11.3.1 Retângulos com @ ... BOX

O comando @ ... BOX é o mais completo da dupla porque você pode determinar qual conjunto de caracteres você irá usar na moldura. A sintaxe do @ ... Box é a seguinte :

Descrição sintática 9

1. Nome : @ ... BOX
2. Classificação : comando.
3. Descrição : Desenha uma caixa na tela.
4. Sintaxe

```
@ <nLinTopo>, <nColTopo>, <nLinBase>, <nColBase>;
 BOX <cCaixa>
```

Fonte : [Nantucket 1990, p. 4-3]

Um exemplo simples pode ser visto na listagem 11.1.

Listagem 11.1: Desenhando um retângulo
Fonte: codigos/arroba00.prg

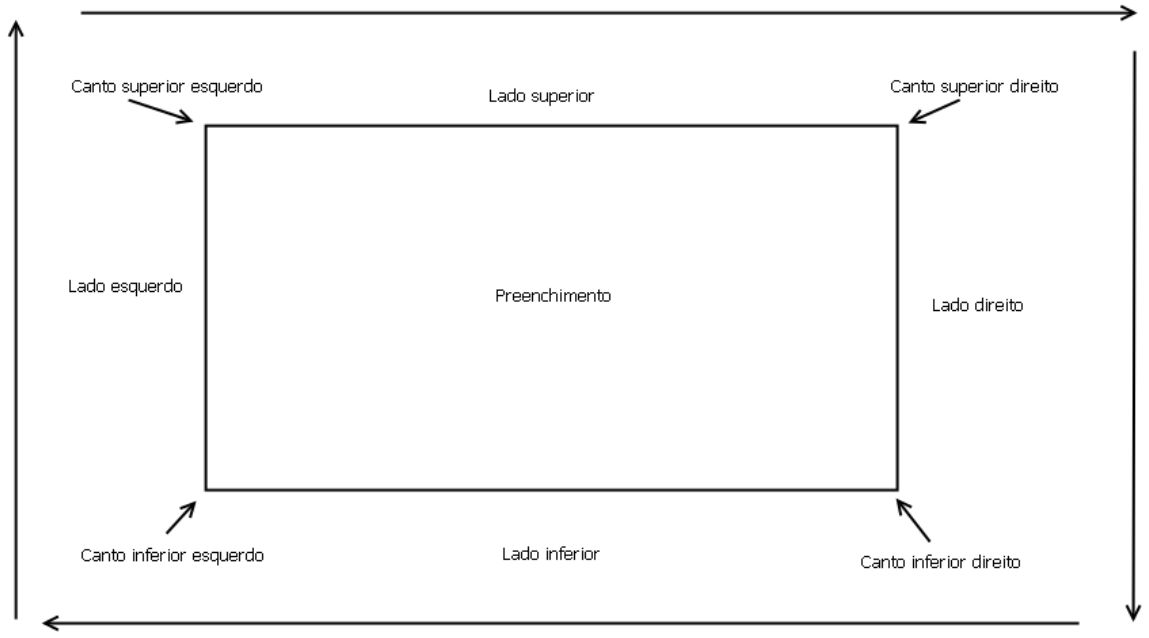
```
/*
Desenhando um retângulo da tela
*/
PROCEDURE Main

   @ 1, 1, 10, 50 BOX "*"

RETURN
```

.:Resultado:.

```
**************************************************
*                                                *
*                                                *
*                                                *
*                                                *
**************************************************
```

Você poderá usar até pode usar nove caracteres como string de moldura. Cada caractere irá corresponder a um lado da moldura e o seu centro, assim como está representado na figura 11.6. As setas representam a ordem com que você deve informar esses caracteres: começando pelo caractere do canto superior esquerdo e terminando no caractere do lado esquerdo. Se quiser você ainda pode incluir um caractere especial extra para preenchimento do seu box.

Figura 11.6: Box

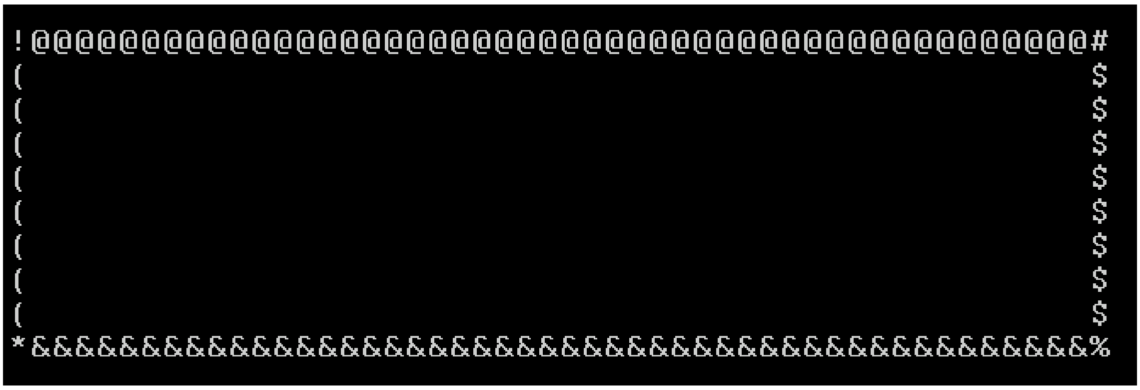


O exemplo a seguir desenha um box usando uma string com nove caracteres. Observe que cada caractere representa uma parte da moldura.

Listagem 11.2: Desenhando um retângulo
Fonte: codigos/arroba000.prg

```
/*
Desenhando um retângulo da tela
*/
PROCEDURE Main

   @ 1, 1, 10, 50 BOX "!@#$%&*( "

RETURN
```

Figura 11.7: Box



.

Como você deve ter percebido essas molduras não são muito práticas para serem criadas, de forma que o Harbour possui formas de facilitar a sua criação. Como as molduras são constituídas por alguns caracteres especiais contidos na tabela ASCII o Harbour possui um arquivo de constantes chamado “box.ch”. Esse arquivo contém algumas constantes convertidas em simbolos para facilitar o seu manuseio. O exemplo a seguir (listagem 11.3) desenha quatro caixas usando constantes simbólicas contidas no arquivo “box.ch”.

Listagem 11.3: Molduras vindas do arquivo box.ch
Fonte: codigos/arroba01.prg

```
/*
Desenhando um retângulo da tela
*/
#include "box.ch"
PROCEDURE Main
LOCAL nLinTop, nColEsq, nLinBase, nColDir // Coordenadas

   nColEsq := 1
   nColDir := 20
   nLinTop := 1
   nLinBase := nLinTop + 5


   @ nLinTop, nColEsq, nLinBase, nColDir BOX B_SINGLE

```

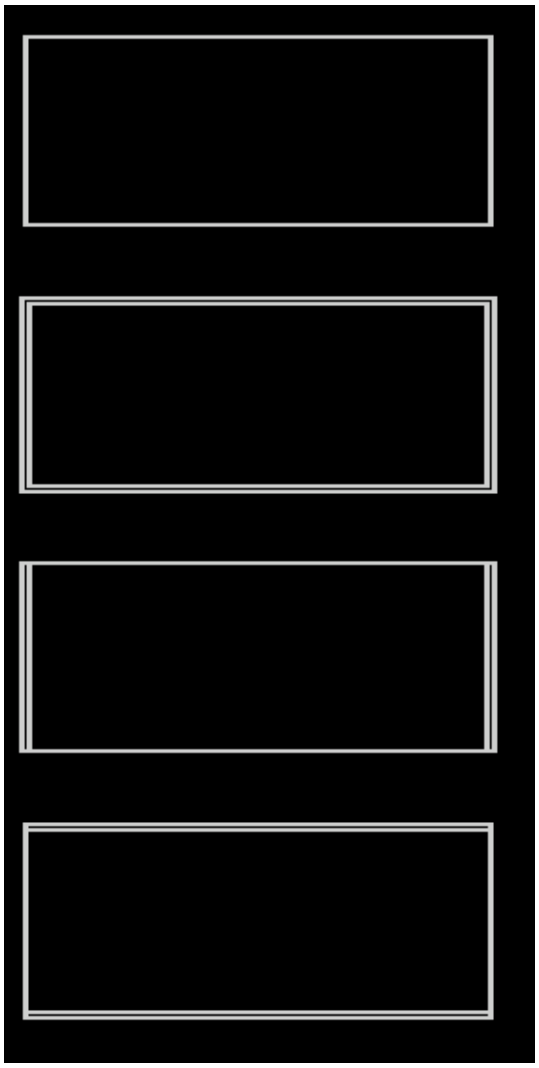
```
   nLinTop := nLinTop + 7                                                    16
   nLinBase := nLinTop + 5                                                   17
   @ nLinTop, nColEsq, nLinBase, nColDir BOX B_DOUBLE                        18
                                                                             19
   nLinTop := nLinTop + 7                                                    20
   nLinBase := nLinTop + 5                                                   21
   @ nLinTop, nColEsq, nLinBase, nColDir BOX B_SINGLE_DOUBLE                 22
                                                                             23
   nLinTop := nLinTop + 7                                                    24
   nLinBase := nLinTop + 5                                                   25
   @ nLinTop, nColEsq, nLinBase, nColDir BOX B_DOUBLE_SINGLE                 26
                                                                             27
RETURN                                                                       28
```

As constantes são :

1. B_SINGLE : Linha simples
2. B_DOUBLE : Linha dupla
3. B_SINGLE_DOUBLE : Linhas horizontais simples e verticais duplas
4. B_DOUBLE_SINGLE : Linhas horizontais duplas e verticais simples

Essas constantes equivalem (em ordem de apresentação) as seguintes molduras da figura 11.8.

Figura 11.8: Box



11.3.2 Retângulos com @ ... TO

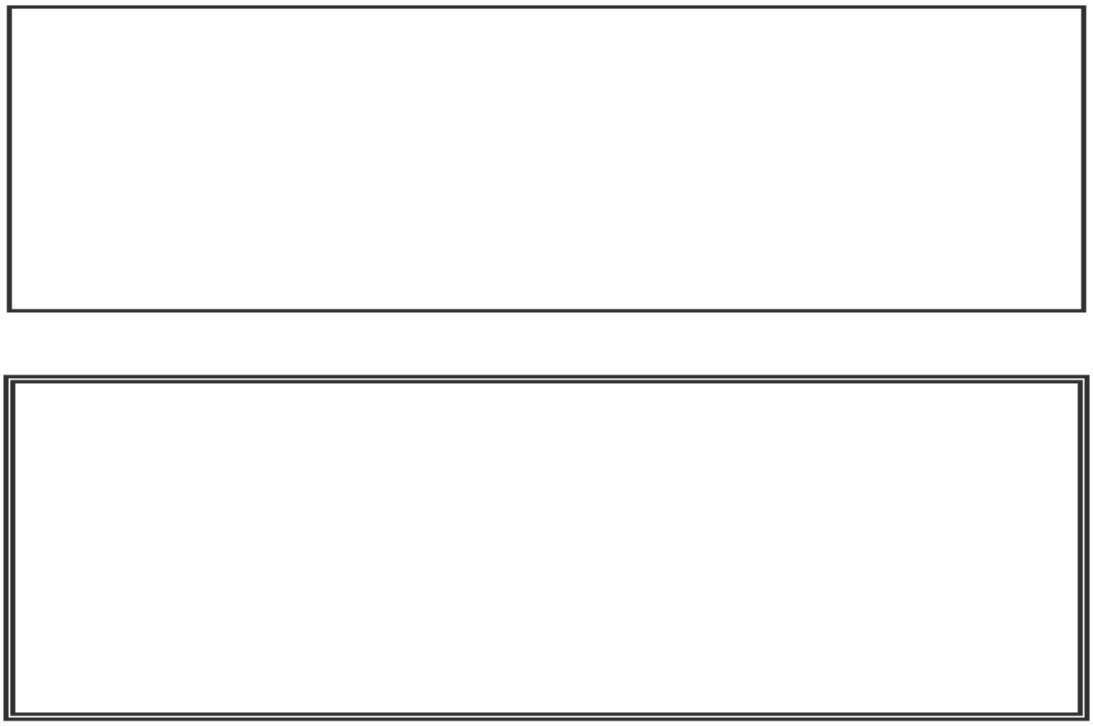
O @ ... TO é mais simples do que o @ ... BOX, porque ele já pressupõe quais caracteres você usará para o contorno do retângulo. O exemplo da listagem 11.4 desenha os dois tipos disponíveis de retângulos.

Listagem 11.4: @ ... TO
Fonte: codigos/arroba02.prg

```
/*
Desenhando um retângulo da tela
*/
PROCEDURE Main

   @ 1, 1 TO 10, 50

   @ 12, 1 TO 22, 50 DOUBLE

RETURN
```

A figura 11.9 ilustra os dois únicos modelos possíveis gerados pela listagem 11.4.

Figura 11.9: @ ... TO



.

A sintaxe do @ ... TO está descrita no quadro seguinte.

Descrição sintática 10

1. Nome : @ ... TO
2. Classificação : comando.
3. Descrição : Desenha um retângulo com bordas simples ou duplas.
4. Sintaxe

```
@ <nLinTopo>, <nColTopo>;
 TO <nLinBase>, <nColBase> [DOUBLE]
```

Fonte : [Nantucket 1990, p. 4-15]

Nesse livro nós usaremos apenas o @ ... TO para desenhar retângulos, mas caso você se depare com algum código com @ ... BOX (em outro local) você já saberá do que se trata.

11.4 Apagando trechos da tela com @ ... CLEAR e CLS

O Harbour possui dois comandos que “limpam” o conteúdo que está sendo exibido na tela. O primeiro deles é @ ... CLEAR e o segundo é o CLS. Como nós já dissemos, o @ significa coordenadas de tela, assim o @ ... CLEAR serve para “limpar” a região delimitado pelas coordenadas informadas. Conforme o exemplo a seguir.

```
@ 10,10 CLEAR TO 20,20
```

Descrição sintática 11

1. Nome : @ ... CLEAR
2. Classificação : comando.
3. Descrição : Limpa uma região retângular da tela.
4. Sintaxe

```
@ <nLinTopo>, <nColTopo> ;
        [CLEAR [TO <nLinBase>, <nColBase>]]
```

Fonte : [Nantucket 1990, p. 4-4]

@ ... CLEAR pode ser especificado sem um CLEAR TO, nesse caso apenas a linha horizontal a partir da coordenada informada será limpa.

O comando CLS não possui o @ porque ele não trabalha com coordenadas. Simplesmente ele “limpa” toda a tela. É simples e rápido, de modo que nós o usaremos para operações de limpeza de tela nesse livro. A partir desse ponto será comum o uso do comando CLS na abertura dos nossos programas.

11.5 Menus com @ ... PROMPT e MENU TO

O comando @ ... PROMPT torna a construção de menus de seleção uma tarefa fácil. Esse comando funciona em conjunto com um outro comando chamado MENU TO. O exemplo a seguir ilustra o seu efeito.

Listagem 11.5: @ ... PROMPT e MENU TO
Fonte: codigos/menuto01.prg

```
/*
Desenhando um retângulo da tela
*/
PROCEDURE Main
LOCAL nOpcao

   CLS // Limpa a tela
```

```
   @ 5, 5 TO 9, 18 DOUBLE // Crio um box com @ ... TO

   // Aqui inicia o menu
   @ 6,6 PROMPT "  Cadastro  "
   @ 7,6 PROMPT " Relatórios "
   @ 8,6 PROMPT " Utilitários"

   // Aqui eu seleciono a opção
   MENU TO nOpcao

   // Aqui eu analiso o valor de nOpcao
   DO CASE
   CASE nOpcao == 1
      ? "Você selecionou cadastro"
   CASE nOpcao == 2
      ? "Você selecionou relatórios"
   CASE nOpcao == 3
      ? "Você selecionou utilitários"
   OTHERWISE
      ? "Nenhuma das alternativas"
   ENDCASE

RETURN
```

Figura 11.10: @ ... PROMPT

.

Descrição sintática 12

1. Nome : @ ... PROMPT
2. Classificação : comando.
3. Descrição : Exibe um ítem de menu e define uma mensagem.
4. Sintaxe

```
@ <nLin>, <nCol> PROMPT <cItem> [MESSAGE <cMessage>]
```

Fonte : [Nantucket 1990, p. 4-11]

Descrição sintática 13

1. Nome : MENU TO
2. Classificação : comando.
3. Descrição : Executa um menu de barra luminosa para PROMPTs definidos anteriormente com o comando @ ... PROMPT
4. Sintaxe

```
MENU TO <nVar>
```

Fonte : [Nantucket 1990, p. 4-70]

O funcionamento do menu criado com @ ... PROMPT é simples, você usa as setas do teclado (cima e baixo) para selecionar a opção desejada e tecla ENTER para ativar a seleção. A partir daí o programa sai do menu e atribui um valor numérico a variável definida em MENU TO. Se o usuário selecionou a primeira opção a variável receberá o valor 1, se selecionou a segunda opção a variável receberá 2 e assim sucessivamente. Caso o usuário resolva abandonar o menu sem selecionar nada ele deve teclar ESC. Com isso o valor da variável definida em MENU TO receberá o valor zero.

Note que você também pode usar esse comando em conjunto com os comandos de construção de retângulos (@ ... BOX ou @ ... TO) para criar um efeito de moldura nos seus menus.

11.5.1 Menus com mensagens adicionais

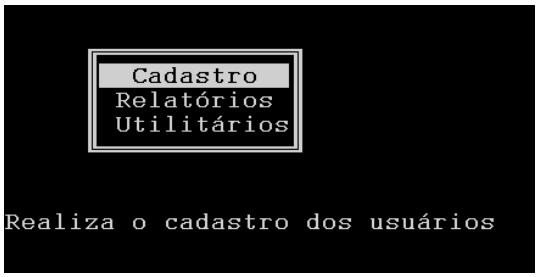
Caso você necessite imprimir alguma mensagem adicional para esclarecer a opção do menu que está selecionada, você deve usar a cláusula MESSAGE no comando @ ... PROMPT, conforme o exemplo da listagem 11.6.

Listagem 11.6: @ ... PROMPT com mensagens adicionais
Fonte: codigos/menuto02.prg

```
/*
Desenhando um retângulo da tela
*/
PROCEDURE Main
LOCAL nOpcao

   CLS // Limpa a tela
   @ 5, 5 TO 9, 18 DOUBLE // Crio um box com @ ... TO


   // Aqui inicia o menu
   @ 6,6 PROMPT "  Cadastro  " ;
         MESSAGE "Realiza o cadastro dos usuários"
   @ 7,6 PROMPT " Relatórios " ;
```

```
         MESSAGE "Imprime os relatórios do sistema"
   @ 8,6 PROMPT " Utilitários" ;
         MESSAGE "Inicia a rotina de utilitários"

   SET MESSAGE TO 12// As mensagens acima serão impressas na linha 12
   // Aqui eu seleciono a opção
   MENU TO nOpcao

   // Aqui eu analiso o valor de nOpcao
   DO CASE
   CASE nOpcao == 1
      ? "Você selecionou cadastro"
   CASE nOpcao == 2
      ? "Você selecionou relatórios"
   CASE nOpcao == 3
      ? "Você selecionou utilitários"
   OTHERWISE
      ? "Nenhuma das alternativas"
   ENDCASE

RETURN
```

Figura 11.11: @ ... PROMPT com mensagens adicionais



.

Note que existe um SET especial chamado MESSAGE que determina a linha onde essas mensagens serão exibidas. Se você não informar essa variável então o valor padrão dela será em uma linha cuja coordenada não será exibida na tela.

11.6 Entrada de dados com @ ... SAY ... GET e READ

O comando @ ... SAY ... GET é usado para criar formulários de entradas de dados. Ele possui algumas características que o fazem bastante diferente dos comandos ACCEPT e INPUT que nós já vimos. O mais importante deles é que o comando @ ... SAY ... GET exige que a variável seja definida previamente. A listagem 11.7 ilustra o funcionamento do comando @ ... SAY ... GET.

Listagem 11.7: @ ... SAY ... GET
Fonte: codigos/get01.prg

```
/*
@ ... SAY ... GET simples
*/
```

```
PROCEDURE Main
LOCAL nValor1 := 0
LOCAL nValor2 := 0


   CLS // Limpa a tela
   @ 5,5 SAY "Informe o valor da parcela 1 : " GET nValor1
   @ 7,5 SAY "Informe o valor da parcela 2 : " GET nValor2
   READ

RETURN
```

Quando você compilou o programa ele foi gerado com sucesso, mas algumas mensagens esquisitas apareceram. As mensagens estão reproduzidas, em parte, a seguir:

.:Resultado:.

```
get01.prg(10) Warning W0001  Ambiguous reference 'GETLIST'
get01.prg(11) Warning W0001  Ambiguous reference 'GETLIST'
```

Você já deve estar familiarizado com esse tipo de mensagem. Esses avisos são gerados quando você não declara uma variável. Mas se você prestou atenção a listagem 11.7 verá que todas as variáveis usadas foram declaradas. Qual é o mistério, então ?

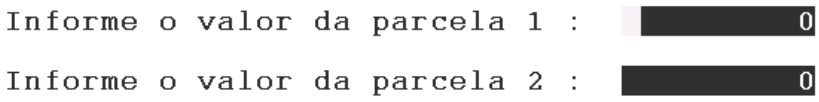
11.6.1 O GetList

O comando @ ... SAY ... GET possui uma variável que não aparece no comando, trata-se de uma variável oculta chamada GetList. Não vamos nos demorar com explicações agora porque ela pertence a uma categoria de variável chamada VETOR, que só será abordada mais adiante. Mas fique tranquilo, basta declarar essa variável, conforme o exemplo da listagem 11.8 a seguir :

Listagem 11.8: @ ... GET
Fonte: codigos/get02.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL nValor1 := 0
LOCAL nValor2 := 0
LOCAL GetList := {}


   CLS // Limpa a tela
   @ 5,5 SAY "Informe o valor da parcela 1 : " GET nValor1
   @ 7,5 SAY "Informe o valor da parcela 2 : " GET nValor2
   READ

RETURN
```

Figura 11.12: @ ... SAY ... GET



.

Dica 69

Sempre que for usar o comando @ ... GET ou @ ... SAY ... GET declare o vetor GetList conforme abaixo:

```
LOCAL GetList := {}
```

Não se preocupe com essas chaves , quando nós formos estudar os vetores você entenderá.

11.6.2 Os Delimitadores do GET

Você pode alterar a forma com que os campos são delimitados através do SET DELIMITERS. Esse SET altera a forma com que os campos são delimitados. Por exemplo :

Listagem 11.9: @ ... GET
Fonte: codigos/get03.prg

```
/*                                                                          1
@ ... SAY ... GET simples                                                   2
*/                                                                          3
PROCEDURE Main                                                              4
LOCAL nValor1 := 0                                                          5
LOCAL nValor2 := 0                                                          6
LOCAL GetList := {}                                                         7
                                                                            8
   SET DELIMITERS ON // Exibe os delimitadores                              9
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs                10
   CLS                                                                      11
   @ 5,5 SAY "Informe o valor da parcela 1 : " GET nValor1                  12
   @ 7,5 SAY "Informe o valor da parcela 2 : " GET nValor2                  13
   READ                                                                     14
                                                                            15
RETURN                                                                      16
```

.:Resultado:.

```
Informe o valor da parcela 1 :  [        0]
```

```
    Informe o valor da parcela 2 :  [         0]
```

Note que temos dois SETs. O primeiro deles “SET DELIMITERS ON” ativa a exibição dos delimitadores, mas não diz quais são os delimitadores. O comando “SET DELIMITERS TO <delimitador>” diz qual delimitador deve ser usado. Se você não informar qual delimitador usar, então o delimitador padrão será “:”, o que é confuso para o operador.

11.6.3 O comando READ

Com certeza você ainda deve ter algumas dúvidas sobre as listagens apresentadas. Afinal de contas, para que serve o comando READ ? Se você tentar compilar as listagens apresentadas sem o comando READ verá que o programa será gerado perfeitamente, mas quando você for executar, notará que o programa não irá “esperar” pela digitação dos valores. Bom, basicamente é isso que faz o comando READ. Ele cria um “estado de espera” para que o usuário digite os valores nos GETs. Esse estado de espera é importante porque permite que eu use as setas para cima e para baixo para navegar a vontade pelos campos do formulário. É para isso que essa dupla foi criada, para permitir o preenchimento de formulários de forma prática, afinal de contas se o usuário errar o preenchimento do primeiro campo e só notar o erro quando estiver no terceiro campo, o estado de espera criado pelo READ permitirá ao usuário “subir” de volta ao primeiro campo e consertar o valor.

11.6.4 Teclas de movimentação principais

Saindo de um formulário

Existem três formas básicas de se sair de um formulário com vários GETs : a primeira delas você já deve ter experimentado, que é ir teclando ENTER até chegar no último GET. A segunda forma é mais sutil, basta pressionar a tecla ESC, já a terceira é usando a tecla Page Down. As duas últimas formas: com ESC e Page Down, permitem ao usuário abandonar o formulário sem precisar estar no último campo. Quando nós estudamos o comando @ ... PROMPT nós vimos que o ESC atribui um valor zero a variável de MENU TO. Já o comando @ ... SAY ... GET **não funciona assim**. O que estamos querendo dizer é : a tecla ESC não faz com que as variáveis percam o valor que o usuário digitou nelas. Faça o seguinte teste: digite o programa da listagem 11.10 e abandone o formulário usando as três formas explicadas. Note que em todas elas os valores que você digitou são preservados.

Listagem 11.10: @ ... GET
Fonte: codigos/get04.prg

```
/*                                                                  1
@ ... SAY ... GET simples                                           2
*/                                                                  3
PROCEDURE Main                                                      4
LOCAL nValor1 := 0                                                  5
LOCAL nValor2 := 0                                                  6
LOCAL GetList := {}                                                 7
                                                                    8
```

```
   SET DELIMITERS ON // Exibe os delimitadores                          9
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs            10
   CLS                                                                  11
   @ 5,5 SAY "Informe o valor da parcela 1 : " GET nValor1              12
   @ 7,5 SAY "Informe o valor da parcela 2 : " GET nValor2              13
   READ                                                                 14
                                                                        15
   ? "O valor nValor1 é " , nValor1                                     16
   ? "O valor nValor2 é " , nValor2                                     17
                                                                        18
RETURN                                                                  19
```

.:Resultado:.

```
     Informe o valor da parcela 1 :  [      100]

     Informe o valor da parcela 2 :  [       10]


O valor nValor1 é          100
O valor nValor2 é           10
```

Navegando entre os campos do formulário

Existem duas formas para se navegar entre os campos do formulário. A tecla ENTER passa para o GET seguinte e abandona o formulário após o último GET, essa você já sabe. Mas você também pode usar as seguintes teclas :

- Setas para cima e para baixo: avançam e retrocedem nos campos.
- TAB e Shift + TAB: o mesmo efeito que as setas para cima e para baixo.[7]

11.6.5 Utilizando valores caracteres nos GETs

Você já sabe que deve inicializar a variável que irá ser usada no GET. Não basta somente declarar a variável, você deve atribuir-lhe um valor inicial. Quando a variável for do tipo caractere você deve inicializar essa variável com o tamanho máximo que ela irá ocupar. Se você criar uma variável caractere para armazenar o CEP de um endereço, então certifique-se de que o tamanho será de pelo menos oito espaços. Já se você criar uma variável caractere para armazenar o nome de uma pessoa, então reserve pelo menos cinquenta espaços para ela.

O Harbour possui uma função chamada SPACE() que retorna uma string do tamanho do número que você informou como argumento da função. Por exemplo : SPACE(50) retorna uma string com cinquenta espaços. Use essa variável para inicializar suas variáveis caracteres, conforme a listagem 11.11.

[7] O símbolo + significa que você deve pressionar a tecla Shift, mantê-la pressionada e apertar a tecla TAB. A tecla Shift na maioria dos teclados é representada por uma seta para cima. Essa tecla situa-se entre a tecla Control (Ctrl) e a tecla Caps Lock. Já a tecla TAB as vezes não vem com o nome TAB gravado nela, mas ela situa-se sobre a tecla Caps Lock.

Listagem 11.11: @ ... GET
Fonte: codigos/get05.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cEndereco := SPACE(50)
LOCAL cBairro := SPACE(30)
LOCAL nNumero := 0
LOCAL cCEP := SPACE(8)
LOCAL cComplemento := SPACE(10)
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o endereço : " GET cEndereco
   @ 7,5 SAY "Informe o número : " GET nNumero
   @ 9,5 SAY "Informe o CEP : " GET cCEP
   @ 11,5 SAY "Informe o complemento : " GET cComplemento
   @ 13,5 SAY "Informe o bairro : " GET cBairro
   READ

   ? cEndereco, nNumero
   ? cComplemento
   ? cBairro, cCEP



RETURN
```

.:Resultado:.

```
     Informe o endereço :  [Rua Afonso Pena

     Informe o número :  [       450]

     Informe o CEP :  [60015100]

     Informe o complemento :  [Apto 789  ]

     Informe o bairro :  [Centro                        ]



Rua Afonso Pena                                      450
Apto 789
Centro                        60015100
```

Note que, os valores são atribuídos mas as respectivas variáveis continuam com os tamanhos originais, ou seja, os espaços em branco finais não foram removidos. Repare que entre o nome da rua e o número existe um grande espaço, e entre o nome do bairro e o CEP também existe um espaço considerável. Isso acontece porque as variáveis não perderam o seu tamanho original. É importante saber disso quando nós formos usar os operadores de comparação "==" e sua respectiva negação, pois um espaço em branco a mais já é suficiente para deixar duas variáveis diferentes uma da outra.

11.7 Criando pós validações com VALID

Você também pode criar validações usando a cláusula VALID. Uma validação é uma comparação que retorna um valor lógico. O exemplo da listagem 11.12 a seguir deve esclarecer o uso de VALID. Note que ele é muito parecido com a listagem 11.11, a única diferença é que no GET da linha 16 nós acrescentamos a cláusula VALID.

Listagem 11.12: @ ... GET
Fonte: codigos/get06.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cEndereco := SPACE(50)
LOCAL cBairro := SPACE(30)
LOCAL nNumero := 0
LOCAL cCEP := SPACE(8)
LOCAL cComplemento := SPACE(10)
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o endereço : " GET cEndereco
   @ 7,5 SAY "Informe o número : " GET nNumero VALID nNumero > 0
   @ 9,5 SAY "Informe o CEP : " GET cCEP
   @ 11,5 SAY "Informe o complemento : " GET cComplemento
   @ 13,5 SAY "Informe o bairro : " GET cBairro
   READ

   ? cEndereco, nNumero
   ? cComplemento
   ? cBairro, cCEP



RETURN
```

No caso a seguir o usuário só vai poder passar para o campo seguinte se atribuir um número válido ao campo "número" do endereço.

.:Resultado:.

```
Informe o endereço :  [Alexandre Simões
   ]

Informe o número :  [         0]

Informe o CEP :  [        ]

Informe o complemento :  [          ]

Informe o bairro :  [                              ]
```

Note também algumas características que talvez tenham passadas desapercebidas:

1. Quando o usuário está preenchendo o campo “número” (que possui a cláusula VALID) ele só irá conseguir sair dele se digitar um valor que satisfaça o VALID correspondente.
2. As teclas de movimentação não deixam o usuário sair do campo. Nem voltar para o anterior.
3. A única tecla que permite abandonar o formulário estando dentro do campo “número” é a tecla ESC.
4. Se o usuário teclar Page Down estando dentro do campo “Nome” o formulário será abandonado. Ou seja, eu posso deixar o valor de “número” igual a zero, basta que eu abandone o formulário prematuramente e não esteja “dentro” desse campo.

Dica 70

A tecla ESC tem um significado especial nas interfaces modo texto. Normalmente ela é usada para abandonar uma operação[a]. Mas como você já deve ter notado anteriormente, os valores das variáveis não se perdem quando você tecla ESC e abandona o formulário, de modo que é você quem deverá tratar as situações em que o usuário resolveu abandonar o formulário e também as situações onde o usuário encerrou o usuário com Page Down ou ENTER. Normalmente isso é feito com um uma declaração IF.

[a]ESC é uma tecla cujo significado é *Escape*.

11.8 Criando pré-validações com WHEN

A cláusula WHEN possui um comportamento muito semelhante a cláusula VALID. A diferença é que a validação feita por WHEN é executada sempre que o usuário tentar entrar no campo. No exemplo da listagem 11.13 a seguir, após a digitação do nome, o usuário deve indicar se a pessoa é física ou jurídica. Caso a pessoa seja física, ela deve teclar um “F” no campo tipo, caso contrário deve teclar um “J”.

Listagem 11.13: @ ... GET
Fonte: codigos/get07.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cNome := SPACE(20)
LOCAL cTipo := SPACE(1)
LOCAL cCPF := SPACE(11)
LOCAL cCNPJ := SPACE(15)
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o nome : " GET cNome

   @ 7,5 SAY "Informe o tipo (F ou J): " GET cTipo VALID UPPER( cTipo ) $ "FJ"
   @ 9,5 SAY "Informe o CPF : " GET cCPF WHEN UPPER(cTipo) == "F"
   @ 11,5 SAY "Informe o CNPJ : " GET cCnpj WHEN UPPER(cTipo) == "J"
   READ

   ? cNome
   IF UPPER(cTipo) == "F"
      ? "CPF : " , cCPF
   ELSE
      ? "CNPJ: " , cCNPJ
   ENDIF

RETURN
```

Ao executar esse programa, note que o campo CPF só será solicitado se o usuário digitar “F” (Pessoa física), e o campo CNPJ só será solicitado caso o usuário digite “J” (Pessoa jurídica).

No exemplo abaixo, o usuário digitou “F”.

.:Resultado:.

```
     Informe o nome :  [Pablo César         ]

     Informe o tipo (F ou J):  [F]

     Informe o CPF :  [1212121212 ]

     Informe o CNPJ :  [              ]



Pablo César
CPF :  1212121212
```

11.9 Criando máscaras para edição de dados com PICTURE

O comando @ ... GET possui uma série de “máscaras” que permitem formatar a entrada de um comando. Por exemplo, vamos supor que você deseje inserir um número de telefone qualquer, por exemplo: (085)3456-9080. Até agora não temos uma forma fácil de obrigar o usuário a digitar os parênteses e o traço que separa o prefixo do telefone do restante da numeração, dessa forma, números “estranhos” podem surgir, por exemplo : (7090)3456=9877 ou 9870098098. O exemplo da listagem 11.14 é o mesmo programa anterior, mas com uma pequena modificação para a inclusão de algumas máscaras.

Listagem 11.14: @ ... GET
Fonte: codigos/get08.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cNome := SPACE(20)
LOCAL cTipo := SPACE(1)
LOCAL cCPF := SPACE(11)
LOCAL cCNPJ := SPACE(15)
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o nome : " GET cNome PICTURE "@!"
   @ 7,5 SAY "Informe o tipo (F ou J): " GET cTipo ;
                  PICTURE "!" VALID cTipo $ "FJ"
   @ 9,5 SAY "Informe o CPF : " GET cCPF ;
                  PICTURE "@R 999.999.999-99" WHEN (cTipo) == "F"
   @ 11,5 SAY "Informe o CNPJ : " GET cCnpj ;
                  PICTURE "@R 9999999999-9999" WHEN (cTipo) == "J"
   READ

   ? cNome
   IF (cTipo) == "F"
      ? "CPF : " , cCPF
   ELSE
      ? "CNPJ: " , cCNPJ
   ENDIF

RETURN
```

O exemplo a seguir simula a execução do código 11.14 levando em conta que o usuário digitou tudo com a tecla CAPS-LOOK desligada. Mas como isso foi possível, se as letras apareceram todas em maiúsculas ? Isso só foi possível porque o símbolo “!” contido na máscara força a entrada dos respectivos caracteres em letras maiúsculas.

.:Resultado:.

```
     Informe o nome :  [ITAMAR LINS         ]

     Informe o tipo (F ou J):  [F]

     Informe o CPF :  [789.000.122-20]

     Informe o CNPJ :  [          -    ]



ITAMAR LINS
CPF :  78900012220
```

Além disso o símbolo “9” obriga a entrada de somente números e o símbolo @R não deixa que os pontos e traços sejam gravados na variável, por isso o CPF foi atribuído a variável sem os pontos ou traços, apesar de ser digitado com os pontos e traços da máscara. Essa é uma pequena amostra das máscaras do @ ... GET. O Harbour possui vários símbolos para a composição de máscaras de entradas de dados, iremos abordar alguns nas sub-seções a seguir. [8]

11.9.1 Convertendo caracteres alfabéticos para maiúsculo

A máscara “!” converte um caractere alfabético para maiúsculo, conforme o exemplo a seguir :

Listagem 11.15: @ ... GET
Fonte: codigos/get09.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cNome := SPACE(20)
LOCAL nIdade := 0
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o nome : " GET cNome PICTURE "!"
   @ 7,5 SAY "Idade : " GET nIdade
   READ

   ? cNome, nIdade

RETURN
```

[8] Se você não entendeu o exemplo anterior não se preocupe pois eles serão abordados detalhadamente.

.:Resultado:.

```
     Informe o nome :  [J]

     Idade :  [23        ]



J                          23
```

O exemplo acima possui algumas características interessantes, a primeira delas é que o símbolo “!” converte apenas um caractere para maiúscula e omite os demais da edição. Note que a variável cNome permanece com o mesmo tamanho (20 caracteres), pois quando ela foi exibida surgiu um espaço em branco grande entre a letra “J” e o número 23.

A listagem a seguir resolve o problema da falta de espaço, basta repetir várias vezes o símbolo “!”. No exemplo a seguir (listagem 11.16) a variável cNome continua tendo um tamanho de 20 caracteres, mas nós só poderemos usar 5 para digitar algo. Isso acontece porque o símbolo “!” limita a quantidade de caracteres de acordo com a quantidade de símbolos digitados, como apenas cinco “!” foram digitados então o tamanho máximo será cinco, embora a variável tenha o tamanho máximo de 20 caracteres.

Listagem 11.16: @ ... GET
Fonte: codigos/get10.prg

```
/*                                                                          1
@ ... SAY ... GET simples                                                   2
*/                                                                          3
PROCEDURE Main                                                              4
LOCAL cNome := SPACE(20)                                                    5
LOCAL nIdade := 0                                                           6
LOCAL GetList := {}                                                         7
                                                                            8
   SET DELIMITERS ON // Exibe os delimitadores                              9
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs                10
   CLS                                                                      11
   @ 5,5 SAY "Informe o nome : " GET cNome PICT "!!!!!"                     12
   @ 7,5 SAY "Idade : " GET nIdade                                          13
   READ                                                                     14
                                                                            15
   ? cNome, nIdade                                                          16
                                                                            17
RETURN                                                                      18
```

.:Resultado:.

```
     Informe o nome :  [JANIO]

     Idade :  [23        ]
```

```
JANIO                          23
```

11.9.2 Máscaras “!” e “@”

Mas, esse método não é muito prático para controlar variáveis com vários caracteres de extensão (já pensou, digitar cinquenta pontos de exclamação para uma variável com cinquenta espaços ?) . Dessa forma foi criado o símbolo “@”. O exemplo a seguir usa o tamanho máximo da variável e aplica o símbolo “!” a todos os caracteres, não importa quantos sejam. Veja o exemplo (listagem 11.17) a seguir:

Listagem 11.17: @ ... GET
Fonte: codigos/get11.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cNome := SPACE(20)
LOCAL nIdade := 0
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o nome : " GET cNome PICTURE "@!"
   @ 7,5 SAY "Idade : " GET nIdade
   READ

   ? cNome, nIdade

RETURN
```

.:Resultado:.

```
     Informe o nome :  [JANIO PROGRAMADOR   ]

     Idade :  [23        ]



JANIO PROGRAMADOR            23
```

Dica 71

Use o símbolo “!” (sem o @) apenas com campos de tamanho fixo, por exemplo “S” ou “N”.

11.9.3 Aceitando somente dígitos

A máscara “9” restringe a digitação a somente números. Ela pode funcionar ou não em conjunto com o símbolo “@”, da mesma forma que a máscara “!”. A diferença entre “9” e “!” é que “9” só permite números, e “!” transforma o caractere no seu equivalente maiúsculo. O exemplo da listagem 11.18 ilustra o seu funcionamento.

Listagem 11.18: @ ... GET
Fonte: codigos/get12.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cCEP := SPACE(10)
LOCAL nIdade := 0
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o CEP " GET cCEP PICTURE "99999-999"
   @ 7,5 SAY "Idade : " GET nIdade PICTURE "999"
   READ

   ? cCEP, nIdade

RETURN
```

.:Resultado:.

```
60014-140   46
```

Observe que, quando a variável é caractere, você pode incluir um símbolo na máscara e ele automaticamente fará parte da variável. No nosso exemplo, a variável cCEP recebeu o traço separador também. Porém, dependendo da situação, você pode querer o traço apenas para facilitar a digitação, ou seja, você não quer armazenar o traço na variável cCEP. Nesse caso, você deve acrescentar um “R” a sua máscara de CEP, ficando assim : “@R 99999-999”. O exemplo da listagem 11.19 ilustra essa situação.

Listagem 11.19: @ ... GET
Fonte: codigos/get13.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cCEP := SPACE(10)
LOCAL cFone := SPACE(10)
LOCAL GetList := {}

   SET DELIMITERS ON // Exibe os delimitadores
```

```
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o CEP " GET cCEP PICT "@R 99999-999"
   @ 7,5 SAY "Telefone : " GET cFone PICT "@R (999) 9999-9999"
   READ

   ? cCEP, cFone

RETURN
```

.:Resultado:.

```
     Informe o CEP  [60015-023]

     Telefone :  [(085) 3456-3421]


60015023   08534563421
```

11.9.4 Incluindo um valor padrão

Caso você precise incluir um valor padrão no seu campo de formulário, ou seja, um valor previamente definido, você pode usar a máscara “@K” (tem que ter o arroba antecedendo o K). Acompanhe o exemplo da listagem 11.20

Listagem 11.20: @ ... GET
Fonte: codigos/get14.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL dDataIni := DATE()
LOCAL dDataFim := DATE()
LOCAL cPedNot := "PEDIDO"
LOCAL GetList := {}

   SET DATE BRITISH // Exibe dadas no formado dd/mm/aa
   SET CENTURY ON // Ativa a exibição do ano com quatro dígitos.
   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe a data inicial: " GET dDataIni PICTURE "@K"
   @ 7,5 SAY "Informe a data final  : " GET dDataFim

   @ 9,5 SAY "Digite 'PEDIDO' para pedido ou 'NOTA' para notas fiscais : " GET
   READ

   ? dDataIni, dDataFim, cPedNot
```

```
                                                                    21
RETURN                                                              22
```

.:Resultado:.

```
    Informe a data inicial:  [02/  /    ]

    Informe a data final  :  [01/10/2016]

    Digite 'PEDIDO' para pedido ou 'NOTA' para notas fiscais :
  [PEDIDO]
```

Em primeiro lugar, veja que as variáveis foram declaradas e inicializadas com seus respectivos valores padrão. Para que serve a máscara "@K", se as variáveis já possuem valores ? Bem, essa máscara apaga o conteúdo padrão caso o usuário digite algo. No exemplo acima a data inicial veio preenchida com a data de hoje, mas o usuário resolveu alterar a data. Quando ele teclou "0" (zero) automaticamente todo o restante foi apagado para que uma nova data seja digitada. Já no campo "data final" a máscara "@K" não foi definida, dessa forma, quando o usuário for alterar a data, o restante do campo não será apagado, já que não tem máscara "@K". Isso serve para ilustrar que nem sempre essa máscara deverá ser usada, afinal de contas, se o usuário quiser apenas alterar o dia de uma data sugerida, a máscara "@K" irá lhe prejudicar, pois lhe obrigará a redigitar o mês e o ano. No terceiro campo, a máscara "@K" trará ganhos de produtividade durante a digitação, porque se o usuário quiser digitar "NOTA" ele não precisará apagar as duas últimas letras da palavra "PEDIDO".

11.9.5 Alterando o separador de decimais

Já vimos, no início do nosso estudo, que as variáveis numéricas não aceitam a vírgula como separador decimal. Pois bem, a máscara "@E" tem o poder de inverter automaticamente a posição de pontos e vírgulas durante a exibição da variável dentro do GET. Ou seja, você pode digitar uma vírgula para separar decimais, mas a variável receberá automaticamente o simbolo "ponto" que é o separador correto. A listagem 11.21 ilustra o uso de "@E":

Listagem 11.21: @ ... GET
Fonte: codigos/get15.prg

```
/*                                                                  1
@ ... SAY ... GET simples                                           2
*/                                                                  3
PROCEDURE Main                                                      4
LOCAL nSalario := 0                                                 5
LOCAL nDesconto := 0                                                6
LOCAL GetList := {}                                                 7
                                                                    8
   SET DATE BRITISH // Exibe dadas no formado dd/mm/aa              9
   SET CENTURY ON // Ativa a exibição do ano com quatro dígitos.    10
   SET DELIMITERS ON // Exibe os delimitadores                      11
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs        12
   CLS                                                              13
```

```
   @ 5,5 SAY "Salário  : " GET nSalario PICTURE "@E 999,999.99"
   @ 7,5 SAY "Desconto : " GET nDesconto PICTURE "@E 99999.9999"
   READ

   ? nSalario, nDesconto

RETURN
```

.:Resultado:.

```
    Salário  :  [  5.755,75]

    Desconto :  [    7,9276]


  5755.75     7.9276
```

Veja que o usuário usa a vírgula para separar decimais, mas a variável é gravada com o ponto separador, que é o correto.

11.9.6 Digitando um valor maior do que o espaço disponível

Existem casos em que o espaço disponível para a digitação é inferior ao tamanho da variável. Por exemplo: geralmente o nome completo de uma pessoa não passa de trinta caracteres, mas existem nomes grandes que podem chegar a até cem caracteres. Temos então um problema de espaço, pois isso nos obrigaria a ter um campo com tamanho 100 que raramente seria preenchido completamente, levando a um desperdício de espaço. A solução para esse problema está na máscara “@S<n>”. Essa máscara reduz o tamanho do campo no formulário, mas não reduz o tamanho da variável. Assim, quando o usuário começar a digitar uma quantidade grande de caracteres, o campo fará automaticamente uma rolagem horizontal dentro do GET. O <n> é um inteiro que especifica a largura do GET. Vamos ao exemplo :

Listagem 11.22: @ ... GET
Fonte: codigos/get16.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL cNome := SPACE(100)
LOCAL GetList := {}

   SET DATE BRITISH // Exibe dadas no formado dd/mm/aa
   SET CENTURY ON // Ativa a exibição do ano com quatro dígitos.
   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe o nome : " GET cNome PICTURE "@S30"
   READ
```

```
                                                                    15
   ? cNome                                                          16
                                                                    17
RETURN                                                              18
```

Na simulação abaixo o nome a ser digitado é “Roberdo de Araújo do Nascimento Pereira”. Note que no início nada de mais acontece:

.:Resultado:.

```
     Informe o nome :  [Roberto de Araújo            ]
```

Mas quando avançamos na digitação o GET faz uma rolagem horizontal graças a máscara “@S”.

.:Resultado:.

```
     Informe o nome :  [aújo do Nascimento Pereira    ]
```

Ao final a variável é exibida com o conteúdo completo:

.:Resultado:.

```
     Informe o nome :  [aújo do Nascimento Pereira    ]

Roberto de Araújo do Nascimento Pereira
```

11.9.7 Resumo das máscaras apresentadas

A seguir temos uma lista das máscaras que foram apresentadas:

- “!” : Converte um caractere para maiúsculo.
- “!!” : Converte 2 caracteres para maiúsculo, e assim sucessivamente.
- “@!” : Converte N caracteres para maiúsculo.
- “9”: Restringe a somente números. Possui o funcionamento análogo ao “!”.
- “K” : Elimina texto assumido caso um caractere qualquer seja digitado.
- “R” : Posso inserir caracteres, tipo ponto ou traço, mas não serão guardados na variável.
- “S” : Permite a rolagem horizontal no GET quando a variável possuir muitos espaços.

Existem outras máscaras, mas o seu estudo tornaria o capítulo muito extenso. Uma explicação detalhada do sistema de máscaras você pode encontrar em `http://www.ousob.com/ng/53guide/ng12bdbd.php`.

11.10 Limitando a edição de datas e variáveis com RANGE

A cláusula RANGE limita a edição de datas e números especificando valores mínimo e máximo aceitáveis. Digite primeiro o mínimo e só depois o máximo. O exemplo da listagem 11.23 ilustra o uso do RANGE com números e datas.

Listagem 11.23: @ ... GET
Fonte: codigos/range.prg

```
/*
@ ... SAY ... GET simples
*/
PROCEDURE Main
LOCAL nNota := 0
LOCAL dData := DATE()
LOCAL GetList := {}

   SET DATE BRITISH // Exibe dadas no formado dd/mm/aa
   SET CENTURY ON // Ativa a exibição do ano com quatro dígitos.
   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe a nota : " GET nNota RANGE 0 , 10

   @ 7,5 SAY "Informa a data da prova : " GET dData RANGE DATE()-10, DATE()+10
   READ

   ? nNota, dData

RETURN
```

No exemplo a seguir o usuário digitou 11. O programa exibe uma mensagem e não permite que o campo “nota” seja abandonado.

.:Resultado:.

```
                        Range: 0 - 10




    Informe a nota :  [        11]

    Informa a data da prova :  [01/10/2016]
```

Note que a cláusula RANGE também aceita expressões, como DATE()-10 e DATE()+10. No exemplo a seguir o usuário digitou uma data que está fora do intervalo definido em RANGE. Essa simulação foi feita no dia primeiro de outubro de 2016, por isso a mensagem de erro reflete o intervalo desse dia.

.:Resultado:.

```
                    Range: 21/09/2016 - 11/10/2016
```

```
Informe a nota :  [        10]

Informa a data da prova :  [01/10/2015]
```

Outro destaque, que você deve ter notado, é uma mensagem de erro que aparece no topo da tela, na linha zero. A linha zero recebe o nome de SCOREBOARD. E é usado pelo Harbour para exibir mensagens diversas, como o pressionamento ou não da tecla Insert (Ins). Caso você não queira exibir essas mensagens você pode usar o SET abaixo :

```
SET SCOREBOARD OFF
```

O valor padrão do SCOREBOARD é ON. Nesse caso o READ exibe mensagens para a RANGE inválida, datas inválidas, o estado da tecla insere (INS) e também mensagens de uma função chamada MEMOEDIT(), que será vista posteriormente.

11.11 Trabalhando com cores

O sistema padrão de cores do Harbour reflete o panorama das tecnologias vigentes na época do dBase e do Clipper. Quando o dBase foi lançado o padrão de cores dos monitores era apenas verde e preto (fósforo verde). O padrão VGA ainda não havia sido criado e o padrão corrente chamava-se CGA. Quando o padrão VGA foi lançado começaram a surgir os primeiros monitores coloridos, e o dBase evoluiu para acompanhar essa inovação da época. O Harbour, portanto, consegue reproduzir um número limitado de cores. Mesmo esse número sendo limitado, nós veremos que isso é mais do que suficiente para uma interface modo texto.

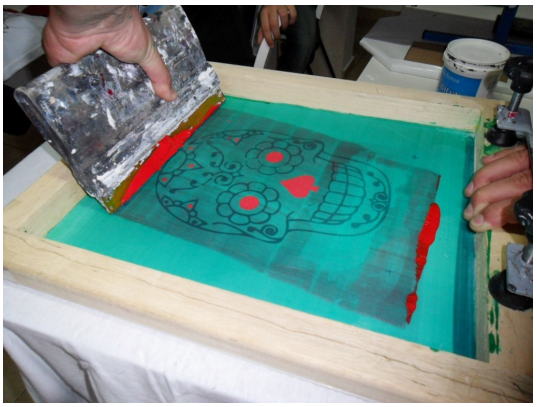
Essa seção trata do sistema de cores do Harbour, mas nós não iremos esgotar o assunto. O objetivo aqui é fazer com que você compreenda que existe um sistema de cores nas interfaces modo texto através de exemplos simples. Vamos começar revendo o sistema de coordenadas do Harbour: cada coordenada corresponde a um quadrado conforme você já viu. Cada quadrado desses comporta um e apenas um caractere, de modo que cada caractere tem a sua própria coordenada. Até aí tudo bem. Vamos agora acrescentar mais uma informação: cada caractere é como se fosse um carimbo, inclusive a origem da palavra caractere remonta as antigas máquinas tipográficas, conforme a figura 11.13.

Figura 11.13: Caracteres tipográficos

Portanto, quando o Harbour imprime o caractere “a”, por exemplo, ele está imprimindo o “quadrado” inteiro, não apenas o caractere “a” isolado. O detalhe está na cor de fundo do quadrado, que é a mesma que a cor de fundo padrão da tela. Tenha sempre essa imagem em mente quando for trabalhar com cores no Harbour, dessa forma, existem sempre duas cores para cada “quadrado”: a cor do caractere em si e a cor de fundo do “quadrado” (background).

Agora vamos utilizar outra analogia: a imagem a seguir (figura 11.14) nos mostra uma etapa na confecção de uma estampa.

Figura 11.14: Serigrafia



Note que o profissional utiliza-se de “telas” (ou camadas) para obter o efeito colorido. Dessa forma, cada camada corresponde a uma cor (se você já ouviu falar das “camadas” usadas pelo software Photoshop você deve saber do que estamos falando). Não basta, portando aplicar a cor, precisa também fazer a aplicação da cor na camada correta para que o colorido fique conforme o esperado.

Em resumo: temos duas cores para cada elemento (a cor do elemento e a cor de fundo) e temos camadas onde essas cores são aplicadas. O Harbour possui um total de cinco níveis de camadas, conforme a lista a seguir :

1. Padrão: é a camada mais usada. Ela corresponde a toda a tela do vídeo, incluindo a utilização de comandos e funções. Isso inclui comandos como o @ ... PROMPT, @ .. SAY ... GET, etc.

2. Destaque : é a cor utilizada para a exibição de barras luminosas, como o GET onde o usuário esta digitando, o ítem atualmente selecionado no menu @ ... PROMPT e outras ainda não vistas.

3. Borda : Essa configuração não é usada mais. Ela corresponde a uma região inacessível da tela, na verdade o contorno dela. Nos monitores modernos essa região não existe mais.

4. Fundo : Essa configuração também não é usada mais.

5. Não selecionado: Corresponde ao par de cores utilizado para configurar os GETs não selecionados.

Na prática temos somente três camadas: padrão (a mais usada), destaque (barra luminosa de PROMPT e GET selecionado) e não selecionado (GETs não selecionados). Guarde a ordem com que elas foram descritas: temos, portanto as camadas 1 (um), 2 (dois) e 5 (cinco).

11.11.1 Códigos das cores

O Harbour possui dezenove códigos de cores, conforme a tabela 11.1[9]

Tabela 11.1: Código de cores

Cor desejada	**Código**
Preto	N
Azul	BG
Verde	G
Ciano	BG
Vermelho	R
Magenta	RB
Marrom	GR
Branco	W
Cinza	N+
Azul Brilhante	B+
Verde Brilhante	G+
Ciano Brilhante	BG+
Vermelho Brilhante	R+
Magenta Brilhante	RB+
Amarelo	GR+
Branco Brilhante	W+
Preto	U
Vídeo Inverso	I
Incolor	X

11.11.2 O SET COLOR

Lembra das variáveis especiais que são definidas através do SET ? Pois bem, o Harbour possui uma variável especial chamada COLOR que serve para aplicar um par

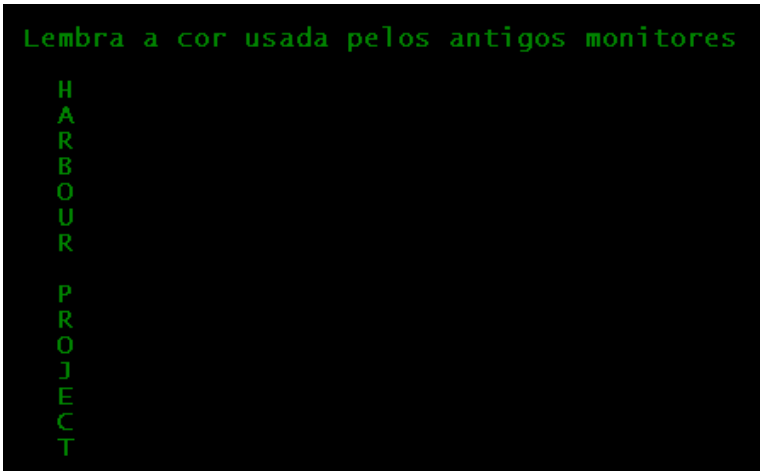
[9]Tabela retirada de `http://harbour.edu.pl/clipper/pt/ng56a80.html` em 08-Out-2016.

de cor a uma determinada camada. A ordem com que eu apresento o par de cores representa a camada. Basta obedecer a ordem descrita na descrição das camadas. Lembre-se que você só vai precisar usar as camadas 1, 2 e 5. Por exemplo, se nós queremos a cor do caractere verde e o fundo preto devemos configurar SET COLOR assim :

Listagem 11.24: Cores no Harbour
Fonte: codigos/cor01.prg

```
/*
CORES
*/
PROCEDURE Main
LOCAL cImprime := "HARBOUR PROJECT"
LOCAL cCaracter

   SET COLOR TO "G/N"
   ?
   ? "   Lembra a cor usada pelos antigos monitores"
   ?
   FOR EACH cCaracter IN cImprime
      ? "    ", cCaracter
   NEXT
   ?

RETURN
```

Figura 11.15: Cores no Harbour



O exemplo a seguir configura uma tela de entrada com GETs coloridos.

Listagem 11.25: Cores no Harbour
Fonte: codigos/cor02.prg

```
/*
CORES
*/
PROCEDURE Main
```

```
LOCAL cProjeto := "SISTEMA PARA CONTROLE DE FROTAS     "
LOCAL cResponsavel := SPACE(20)
LOCAL nValor := 0
LOCAL GetList := {}
                /*Padrão , Get ativo,,, Get Inativo*/
   SET COLOR TO "BG+/N, N/W,,,W+/GR+"

   CLS

   @ 8,8 TO 16,75 DOUBLE

   SET DELIMITERS ON
   SET DELIMITERS TO "[]"

   @ 10,10 SAY "Nome do projeto     : " GET cProjeto PICTURE "@K"
   @ 12,10 SAY "Nome do responsável : " GET cResponsavel

   @ 14,10 SAY "Valor do projeto    : " GET nValor PICTURE "@E 999,999.99"
   READ


RETURN
```

Figura 11.16: Cores no Harbour

O próximo exemplo aplica cores a um menu com @ ... PROMPT.

Listagem 11.26: Cores no Harbour
Fonte: codigos/cor03.prg

```
/*
CORES
*/
PROCEDURE Main
LOCAL nOpc // Opção selecionada

                /*Padrão , Barras luminosas*/
   SET COLOR TO "B/W, W+/B+"

   CLS
```

```
   @ 8,8 TO 16,25 DOUBLE

   @ 10,10 PROMPT "   Cadastros   "
   @ 12,10 PROMPT "   Relatórios  "
   @ 14,10 PROMPT "      Sair     "
   MENU TO nOpc

   SWITCH nOpc
   CASE 1
      ? "Cadastros"
      EXIT
   CASE 2
      ? "Relatórios"
      EXIT
   CASE 0
   CASE 3
   OTHERWISE
      ? "Sair"
   END



RETURN
```

Figura 11.17: Cores no Harbour

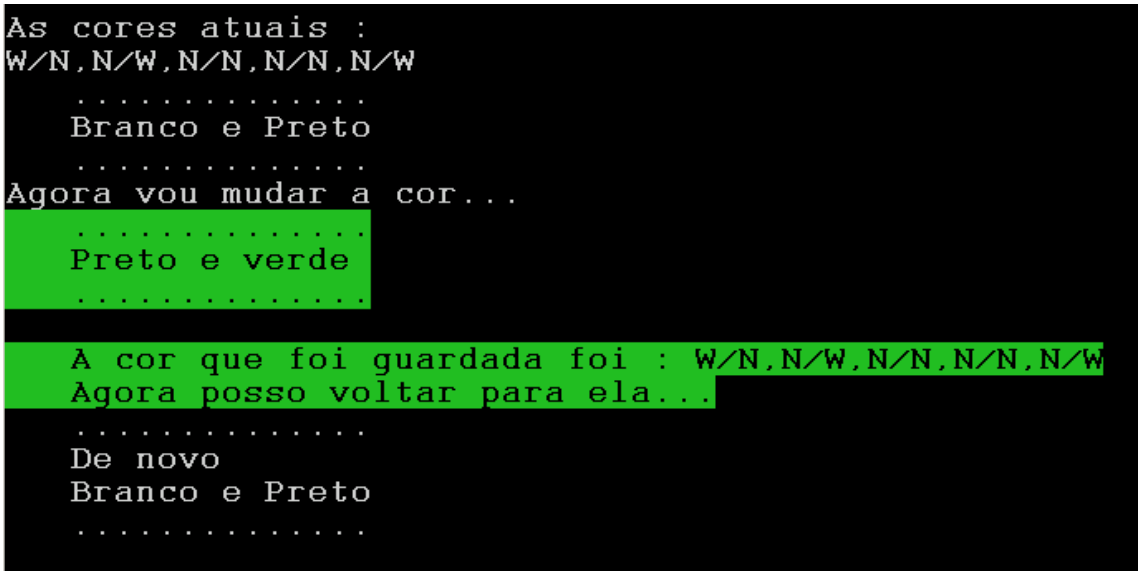
11.11.3 A função SETCOLOR()

A função SETCOLOR tem uma vantagem sobre o SET COLOR: ela retorna a configuração atual de cores. Isso fica mais fácil se nós desejamos gravar o estado anterior para restaurar depois. O exemplo da listagem 11.18 ilustra o uso da função SETCOLOR().

Listagem 11.27: Cores no Harbour
Fonte: codigos/cor04.prg

```
/*
SETCOLOR
*/
PROCEDURE Main
LOCAL cCorAnterior


   CLS
   ? "As cores atuais :"
   ? SETCOLOR()
   ? "   ............."
   ? "   Branco e Preto"
   ? "   ............."
   ? "Agora vou mudar a cor..."
   cCorAnterior := SETCOLOR( "N/G" )
   ? "   ............."
   ? "   Preto e verde "
   ? "   ............."
   ?
   ? "   A cor que foi guardada foi : " + cCorAnterior
   ? "   Agora posso voltar para ela..."
   SetColor( cCorAnterior )
   ? "   ............."
   ? "   De novo       "
   ? "   Branco e Preto"
   ? "   ............."
   ?
   ?

RETURN
```

Figura 11.18: Cores no Harbour



Uma excelente discussão sobre as cores pode ser acompanhada no fórum PCToledo

em `http://www.pctoledo.com.br/forum/viewtopic.php?f=1&t=13559`.

11.12 O cursor

O cursor é uma marca registrada das interfaces caracteres, trata-se de um quadrado (pode ser meio quadrado ou uma barrinha) que fica piscando na tela para lhe informar onde será escrito o próximo comando. As interfaces gráficas também tem os seus cursores, mas é na tradicional interface modo texto que o cursor tem um papel maior. Enquanto que nas interfaces gráficas o cursor serve apenas para informar onde será impresso o próximo caractere digitado, no modo texto o cursor também serve para informar onde será desenhado a próxima tela, retângulo, etc. Como você já deve ter percebido, as interfaces modo texto utilizam um mesmo mecanismo para exibir seus dados e também para desenhar sua própria interface. É diferente da interface gráfica, que diferencia os dados das janelas. O Harbour possui alguns comandos que controlam a exibição do cursor, a aparência do mesmo e também a sua posição. Vamos enumerar alguns a seguir :

A função COL() informa a coluna onde o cursor se encontra.

Descrição sintática 14

1. Nome : COL
2. Classificação : função.
3. Descrição : Retorna a posição da coluna do cursor na tela.
4. Sintaxe

```
COL() -> nCol
```

Fonte : [Nantucket 1990, p. 5-41]

Da mesma forma, o Harbour também possui a função ROW(), que retorna a posição da linha corrente.

Descrição sintática 15

1. Nome : ROW
2. Classificação : função.
3. Descrição : Retorna a posição da linha do cursor na tela.
4. Sintaxe

```
ROW() -> nRow
```

Fonte : [Nantucket 1990, p. 5-200]

O comando SET CURSOR serve para exibir ou esconder o cursor.

Descrição sintática 16

1. Nome : SET CURSOR
2. Classificação : comando.
3. Descrição : Torna o cursor visível (ON) ou invisível (OFF)
4. Sintaxe

```
SET CURSOR ON | OFF | <lComuta>

   ON : Cursor visível
   OFF : Cursor invisível
   <lComuta> : Valor lógico verdadeiro ou falso que deve ser info
               entre parênteses. Se for .t. exibe o cursor, e se
               esconde o cursor.
```

Fonte : [Nantucket 1990, p. 4-111]

11.13 Exibição de dados com @ ... SAY

Se você entendeu o comando @ ... GET não terá problemas com o comando @ ... SAY. Ele funciona de maneira muito semelhante ao comando @ ... GET, mas só serve para exibir informações em locais pré-determinados, sem permitir alterações. A consequência disso é que as cláusulas que controlam a entrada de dados, como o VALID, WHEN e RANGE não se aplicam ao comando SAY. A sintaxe do comando SAY é a seguinte :

Descrição sintática 17

1. Nome : @ ... SAY
2. Classificação : comando.
3. Descrição : Exibe dados em uma linha e coluna especificadas.
4. Sintaxe

```
@ <nlin>,<ncol> [SAY <exp> ;
[PICTURE <cPicture>] [COLOR <cColor>]  ]
```

Fonte : [Nantucket 1990, p. 4-12]

11.13.1 A cláusula COLOR

A cláusula COLOR é nova para nós, pois o comando @ ... GET não a possui. Essa cláusula serve para atribuir uma cor em especial a apenas o SAY que a possui, por exemplo: suponha que você quer chamar a atenção para um campo de preenchimento obrigatório. Você pode atribuir uma cor vermelha a ele, enquanto os demais permanecem com a cor branca.

11.13.2 A cláusula PICTURE

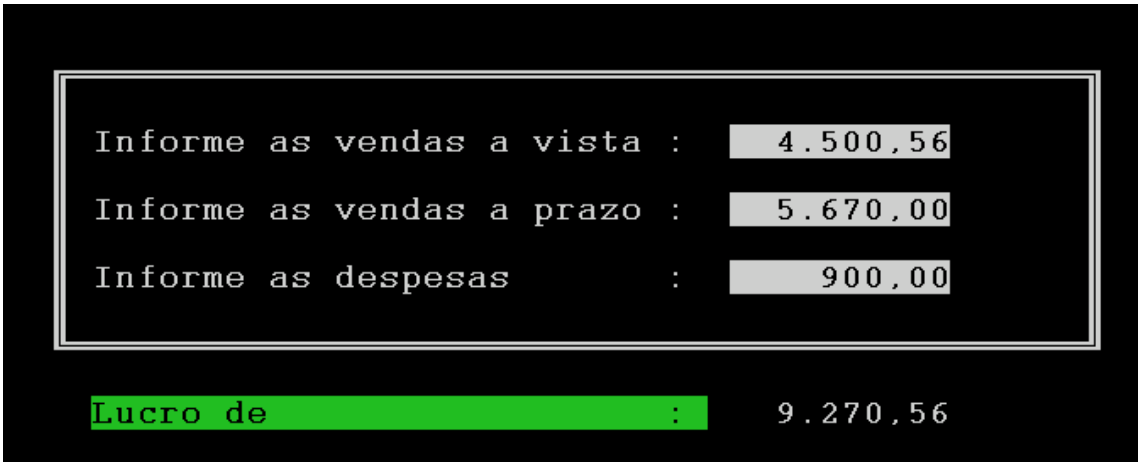
O funcionamento da cláusula PICTURE é exatamente igual ao comando @ ... GET, mas você pode estar com dificuldades para arranjar uma aplicação para ela. Bem, a cláusula PICTURE geralmente é usada para formatar o resultado de uma operação, ou para exibir campos que são somente para consulta. O exemplo da listagem 11.28 ilustra o uso da cláusula COLOR e PICTURE com o comando SAY :

Listagem 11.28: Máscaras e Cores no Harbour
Fonte: codigos/say01.prg

```
/*
CORES
*/
#define PREJUIZO "N/R" // Preto / Vermelho
#define LUCRO "N/G" // Preto / Verde
PROCEDURE Main
LOCAL nAVista := 0
LOCAL nAPrazo := 0
LOCAL nDespesas := 0
LOCAL nResultado
LOCAL GetList := {}

                /*Padrão , Barras luminosas*/
   SET COLOR TO "W/N" // Branco / Preto

   CLS
```

```
                                                                    17
   @ 8,8 TO 16,55 DOUBLE                                            18
                                                                    19
   @ 10,10 SAY "Informe as vendas a vista : " GET nAVista ;         20
               PICTURE "@E 999,999.99" RANGE 0, 999999.99           21
   @ 12,10 SAY "Informe as vendas a prazo : " GET nAPrazo ;         22
               PICTURE "@E 999,999.99" RANGE 0, 999999.99           23
   @ 14,10 SAY "Informe as despesas       : " GET nDespesas ;       24
               PICTURE "@E 999,999.99" RANGE 0, 999999.99           25
   READ                                                             26
                                                                    27
   nResultado := ( nAVista + nAPrazo ) - nDespesas                  28
   IF nResultado > 0                                                29
      @ 18,10 SAY "Lucro de                  : " COLOR LUCRO        30
   ELSEIF nResultado < 0                                            31
      @ 18,10 SAY "Prejuízo de               : " COLOR PREJUIZO     32
   ELSE                                                             33
      @ 18,10 SAY "Resultado neutro          : "                    34
   ENDIF                                                            35
   @ 18,COL()+1 SAY nResultado PICTURE "@E 999,999.99"              36
   ?                                                                37
   ? "Fim"                                                          38
   ?                                                                39
                                                                    40
                                                                    41
RETURN                                                              42
```

Figura 11.19: Cláusula COLOR e PICTURE



Você deve ter percebido a função COL() descrita anteriormente. Ela foi usada para poder imprimir o valor logo após o comando @ ... SAY que informa se foi lucro, prejuízo ou neutro.

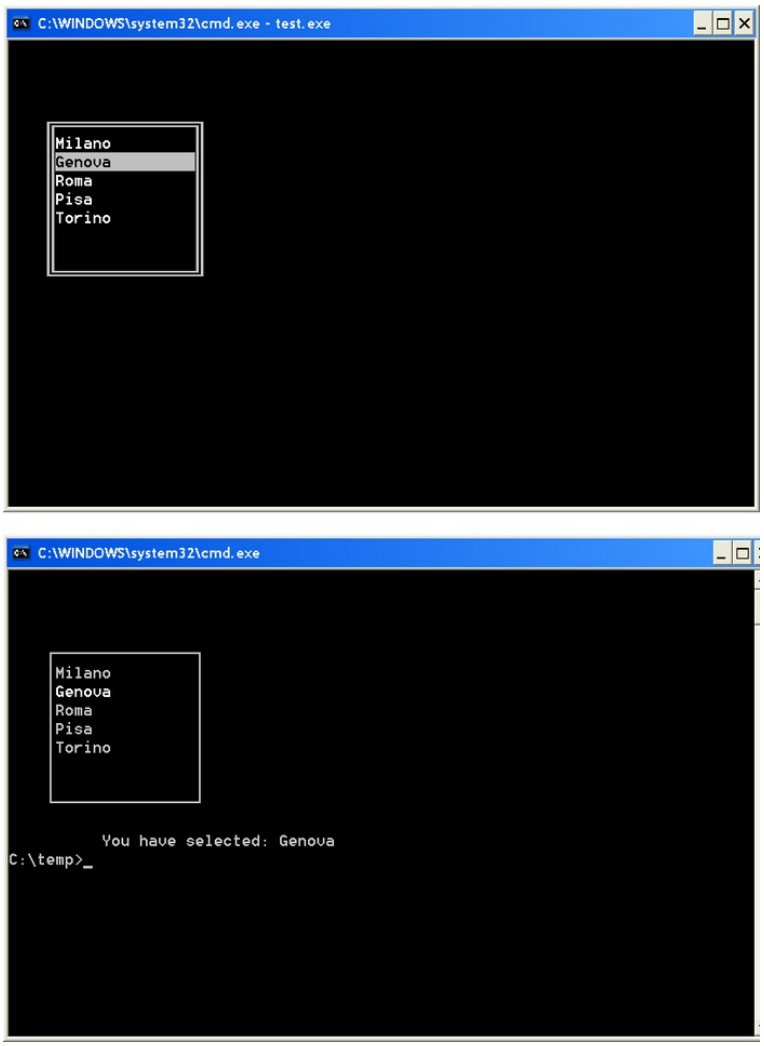
11.14 O comando LISTBOX

Veremos também uma pequena introdução ao comando @ ... LISTBOX. As seguintes imagens e explicações dessa seção foram retiradas do excelente tutorial de Giovani Di Maria, em `http://www.elektrosoft.it/tutorials/harbour-how-to/harbour-how-asp#listbox`. Elas nos mostram alguns exemplos de uso do LISTBOX e o código está na listagem 11.29.

Listagem 11.29: Listbox
Fonte: codigos/listbox.prg

```
PROCEDURE Main()
LOCAL GetList := {}
LOCAL cCity



   CLS
   cCity = "Milano"

   @ 4, 4, 12, 20 GET cCity ;
         LISTBOX { "Milano", "Genova", "Roma", "Pisa", "Torino" }
   READ

   @ 14, 10 SAY "Você selecionou: " + cCity

RETURN
```

Figura 11.20: LISTBOX



Esse GET é especial porque ele **não** admite a possibilidade de um SAY. Você **NÃO** pode fazer conforme a listagem abaixo :

```
@ 4, 4, 12, 20 SAY "Informe a cidade : " ;
GET cCity LISTBOX { "Milano", "Genova", "Roma", "Pisa", "Torino" }
```

Isso irá gerar um erro de compilação. Também não use VALIDs nem qualquer outra cláusula de GET.

Outra observação importante diz respeito as chaves que delimitam a lista de cidades. Essa estrutura chama-se ARRAY, VETOR ou MATRIZ. Ela será abordada futuramente nas estruturas de dados compostas. Mas a ideia é simples e intuitiva: abra com uma chave, insira os valores separados por vírgula e feche com o outro par da chave.

11.15 Usos do Mouse em aplicações modo texto

Para ativar o mouse basta você incluir um SET específico no início do seu código. Essa característica é outra novidade do Harbour e não está presente nas linguagens

antigas, como Clipper e dBase. A listagem 11.30 é uma alteração da listagem 11.20 e exemplifica o procedimento.

Listagem 11.30: Mouse
Fonte: codigos/mouse.prg

```
/*
MOUSE
*/
#include "inkey.ch"
PROCEDURE Main
LOCAL dDataIni := DATE()
LOCAL dDataFim := DATE()
LOCAL cPedNot := "PEDIDO"
LOCAL GetList := {}

   // Configuraï¿½ï¿½o do mouse
   SET EVENTMASK TO INKEY_ALL // Mouse


   SET DATE BRITISH // Exibe dadas no formado dd/mm/aa
   SET CENTURY ON // Ativa a exibição do ano com quatro dígitos.
   SET DELIMITERS ON // Exibe os delimitadores
   SET DELIMITERS TO "[]" // Cria delimitadores para os GETs
   CLS
   @ 5,5 SAY "Informe a data inicial: " GET dDataIni PICTURE "@K"
   @ 7,5 SAY "Informe a data final  : " GET dDataFim
   @ 9,5 SAY ;
      "Digite 'PEDIDO' para pedido ou 'NOTA' para notas fiscais : " ;
      GET cPedNot PICTURE "@K"
   READ

   ? dDataIni, dDataFim, cPedNot

RETURN
```

Para inserir o mouse na sua aplicação você precisa somente :

1. Incluir o arquivo de cabeçalho “inkey.ch” antes de main. Linha 4 da listagem.
2. Incluir “SET EVENTMASK TO INKEY_ALL” nas primeiras linhas de main. Linha 12 da listagem.

A partir daí o programa incluirá o mouse em alguns controles, como o @ ... GET e o @ ... PROMPT.

Dica 72

Eu, particularmente acredito que a inclusão do mouse em interfaces caracteres pode ser bom para o programador e para o usuário, desde que o programador deixe claro que as interfaces modo texto nunca serão iguais a uma interface gráfica do tipo Windows. Por exemplo: você não pode arrastar, nem copiar e nem colar. Além disso as interfaces modo texto caracterizam-se pela "linearidade" no seu comportamento, e as interfaces gráficas (estilo Windows) caracterizam-se por uma forma de programar completamente diferente, conhecida pelo nome de "programação orientada a eventos". Se o seu cliente estiver ciente dessas limitações, o que algumas vezes não acontece, você pode usar o mouse sem problemas.
Portanto, se você quiser apresentar o mouse em interface modo texto ao seu cliente faça isso, mas não esqueça também de apresentar também as limitações e deixar tudo isso bem claro. Não tente disfarçar sua "interface modo texto" de "interface windows" pois com o passar do tempo o seu cliente vai se sentir enganado.

11.16 Salvando e restaurando telas

A medida que o seu sistema vai ficando mais complexo, ele vai criando várias funcionalidades, tais como : formulários, rotinas de configuração e cálculo, relatórios, grids de dados, etc. Com isso surge a necessidade de guardar a tela anterior para poder restaurar depois. O Harbour possui dois comandos para esse fim e duas funções também, você poderá usar esses recursos sempre que quiser restaurar a aparência de uma tela anterior que foi "encoberta".

11.16.1 Os comandos SAVE SCREEN TO e RESTORE SCREEN FROM

O funcionamento desses comandos é bem simples. Para salvar a tela você deve digitar SAVE SCREEN TO <cTela>, onde <cTela> é uma variável que armazenará a tela que foi salva. Para restaurar essa tela salva, você deve usar o comando RESTORE SCREEN FROM <cTela>, onde <cTela> é uma variável que foi usada previamente pelo comando SAVE SCREEN. Ou seja, você não pode usar um comando RESTORE SCREEN se não tiver salvo a tela anteriormente com SAVE SCREEN. Você pode salvar quantas telas quiser, desde que utilize variáveis diferentes para cada tela que você salvou. Caso você use o mesmo nome de variável, a tela salva anteriormente não poderá mais ser restaurada.

Dica 73

Se você já tem alguma experiência com programação WINDOWS ou programação WEB deve achar isso muito estranho, pois no Windows ou no navegador basta a você enviar o comando “fechar” para uma janela e não é precisa restaurar o que ficou sob ela. Nas interfaces modo texto as coisas não funcionam dessa forma pois não existem janelas a serem fechadas, existem telas a serem salvas e restauradas posteriormente. O que estamos querendo dizer é : caso você seja um programador WINDOWS a sua experiência prévia com janelas não servirá muito no ambiente modo texto.

É bom ressaltar que ,quando você restaura uma tela, você está apenas restaurando a imagem que foi sobreposta. Ou seja, o esquema de cores definido por SET COLOR, a aparência do cursor, as coordenadas atuais do cursor e outros elementos da interface modo texto não são redefinidas.

As sintaxes dos SAVE SCREEN TO e do RESTORE SCREEN FROM estão descritas a seguir.

Descrição sintática 18

1. Nome : SAVE SCREEN
2. Classificação : comando.
3. Descrição : Grava a tela corrente num buffer ou variável
4. Sintaxe

```
SAVE SCREEN [TO <cTela>]
```

Fonte : [Nantucket 1990, p. 4-95]

Descrição sintática 19

1. Nome : RESTORE SCREEN
2. Classificação : comando.
3. Descrição : Exibe dados em uma linha e coluna especificadas.
4. Sintaxe

```
RESTORE SCREEN [FROM <cTela>]
```

Fonte : [Nantucket 1990, p. 4-89]

O código a seguir (listagem 11.31) salva a tela, realiza uma operação e depois restaura a tela que foi salva.

Listagem 11.31: Screen
Fonte: codigos/screen.prg

```
/*
SAVE SCREEN E RESTORE SCREEN
*/
PROCEDURE Main
LOCAL cScreen
LOCAL x, y, z

   CLS
   @ 10,10 TO 12,74
   @ 11,11 SAY PADC( "Essa tela será restaura ao final." , 60 )
   SAVE SCREEN TO cScreen
   ?
   WAIT "TECLE ALGO PARA CONTINUAR"
   CLS
   ?
   ? "Aplicaçäo da fórmula"
   ?
   INPUT "Insira o valor de x : " TO x
   INPUT "Insira o valor de y : " TO y
   z = ( 1 / ( x + y ) )
   ? "O valor de z é : " , z
   ?
   WAIT "Tecle algo para restaurar a tela anterior"
   RESTORE SCREEN FROM cScreen


RETURN
```

Repare no uso do comando WAIT que realiza uma pausa e aguarda que o usuário pressione qualquer tecla para continuar. O comando WAIT está descrito a seguir:

Descrição sintática 20

1. Nome : WAIT
2. Classificação : comando.
3. Descrição : Suspende a execução de um programa até que seja pressionada uma tecla. Caso o usuário não digite nada em cTexto o texto exibido será “Press any key to continue”. A variável cValor armazena o valor caractere da tecla pressionada, caso o programador deseje tratar a resposta obtida.
4. Sintaxe

```
WAIT [cTexto] [TO cValor]
```

Fonte : [Nantucket 1990, p. 4-89]

11.16.2 As funções SAVESCREEN() e RESTSCREEN()

As funções SAVESCREEN() e RESTSCREEN() não serão usadas nesse livro, mas são amplamente usadas por programadores. A ideia é simples: em vez de salvar a tela inteira (como faz o comando SAVE SCREEN TO) você pode salvar apenas um pedaço retangular da tela usando a função SAVESCREEN(). Para restaurar você deve usar a função RESTSCREEN(). Por que salvar e restaurar apenas um “pedaço” da tela ? A razão principal é a economia dos recursos de hardware e rede. Isso parece um exagero nos dias de hoje, quando temos fartura de recursos computacionais, mas ainda hoje, em algumas situações, é importante economizar recursos na hora de salvar e restaurar telas. Se você vai desenvolver alguma aplicação que funcione em modo texto e seja acessada remotamente por uma conexão de baixa velocidade (um terminal em uma fazenda remota que acessa o servidor na capital, por exemplo) o uso das funções SAVESCREEN() e RESTSCREEN() trará ganhos significativos na performance da aplicação.

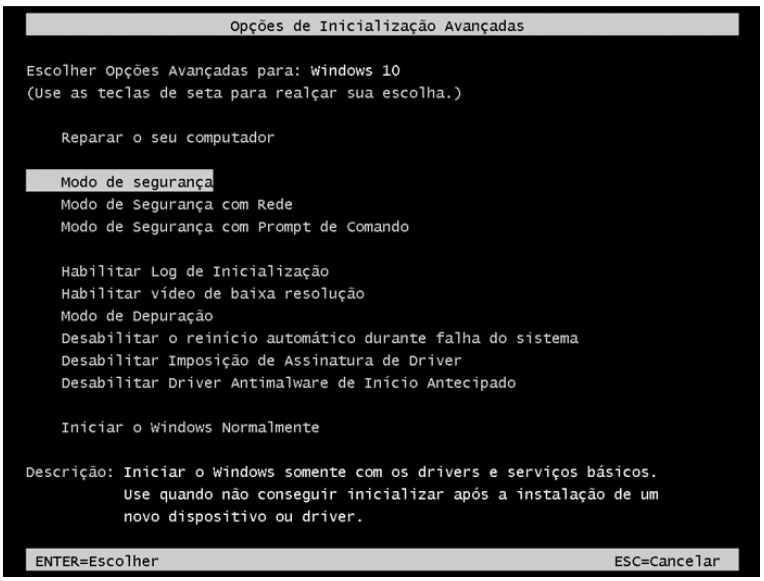
11.17 Exemplos de interface para prática do aprendizado

A seguir nós listamos alguns exemplos de interfaces em modo texto. **Nenhuma delas foi feita em Harbour**, mas você pode tentar reproduzí-las através dos recursos que você aprendeu nesse capítulo.

11.17.1 Menu do windows

Essa figura exibe o clássico menu do windows que é obtido durante o processo de inicialização do sistema. Se você já iniciou o windows pressionando F8 sabe do que estamos falando. Você pode obter um resultado semelhante com o @ ... PROMPT.

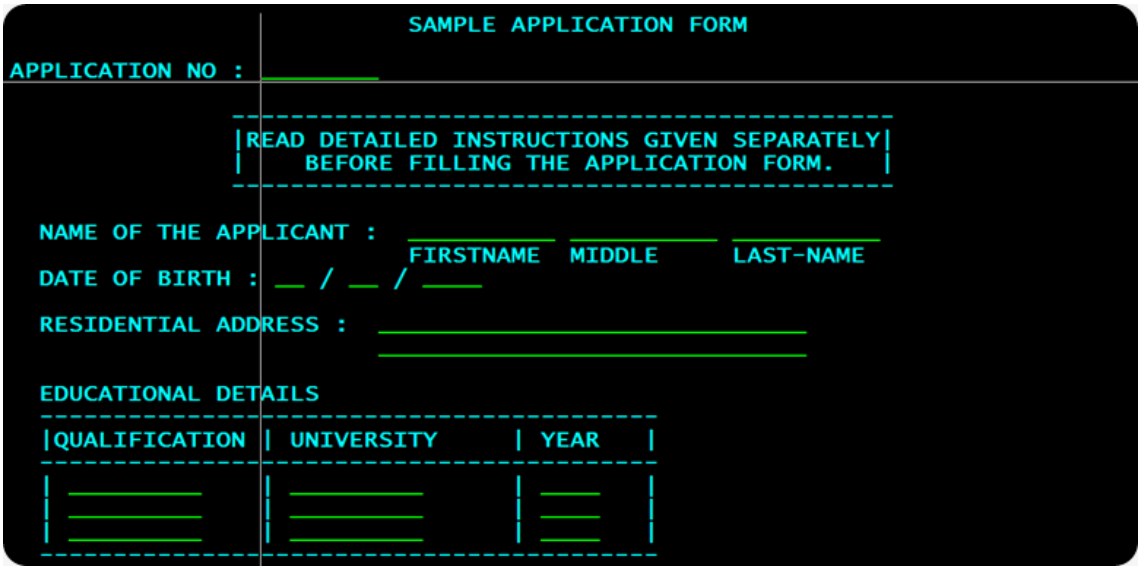
Figura 11.21: Windows modo de segurança



11.17.2 Mainframe Form

A figura 11.22 exibe uma tela de uma aplicação rodando em um mainframe.

Figura 11.22: Mainframe Form



Note que existem alguns elementos que não podem ser reproduzidos na íntegra, como as linhas horizontais verdes que marcam o campo de formulário. Nesse caso você pode usar o SET DELIMITERS e criar seus próprios delimitadores.

12 Modularização: rotinas e sub-rotinas

Divida cada dificuldade em tantas parcelas quanto seja possível e quantas sejam necessárias para resolvê-las.

René Descartes - O discurso do método

Objetivos do capítulo

- Entender o que é modularização.
- Compreender a natureza do problema que a modularização resolve.
- Estabelecer analogias entre a estrutura de um programa e um organograma corporativo.
- Aprender a usar o Harbour para implementar soluções modulares.
- Classificação de variáveis: escopo e tempo de vida.
- Parâmetros e tipos de passagem de parâmetros.
- Entender o que é método Top-Down e suas vantagens.
- Saber o que é refinamento sucessivo.

12.1 O problema que o software enfrenta

Nos fins da década de 1960, o mundo conheceu a sua primeira “crise do software”. Foi uma crise que passou despercebida pela população de um modo geral, afinal de contas, os livros de história nos contam sobre guerras, regimes que caem, epidemias que matam milhares e até mesmo crises econômicas. Mas crise de software é algo desconhecido para a maioria das pessoas. O que é essa tal “crise do software” ? Por que ela ocorreu ? De que ela consistiu ? São perguntas que procuraremos responder nessa nossa rápida introdução.

12.1.1 Problemas na construção do software

Ainda na década de 1960, os custos das atividades de programação aumentavam a cada ano. Naquela época os computadores sofreram uma evolução muito grande, mas o software não conseguia acompanhar esse ritmo. A causa principal disso foi a ausência de uma metodologia que possibilitasse um desenvolvimento com um mínimo de erros. A solução para esse problema, é claro, não estava no desenvolvimento de máquinas mais rápidas, mas na criação de novas técnicas que possibilitem a criação de um software "suficientemente bom"[1]. Em resumo, as ideias que ajudaram a superar a primeira crise de software foram :

1. desenvolvimento de algoritmos por fases ou refinamentos;
2. uso de um número limitado de estruturas de controle;
3. decomposição do código em partes menores chamadas de módulos.

Nesse capítulo nós aprenderemos a modularizar um código, ele fecha um ciclo de técnicas de desenvolvimento que nos auxiliam na criação de um software de qualidade.

12.1.2 A segunda crise do software: problemas na manutenção do código

O conteúdo desse capítulo irá lhe introduzir nas técnicas de modularização de software, de modo que, ao final dele, você irá entender que esse termo não é nenhum “bicho- de-sete-cabeças”. Contudo, nós não queremos lhe iludir. Apenas as técnicas de modularização (que serão apresentadas) não são suficientes para o desenvolvimento de um software complexo. Elas são etapas pelas quais um aprendiz deve necessariamente passar, mas existem outras técnicas que servem para complementar a modularização de software. **Não abordaremos essas técnicas detalhadamente nesse capítulo**, o nosso intuito imediato com essa seção é lhe convencer a não parar o estudo ao final da leitura da primeira parte desse livro. Sim, é claro que você deve praticar o que aprendeu, afinal de contas essas técnicas não deixaram (e nem deixarão) de ser usadas, mas não caia no erro de construir softwares complexos apenas com essas técnicas. Os programas construídos usando os antigos Clipper e dBase são programas modularizados e ainda hoje funcionam perfeitamente em muitas organizações; mas sejamos francos : a programação estruturada, que caracteriza a linguagem Clipper, pode resolver muitos problemas na criação do software, mas já existem técnicas e

[1] O termo "software suficientemente bom"é encontrado nas obras clássicas de Yourdon, não quer dizer "software de baixa qualidade".

recursos que tornam a manutenção do software um obstáculo menos difícil de ser transposto.

Crise de manutenção

Durante a década de 1980 as organizações passaram por outra crise de software, desta vez de uma natureza diferente. A primeira crise aconteceu porque não existiam métodos para a construção de software, mas essa segunda foi ocasionada pela inabilidade em manter o software já construído.

Desenvolver softwares é algo relativamente novo na história da humanidade. As pirâmides antigas foram construídas a milhares de anos, de modo que a Engenharia Civil já superou suas crises e vive agora momentos de estabilidade e calmaria. É claro que o engenheiro necessita sempre estar atualizado, mas o fato é que não temos problemas graves com prédios desabando, exceto quando ocorrem terremotos em um grau além do esperado ou quando surge algum picareta que decide construir um imóvel com areia da praia. Com o software é diferente: mesmo tendo uma equipe preparada, um orçamento adequado e uma estrutura de hardware moderna, existe a possibilidade (embora menor) do software não funcionar conforme o esperado. A humanidade ainda está evoluindo nessa área e, em um futuro próximo, esperamos desfrutar da calmaria das Engenharias.

Mas por que ocorrem problemas com softwares e não com prédios ? Bem, vamos imaginar que uma empresa foi contratada para construir um prédio de cinquenta andares com um orçamento adequado e com um tempo aceitável para a tarefa. É muito provável que ela consiga entregar o imóvel no prazo. Imagine agora que, passados três anos, o dono do imóvel contacte a referida construtora e diga assim : “precisamos aumentar esse prédio em mais vinte andares, quanto vocês me cobram ?” O dono da construtora irá tomar um susto e dirá : “nós não podemos fazer isso, esse prédio foi projetado desde os alicerces para suportar esse peso e essas condições de tempo. Aumentar esse imóvel em mais vinte andares é impossível.”

É por isso que o famoso prédio Empire State Building (figura 12.1), construído em Nova York, ainda tem os seus 102 andares. De 1931 até os dias de hoje, o Empire State não sofreu alterações na sua estrutura, ele só pode sofrer alterações pequenas, como a inclusão de uma torre de antena ou uma iluminação especial de acordo com alguma data considerada importante.

Com um sistema de informação é diferente. Ele **não** foi construído para permanecer inalterado na sua estrutura básica. Na verdade, ele sofre constantemente demandas externas para que mudanças sejam realizadas. Se tais mudanças não forem realizadas corretamente o sistema de informação pode produzir informações desencontradas, ocasionando a tomada de decisões erradas e a consequente falência da organização. Sistemas de informações são a “bússola” de qualquer organização, e quando uma bússola (esse pequeno objeto subestimado por muitos) não funciona corretamente, o caminhante não pode chegar ao seu destino.

Figura 12.1: Empire State Building

Em resumo, a industria do software passou por duas grandes crises: uma crise decorrente de problemas na construção do software e uma crise de manutenção do software criado[2]. As ferramentas que aprenderemos a usar nesse capítulo solucionam a primeira crise, ou seja, nós aprenderemos a criar um software. Por outro lado, todo sistema de informação é criado para interagir com um ambiente que constantemente está mudando. Por exemplo: uma nova lei fiscal que surge e exige que um cálculo seja alterado, uma tecnologia nova de troca de informações, uma nova forma de interface com o usuário, uma exigência feita por um consultor organizacional, etc. Se tais mudanças não forem conduzidas de forma correta, é certo que tempo e dinheiro serão perdidos.

Como nós dissemos, as crises de manutenção não foram solucionadas completamente com as ferramentas que apresentaremos. Elas podem até ser amenizadas, mas não radicalmente diminuídas. Para tratar as crises de manutenção, você terá que concluir esse capítulo e continuar seus estudos abordando novas técnicas, como

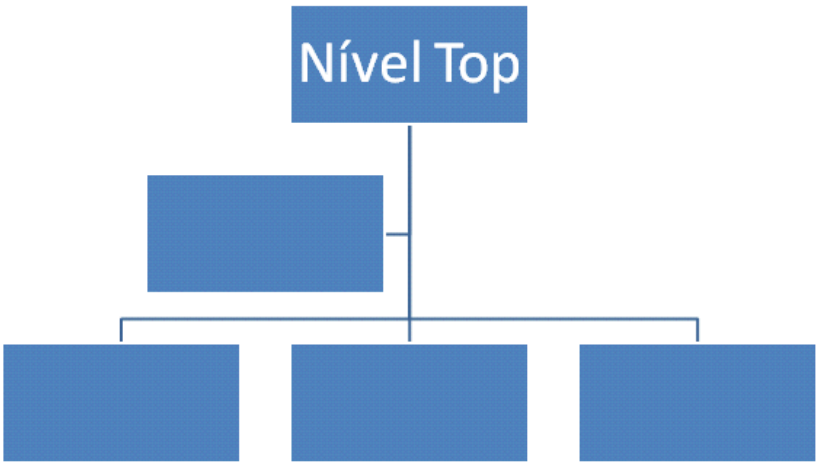
[2]Essa classificação é pessoal. Eu acredito que ela ajuda o iniciante a entender as mudanças ocorridas entre as décadas de 1970 até 1990. O fato, historicamente comprovado, é que as técnicas de programação estruturada, popularizadas a partir de 1970, não solucionaram totalmente os problemas na manutenção do software. As técnicas de orientação a objetos dominaram o cenário a partir da segunda metade da década de 1980 e se propuseram a resolver a lacuna deixada pela programação estruturada.

programação orientada a objetos e padrões de projetos.[3].

12.2 Programação Modular

A Programação Modular foi uma das primeiras técnicas adotadas para a redução dos custos no desenvolvimento do software. A imagem mental de um programa desenvolvido usando a Programação Modular é semelhante a um organograma, como o da figura 12.2.

Figura 12.2: Um organograma



De acordo com Yourdon e Constantine,

> um dos aspectos fascinantes do projeto de programa é seu relacionamento com as estruturas organizacionais humanas - particularmente a hierarquia de administração encontrada em corporações maiores. Sempre que quisermos ilustrar um certo ponto sobre um projeto de programa [...] quase sempre poderemos fazer isso desenhando analogias com uma situação de administração [Yourdon 1992, p. 27].

Imagine uma empresa com um alto grau de burocracia. Em organizações burocráticas, o fluxo de informação é estritamente vertical, obedecendo sempre o organograma. Por exemplo, Mario trabalha como Auxiliar administrativo no Setor Contábil de uma empresa, e Jonas trabalha como assistente no Departamento de Marketing. Jonas precisa de uma informação que só o Departamento Contábil pode fornecer, ele tem duas opções:

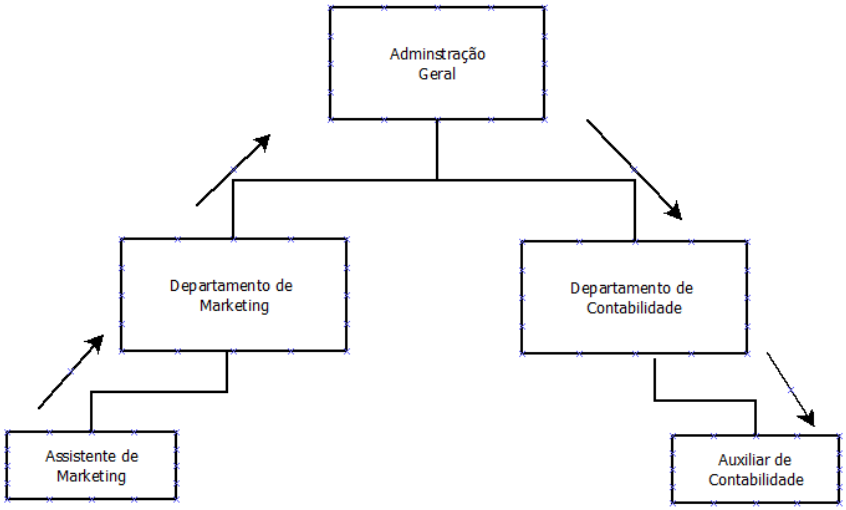
1. Fazer uma solicitação formal ao Departamento de Contabilidade, dessa forma, os dois gerentes ficam sabendo da solicitação.

2. Solicitar informalmente ao seu colega Mario, da Contabilidade, essa informação.

[3]Todas as técnicas de desenvolvimento que você está aprendendo agora serão usadas também no futuro, mas no contexto da orientação ao objeto e obedecendo a padrões de projetos atuais

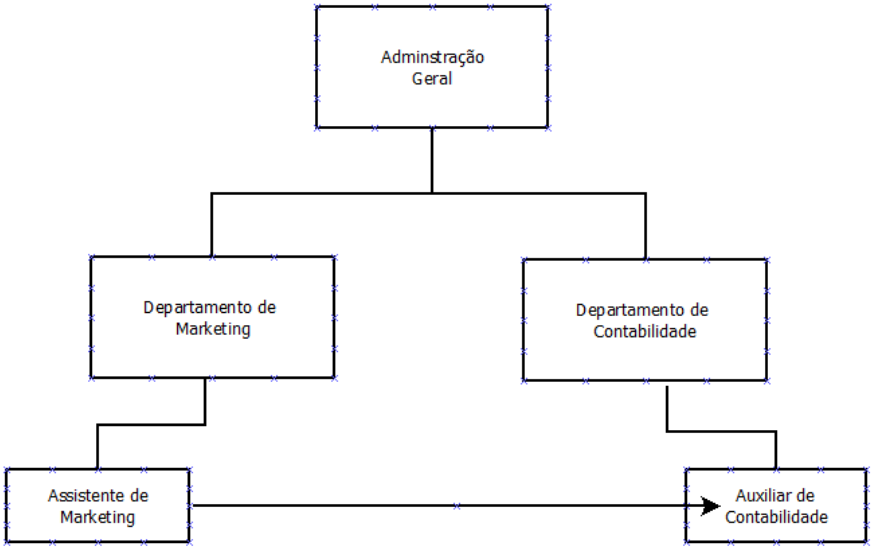
Pois bem, a Programação Modular equivale a primeira forma de tráfego de dados.Isso equivale a figura 12.3.

Figura 12.3: A programação modular é semelhante a uma organização formal burocrática.



As formas de comunicação informais, tipo as da figura 12.4, equivalem a conexões anormais, de acordo com a Programação Modular.

Figura 12.4: Um módulo não deve acessar diretamente os dados de outro módulo.



Na prática, isso quer dizer que a comunicação entre rotinas deve ser feita apenas mediante **passagem de parâmetros**, não através do acesso direto a variáveis. Veremos exemplos dessa prática mais adiante, não se preocupe com os detalhes. Por enquanto, é importante apenas que você saiba que a estrutura de um sistema modular assemelha-se a um organograma corporativo.

Agora vamos estudar a melhor forma de se construir esse organograma. Existem duas formas desse organograma ser construído: vamos construindo ele dos setores

mais "inferiores"até chegarmos a Administração Geral, ou partimos desta até chegarmos aos setores mais básicos da organização.

1. Quando começamos a desenvolver o sistema partindo das rotinas mais básicas até chegarmos às rotinas principais, dizemos que esse desenvolvimento é Bottom-up. Ou seja, de baixo para cima.

2. Agora, quando começamos dos módulos "superiores"até chegarmos aos módulos mais básicos, dizemos que o desenvolvimento é Top-Down.

Os primeiros desenvolvedores utilizavam o método Bottom-Up, porque ele parece ser o mais natural. Afinal de contas, eu vou construindo um prédio a partir dos alicerces, me concentrando em estruturas menores, como tijolos, colunas, etc. Contudo, essa abordagem se mostrou improdutiva. Alguém poderia objetar: como eu posso iniciar a construção a partir de um módulo superior, se os básicos não estão prontos ainda ? As próximas seções respondem a essa pergunta.

12.3 O método dos refinamentos sucessivos e o método Top-Down

Vamos agora aprender as ferramentas e técnicas que foram desenvolvidas para resolver a crise decorrente da criação do software (a primeira crise).

De acordo com Manzano e Oliveira :

> O método mais adequado para a programação de um computador é o de trabalhar com o conceito de programação estruturada, pois a maior parte das linguagens de programação utilizadas atualmente também são, o que facilita a aplicação deste processo de trabalho. O método mais adequado para a programação estruturada é o Top-Down (De cima para baixo). [Manzano e Oliveira 2008, p. 159]

Algumas características do método Top-Down :

1. O programador precisa ter em mente as tarefas principais que o programa deverá executar. **Não se preocupe em saber como essas tarefas serão executadas**, basta saber quais são elas.

2. Crie uma lista ordenada das tarefas do item anterior.

3. Agora procure detalhar o interior de cada item da lista.

Se você não entendeu o método Top-Down não se preocupe, vamos agora simular através de exemplos as etapas no desenvolvimento de um programa simples de computador. Não é a nossa intenção criar processos rígidos para que você os siga sem questionar. O nosso objetivo é lhe dar uma direção a ser seguida quando lhe pedirem para desenvolver programas simples de computador. O seguinte processo é baseado no estudo de caso formulado por Deitel e Deitel [Deitel e Deitel 2001, p. 120]. O estudo de caso envolve as seguintes etapas :

1. Definição do problema

2. Lista de observações
3. Criação do pseudo-código através de refinamento top-down
4. Criação do programa

12.3.1 Definição do problema

Considere o seguinte problema :

Uma escola oferece um curso preparatório para um concurso público. A escola deseja saber qual foi o desempenho dos seus estudantes. Você é o responsável pelo desenvolvimento do programa que irá resumir os resultados. Uma lista de dez estudantes com os resultados lhe foi entregue e na lista está escrito 1 se o estudante passou ou um 2 se o estudante foi reprovado. Se mais de 8 estudante passaram imprima a mensagem : "Aumente o preço do curso".

12.3.2 Lista de observações

Temos agora as seguintes observações tiradas da descrição do problema :

1. O programa irá processar 10 resultados. Sempre que surgirem listas a serem processadas considere a possibilidade de criar um laço (WHILE).
2. Devemos determinar se o resultado é 1 ou se é 2. Note que teremos que usar uma estrutura de seleção. Sempre que surgirem decisões a serem tomadas, ou dados a serem classificados (se é 1 ou se é 2), considere usar uma estrutura de seleção (IF).
3. Devemos contar o número de aprovações e o número de reprovações. Considere usar um contador (uma variável que é incrementada dentro da estrutura de seleção).
4. Por fim, devemos testar se mais de oito passaram no concurso e exibir uma mensagem. Novamente deveremos utilizar uma estrutura de seleção (IF) para exibir uma mensagem ("Aumente o preço do curso").

12.3.3 Criação do pseudo-código

O refinamento top-down é uma técnica de análise que pode ser usada na confecção de procedimentos como o desse estudo de caso. Como nós já vimos, essa técnica consiste em definir o problema principal e ir subdividindo-o até chegar nos níveis mais baixos do problema. Os itens a seguir exemplificam essa técnica :

- Representação do pseudo-código do topo : *Analise os resultados do exame e decida se o preço deve ser aumentado*
- Faça o primeiro refinamento
    1. Inicialize as variáveis
    2. Obtenha as dez notas e conte as aprovações e reprovações
    3. Imprima um resumo dos resultados e decida se o preço deve ser aumentado

- Faça o segundo refinamento

Algoritmo 18: Pseudo-código da análise dos resultados do teste

```
1  início
2      // Inicialize as variáveis
3      Inicialize as aprovações com 0
4      Inicialize as reprovações com 0
5      Inicialize o contador com 1
6      // Obtenha as 10 notas
7      enquanto contador <= 10 faça
8          Leia o próximo resultado
9          se o estudante foi aprovado então
10             Some 1 as aprovações
11         Senao Some 1 as reprovações
12     fim
13     // Imprima o resultado e decida se o curso
14     // deve ser aumentado
15     Imprima o número de aprovações
16     Imprima o número de reprovações
17     se mais de 8 estudantes foram aprovados então
18         Imprima "Aumente o preço do curso"
19 fim
```

- Faça o terceiro e último refinamento

Algoritmo 19: Pseudo-código da análise dos resultados do teste

```
1  início
2      // Inicialize as variáveis
3      aprovações ← 0
4      reprovações ← 0
5      contador ← 1
6      // Obtenha as 10 notas
7      enquanto contador <= 10 faça
8          leia resultado
9          se resultado = 1 então
10             Some 1 as aprovações
11         Senao se resultado = 2 então
12             Some 1 as reprovações
13     fim
14     // Imprima o resultado e decida se o curso
15     // deve ser aumentado
16     escreva aprovações
17     escreva reprovações
18     se contador > 8 então
19         escreva "Aumente o preço do curso"
20 fim
```

Mais uma vez enfatizamos : você não precisa se ater precisamente a esses passos. O que queremos dizer é que o código do programa antes de ser concluído deve passar

por sucessivos refinamentos. O refinamento inicial corresponde ao mais geral possível, como se fosse um avião sobrevoando uma floresta e vendo as topografia do terreno, as estradas e os lagos, mas tudo sem se ater aos detalhes. Esse nível é o nível “top” (ou superior). A partir daí outros refinamentos são feitos, até chegarmos aos detalhes (nível “down”) de uma árvore, uma caverna, uma casa, etc. Por isso que o nosso estudo sobre os algoritmos (pseudo-códigos principalmente) é importante, porque um algoritmo consegue abstrair detalhes e nos permite ver o problema de forma mais ampla.

12.3.4 Criação do programa

Feito todos esses passos, a codificação do programa fica muito mais fácil porque muitos questionamentos foram respondidos nas fases anteriores. Quando muitas dúvidas surgirem enquanto você estiver codificando, e essas dúvidas não se relacionarem com algum detalhe técnico, isso é um forte indício de que essas dúvidas não são propriamente dúvidas técnicas de programação, mas questionamentos referentes a fases anteriores a da codificação. Hoje em dia, é cada vez menor o número de pessoas que apenas sentam e desenvolvem um programa sem passar por etapas anteriores que definem realmente o problema.

Listagem 12.1: Análise dos resultados do teste
Fonte: codigos/loop04.prg

```
/*
Estruturas de controle de fluxo
Análise dos resultados do teste
Adaptada de (DEITEL; DEITEL, 2001, p.122)
*/
PROCEDURE Main
LOCAL nPasses,;  // Sucesso
      nFailures,;  // Falhas
      nStudentCounter,; // Estudantes
      nResult   // Desempenho

   // Inicialização
   nPasses := 0
   nFailures := 0
   nStudentCounter := 1
   nResult := 0

   // Processamento
   DO WHILE ( nStudentCounter <= 10 )
      INPUT "Forneça o resultado (1=aprovado, 2=reprovado) : ";
                                                    TO nResult
      IF ( nResult == 1 )
         nPasses++
      ELSE
         nFailures++
      ENDIF

      nStudentCounter++
```

```
                                                                    29
   ENDDO                                                            30
                                                                    31
   // Fase de término                                               32
   ? "Aprovados : ", nPasses                                        33
   ? "Reprovados: ", nFailures                                      34
                                                                    35
   IF ( nPasses > 8 )                                               36
      ?                                                             37
      ? "Aumente o preço do curso."                                 38
   ENDIF                                                            39
                                                                    40
RETURN                                                              41
```

A seguir temos uma simulação da execução do código quando nove alunos são aprovados e apenas um é reprovado.

.:Resultado:.

```
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Forneça o resultado (1=aprovado, 2=reprovado) : 2
Forneça o resultado (1=aprovado, 2=reprovado) : 1
Aprovados :           9
Reprovados:           1

Aumente o preço do curso.
```

12.4 Modularização de programas

A consequência direta do método Top-Down é a modularização do código. Até o presente momento nós usamos algumas funções do Harbour e construímos, em todos os exemplos, uma procedure : a **Main**. Sabemos que a procedure **Main** é usada pelo Harbour para identificar o ponto onde o programa deve iniciar. Mas a medida que o programa vai ficando mais complexo, a procedure Main vai ficando cada vez maior em número de linhas, e quando isso acontece, o código vai ficando muito mais difícil de se manter. A partir desse ponto chegamos ao conceito de “modularização do código”, ou seja: o código com muitas linhas é dividido em pequenos módulos (ou peças) que interagem entre si. O princípio é simples: se der um problema em um módulo qualquer do seu código, apenas esse módulo deverá ser alterado pois ele é “independente” dos demais módulos.

Vamos elaborar um pequeno programa para ilustrar a modularização. O programa é simples, vamos a sua análise :

12.4.1 O problema

Uma escola precisa de um programa para treinar os seus alunos nas quatro operações matemáticas básicas: soma, subtração, multiplicação e divisão. O aluno deve informar qual das quatro operações ele irá estudar, e o programa irá gerar randomicamente 5 questões matemáticas (tipo tabuada) para que o aluno informe a resposta (os números devem variar de 1 a 10). Ao final, se o aluno acertou todas as cinco o programa deve parabenizar o aluno com uma mensagem.

12.4.2 Lista de observações

Do programa descrito acima tiramos as seguintes observações :

1. O aluno deverá selecionar qual a operação que ele irá realizar. Podemos usar um menu com os comandos @ ... PROMPT e MENU TO.
2. O programa lerá a resposta e, de acordo com a opção escolhida, irá para a rotina selecionada: soma, subtração, multiplicação e divisão. Podemos usar o comando DO CASE para avaliar a resposta.
3. O programa irá sortear cinco questões referidas, de acordo com a operação. Podemos usar as funções HB_RandonInt() para sortear as operações.
4. Podemos guardar os acertos em um contador.
5. Ao final avaliamos o contador, se ele for igual a 5 então imprimimos “Parabéns, você está no caminho certo!”.

12.4.3 Criação do pseudo-código através do refinamento

O primeiro refinamento :

1. Crie o menu principal.
2. Leia e avalie as opções escolhidas.
3. Execute a rotina de acordo com a escolha.
4. Parabenize ou não o usuário.

O segundo refinamento :
Após uma análise do problema, você criou o seguinte pseudo-código em um nível bem superior (o nível “top”), descrito na listagem 20.

Algoritmo 20: Cálculadora.

```
1  início
2      Menu de seleção
3      escreva “Selecione a operação”
4      escreva “Soma”
5      escreva “Subtração”
6      escreva “Multiplicação”
7      escreva “Divisão”
8      leia Escolha
9      se Escolha = soma então
10         para contador = 1 até 5 faça
11             Rotina de soma
12         fim
13     se Escolha = subtração então
14         para contador = 1 até 5 faça
15             Rotina de subtração
16         fim
17     se Escolha = multiplicação então
18         para contador = 1 até 5 faça
19             Rotina de multiplicação
20         fim
21     se Escolha = divisão então
22         para contador = 1 até 5 faça
23             Rotina de divisão
24         fim
25     se Acertou 5 então
26         escreva “Parabéns, você está no caminho certo!”
27 fim
```

Até agora temos uma ideia geral do que devemos fazer. Alguns detalhes foram omitidos, como as rotinas de soma, subtração, multiplicação e divisão, além da inicialização das variáveis. Vamos agora usar esse modelo para codificar o nosso programa.

12.4.4 O programa calculadora

Vamos codificar o algoritmo desenvolvido na subseção anterior. Ele agora pode ser representado na listagem 12.2.

Listagem 12.2: Calculadora 1
Fonte: codigos/calc01.prg

```
/*
Calculadora
*/
PROCEDURE Main
LOCAL nEscolha

   // Inicialização do ambiente
   CLS
```

```
   // Menu inicial
   @ 09,10 SAY "Teste suas habilidades matemáticas"

   @ 11,10 PROMPT "       Soma       "
   @ 12,10 PROMPT "     Subtraçäo    "
   @ 13,10 PROMPT "   Multiplicaçäo  "
   @ 14,10 PROMPT "       Divisäo    "
   MENU TO nEscolha

   DO CASE
   CASE nEscolha == 1
      ? "Soma"
   CASE nEscolha == 2
      ? "Subtraçäo"
   CASE nEscolha == 3
      ? "Multiplicaçäo"
   CASE nEscolha == 4
      ? "Divisäo"
   ENDCASE

RETURN
```

Até agora só temos o menu principal com o resultado das escolhas. Na representação abaixo temos uma cópia de uma execução desse pequeno programa, nele o usuário selecionou a opção 1 (soma).

.:Resultado:.

```
          Teste suas habilidades matemáticas

                 Soma
               Subtração
            Multiplicação
                Divisão



Soma
```

Vamos colocar agora a rotina de soma para funcionar. A listagem 12.3 faz isso.

Listagem 12.3: Calculadora 2
Fonte: codigos/calc02.prg

```
/*
Calculadora
*/
PROCEDURE Main
LOCAL nEscolha // Escolha do menu
LOCAL n1, n2 // Os dois valores da pergunta
LOCAL nResp // A resposta
LOCAL nCont // O contador do laço
LOCAL nAcerto := 0 // Contador de acertos
```

```
                                                                    10
   // Inicialização do ambiente                                     11
   CLS                                                              12
                                                                    13
   // Menu inicial                                                  14
   @ 09,10 SAY "Teste suas habilidades matemáticas"                 15
                                                                    16
   @ 11,10 PROMPT "       Soma       "                             17
   @ 12,10 PROMPT "     Subtraçäo    "                             18
   @ 13,10 PROMPT "   Multiplicaçäo  "                             19
   @ 14,10 PROMPT "       Divisäo    "                             20
   MENU TO nEscolha                                                 21
                                                                    22
   DO CASE                                                          23
   CASE nEscolha == 1                                               24
      FOR nCont := 1 TO 5                                           25
         n1 := HB_RandomInt( 1 , 10 )                               26
         n2 := HB_RandomInt( 1 , 10 )                               27
         ? "Quanto é ", n1, " + " , n2 , " ? "                      28
         INPUT "Resposta : " TO nResp                               29
         IF ( nResp == ( n1 + n2 ) )                                30
             nAcerto++                                              31
             ? "Certo!"                                             32
         ELSE                                                       33
             ? "Errado!"                                            34
         ENDIF                                                      35
      NEXT                                                          36
      IF nAcerto == 5                                               37
         ? "Parabéns, você está no caminho certo!"                  38
      ENDIF                                                         39
   CASE nEscolha == 2                                               40
      ? "Subtraçäo"                                                 41
   CASE nEscolha == 3                                               42
      ? "Multiplicaçäo"                                             43
   CASE nEscolha == 4                                               44
      ? "Divisäo"                                                   45
   ENDCASE                                                          46
                                                                    47
RETURN                                                              48
```

Veja que apenas a soma está implementada. Vamos, então, testar e depois implementar para as outras operações.

.:Resultado:.

```
Quanto é           4  +           9  ?
Resposta : 13
Certo!
Quanto é           2  +           1  ?
Resposta : 3
Certo!
Quanto é           3  +           3  ?
Resposta : 6
```

```
Certo!
Quanto é           3  +           3  ?
Resposta : 6
Certo!
Quanto é          10  +           9  ?
Resposta : 19
Certo!

Parabéns, você está no caminho certo!
```

A listagem 12.4 implementa para as outras operações. Veja que nós só fizemos copiar e colar a rotina de soma para as demais operações e realizar pequenas alterações, que são :

1. Trocar o sinal de “+” pelo sinal correspondente na pergunta.
2. Trocar o sinal de “+” pelo sinal correspondente no IF que faz a avaliação da resposta.

Listagem 12.4: Calculadora 3
Fonte: codigos/calc03.prg

```
/*
Calculadora
*/
PROCEDURE Main
LOCAL nEscolha // Escolha do menu
LOCAL n1, n2 // Os dois valores da pergunta
LOCAL nResp // A resposta
LOCAL nCont // O contador do laço
LOCAL nAcerto := 0 // Contador de acertos

   // Inicialização do ambiente
   CLS

   // Menu inicial
   @ 09,10 SAY "Teste suas habilidades matemáticas"

   @ 11,10 PROMPT "       Soma       "
   @ 12,10 PROMPT "     Subtraçäo    "
   @ 13,10 PROMPT "   Multiplicaçäo  "
   @ 14,10 PROMPT "       Divisäo    "
   MENU TO nEscolha

   DO CASE
   CASE nEscolha == 1
      FOR nCont := 1 TO 5
         n1 := HB_RandomInt( 1 , 10 )
         n2 := HB_RandomInt( 1 , 10 )
         ? "Quanto é ", n1, " + " , n2 , " ? "
         INPUT "Resposta : " TO nResp
         IF ( nResp == ( n1 + n2 ) )
```

```
            nAcerto++
            ? "Certo!"
         ELSE
            ? "Errado!"
         ENDIF
      NEXT
      IF nAcerto == 5
         ? "Parabéns, você está no caminho certo!"
      ENDIF
   CASE nEscolha == 2
      FOR nCont := 1 TO 5
         n1 := HB_RandomInt( 1 , 10 )
         n2 := HB_RandomInt( 1 , 10 )
         ? "Quanto é ", n1, " - " , n2 , " ? "
         INPUT "Resposta : " TO nResp
         IF ( nResp == ( n1 - n2 ) )
            nAcerto++
            ? "Certo!"
         ELSE
            ? "Errado!"
         ENDIF
      NEXT
      IF nAcerto == 5
         ? "Parabéns, você está no caminho certo!"
      ENDIF
   CASE nEscolha == 3
      FOR nCont := 1 TO 5
         n1 := HB_RandomInt( 1 , 10 )
         n2 := HB_RandomInt( 1 , 10 )
         ? "Quanto é ", n1, " * " , n2 , " ? "
         INPUT "Resposta : " TO nResp
         IF ( nResp == ( n1 * n2 ) )
            nAcerto++
            ? "Certo!"
         ELSE
            ? "Errado!"
         ENDIF
      NEXT
      IF nAcerto == 5
         ? "Parabéns, você está no caminho certo!"
      ENDIF
   CASE nEscolha == 4
      FOR nCont := 1 TO 5
         n1 := HB_RandomInt( 1 , 10 )
         n2 := HB_RandomInt( 1 , 10 )
         ? "Quanto é ", n1, " / " , n2 , " ? "
         INPUT "Resposta : " TO nResp
         IF ( nResp == ( n1 / n2 ) )
            nAcerto++
            ? "Certo!"
         ELSE
```

```
            ? "Errado!"                                                    82
         ENDIF                                                             83
      NEXT                                                                 84
      IF nAcerto == 5                                                      85
         ? "Parabéns, você está no caminho certo!"                         86
      ENDIF                                                                87
   ENDCASE                                                                 88
                                                                           89
RETURN                                                                     90
```

No exemplo a seguir o usuário selecionou multiplicação.

.:Resultado:.

```
Quanto é           1  *           4  ?
Resposta : 4
Certo!
Quanto é           1  *           8  ?
Resposta : 8
Certo!
Quanto é           7  *           2  ?
Resposta : 14
Certo!
Quanto é          10  *           3  ?
Resposta : 30
Certo!
Quanto é           7  *           4  ?
Resposta : 28
Certo!
Parabéns, você está no caminho certo!
```

Nós acabamos de usar técnicas de programação estruturada para poder desenvolver uma solução para um problema computacional. Mas essa solução apresentada ainda é passível de melhorias. Os métodos que usamos até agora foram : o método do refinamento (implementamos várias soluções até chegar na ideal) e o método Top-Down (de cima para baixo). O método Top-Down ainda não foi totalmente explorado, na verdade, a parte mais interessante do método Top-Down ainda será usada, que é a criação de módulos ou sub-rotinas.

Segundo Manzano e Oliveira,

> Uma sub-rotina é, na verdade, um programa, e sendo um programa poderá efetuar diversas operações computacionais (entrada, processamento e saída). As sub-rotinas são utilizadas na divisão de algoritmos complexos, permitindo assim possuir a modularização de um determinado problema, considerado grande e de difícil solução [Manzano e Oliveira 2008, p. 158].

Hyman complementa essa definição apelando para a clareza:

> os programas são frequentemente muito complexos e longos. Na verdade, alguns programas exigem milhares, ou até milhões, de linhas de código. Quando você está criando um programa grande, é uma boa estratégia agrupá-los em seções manuseáveis que você pode entender facilmente (e as outras pessoas que o lêem também) [Hyman 1995, p. 193].

Uma sub-rotina é composta basicamente por :

1. Um nome
2. Um par de parenteses
3. Uma lista de valores de entrada (separados por vírgulas)

Nem todas as rotinas apresentam os três itens enumerados acima, você verá que, no mínimo, uma rotina é composta por um nome e um par de parênteses. No Harbour as sub-rotinas ou módulos recebem o nome de “procedimentos” ou procedures (em inglês). A procedure Main é a rotina principal, mas outras podem existir como veremos a seguir.

O nosso programa calculadora está muito complexo, porque existe uma quantidade grande de processamento em um único bloco ou rotina (a procedure Main), vamos agora, na listagem 12.5 dividir a rotina em partes menores.

Listagem 12.5: Calculadora 4
Fonte: codigos/calc04.prg

```
/*
Calculadora
*/
PROCEDURE Main
LOCAL nEscolha // Escolha do menu

   // Inicialização do ambiente
   CLS

   // Menu inicial
   @ 09,10 SAY "Teste suas habilidades matemáticas"

   @ 11,10 PROMPT "       Soma       "
   @ 12,10 PROMPT "     Subtração    "
   @ 13,10 PROMPT "   Multiplicação  "
   @ 14,10 PROMPT "       Divisão    "
   MENU TO nEscolha

   DO CASE
   CASE nEscolha == 1
      Soma()
   CASE nEscolha == 2
      Subtracao()
   CASE nEscolha == 3
      Multiplicacao()
   CASE nEscolha == 4
      Divisao()
   ENDCASE

RETURN

PROCEDURE Soma
LOCAL n1, n2 // Os dois valores da pergunta
```

```
LOCAL nResp // A resposta
LOCAL nCont // O contador do laço
LOCAL nAcerto := 0 // Contador de acertos

      FOR nCont := 1 TO 5
         n1 := HB_RandomInt( 1 , 10 )
         n2 := HB_RandomInt( 1 , 10 )
         ? "Quanto é ", n1, " + " , n2 , " ? "
         INPUT "Resposta : " TO nResp
         IF ( nResp == ( n1 + n2 ) )
             nAcerto++
             ? "Certo!"
         ELSE
             ? "Errado!"
         ENDIF
      NEXT
      IF nAcerto == 5
         ? "Parabéns, você está no caminho certo!"
      ENDIF

RETURN

PROCEDURE Subtracao
LOCAL n1, n2 // Os dois valores da pergunta
LOCAL nResp // A resposta
LOCAL nCont // O contador do laço
LOCAL nAcerto := 0 // Contador de acertos

      FOR nCont := 1 TO 5
         n1 := HB_RandomInt( 1 , 10 )
         n2 := HB_RandomInt( 1 , 10 )
         ? "Quanto é ", n1, " - " , n2 , " ? "
         INPUT "Resposta : " TO nResp
         IF ( nResp == ( n1 - n2 ) )
             nAcerto++
             ? "Certo!"
         ELSE
             ? "Errado!"
         ENDIF
      NEXT
      IF nAcerto == 5
         ? "Parabéns, você está no caminho certo!"
      ENDIF

RETURN

PROCEDURE Multiplicacao
LOCAL n1, n2 // Os dois valores da pergunta
LOCAL nResp // A resposta
LOCAL nCont // O contador do laço
LOCAL nAcerto := 0 // Contador de acertos
```

```
                                                                    85
      FOR nCont := 1 TO 5                                           86
         n1 := HB_RandomInt( 1 , 10 )                               87
         n2 := HB_RandomInt( 1 , 10 )                               88
         ? "Quanto é ", n1, " * " , n2 , " ? "                      89
         INPUT "Resposta : " TO nResp                               90
         IF ( nResp == ( n1 * n2 ) )                                91
             nAcerto++                                              92
             ? "Certo!"                                             93
         ELSE                                                       94
             ? "Errado!"                                            95
         ENDIF                                                      96
      NEXT                                                          97
      IF nAcerto == 5                                               98
         ? "Parabéns, você está no caminho certo!"                  99
      ENDIF                                                         100
                                                                    101
RETURN                                                              102
                                                                    103
PROCEDURE Divisao                                                   104
LOCAL n1, n2 // Os dois valores da pergunta                         105
LOCAL nResp // A resposta                                           106
LOCAL nCont // O contador do laço                                   107
LOCAL nAcerto := 0 // Contador de acertos                           108
                                                                    109
      FOR nCont := 1 TO 5                                           110
         n1 := HB_RandomInt( 1 , 10 )                               111
         n2 := HB_RandomInt( 1 , 10 )                               112
         ? "Quanto é ", n1, " / " , n2 , " ? "                      113
         INPUT "Resposta : " TO nResp                               114
         IF ( nResp == ( n1 / n2 ) )                                115
             nAcerto++                                              116
             ? "Certo!"                                             117
         ELSE                                                       118
             ? "Errado!"                                            119
         ENDIF                                                      120
      NEXT                                                          121
      IF nAcerto == 5                                               122
         ? "Parabéns, você está no caminho certo!"                  123
      ENDIF                                                         124
                                                                    125
RETURN                                                              126
```

Quando o programa inicia e descobre a procedure Soma(), Subtracao(), Multiplicacao() ou Divisao(), ele “salta” para dentro dela, executa tudo o que está lá até achar o comando RETURN. Quando o programa encontra o RETURN ele volta e prossegue a execução a partir da procedure que ele entrou.

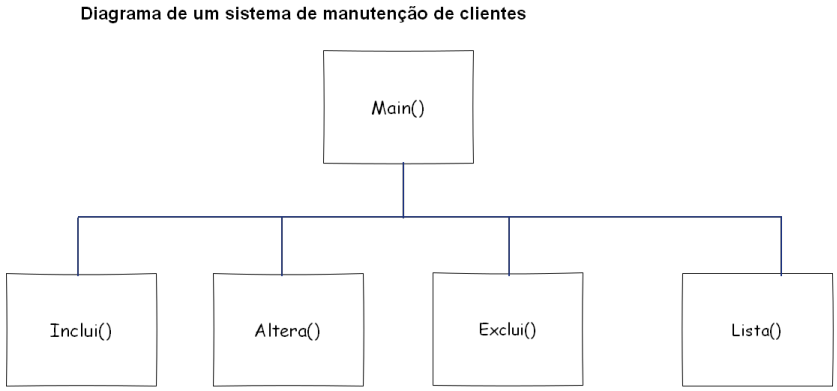
O resultado final é o mesmo, mas a calculadora agora está mais fácil de se manter. Basicamente as mudanças internas efetuadas foram a declaração de variáveis no início de cada módulo. Nós dividimos um problema complexo em quatro sub-rotinas bem mais simples. Ao final do capítulo esse mesmo programa ficará bem mais fácil de se

manter com a ajuda de parâmetros.

12.5 Um esboço de um sistema com sub-rotinas

Vamos agora criar um esboço de um programa com subrotinas (figura 12.6). É bom ressaltar que um programa raramente é composto por apenas um bloco, mas devido a sua complexidade ele acaba sendo subdividido em inúmeras partes. Por exemplo, um cadastro de clientes pode possuir 5 blocos : menu principal, inclusão de clientes, alteração de clientes, exclusão de clientes e listagem de clientes.

Figura 12.5: Manutenção de usuários



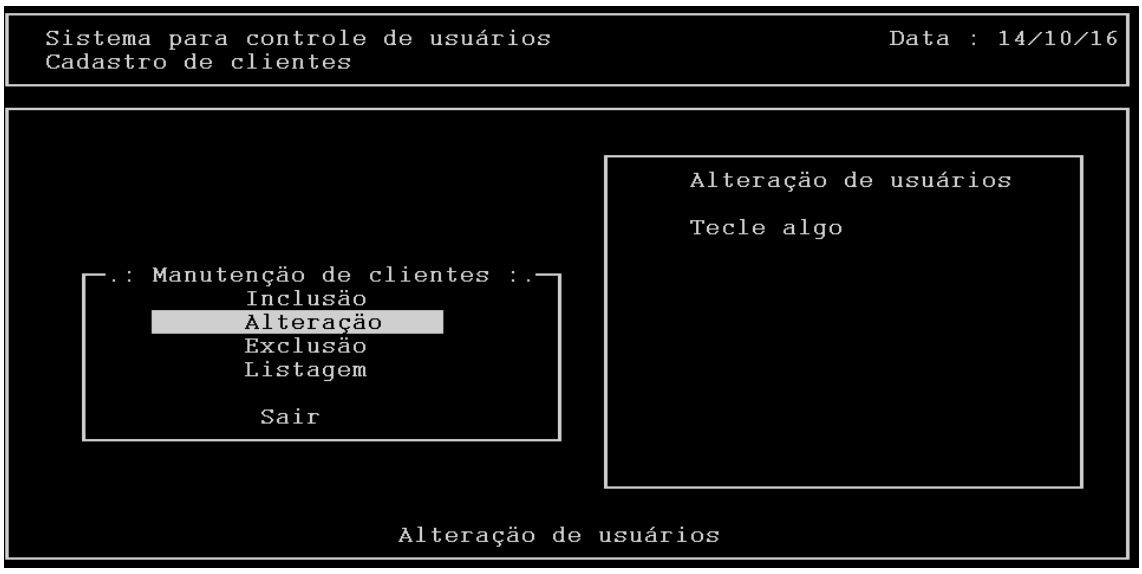
Listagem 12.6: Manutenção de clientes (Fonte)
Fonte: codigos/cadcli.prg

```
/*
Manutenção de clientes (menu principal)
*/
#define ULTIMA_LINHA 23
PROCEDURE Main
LOCAL nOpc

   SET DATE BRITISH
   SET DELIMITERS ON
   SET DELIMITERS TO "[]"
   SET WRAP ON
   SET MESSAGE TO ULTIMA_LINHA-1 CENTER

   CLS
   @ 0,0 TO 3,MAXCOL()
   @ 1,3 SAY "Sistema para controle de usuários"
   @ 1,MAXCOL()-15 SAY "Data : " + DTOC( DATE() )
   @ 2,3 SAY "Cadastro de clientes"

   @ 4,0 TO ULTIMA_LINHA,MAXCOL()
```

```
   @ 11,05 TO 18,36 // Contorno do menu                                    21
                                                                           22
   @ 06,39 TO ULTIMA_LINHA-3,MAXCOL()-3 // Contorno do box a direita do menu
                                                                           23
   @ 11,07 SAY ".: Manutenção de clientes :."                              24
                                                                           25
   DO WHILE .T.                                                            26
                                                                           27
                                                                           28
   @ 12,10 PROMPT "      Inclusäo     " MESSAGE "Inclusäo de usuários "
                                                                           29
   @ 13,10 PROMPT "      Alteraçäo    " MESSAGE "Alteraçäo de usuários"
                                                                           30
   @ 14,10 PROMPT "      Exclusäo     " MESSAGE "Exclusäo de usuários "
                                                                           31
   @ 15,10 PROMPT "      Listagem     " MESSAGE "Listagem de usuários "
                                                                           32
   @ 17,10 PROMPT "       Sair        " MESSAGE "Sair do sistema      "
      MENU TO nOpc                                                         33
                                                                           34
      DO CASE                                                              35
      CASE nOpc == 1                                                       36
         Inclui()                                                          37
      CASE nOpc == 2                                                       38
         Altera()                                                          39
      CASE nOpc == 3                                                       40
         Exclui()                                                          41
      CASE nOpc == 4                                                       42
         Lista()                                                           43
      CASE nOpc == 5                                                       44
         EXIT // Sai do laço e, consequentemente, do programa              45
      ENDCASE                                                              46
                                                                           47
   @ 07,40 CLEAR TO ULTIMA_LINHA-4,MAXCOL()-4 // Limpa o interior do box
   ENDDO                                                                   48
                                                                           49
RETURN                                                                     50
                                                                           51
/*                                                                         52
Inclusão de clientes                                                       53
*/                                                                         54
PROCEDURE INCLUI()                                                         55
                                                                           56
  @ 07,45 SAY "Inclusäo de usuários  "                                     57
  @ 09,45 SAY "Tecle algo"                                                 58
  INKEY(0)                                                                 59
                                                                           60
RETURN                                                                     61
                                                                           62
/*                                                                         63
Alteração de clientes                                                      64
```

```
*/
PROCEDURE ALTERA()

  @ 07,45 SAY "Alteração de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN

/*
Exclus?o de clientes
*/
PROCEDURE EXCLUI()

  @ 07,45 SAY "Exclusão de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN

/*
Listagem de clientes
*/
PROCEDURE LISTA()

  @ 07,45 SAY "Listagem de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN
```

Figura 12.6: Resultado



Vamos usar esse programa (listagem 12.6) para praticar alguns conceitos já vistos

e aprender novos conceitos. Antes de tudo, é importante que você digite o código e gere o referido programa. Depois que você fizer isso, retorne a esse ponto para prosseguirmos com o nosso estudo.

12.5.1 Criando uma rotina de configuração do sistema

O objetivo do exemplo da listagem listagem 12.6 é realizar operações de inclusão, alteração, exclusão e de listagem em um arquivo de usuários de um laboratório de informática. Ainda não sabemos como realizar tais operações, **mas sabemos ser esse o objetivo principal do programa**. Nós dividimos as operações em cinco sub-rotinas: a rotina principal (Main) que apresenta o menu com as opções e monta a tela principal e as demais rotinas com as operações de manutenção e listagem. Essa forma de organização contém um princípio muito importante : **cada rotina deve realizar uma tarefa bem definida**. Veja que eu até posso colocar todo o código em apenas uma rotina (como fizemos anteriormente) mas o programa ficaria difícil de se manter.

O nosso programa já evoluiu bastante, mas ainda existe um pequeno problema na nossa rotina principal (Main): o objetivo dessa rotina é iniciar o programa e exibir o menu com as escolhas, mas nós temos alguns comandos que não estão relacionados com essa tarefa, ou seja, repare que aparecem dentro da rotina Main o desenho da tela inicial e o comando **SET DATE BRITISH**. Eles são importantes para o programa funcionar corretamente, mas eles não fazem parte do objetivo final do programa. Eles facilitam a visualização de datas e de acentos, mas a essência do que eu quero fazer não depende diretamente deles.

Dica 74

O objetivo principal e o objetivo secundário são dois conceitos que costumam causar confusão na nossa mente. Mesmo entendendo o significado deles, em alguns momentos a diferença entre eles não é muito clara. É importante ter em mente que tudo é importante para o objetivo final, mas algumas coisas não cooperam diretamente para o seu alcance. Por exemplo: ter uma boa saúde coopera indiretamente para que você faça uma boa prova. Ter estudado a matéria coopera diretamente para que você tenha um bom desempenho na prova.

Dica 75

Cuidado quando for delimitar o objetivo de um programa e das suas rotinas. Lembre-se que uma rotina deve fazer uma e apenas uma coisa. Esse princípio não é uma lei a ser seguida, na verdade ela mais se parece com um conselho a ser levado em consideração pelo programador. Os programadores que se utilizam do sistema operacional UNIX (hoje famoso por causa do LINUX) levam esse princípio muito a sério. Os programas que auxiliam o sistema operacional realizam uma e apenas uma tarefa, e a realizam bem. Por exemplo o comando “ls” lista os arquivos de uma pasta [a], caso você digite “ls –help” verá que o comando possui uma miríade de opções, mas todas elas relacionam-se com a listagem de arquivos de uma pasta.

[a] O termo “diretório” quer dizer a mesma coisa e é frequentemente usado por usuários que usam o modo texto.

Temos então, um exemplo simples de complexidade que pode ser eliminado com o uso de procedures. A listagem 12.7 nos mostra como seria a resolução desse problema com o uso de duas procedures (rotinas) extras que nós decidimos chamar de “Config” e TelaInicial[4].

Listagem 12.7: Aprendendo sobre procedures e funções (Parte II).
Fonte: codigos/cadcli2.prg

```
/*
Manutenção de clientes (menu principal)
*/
#define ULTIMA_LINHA 23
#include "inkey.ch"
PROCEDURE Main
LOCAL nOpc

   Config()
   TelaInicial()




   DO WHILE .T.
      @ 11,05 TO 18,36 // Contorno do menu

   @ 06,39 TO ULTIMA_LINHA-3,MAXCOL()-3 // Contorno do box a direita do menu
      @ 11,07 SAY ".: Manutenção de clientes :."


   @ 12,10 PROMPT "    Inclusão     " MESSAGE "Inclusão de usuários "

   @ 13,10 PROMPT "    Alteração    " MESSAGE "Alteração de usuários"

   @ 14,10 PROMPT "    Exclusão     " MESSAGE "Exclusão de usuários "

   @ 15,10 PROMPT "    Listagem     " MESSAGE "Listagem de usuários "

   @ 17,10 PROMPT "     Sair        " MESSAGE "Sair do sistema      "
      MENU TO nOpc

      DO CASE
      CASE nOpc == 1
         Inclui()
      CASE nOpc == 2
         Altera()
      CASE nOpc == 3
         Exclui()
      CASE nOpc == 4
         Lista()
      CASE nOpc == 5
         EXIT // Sai do laço e, consequentemente, do programa
```

[4]Lembre-se de escolher um nome facilmente memorizável e que tenha relação com o objetivo da procedure ou função.

```
      ENDCASE
      @ 07,40 CLEAR TO ULTIMA_LINHA-4,MAXCOL()-4
   ENDDO

RETURN

/*
Inclusão de clientes
*/
PROCEDURE Inclui()

  @ 07,45 SAY "Inclusão de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN

/*
Alteração de clientes
*/
PROCEDURE Altera()

  @ 07,45 SAY "Alteração de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN

/*
Exclusão de clientes
*/
PROCEDURE Exclui()

  @ 07,45 SAY "Exclusão de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN

/*
Listagem de clientes
*/
PROCEDURE LISTA()

  @ 07,45 SAY "Listagem de usuários  "
  @ 09,45 SAY "Tecle algo"
  INKEY(0)

RETURN

/*
```

```
Config                                                                    88
*/                                                                        89
PROCEDURE Config()                                                        90
                                                                          91
   SET DATE BRITISH // Converte data para formato dd/mm/yy                92
   SET CENTURY ON // Exibe o ano no formato de quatro dígitos              93
   SET DELIMITERS ON // Ativa delimitadores de GETs                        94
   SET DELIMITERS TO "[]" // Informa como serão os delimitadores           95
                                                                          96
   SET WRAP ON // Quando o usuário chegar no último PROMPT passa para o primei
                                                                          97
   SET MESSAGE TO ULTIMA_LINHA-1 CENTER //Linha de mensagem do menu @ ... PROM
   SET EVENTMASK TO INKEY_ALL // Mouse                                    98
   hb_cdpSelect("UTF8")                                                   99
RETURN                                                                    100
                                                                          101
/*                                                                        102
Tela inicial                                                              103
*/                                                                        104
PROCEDURE TelaInicial()                                                   105
                                                                          106
   CLS                                                                    107
   @ 0,0 TO 3,MAXCOL()                                                    108
   @ 1,3 SAY "Sistema para controle de usuários"                          109
   @ 1,MAXCOL()-15 SAY "Data : " + DTOC( DATE() )                         110
   @ 2,3 SAY "Cadastro de clientes"                                       111
                                                                          112
   @ 4,0 TO ULTIMA_LINHA,MAXCOL()                                         113
                                                                          114
RETURN                                                                    115
```

Essa listagem merece alguns comentários.

1. As procedures Config e TelaInicial retiraram de Main tudo o que se refere a configuração do ambiente e desenho inicial de telas.
2. Quando o programa inicia e descobre a procedure Config, por exemplo, ele “salta” para dentro dela, executa tudo o que está lá até achar o comando RETURN.
3. Quando o programa encontra o RETURN ele volta e prossegue a execução a partir da procedure Config.
4. O mesmo acontece com a procedure TelaInicial

Você pode estar pensando : “eu apenas mudei de posição alguns comandos. Não foi uma grande troca.” O seu pensamento faz sentido, mas quando a complexidade de um programa aumenta você pode usar procedures e funções para reduzi-la. Suponha agora que você deseje realizar outras configurações no ambiente antes de exibir o menu inicial. A linguagem Harbour possui dezenas de SETs que podem ser chamados de um lugar centralizado e organizados para que fiquem fáceis de serem localizados. Pode ser, também, que você crie um programa novo no futuro e deseje usar a rotina de tela e de configuração de ambiente que você desenvolveu para o programa de cadastro

de usuários. Nesse caso, tudo o que você tem que fazer é “ligar” (“linkar”) essas rotinas novas ao seu novo programa. Veremos como fazer isso na próxima seção.

12.5.2 Programas com mais de um arquivo

Vamos expandir um pouco mais a nossa ideia sobre rotinas. Para tornar a procedure Config disponível para outros programas futuros você deve, primeiramente, desvincular essa procedure do arquivo principal, tornando-a independente. Fazer isso é fácil, basta criar um arquivo qualquer (vamos chamar esse arquivo de rotinas.prg) e colar nele a procedure Config. O arquivo deve ficar conforme a listagem 12.8.

Listagem 12.8: Rotinas.
Fonte: codigos/rotinas.prg

```
#define ULTIMA_LINHA 23                                                    1
#include "inkey.ch"                                                        2
/*                                                                         3
Config                                                                     4
*/                                                                         5
PROCEDURE Config()                                                         6
                                                                           7
   SET DATE BRITISH // Converte data para formato dd/mm/yy                 8
   SET CENTURY ON // Exibe o ano no formato de quatro dígitos               9
   SET DELIMITERS ON // Ativa delimitadores de GETs                         10
   SET DELIMITERS TO "[]" // Informa como serão os delimitadores            11
                                                                           12
   SET WRAP ON // Quando o usuário chegar no último PROMPT passa para o primei
                                                                           13
   SET MESSAGE TO ULTIMA_LINHA-1 CENTER //Linha de mensagem do menu @ ... PROM
   SET EVENTMASK TO INKEY_ALL // Mouse                                     14
                                                                           15
RETURN                                                                     16
```

Não precisa compilar esse arquivo agora, pois a função dele é ficar disponível para outros projetos futuros. Dessa forma, você não precisará duplicar (nem triplicar ou quadruplicar) o seu código a cada novo projeto, basta “ligar” o seu projeto futuro com esse arquivo (rotinas.prg) e as procedures dele ficarão disponíveis para o novo projeto.

12.5.3 Reaproveitando a rotina config.prg em outro programa

Vamos supor que você agora deseje criar um outro programa. O programa novo (listagem 12.9) cria um pequeno formulário para entrada de dados. Vamos salvar esse arquivo novo com o nome de “doisarq.prg”.

Listagem 12.9: Programa com mais de um arquivo (Parte I).
Fonte: codigos/doisarq.prg

```
/*                                                                         1
Chamando rotinas de outros arquivos                                        2
*/                                                                         3
PROCEDURE Main                                                             4
LOCAL cNome := SPACE(20)                                                   5
```

```
LOCAL cTipo := SPACE(1)                                                    6
LOCAL dNascimento := CTOD("//") // Inicializa com uma data vazia           7
LOCAL GetList := {}                                                        8
                                                                           9
   CLS                                                                     10
   @ 5,5 SAY "Informe o nome : " GET cNome                                 11
                                                                           12
   @ 7,5 SAY "Informe o tipo (F ou J): " GET cTipo VALID UPPER( cTipo ) $ "FJ"
   @ 9,5 SAY "Data de nascimento : " GET dNascimento                       13
   READ                                                                    14
                                                                           15
                                                                           16
                                                                           17
RETURN                                                                     18
```

Para não ter que repetir as configurações de inicialização você pode chamar a rotina de configuração que está em config.prg, basta incluir a chamada a essa rotina na linha 10 da listagem a seguir.

Listagem 12.10: Programa com mais de um arquivo (Parte II).
Fonte: codigos/doisarq2.prg

```
/*                                                                         1
Chamando rotinas de outros arquivos                                        2
*/                                                                         3
PROCEDURE Main                                                             4
LOCAL cNome := SPACE(20)                                                   5
LOCAL cTipo := SPACE(1)                                                    6
LOCAL dNascimento := CTOD("//") // Inicializa com uma data vazia           7
LOCAL GetList := {}                                                        8
                                                                           9
   Config()                                                                10
                                                                           11
   CLS                                                                     12
   @ 5,5 SAY "Informe o nome : " GET cNome                                 13
                                                                           14
   @ 7,5 SAY "Informe o tipo (F ou J): " GET cTipo VALID UPPER( cTipo ) $ "FJ"
   @ 9,5 SAY "Data de nascimento : " GET dNascimento                       15
   READ                                                                    16
                                                                           17
RETURN                                                                     18
```

Se você tentar gerar o programa agora irá dar errado, pois o programa tentará achar a procedure Config e não irá encontrá-la. Veja a seguir a mensagem de erro:

.:Resultado:.

```
hbmk2: Erro: Referenciado, faltando, mas funções desconhecida(s):
   CONFIG()
```

Para que o programa “doisarq.prg” encontre a procedure Config você deve fazer assim :

.:Resultado:.

```
hbmk2 doisarq rotinas
```

Pronto, agora a procedure Config (que está dentro de rotinas.prg) é visível pelo programa principal. Lembre-se que qualquer alteração que você fizer em rotinas.prg requer a recompilação do projeto principal, caso você queira que as mudanças sejam reconhecidas por ele. Por exemplo: se você quiser inserir um SET novo na procedure Config, você deve recompilar o arquivo (*hbmk2 doisarq rotinas*) para que o SET novo seja visto pelo programa doisarq.

Essa forma de trabalhar é interessante pois, caso você deseje criar um novo programa, basta compilar o mesmo chamando "rotinas.prg" para que as procedures contidas nesse arquivo se tornem disponíveis para o seu novo projeto.

Dica 76

Lembra de uma dica anterior (nesse capítulo) quando nós falamos sobre "objetivo principal" e "objetivo secundário" ? Pois bem, você pode usar esse raciocínio para ajudar na classificação das rotinas externas. Ou seja, um objetivo secundário não possui um fim em si mesmo, mas existe para que outras coisas possam ser construídas. Da mesma forma, quando alguém economiza dinheiro, ela não está economizando porque o dinheiro é um meio e não um fim. Ou seja, o seu valor não se justifica nele mesmo, mas nos objetivos finais que ele alcança. Ter dinheiro é um objetivo secundário, ter uma vida próspera poderia ser um objetivo principal.
Dessa forma, tudo o que for "objetivo principal" poderia ficar na rotina principal (no nosso exemplo: doisarq.prg) e tudo o que for "objetivo secundário" poderia ficar no arquivo chamado rotinas.prg. Mais exemplos de coisas que poderiam ficar no rotinas.prg :

- Rotinas que fazem cálculos financeiros que serão usados por diversos programas.
- Rotinas que manipulam datas e calculam feriados locais.
- Rotinas que configuram ambientes (Config) e telas.

Com o passar do tempo você poderia criar vários arquivos no estilo de rotinas.prg com procedures e funções bem específicas, por exemplo: você poderia criar o financeiro.prg apenas com rotinas de cálculos financeiros, depois poderia criar o datas.prg apenas com rotinas de cálculos de datas, e poderia manter o rotinas.prg com rotinas que não puderam ser classificadas. Por enquanto não se preocupe em classificar nada, basta ter o arquivo rotinas.prg com as rotinas que iremos criar, depois você pode se preocupar em classificar em arquivos diferentes. O que importa no momento é que você coloque as rotinas de apoio em um arquivo separado para utilização posterior em vários projetos.

12.6 Parâmetros

Parâmetros são valores que nós passamos para as rotinas na forma de variáveis. O exemplo a seguir nos dá uma ideia do que é um parâmetro.

Listagem 12.11: Parâmetros.
Fonte: codigos/para1.prg

```
/*
Chamando rotinas de outros arquivos
*/
PROCEDURE Main
LOCAL nParcela1 := 0
LOCAL nParcela2 := 0
LOCAL GetList := {}

   CLS

   @ 5,5 SAY "Informe a parcela 1 : " GET nParcela1 PICTURE "@RE 999,999.99"

   @ 7,5 SAY "Informe a parcela 2 : " GET nParcela2 PICTURE "@RE 999,999.99"
   READ

   Soma( nParcela1, nParcela2 )

RETURN

PROCEDURE Soma( nVal1, nVal2 )

   @ 9,5 SAY nVal1 + nVal2 PICTURE  "@RE 999,999.99"

RETURN
```

.:Resultado:.

```
Informe a parcela 1 :       12,00

Informe a parcela 2 :        5,00

     17,00
```

A passagem de um parâmetro da rotina chamadora para a rotina chamada aparentemente é um processo simples, mas você deve ter cuidado com a qualidade e a quantidade dos parâmetros passados. Se a rotina espera que você , obrigatoriamente, passe dois parâmetros numéricos (como é o caso da rotina Soma), você não pode passar somente um parâmetro numérico, ou então um parâmetro numérico e outro caractere. Felizmente o Harbour possui algumas funções que avaliam a qualidade e a quantidade dos parâmetros passados. Iremos, nos próximos exemplos, expandir a rotina Soma para que ela passe a verificar os parâmetros passados. A listagem 12.12 a seguir usa uma função chamada PCOUNT() que avalia a quantidade de parâmetros passados.

Listagem 12.12: Parâmetros e PCount().
Fonte: codigos/para2.prg

```
/*
```

```
/*
Chamando rotinas de outros arquivos
*/
PROCEDURE Main
LOCAL nParcela1

   nParcela1 := 2
   Soma( nParcela1 )

RETURN

PROCEDURE Soma( nVal1, nVal2 )

   IF PCOUNT() <> 2
      ? "Quantidade incorreta de parâmetros."
      ? "Essa rotina só aceita 2 parâmetros."
      RETURN
   ENDIF

   @ 9,5 SAY nVal1 + nVal2 PICTURE  "@RE 999,999.99"

RETURN
```

Como só foi passado um parâmetro, a rotina exibe uma mensagem de erro. Se não tivesse essa verificação (na linha 14) o valor de nVal1 seria 2, mas o valor de nVal2 seria NIL, o programa tentaria somar 2 + NIL (linha 20) e um erro de execução seria gerado. A checagem prévia nos ajudou a evitar esse erro.

.:Resultado:.

```
Quantidade incorreta de parâmetros.
Essa rotina só aceita 2 parâmetros.
```

A listagem 12.13 ilustra uma passagem correta de parâmetros. Foram passados dois valores: 2 e 7.

Listagem 12.13: Parâmetros e PCount().
Fonte: codigos/para3.prg

```
/*
Chamando rotinas de outros arquivos
*/
PROCEDURE Main
LOCAL nParcela1
LOCAL nParcela2

   nParcela1 := 2
   nParcela2 := 7
   Soma( nParcela1, nParcela2 )

RETURN

PROCEDURE Soma( nVal1, nVal2 )

```

```
   IF PCOUNT() <> 2
      ? "Quantidade incorreta de parâmetros."
      ? "Essa rotina só aceita 2 parâmetros."
      RETURN
   ENDIF

   @ 9,5 SAY nVal1 + nVal2 PICTURE  "@RE 999,999.99"

RETURN
```

.:Resultado:.

```
          9,00
```

Dica 77

A maneira mais segura de se trabalhar com [rotinas] é verificar a quantidade e o tipo dos parâmetros passados [Ramalho 1991, p. 470].

Porém, mesmo que a rotina Soma verifique a quantidade de parâmetros passados, a rotina chamadora (Main) pode mandar a quantidade correta, mas com o tipo diferente de dado. Por exemplo, se a rotina chamadora chamar Soma com um parâmetro numérico e outro caractere, um erro de execução será gerado. A listagem 12.14 ilustra essa situação.

Listagem 12.14: Parâmetros e PCount().
Fonte: codigos/para4.prg

```
/*
Chamando rotinas de outros arquivos
*/
PROCEDURE Main
LOCAL nParcela1
LOCAL nParcela2

   nParcela1 := "Era para ser um número"
   nParcela2 := 7
   Soma( nParcela1, nParcela2 )

RETURN

PROCEDURE Soma( nVal1, nVal2 )

   IF PCOUNT() <> 2
      ? "Quantidade incorreta de parâmetros."
      ? "Essa rotina só aceita 2 parâmetros."
      RETURN
   ENDIF

   @ 9,5 SAY nVal1 + nVal2 PICTURE  "@RE 999,999.99"

```

```
RETURN                                                                    24
```

.:Resultado:.

```
Error BASE/1081  Argument error: +
Called from SOMA(22)
Called from MAIN(10)
```

Nesse caso a função PCOUNT() não iria ajudar muito. O Harbour possui uma outra forma mais eficaz de tratar os parâmetros passados: a função hb_DefaultValue(). Antes de apresentar essa função, vamos explicar o que é um valor DEFAULT. A expressão DEFAULT é uma palavra inglesa, termo técnico muito usado em vários contextos de informática, normalmente com o significado de "padrão" ou de "algo já previamente definido". No processador de texto, por exemplo, existem valores ou configurações default para o tamanho e o tipo da fonte [5]. No nosso contexto, DEFAULT é o valor que um parâmetro assume caso ele não seja passado. A função hb_DefaultValue() faz mais ainda, ela verifica se o tipo de dado passado é o mesmo tipo do valor DEFAULT, se não for, então ela atribui o valor DEFAULT sobrepondo o que foi passado. Tudo bem, parece complicado mas não é, vamos aos exemplos.

Listagem 12.15: Parâmetros e valores DEFAULT.
Fonte: codigos/para5.prg

```
/*                                                                        1
Chamando rotinas de outros arquivos                                       2
*/                                                                        3
PROCEDURE Main                                                            4
LOCAL nParcela1                                                           5
LOCAL nParcela2                                                           6
                                                                          7
   nParcela1 := "Era para ser um número"                                  8
   nParcela2 := 7                                                         9
   Soma( nParcela1, nParcela2 )                                           10
                                                                          11
RETURN                                                                    12
                                                                          13
PROCEDURE Soma( nVal1, nVal2 )                                            14
                                                                          15
   nVal1 := hb_DefaultValue( nVal1, 0 )                                   16
   nVal2 := hb_DefaultValue( nVal2, 0 )                                   17
                                                                          18
   @ 9,5 SAY nVal1 + nVal2 PICTURE  "@RE 999,999.99"                      19
                                                                          20
RETURN                                                                    21
```

Note que a função hb_DefaultValue() desprezou o caractere e atribuiu zero ao parâmetro errado. Zero, nesse caso é o valor DEFAULT, note que ele é o segundo parâmetro da função hb_DefaultValue().

.:Resultado:.

[5] Fonte: https://www.significados.com.br/default-1/ - On line: 15/10/2016

```
   7,00
```

12.6.1 O programa calculadora com parâmetros

O programa dessa seção coloca em prática o conceito de parâmetros. Você certamente deve estar lembrado do programa da tabuada, no início desse capítulo. Vamos agora fazer o mesmo programa, mas evitando a criação de rotinas de soma, subtração, multiplicação e divisão. Em vez de criar essas quatro rotinas nós iremos criar apenas uma chamada de “calcular”, e nós iremos passar para ela a operação a ser realizada através de um sinal : “+”, “-”, “*” ou “/”.

Bem, a rotina o programa a seguir (listagem 12.16) faz isso através de parâmetros de entrada. Observe atentamente.

Listagem 12.16: Calculadora 5
Fonte: codigos/calc05.prg

```
/*
Calculadora
*/
PROCEDURE Main
LOCAL nEscolha // Escolha do menu

   // Inicialização do ambiente
   CLS

   // Menu inicial
   @ 09,10 SAY "Teste suas habilidades matemáticas"

   @ 11,10 PROMPT "       Soma       "
   @ 12,10 PROMPT "     Subtraçäo    "
   @ 13,10 PROMPT "   Multiplicaçäo  "
   @ 14,10 PROMPT "       Divisäo    "
   MENU TO nEscolha

   DO CASE
   CASE nEscolha == 1
      Calcular( "+" )
   CASE nEscolha == 2
      Calcular( "-" )
   CASE nEscolha == 3
      Calcular( "*" )
   CASE nEscolha == 4
      Calcular( "/" )
   ENDCASE

RETURN

PROCEDURE Calcular( cSinal )
LOCAL n1, n2 // Os dois valores da pergunta
LOCAL nResp // A resposta
```

```
LOCAL nCont // O contador do laço
LOCAL nAcerto := 0 // Contador de acertos
LOCAL nSolucao // Solução computada

   FOR nCont := 1 TO 5
      n1 := HB_RandomInt( 1 , 10 )
      n2 := HB_RandomInt( 1 , 10 )
      ? "Quanto é ", n1, cSinal , n2 , " ? "
      INPUT "Resposta : " TO nResp

      SWITCH cSinal
      CASE "+"
         nSolucao := ( n1 + n2 )
         EXIT
      CASE "-"
         nSolucao := ( n1 - n2 )
         EXIT
      CASE "*"
         nSolucao := ( n1 * n2 )
         EXIT
      CASE "/"
         nSolucao := ( n1 / n2 )
         EXIT
      END

      IF ( nSolucao == nResp )
          nAcerto++
          ? "Certo!"
      ELSE
          ? "Errado!"
      ENDIF
   NEXT
   IF nAcerto == 5
      ? "Parabéns, você está no caminho certo!"
   ENDIF

RETURN
```

A seguir temos um exemplo do funcionamento do programa. Note que nada foi alterado no seu comportamento, apenas ele ficou bem mais fácil de se manter.

.:Resultado:.

```
 Quanto é           6 -         10  ?
Resposta : -4
Certo!
Quanto é           9 -          7  ?
Resposta : -2
Errado!
Quanto é           2 -          4  ?
Resposta : -2
Certo!
Quanto é           9 -          9  ?
```

```
Resposta : 0
Certo!
Quanto é          6 -          3  ?
Resposta : 3
Certo!
```

12.7 Conclusão

Encerramos mais um capítulo crucial no aprendizado da programação de computadores. Alguns conceitos importantes foram vistos, mas o mais importante de todos com certeza é a divisão de um problema em partes menores e gerenciáveis independentemente umas das outras.

13 Variável Local, Funções e Passagem de Valor

A funcionalidade vem antes da forma.

Louis Sullivan

Objetivos do capítulo

- Entender um pouco mais sobre variáveis Locais
- Aprender a trabalhar com sub-rotinas.
- Passagem de parâmetros: valor e referência.
- Criar dois pequenos exemplos com as técnicas aprendidas.

13.1 A declaração LOCAL

Lembra da palavra LOCAL para declarar variáveis ? Você foi aconselhado, desde o início a sempre usá-la, mas não sabia o real motivo. Bem, a partir de agora você poderá entender o seu real significado. Vamos começar com um exemplo simples, na listagem 13.1.

Listagem 13.1: LOCAL
Fonte: codigos/local01.prg

```
/*
Declaração local
*/
PROCEDURE Main
LOCAL x

   x := 10
   Quadrado()

RETURN

PROCEDURE Quadrado()

   ? x * x

RETURN
```

A listagem 13.1 possui uma sub-rotina chamada quadrado. A ideia é simples. A variável x é declarada na função Main e a sub-rotina Quadrado exibe o valor de x ao quadrado. Porém, quando executamos obtemos o seguinte erro:

.:Resultado:.

```
Error BASE/1003  Variable does not exist: X
Called from QUADRADO(14)
Called from MAIN(8)
```

A mensagem de erro nos informa que a variável X não existe. Mas como, se nós a declaramos no início do programa ? A resposta é simples: quando nós declaramos uma variável como LOCAL, nós estamos dizendo ao programa que ela só será visível dentro da rotina onde ela se encontra. Como a variável x foi declarada e inicializada em uma rotina (Main), mas nós tentamos usar o seu valor em uma outra rotina (Quadrado), então uma mensagem de erro foi gerada. Em resumo: variáveis declaradas com LOCAL só são vistas dentro da rotina onde elas se encontram. Como o exemplo que nós usamos foi bem simples, talvez você ainda não tenha reconhecido isso como vantagem, mas acredite: declarar suas variáveis com LOCAL fará com que os seus programas fiquem mais fáceis de manter.

Se você não usar a declaração LOCAL, então a variável será visível também pelas sub-rotinas. Vamos fazer uma pequena alteração no nosso programa e retirar a declaração LOCAL x. Acompanhe o código na listagem 13.2 e a simulação a seguir.

Listagem 13.2: LOCAL
Fonte: codigos/local02.prg

```
/*
Declaração sem a variável local
*/
PROCEDURE Main

   x := 10
   Quadrado()

RETURN

PROCEDURE Quadrado()

   ? x * x

RETURN
```

.:Resultado:.

```
      100
```

É por isso que nós insistimos para que você sempre declare as suas variáveis com a declaração LOCAL. Quando nós omitimos a declaração LOCAL, a variável recebe uma outra classificação chamada PRIVATE, e passa a ser vista pelo seu módulo e por todos os módulos que são chamados a partir deste. Isso fará com que você corra o risco de ter o valor da variável acidentalmente alterado em algum módulo chamado. Esse problema é considerado grave pois costuma gerar erros de execução difíceis de se identificar a causa real.

Yourdon e Constantine colocam esse problema em forma de pergunta:

> A questão chave é: quanto de um módulo tem que ser conhecido para compreendermos um outro módulo ? Quanto mais precisarmos saber do módulo B para compreendermos o módulo A, mais fortemente A está conectado a B. O fato de precisarmos saber alguma coisa sobre um outro módulo é uma evidência a priori de alguns graus de interconexão, mesmo que o tipo de interconexão não seja conhecido. [Yourdon 1992, p. 95]

Voltaremos a esse assunto no capítulo seguinte.

13.1.1 Passagem de parâmetros por valor

Quando você passa uma variável como parâmetro, como por exemplo, na função Soma, você está gerando uma cópia da variável original. Vamos ilustrar isso na listagem 13.3.

Listagem 13.3: LOCAL
Fonte: codigos/local03.prg

```
/*
Declaração com a variável local
passagem de parâmetro por valor
*/
PROCEDURE Main
```

```
LOCAL x

   x := 10
   Muda( x )
   ? x

RETURN

PROCEDURE Muda( x )

   x :=  200
   ? x

RETURN
```

.:Resultado:.

```
        10
       200
        10
```

Preste atenção nesse exemplo simples. O valor x foi passado como parâmetro para a função Muda e lá ele recebeu uma nova atribuição, ficando o seu valor igual a 200. Mas quando o valor de x foi exibido fora da função Muda, o seu valor continuou como 10. Como uma variável pode ter dois valores ? Bem, vamos por partes. Em primeiro lugar o valor de x foi declarado como LOCAL, portanto ele é restrito a procedure Main. Em segundo lugar, quando ele é passado para a função Muda na forma de parâmetro, o que o programa fez foi gerar uma outra cópia LOCAL de x e trabalhar com ela dentro da função Muda. Assim, apesar de termos o mesmo nome (variável x), o que temos na realidade são duas variáveis: a variável x que é local a função Main e a variável x que é local a função Muda. Temos, então duas variáveis. Nós não podemos ter duas variáveis com o mesmo nome dentro da mesma rotina, isso geraria um erro. Mas nós podemos ter variáveis diferentes, totalmente independentes, em rotinas diferentes. Pode parecer complicado no início, mas com a prática você verá que isso é uma grande vantagem que as modernas linguagens de programação possuem.

13.1.2 Passagem de parâmetros por referência

De acordo com Manzano e Oliveira,

> Parâmetros têm por finalidade servir como um ponto de comunicação bidirecional entre uma rotina e o programa principal, ou com uma outra sub-rotina hierarquicamente de nível mais alto. Desta forma, é possível passar valores de uma sub-rotina ou rotina chamadora à outra sub-rotina e vice-versa. [Manzano e Oliveira 2008, p. 179]

Por enquanto nós estudamos apenas a passagem de parâmetros de uma rotina chamadora para uma sub-rotina chamada. No nosso exemplo (listagem 13.3) a rotina chamadora é a Main e a rotina chamada é a Muda. Os valores são passados de Main para Muda e uma nova variável é criada. Existem casos, porém, em que você

precisa modificar o valor de x dentro da rotina Muda. Nesses casos, você deve alterar a forma com que o parâmetro é passado. Chamamos essa passagem de passagem de parâmetros por referência. Veja um exemplo logo abaixo, na listagem 13.4.

Listagem 13.4: LOCAL
Fonte: codigos/local04.prg

```
/*
Declaração com a variável local
passagem de parâmetro por referência
*/
PROCEDURE Main
LOCAL x

   x := 10
   ? x
   Muda( @x )
   ? x

RETURN

PROCEDURE Muda( x )

   x :=  200
   ? x

RETURN
```

.:Resultado:.

```
        10
       200
       200
```

Pronto, a única coisa que nós precisamos fazer para passar um valor por referência é prefixar o valor com um arroba (@), na linha 10. O arroba diz ao compilador para ele considerar o valor passado não como uma cópia da variável, mas como a própria variável. O termo “referência” usado para definir essa passagem de parâmetros remonta um recurso presente nas linguagens de programação chamado de “ponteiro”.

13.2 Função: um tipo especial de Procedure

De acordo com Ramalho, “uma função sempre retorna um valor, enquanto que uma procedure não” [Ramalho 1991, p 467]. Se a única diferença entre a procedure e uma função é a ausência de valores de retorno, então tudo o que vale para as procedures vale para as funções também, com a diferença do tal “valor de retorno” que a função tem e a procedure não. Nós já vimos alguma coisa sobre as funções. Vamos rever rapidamente um exemplo através da função que calcula a raiz quadrada.

```
PROCEDURE Main

```

```
   ? SQRT( 64 )

RETURN
```

.:Resultado:.

```
  8
```

O valor de retorno de uma função, portanto, é o valor que ela retorna para o contexto que a invocou. Lembra do exemplo do oráculo do capítulo 3 ? Se tiver alguma dúvida revise a seção desse capítulo que introduz ao uso de funções.

Você deve estar lembrado que SQRT() retorna a raiz quadrada de um número positivo. E se você mesmo fosse escrever essa função, como você faria ? O código da listagem 13.5 mostra que a função nada mais é do que uma procedure com um valor de retorno. Simples.

Listagem 13.5: Minha primeira função
Fonte: codigos/myfunc.prg

```
/*
Minha primeira função
*/
PROCEDURE Main
LOCAL nX

    ? "Cálculo da raiz quadrada"
    INPUT "Informe um inteiro positivo : " TO nX
    ? MySQRT( nX )

RETURN

FUNCTION MySQRT( nValor )

RETURN ( nValor ^ ( 1 / 2 ) )
```

.:Resultado:.

```
Informe um inteiro positivo para cálculo de raiz quadrada : 64
         8.00
```

Pronto. As diferenças entre uma PROCEDURE e uma FUNCTION são :

1. A procedure inicia com o identificador PROCEDURE, a função inicia com o identificador FUNCTION;
2. A procedure não tem valor de retorno. A função tem um valor de retorno através do comando RETURN <valor>.

14 Classes de variáveis

A sorte favorece a mente preparada.

Louis Pasteur

Objetivos do capítulo

- Ser apresentado as quatro classes de variáveis.
- Compreender o significado de grau de acoplamento.
- Entender o significado de escopo.
- Saber porque as variáveis LOCAIS são as mais indicadas para uso.
- Entender o uso de variáveis STATIC.
- Entender o que é tempo de vida de uma variável.

14.1 Classes básicas de variáveis

Iremos agora ver as quatro classes de variáveis do Harbour. Esse assunto é muito importante e, caso não seja bem compreendido, poderá influenciar negativamente todo o aprendizado a seguir. Antes de mais nada, precisamos esclarecer que essa classificação nada tem a ver com os tipos de dados estudados anteriormente. O Harbour possui quatro tipo de classe de variáveis :

1. Private
2. Public
3. Local
4. Static

Dessas classes nós já vimos parcialmente duas: a Private (que é o padrão) e a Local, que é a classe que você deve usar na maioria dos casos. Iremos usar uma abordagem histórica para explicar as classes de variáveis do Harbour, pois acreditamos ser essa a melhor forma de ensino.

14.1.1 Antigamente todas as variáveis eram PUBLIC

Nos primórdios da programação de computadores todas as variáveis eram públicas. O que isso significa ? Bem, vamos explicar isso através do Harbour usando a listagem 14.1.

Listagem 14.1: Variável PUBLIC
Fonte: codigos/escopo01.prg

```
/*
Escopo e tempo de vida
*/
PROCEDURE Main

   SeuNome()
   ? "O seu nome é : ", cNome

RETURN

PROCEDURE SeuNome

   PUBLIC cNome
   /* Pede e exibe o nome do usuário */
   ACCEPT "Informe o seu nome : " TO cNome

RETURN
```

O programa acima funciona perfeitamente, porém ele se utiliza de técnicas comprovadamente arcaicas de programação. Abaixo temos uma execução desse programa.

.:Resultado:.

```
Informe o seu nome : Vlademiro
O seu nome é :  Vlademiro
```

Mas, o que há de errado com as técnicas usadas por esse programa ? Para que você entenda o erro, vamos voltar no tempo até a década de 1970. Lá surgiu um conceito fundamental na programação de computadores chamado de “acoplamento”. O que é isso ?

Acoplamento é uma medida de interação entre duas rotinas. Quanto maior o grau de acoplamento, mais uma rotina depende da outra.
Segundo Schach:

> Dois módulos apresentam acoplamento comum se ambos tiverem acesso aos mesmos dados globais. Em vez de se comunicarem entre si por intermédio de passagem de argumentos. [Schach 2009, p. 187]

No nosso código (listagem 14.1) temos um caso clássico de acoplamento. A procedure Main usa uma variável que foi declarada e inicializada dentro de uma outra rotina. Em rotinas mais complexas, com um número muito grande de sub-rotinas e com milhares de linhas, essa técnica de “expor a variável” para que ela possa ser alterada fora da rotina em que ela foi criada, já levou a inúmeros erros de programação. Quanto maior o acoplamento, maior a dependência que um módulo tem do outro, e isso não é bom. **Quanto menor for o grau de acoplamento, melhor para o módulo, pois diminui a dependência que um módulo tem de outro**. Imagine o software como uma máquina. Um carro, por exemplo. Um automóvel é composto de inúmeras peças que, juntas, trabalham para que ele consiga desempenhar a sua função de veiculo. Uma peça que está no interior do motor (um cilindro, por exemplo) não interage diretamente com uma peça que está no interior do sistema de freios. A vantagem disso é que, se der um problema no sistema de freios, eu só vou “mecher” no sistema de freios, sem precisar me preocupar com as demais peças. O sistema todo funciona como uma unidade, mas dentro dele existem diversos componentes independentes e interrelacionados. Da mesma forma é um programa de computador, se eu expor um componente de uma rotina (uma variável) para que ela possa ser alterada por outra rotina eu estou criando um sistema cujas fronteiras entre as “peças” não existem na realidade, e isso é um grande problema.

Não se engane: o seu sistema sempre terá um certo grau de acoplamento, afinal de contas a própria palavra "sistema"já nos remete a cooperação entre partes de um todo. O trabalho do programador, em qualquer linguagem, é diminuir esse grau de acoplamento. Por exemplo: a substituição de variáveis públicas por variáveis passadas por parâmetros irá diminuir o grau de acoplamento, mas não irá eliminá-lo. Uma função que possui dez parâmetros possui um grau de acoplamento maior do que uma função com apenas um parâmetro em forma de array. Assim, a quantidade de parâmetros de uma função é uma medida de acoplamento, ou nas palavras de Yourdon e Constantine: "o número de ítens diferentes sendo passados (não a quantidade de dados) - quanto mais ítens, maior é o acoplamento"[Yourdon 1992, p. 97]. Isso não quer dizer que você deve ir trocando todos os parâmetros por arrays, as regras não devem ser seguidas ao

pé-da-letra. Esse nosso estudo inicial serve mais para uma compreensão do problema a ser enfrentado. Não esqueça: programação é uma atividade prática.

Dica 78

No Harbour as variáveis PUBLIC representam uma exceção em relação as outras classes. Nas outras classes, quando nós declaramos uma variável e não a inicializamos, o valor dela é NIL. Já as variáveis PUBLIC quando são declaradas sem inicialização recebem o valor lógico falso. Uma curiosidade é que, se você declarar uma variável PUBLIC chamada CLIPPER (ou HARBOUR) e não a inicializar, o seu valor inicial será um lógico verdadeiro. Você pode achar que isso é algum tipo de easter egg [a] mas não é nada disso. Segundo Vidal, esse valor diferente serviu para que um programador escrevesse um mesmo código tanto para o dBase quanto para o Clipper. "Dessa forma, no Clipper, os novos comandos serão executados, enquanto no dBase III não" [Vidal 1989, p. 15]. Isso permitia a execução do mesmo programa tanto em um como no outro. O fragmento a seguir ilustra isso:

```
PUBLIC clipper

IF clipper
    // Código que só funciona no clipper e harbour
ELSE
    // Código que só funciona em um interpretador.
    // dBase III por exemplo
ENDIF
```

O Harbour copiou o mesmo comportamento, de modo que se você criar uma variável pública chamada HARBOUR, esta terá o valor inicial .T.

```
PROCEDURE Main                                                  1
                                                                2
    PUBLIC CLIPPER                                              3
    PUBLIC HARBOUR                                              4
    PUBLIC OUTRA                                                5
                                                                6
    ? M->CLIPPER , M->HARBOUR, M->OUTRA                         7
    //  .T.            .T.          .F.                         8
                                                                9
RETURN                                                          10
```

[a] Em informática, um ovo de páscoa (ou easter egg, tradução para o inglês, como é mais conhecido) é qualquer coisa oculta, podendo ser encontrada em qualquer tipo de sistema virtual, incluindo músicas, filmes, videogames etc. - Fonte: https://en.wikipedia.org/wiki/Easter_egg_(media) - On Line: 16-Out-2016

14.1.2 Variáveis PRIVATE: A solução (não tão boa) para o problema das variáveis PUBLIC

A solução encontrada, na época, para resolver o problema gerado pelas variáveis PUBLIC, foi criar uma classe especial de variável cujo conteúdo era privativo ao módulo em que ela foi criada. Esse ideia originou uma classe de variáveis chamada de PRIVATE. Essa seria a solução ideal, mas ela não resolveu completamente o problema, talvez (opinião pessoal minha) por não ter sido implementada corretamente pelos desenvolvedores da época.

Tínhamos, então duas classes de variáveis: PUBLIC e PRIVATE. As variáveis PRIVATE ainda hoje são o tipo padrão (default) de variável. Ou seja, se você simplesmente criar uma variável sem dizer a que classe ela pertence ela será do tipo PRIVATE. A listagem 14.2 nos mostra um uso típico de uma variável PRIVATE.

Listagem 14.2: Variável PRIVATE
Fonte: codigos/escopo02.prg

```
/*
Escopo e tempo de vida
*/
PROCEDURE Main
PUBLIC x

   Quadrado()

   ? "O valor de x é " , x

RETURN

PROCEDURE Quadrado

   PRIVATE x

   INPUT "Informe um número : " TO x

   ? "O valor ao quadrado é ", x ^ 2

RETURN
```

Veja, na execução abaixo, que nós declaramos duas variáveis x: uma PUBLIC, e uma PRIVATE. Note que o valor de uma não interfere no valor da outra.

.:Resultado:.

```
Informe um número : 2
O valor ao quadrado é           4.00
O valor de x é  .F.
```

Com o surgimento das variáveis PRIVATE surgiu um conceito novo na programação: o conceito de “escopo” de variável. Não é um conceito difícil, vamos explicar por partes: a palavra escopo significa delimitação de atividade ou abrangência. No nosso contexto seria mais ou menos assim: “até onde o valor de x é visível ?” Nas antigas variáveis

PUBLIC, o escopo era o programa inteiro, ou seja, após a variável PUBLIC ser criada ela só era destruída ao final do programa e o seu valor era visível de qualquer local. Mas com o surgimento das variáveis PRIVATE o escopo passou a ser a rotina em que a variável se encontra e todas as rotinas “abaixo” dela. A listagem 14.3 exemplifica o conceito de escopo de uma variável PRIVATE.

Listagem 14.3: Variável PRIVATE - Escopo de variável
Fonte: codigos/escopo03.prg

```
/*
Escopo e tempo de vida
*/
PROCEDURE Main
PUBLIC x

   Quadrado()

   ? "O valor de x é " , x

RETURN

PROCEDURE Quadrado

   PRIVATE x

   INPUT "Informe um número : " TO x
   ? "O valor ao quadrado é ", x ^ 2

   Cubo()

RETURN

PROCEDURE Cubo

   ? "O valor ao cubo é ", x ^ 3

RETURN
```

.:Resultado:.

```
Informe um número : 3
O valor ao quadrado é           9.00
O valor ao cubo é          27.00
O valor de x é  .F.
```

Note que a variável x, que foi declarada como PRIVATE na procedure Quadrado, pode ser usado pela procedure Cubo também. É bom ressaltar que a procedure Cubo só pode usar o valor de x porque ela está “abaixo” de (sendo chamada por) Quadrado. Conforme a figura 14.1.

Figura 14.1: Escopo

Lembre-se que, se eu tiver duas variáveis com o mesmo nome, mas declaradas em rotinas diferentes a variável que prevalecerá sempre será a PRIVATE. Ou seja, se eu tiver uma PRIVATE e uma PUBLIC (como aconteceu nas procedures Quadrado e Cubo com x) o valor que prevalecerá será o da variável PRIVATE. Quando a execução do programa "sair" da rotina Quadrado (que foi a rotina que declarou x como PRIVATE), então a variável x PRIVATE perderá o seu valor e apenas a variável x PUBLIC existirá (por isso que o último valor exibido foi um valor lógico (.f.), já que a variável PRIVATE tinha sido descartada e sobrou apenas a variável x PUBLIC).

A variável PRIVATE resolve parcialmente o problema do acoplamento de rotinas. Vamos criar um outro exemplo.

Listagem 14.4: Variável PRIVATE
Fonte: codigos/escopo04.prg

```
/*
Escopo e tempo de vida
*/
PROCEDURE Main
PUBLIC x

   Quadrado()

   ? "O valor de x é " , x

RETURN

PROCEDURE Quadrado

   PRIVATE x

   INPUT "Informe um número : " TO x
```

```
   ? "O valor ao quadrado é ", x ^ 2

   Cubo()
   RaizQuadrada()

RETURN

PROCEDURE Cubo

   ? "O valor ao cubo é ", x ^ 3

RETURN

PROCEDURE RaizQuadrada

   ? "A raiz quadrada é ", SQRT( x )

RETURN
```

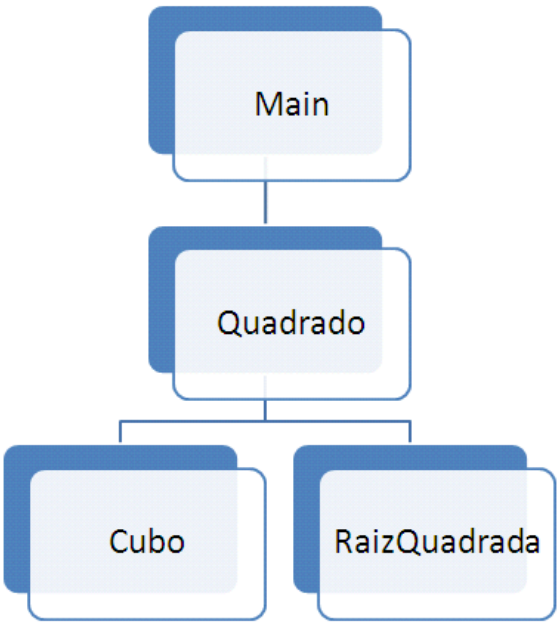
Agora nós adicionamos outra rotina chamada RaizQuadrada, dentro da rotina Quadrado. Ela funciona perfeitamente.

.:Resultado:.

```
Informe um número : 2
O valor ao quadrado é           4.00
O valor ao cubo é           8.00
A raiz quadrada é           1.41
O valor de x é  .F.
```

A nossa representação gráfica das rotinas ficou agora assim:

Figura 14.2: Escopo 2



.

Atenção, pois vamos agora simular um problema com as variáveis PRIVATE. Imagine que o programador, inadvertidamente, alterou o interior da procedure Cubo e atribui um valor caractere ao valor de x, conforme o exemplo da listagem 14.5 (linha 28).

Listagem 14.5: Variável PRIVATE
Fonte: codigos/escopo05.prg

```
/*
Escopo e tempo de vida
*/
PROCEDURE Main
PUBLIC x

   Quadrado()

   ? "O valor de x é " , x

RETURN

PROCEDURE Quadrado

   PRIVATE x

   INPUT "Informe um número : " TO x
   ? "O valor ao quadrado é ", x ^ 2

   Cubo()
   RaizQuadrada()

RETURN

PROCEDURE Cubo

   ? "O valor ao cubo é ", x ^ 3
   x := "Atribuição incorreta"

RETURN

PROCEDURE RaizQuadrada

   ? "A raiz quadrada é ", SQRT( x )

RETURN
```

Veja o resultado: um erro de execução. Note que o erro aconteceu na linha 34 (RaizQuadrada), mas a causa do erro estava em outra rotina (na linha 28 da rotina Cubo). O problema com o alto grau de acoplamento não foi completamente solucionado com as variáveis PRIVATE. Erros não devem ocorrer (é claro), mas caso eles ocorram, pelo menos devem envolver apenas uma rotina e não duas ou mais, como esse exemplo.

.:Resultado:.

```
Informe um número : 4
O valor ao quadrado é          16.00
O valor ao cubo é          64.00
Error BASE/1097  Argument error: SQRT
Called from SQRT(0)
Called from RAIZQUADRADA(34)
Called from QUADRADO(21)
Called from MAIN(7)
```

Dica 79

Quando uma variável PRIVATE é declarada na rotina Main de um programa ela se comporta como uma variável PUBLIC, pois ela passa a ser vista por todas as rotinas abaixo de Main. A única diferença, nós já vimos, é que uma variável PRIVATE sem inicialização recebe o valor NIL, e uma variável PUBLIC, sem inicialização recebe o valor lógico falso.

14.1.3 Variáveis LOCAL: A solução definitiva para o problema do acoplamento

Finalmente chegamos a solução definitiva para o problema gerado pelo alto grau de acoplamento: as variáveis com escopo local ou de rotina. Nós já vimos um pouco sobre as variáveis locais no capítulo anterior e temos insistido, desde o começo do livro, que elas devem ser usadas no lugar das demais. Agora a transmissão de valores entre rotinas deixa de ser feito através do acesso direto ao valor de uma variável em outra rotina (acoplamento alto) e passa a ser feito através de passagem de argumentos (parâmetros). O exemplo da listagem 14.6 ilustra essa solução.

Listagem 14.6: Variável LOCAL
Fonte: codigos/escopo06.prg

```
/*
Escopo e tempo de vida
*/
PROCEDURE Main
PUBLIC x

   Quadrado()

   ? "O valor de x é " , x

RETURN

PROCEDURE Quadrado
LOCAL x

   INPUT "Informe um número : " TO x
   ? "O valor ao quadrado é ", x ^ 2

   Cubo( x )
   RaizQuadrada( x )
```

```
                                                                    21
RETURN                                                              22
                                                                    23
PROCEDURE Cubo( x )                                                 24
                                                                    25
   ? "O valor ao cubo é ", x ^ 3                                    26
   x := "Atribuição incorreta"                                      27
                                                                    28
RETURN                                                              29
                                                                    30
PROCEDURE RaizQuadrada( x )                                         31
                                                                    32
   ? "A raiz quadrada é ", SQRT( x )                                33
                                                                    34
RETURN                                                              35
```

Deixamos, propositadamente, a atribuição na linha 27 para que você veja que ela ficou restrita a procedure Cubo, e não interfere em valores externos.

.:Resultado:.

```
Informe um número : 2
O valor ao quadrado é           4.00
O valor ao cubo é           8.00
A raiz quadrada é           1.41
O valor de x é  .F.
```

Fique atento, também ao ponto onde a declaração LOCAL deve ser feita. O posicionamento da declaração é muito importante, pois a variável LOCAL é definida em tempo de compilação, economizando recursos de implementação. Os antigos desenvolvedores do Clipper chamavam isso de “abrangência léxica” [Spence 1991, p. 87]. A consequência prática disso é que você não pode declarar uma variável local em qualquer parte do programa, você deve sempre usar a declaração LOCAL nas primeiras linhas da rotina, antes de qualquer declaração, comando ou função.

14.1.4 Variáveis STATIC e o surgimento de um novo conceito

Existem dois tipos de variáveis STATIC: interna e externa. Vamos, primeiro abordar as variáveis STATIC externas e em seguida abordaremos as variáveis STATIC internas.

STATIC externa: a alternativa viável às variáveis PUBLIC

Os problemas gerados com o alto grau de acoplamento foram eliminados com o uso de variáveis locais, mas existem casos que um certo grau de acoplamento é desejável. As variáveis PUBLIC favorecem a um grau altíssimo de acoplamento, pois a variável possui um escopo de aplicação. Portanto, as variáveis PUBLIC não são indicadas para resolver esses problemas (na verdade as variáveis PUBLIC não são indicadas em caso algum). Pensando nisso os desenvolvedores criaram um tipo especial de variável chamada de STATIC externa. A ideia por trás de uma variável STATIC externa é a seguinte: existem valores que você pode compartilhar entre um grupo de rotinas, dessa forma, apenas um grupo de rotinas irá compartilhar a variável, e não o programa todo. Como isso acontece na prática ? Bem, sabemos que um programa pode ser

composto de vários arquivos fonte (arquivos com extensão PRG). Se você agrupar as rotinas por arquivo, então você pode criar uma variável STATIC externa, cujo escopo é o arquivo. As variáveis STATIC externas, portanto, substituem as variáveis PUBLIC (você só não deve esquecer de privilegiar o uso das variáveis locais). Só use a variável STATIC externa quando for estritamente necessário.

Em que situação uma variável STATIC externa é aconselhada ? Essa pergunta é muito importante, já que na grande maioria das vezes você deverá usar a variável LOCAL. Bem, vamos exemplificar através do uso de duas funções já estudadas: SaveScreen e RestScreen. Você deve estar lembrado, SaveScreen salva um trecho da tela e RestScreen restaura esse trecho que foi salvo. Mas tem um pequeno inconveniente: você deve repetir as coordenadas desnecessariamente usando RestScreen. Vamos exemplificar a seguir.

Listagem 14.7: Variável STATIC externa
Fonte: codigos/saverest.prg

```
1
FUNCTION Main                                                                   2
LOCAL cTela, x                                                                  3
                                                                                4
   hb_cdpSelect("UTF8")                                                         5
   CLS                                                                          6
   @ 10,10 TO 14,20                                                             7
   @ 12,12 SAY "Harbour"                                                        8
   cTela := SaveScreen( 10 , 10 , 14, 20 )                                      9
   @ 16,10 SAY ;                                                                10
                                                                                11
   "Um trecho dessa tela será apagada para ser restaurada logo depois."
   @ 17,10 SAY "TECLE ALGO PARA COMEÇAR!"                                       12
   INKEY( 0 )                                                                   13
   FOR x := 1 TO 100                                                            14
       ? x , SPACE(70)                                                          15
   NEXT                                                                         16
   ? "Tecle algo para restaurar a tela..."                                      17
   INKEY(0)                                                                     18
                                                                                19
   RestScreen( 10 , 10, 14, 20, cTela )                                         20
                                                                                21
RETURN NIL                                                                      22
```

Compile e execute o programa acima.

O programa funciona perfeitamente, mas não seria mais prático para você se não fosse preciso repetir as coordenadas informadas para SaveScreen (na linha 9) em RestScreen (na linha 19) ? Podemos desenvolver uma solução para esse pequeno inconveniente. Acompanhe a listagem 14.8 e 14.9. Na listagem 14.8 temos um código semelhante a listagem 14.7, as duas únicas diferenças foram nas linhas 9 e 19, pois nós substituimos as funções SaveScreen e RestScreen por umas procedures que nós criamos: BoxSave e BoxRest (essas procedures estão em telas.prg). Acompanhe com atenção.

O arquivo abaixo chama-se saverest.prg. As diferenças estão nas linhas 9 e 19.

Listagem 14.8: Variável STATIC externa
Fonte: codigos/saverest2.prg

```
FUNCTION Main
LOCAL x

   hb_cdpSelect("UTF8")
   CLS
   @ 10,10 TO 14,20
   @ 12,12 SAY "Harbour"
   cTela := BoxSave( 10 , 10 , 14, 20 )
   @ 16,10 SAY ;

   "Um trecho dessa tela será apagada para ser restaurada logo depois."
   @ 17,10 SAY "TECLE ALGO PARA COMEÇAR!"
   INKEY( 0 )
   FOR x := 1 TO 100
       ? x , SPACE(70)
   NEXT
   ? "Tecle algo para restaurar a tela..."
   INKEY(0)

   BoxRest()

RETURN NIL
```

O arquivo abaixo chama-se telas.prg.

Listagem 14.9: Variável STATIC externa
Fonte: codigos/escopo07.prg

```
/*
Escopo e tempo de vida
*/
STATIC nLin1
STATIC nLin2
STATIC nCol1
STATIC nCol2
STATIC cTela
PROCEDURE BoxSave( nL1, nC1, nL2, nC2 )

    nLin1 := nL1
    nCol1 := nC1
    nLin2 := nL2
    nCol2 := nC2
    cTela := SaveScreen( nLin1, nCol1, nLin2, nCol2 )

RETURN

PROCEDURE BoxRest()

    RestScreen( nLin1, nCol1, nLin2, nCol2, cTela )
```

```
                                                                    22
RETURN                                                              23
```

Parece complicado mas não é. O programa possui dois arquivos. Eles são compilados conforme abaixo:

.:Resultado:.

```
hbmk2 saverest telas
```

Vamos analisar o arquivo telas.prg. O nosso objetivo é não ter que repetir nomes e coordenadas quando for restaurar a tela. Por isso os nomes e as coordenadas foram definidas como variáveis STATIC externas. A vantagem dessas variáveis sobre as PUBLIC é porque elas (nLin1, nLin2, nCol1, nCol2 e cTela) só são visíveis na rotina telas.prg, e estão ocultas (são invisíveis) na rotina saverest.prg (ou em qualquer outro arquivo PRG que possa vir a ser adicionado ao projeto no futuro).

STATIC interna: quando você desejar preservar o valor

Existem situações (bem específicas) em que você necessita preservar o valor de uma variável após o término da rotina. Nesse caso, você pode usar variáveis STATIC interna. O exemplo 14.10 ilustra uma situação assim.

Listagem 14.10: Variável STATIC interna
Fonte: codigos/escopo08.prg

```
/*                                                                  1
Escopo e tempo de vida                                              2
*/                                                                  3
PROCEDURE Main                                                      4
                                                                    5
   Teste(1)                                                         6
   Teste(1)                                                         7
   Teste(1)                                                         8
   Teste(1)                                                         9
                                                                    10
RETURN                                                              11
                                                                    12
PROCEDURE Teste( nIncrementa )                                      13
STATIC nAtual := 0 // Só será feito a primeira vez                  14
                                                                    15
   nAtual += nIncrementa                                            16
   ? "O valor de x é : " , nAtual                                   17
                                                                    18
RETURN                                                              19
```

.:Resultado:.

```
O valor de x é :           1
O valor de x é :           2
O valor de x é :           3
O valor de x é :           4
```

Note que o valor com que x é inicializado (na linha 14) **SÓ TEM EFEITO NA PRIMEIRA EXECUÇÃO DA ROTINA**. A partir da segunda execução o programa não executa mais a linha 14, e passa a reaproveitar o valor anterior para realizar os incrementos.

14.1.5 Escopo e Tempo de Vida

Depois que surgiram as variáveis STATIC, dois conceitos que eram a mesma coisa passaram a ser tratados distintamente, são eles : escopo e tempo de vida. Você deve estar lembrado do escopo (abrangência) de uma variável. Pois bem, quando o escopo das variáveis PUBLIC, PRIVATE e LOCAL acaba a variável deixa de existir. Por exemplo: quando uma rotina que usa uma variável LOCAL termina, a variável LOCAL deixa de existir (a memória é liberada). Ou seja, escopo e o tempo de vida são os mesmos.

Com as variáveis STATIC esses conceitos, que antes eram juntos, passaram a ser tratados em separado. **O tempo de vida de uma variável STATIC sempre é o programa todo**, da mesma forma que a variável PUBLIC. A diferença é que as variáveis STATIC possuem dois escopos distintos: a variável STATIC externa possui escopo de arquivo (ela é vista por todos do arquivo PRG ao qual pertence) e a variável STATIC interna possui escopo de rotina (ela é vista apenas na rotina a qual pertence). Veja bem que a variável STATIC interna é bem parecida com a variável LOCAL, pois o escopo de ambas é a rotina. A diferença é que, quando a rotina acaba, a variável LOCAL é destruída, e a variável STATIC interna não é destruída, mas fica invisível (sem efeito) fora da rotina na qual foi criada[1].

14.2 Rotinas estáticas

Uma rotina (PROCEDURE ou FUNÇÃO) pode receber o escopo de uma variável estática. Quando isso acontecer quer dizer que a rotina **apenas é visível dentro do arquivo PRG em que ela está definida**. A maioria das rotinas não são estáticas, portanto elas podem ser “vistas” de qualquer lugar do programa, mas caso essa rotina seja estática, então o seu escopo é apenas o PRG onde ela foi criada. As consequências práticas disso são :

1. Você pode criar duas rotinas com o mesmo nome mas em PRGs diferentes. Pelo menos uma delas deve ser estática.

2. Você pode ter rotinas bem exclusivas sem se preocupar se o programa tem uma outra rotina com o mesmo nome.

Criar rotinas estáticas é simples, basta atribuir o identificador STATIC antes do nome FUNCTION ou PROCEDURE, conforme o exemplo abaixo :

```
STATIC FUNCTION Delta( a , b , c )
```

[1] Esse conceito de variáveis estáticas internas e externas não é exclusivo do Harbour. A Linguagem C também tem essa característica.

. . .

Essa função só será visível de dentro do PRG onde ela foi criada. Quando usar essas funções ? Bem, um caso onde o uso desse tipo de função é aconselhada é quando você precisa realizar operações auxiliares para uma função principal. Ou seja, a função estática nunca é chamada diretamente de outras rotinas, mas ela serve de apoio para uma função complexa. Por exemplo: se você criar um arquivo chamado matematica.prg e dentro dele tiver uma função que calcula as raízes de uma equação de segundo grau, você pode, dentro desse arquivo, ter uma função auxiliar estática para calcular o valor de delta (que é usado pela função principal que calcula as raízes da equação de segundo grau).

14.3 Conclusão

Esse capítulo tratou de conceitos importantes no tocante a variáveis. Vimos que, historicamente, temos duas subdivisões: a primeira envolve variáveis PUBLIC e PRIVATE (eram usados pelo antigo dBase e Clipper Summer, na década de 1980), e depois temos o segundo tipo, cujas variáveis são LOCAL e STATIC (usados a partir da década de 1990 até hoje). Você sempre deve usar variáveis LOCAL e STATIC. Se, por acaso, você sentir a necessidade de usar uma variável PRIVATE ou (o que é pior) uma variável PUBLIC, isso quer dizer que o seu código está com algum problema de legibilidade [2].

[2]Não se iluda achando que não voltará a alterar um código, pois sempre irá surgir uma situação que irá requerer de você uma alteração em um trecho antigo.

15 Controle de erros

E quando você perder o controle, você colherá o que plantou.

David Gilmour / Roger Waters - Dogs

Objetivos do capítulo

- Os tipos de erros.
- Aprender a tratar os erros que podem surgir na sua aplicação.
- Entender a lógica por trás do gerenciador de erros do Harbour.
- Compreender o funcionamento do BEGIN SEQUENCE ... END.

15.1 O erro

O erro é uma condição humana, não tem como evitar. O que nós podemos fazer é programar defensivamente, usar boas práticas e modularizar a nossa aplicação. Sabendo disso os projetistas de linguagens de programação criaram estruturas específicas para ajudar na manipulação de erros. Nesse capítulo iremos estudar algumas estruturas de controles adicionais que foram criadas especificamente para manipular erros de execução. Mas antes vamos dar uma parada para classificar os erros que podem ocorrer em um sistema de informação. Não será uma parada muito longa pois nós já tratamos desse assunto nos capítulos anteriores, será mais uma revisão.

15.1.1 Os tipos de erros

Os erros classificam-se em : erros de compilação, linkedição, execução e de lógica. As subseções a seguir trata os erros separadamente.[1]

Erros de compilação

Erros de compilação, também conhecido como erros de compilador, são erros que impedem seu programa de executar. Quando o Harbour compila o código em um idioma binário que o computador compreende ele pode encontrar algum código que ele não entende, então ele emite um erro de compilador. A maioria dos erros de compilador são causados por erros que você faz ao digitar o código. Por exemplo, você pode errar uma palavra-chave, esquecer algumas pontuações necessárias ou tentar usar uma instrução End If sem primeiro usar uma instrução If.

Erros de linkedição

Erros de linkedição são erros que ocorrem antes do programa ser gerado, mas após o processo de compilação. Vimos no capítulo 2 o processo de criação de um programa. O processo de linkedição é o processo intermediário, quando alguns arquivos auxiliares (chamados de bibliotecas) são acrescentados ao código compilado. No exemplo a seguir nós simulamos um erro de linkedição.

Listagem 15.1: Erro de linkedição

```
/*
Erro de linkedição
*/
PROCEDURE Main

  ? Year2()

RETURN
```

Note que nós usamos, propositadamente, uma função que não existe (YEAR2()). Quando nós formos compilar o programa ele irá compilar normalmente, mas a etapa de linkedição irá retornar um erro, cancelando a geração do programa.

[1] Essas subseções foram adaptadas de `https://msdn.microsoft.com/pt-br/library/s9ek7a19(v=vs.90).aspx` (Acessada em 10-12-2016)

.:Resultado:.

```
> hbmk2 errolink
hbmk2: Processando script local: hbmk.hbm
hbmk2: Harbour: Compilando módulos...
Harbour 3.2.0dev (r1507030922)
Copyright (c) 1999-2015, http://harbour-project.org/
Compiling 'errolink.prg'...
Lines 10, Functions/Procedures 1
Generating C source output to '.hbmk\win\mingw\errolink.c'... Done.
hbmk2: Compilando...
hbmk2: Linkando... errolink.exe
.hbmk/win/mingw/errolink.o:errolink.c:(.data+0x48): undefined
   reference to `HB_FUN_YEAR2'

hbmk2: Erro: Referenciado, faltando, mas funções desconhecida(s):
   YEAR2()
```

Note que o processo de compilação ocorre normalmente, mas na linkagem ocorre o erro. Note também que o Harbour retorna o nome do arquivo PRG que chamou a função inexistente, no nosso exemplo o arquivo é errolink.prg. Observe que o Harbour também retorna o nome da função que ele não conseguiu encontrar, mas esse nome tem um prefixo HB_FUN_. Ao final o Harbour emite uma mensagem com o nome da função sem o prefixo, conforme abaixo:

.:Resultado:.

```
hbmk2: Erro: Referenciado, faltando, mas funções desconhecida(s):
   YEAR2()
```

Erros de execução

Erros em tempo de execução são erros que ocorrem enquanto o programa é executado. Eles normalmente ocorrem quando o programa tenta uma operação que é impossível executar. Um exemplo disso é a divisão por zero. Suponha que você tinha a instrução a seguir:

```
Speed := Miles / Hours
```

Se a variável Hours possui um valor de 0, a operação de divisão falha e faz com que ocorra um erro em tempo de execução. O programa deve ser executado de modo a esse erro ser detectado, e se Hours contiver um valor válido, ele não ocorrerá.

Erros de lógica

Erros lógicos são erros que impedem seu programa de fazer o que você pretendia fazer. Seu código pode ser compilado e executado sem erros, mas o resultado de uma operação pode produzir um resultado que você não esperava. Por exemplo, você pode ter uma variável chamada FirstName que é inicialmente definida como uma sequência de caracteres em branco. Posteriormente no seu programa, você pode concatenar

FirstName com outra variável chamada LastName para exibir um nome completo. Se você esqueceu de atribuir um valor para FirstName, apenas o último nome seria exibido, não o nome completo como você pretendia.

Os erros lógicos são os mais difíceis de se localizar e corrigir, pois eles não impedem o programa de funcionar.

15.2 Saindo prematuramente do programa

Antes de prosseguirmos vamos estudar um comando que é bastante usado nas funções de gerenciamento: o comando QUIT. Esse comando faz com que o programa seja finalizado mesmo sem ter encontrado a última linha de Main. Veja um exemplo simples:

Listagem 15.2: Quit

```
/*
Quit
*/
PROCEDURE Main

  ? "O comando QUIT será executado a seguir..."
  QUIT
  ? "Essa linha näo será executada."

RETURN
```

Quando o programa encontra esse comando ele é finalizado.

.:Resultado:.

```
O comando QUIT será executado a seguir...
```

Descrição sintática 21

1. Nome : QUIT
2. Classificação : comando.
3. Descrição : Finaliza a execução de um programa.
4. Sintaxe

```
QUIT
```

Fonte : [Nantucket 1990, p. 4-74]

Porque esse comando é importante ? Bem, geralmente quando um erro de execução ocorre, o programa precisa ser finalizado no ponto onde o erro ocorreu. Nessa hora o comando QUIT entra em ação, ele é usado pela função de manipulação de erros para finalizar uma aplicação prematuramente. Você pode usá-lo nas suas funções de

manipulação de erro ou pode usar sempre que desejar encerrar prematuramente a sua aplicação.

15.3 O gerenciador de erros do Harbour

O gerenciador de erros do Harbour entra em ação quando um erro de execução é gerado pelo programa. Quando isso ocorre, a sequência do programa é quebrada, e o controle passa para o gerenciador de erros do Harbour. O gerenciador de erros do Harbour também chama uma função que assume o controle do programa e providencia as etapas necessárias para que ele seja finalizado. Durante esse processo, os arquivos são fechados (através do comando QUIT) e uma mensagem de erro é gerada juntamente com algumas informações adicionais para auxiliar o programador no processo de descoberta do erro.

Nós dissemos que o gerenciador de erros do Harbour chama uma função que é executada durante o processo de erros. Essa função chama-se Errorsys() e pode ser alterada por você livremente caso você deseje.

Dica 80

Caso você seja um iniciante nós não aconselhamos a alteração da função ERRORSYS() que faz o tratamento de erros.

Um fato interessante é que você pode trocar essa função ErrorSys pela sua própria função gerenciadora de erros sem precisar modificar a função ErrorSys original. Vamos realizar algumas simulações nas subseções a seguir.

Dica 81

Em vez de substituir a ERRORSYS padrão, prefira criar a sua própria função de manipulação de erros e informar a sua aplicação que ela será a função padrão de manipulação de erros. Veremos como fazer isso adiante.

15.3.1 Trocando a função ErrorSys do Harbour por uma função que não faz nada

Nessa seção iremos trocar a função padrão de manipulação de erros do Harbour por uma função sem conteúdo. Qual a utilidade desse exemplo ? Apesar de não ter utilidade prática esse exemplo serve para :

- ensinar como realizar a troca de uma função gerenciadora de erros por outra;
- mostrar o que acontece quando ocorre um erro e o Harbour não tem um gerenciador eficaz para realizar o tratamento do erro;
- lhe convencer da importância de se ter um bom gerenciador de erros.

Primeiramente veremos um exemplo sem erros para entender o que ele faz.

Listagem 15.3: Errorsys

```
/*                                                                    1
```

```
Gerenciador de erros                                                    2
*/                                                                      3
PROCEDURE Main                                                          4
LOCAL xVar                                                              5
                                                                        6
  SET DATE BRITISH                                                      7
  SET CENTURY ON                                                        8
                                                                        9
  ? "Somando um valor a uma variável"                                   10
  xVar := DATE()                                                        11
  ? "A variável é " , xVar                                              12
  ? xVar += 3 // Somando 3 dias                                         13
  ? "A variável agora é " , xVar                                        14
                                                                        15
RETURN                                                                  16
```

A rotina simplesmente adiciona um valor a uma variável. No nosso caso adicionamos 3 dias a uma variável do tipo data.

.:Resultado:.

```
Somando um valor a uma variável
A variável é  14/12/2016
17/12/2016
A variável agora é  17/12/2016
```

Agora vamos gerar um erro mas não iremos alterar o modo como o Harbour trata esse erro. Acompanhe a seguir:

Listagem 15.4: Errorsys

```
/*                                                                      1
Gerenciador de erros                                                    2
*/                                                                      3
PROCEDURE Main                                                          4
LOCAL xVar                                                              5
                                                                        6
  SET DATE BRITISH                                                      7
  SET CENTURY ON                                                        8
                                                                        9
  ? "Somando um valor a uma variável"                                   10
  xVar := DATE()                                                        11
  ? "A variável é " , xVar                                              12
  ? xVar += "A" // Somando um caractere a uma data (ERRO!!)             13
  ? "A variável agora é " , xVar                                        14
                                                                        15
RETURN                                                                  16
```

Note que na linha 13 um valor caractere foi somado a uma variável do tipo data. O erro gerado pode ser visto abaixo.

.:Resultado:.

```
Somando um valor a uma variável
A variável é  14/12/2016
```

```
Error BASE/1081  Argument error: +
Called from MAIN(13)
```

Até agora não aprendemos nada de novo. No exemplo a seguir nós iremos trocar a função gerenciadora de erros por uma função sem conteúdo relevante. Acompanhe.

Listagem 15.5: Errorsys

```
/*
Gerenciador de erros
*/
PROCEDURE Main
LOCAL xVar

  SET DATE BRITISH
  SET CENTURY ON

  ERRORBLOCK( {|e| MeuManipuladorDeErros( e )} )

  ? "Somando um valor a uma variável"
  xVar := DATE()
  ? "A variável é " , xVar
  ? xVar += "A" // Somando um caractere a uma data (ERRO!!)
  ? "A variável agora é " , xVar

RETURN


FUNCTION MeuManipuladorDeErros( objError )

   ? "Estou dentro da funçäo manipuladora de erros."
   ? "Mas aqui näo tem nada!"

RETURN NIL
```

Na linha 10 nós trocamos a função gerenciadora padrão do Harbour pela nossa própria função chamada de MeuManipuladorDeErros(). Essa troca é feita pela função ERRORBLOCK() do Harbour. A partir desse ponto os erros de execução serão tratados por essa nova função. Note que essa nova função manipuladora não faz coisa alguma com o erro gerado, apenas imprime uma mensagem na tela e retorna o controle para a linha original.

.:Resultado:.

```
Somando um valor a uma variável
A variável é  14/12/2016
Estou dentro da função manipuladora de erros.
Mas aqui não tem nada!
NIL
A variável agora é  NIL
```

Note também que o programa não foi interrompido, pois o comando que o interrompe (QUIT) deveria estar dentro da função manipuladora de erros.

Em resumo: o erro de execução não foi reportado, apesar de ainda estar lá. Outro fato importante: as consequências do erro ainda existem, pois a variável xVar perdeu o valor original e agora é NIL. Os pontos principais que você deve ter em mente são três :

1. O Harbour irá identificar automaticamente um erros de execução assim que ele ocorre. Quando isso acontecer o controle (a sequência) será transferido para a função manipuladora de erros.
2. “O que fazer com o erro reportado” é algo que acontece dentro da função manipuladora de erros. Se for a função padrão do Harbour o programa geralmente será interrompido e algumas ações serão tomadas. Mas você pode trocar essa função padrão pela sua própria função manipuladora e tratar o erro livremente. Isso é feito através da função ERRORBLOCK().
3. As consequências do erro permanecem, caso o programa não seja abortado.

A função ERRORBLOCK

A função ERRORBLOCK() envia um bloco de código[2] a ser executado quando ocorre um erro em tempo de execução. O manipulador de erros é especificado como um bloco de código da seguinte forma:

```
{ |<objError>| <lista de expressões>,... }
```

Onde <objError> é uma variável especial[3] que contém informações sobre o erro. O bloco de tratamento de erros pode ser especificado como uma lista de expressões ou como uma chamada a uma função definida por usuário (como nós fizemos no exemplo da listagem 15.5. Uma chamada a uma função é mais útil pois assim você pode utilizar declarações de controle do Harbour ao invés de expressões.

15.3.2 Devolvendo o controle a função manipuladora do Harbour

Como você deve ter percebido, a função manipuladora do Harbour deve ser usada na maioria dos casos, porém, existem pequenas situações em que você pode preferir usar a sua própria rotina de manipulação de erros. Nesse ponto surge um problema: você pode até controlar os erros temporariamente com a sua rotina de manipulação de erros, mas vai chegar a hora de trocar pela rotina original do Harbour. Vamos aprender a devolver o controle no exemplo a seguir, ele é uma continuação da listagem 15.5.

Listagem 15.6: Errorsys

```
/*                                                                    1
```

[2] Ainda não estudamos o que é um bloco de código, mas não se preocupe. Você não precisa entender o que é um bloco de código para entender como o gerenciador de erros funciona. Nos capítulos a seguir nós veremos detalhadamente o que é um bloco de código, por ora fique atento aos exemplos e você entenderá o sistema de gerenciamento de erros mesmo sem saber o que é um bloco de código.

[3] Essa “variável especial” chama-se “objeto”. Você verá o que é um objeto nos capítulos posteriores, por enquanto não se preocupe caso não esteja entendendo tudo nos mínimos detalhes. Fique atento aos exemplos dados.

```
*Gerenciador de erros
*/
PROCEDURE Main
LOCAL xVar, bOriginal

  SET DATE BRITISH
  SET CENTURY ON

  bOriginal := ERRORBLOCK( {|e| MeuManipuladorDeErros( e )} )

  ? "Somando um valor a uma variável"
  xVar := DATE()
  ? "A variável é " , xVar
  ? xVar += "A" // Somando um caractere a uma data (ERRO!!)
  ? "A variável agora é " , xVar


   ? "Agora iremos controlar os próximos erros através da função original"
  ERRORBLOCK( bOriginal )
  ? "Vamos repetir o erro para testar..."

  ? "Somando um valor a uma variável"
  xVar := DATE()
  ? "A variável é " , xVar
  ? xVar += "A" // Somando um caractere a uma data (ERRO!!)
  ? "A variável agora é " , xVar


RETURN


FUNCTION MeuManipuladorDeErros( objError )

   ? "Estou dentro da função manipuladora de erros."
   ? "Mas aqui não tem nada!"

RETURN NIL
```

.:Resultado:.

```
Somando um valor a uma variável
A variável é  14/12/2016
Estou dentro da função manipuladora de erros.
Mas aqui não tem nada!
NIL
A variável agora é  NIL
Agora iremos controlar os próximos erros através da função original
Vamos repetir o erro para testar...
Somando um valor a uma variável
A variável é  14/12/2016
Error BASE/1081  Argument error: +
```

```
Called from MAIN(25)
```

Note que a função ERRORBLOCK() muda para outro bloco de manipulação (conforme já vimos) mas retorna o bloco de manipulação de erros que foi substituído. Isso é muito útil, pois nós podemos armazenar esse bloco anterior em uma variável (no nosso exemplo, a variável é chamada de bOriginal), para pode ser restaurado depois (linha 19). Assim, entre as linhas 10 e 19 os erros foram controlados pela nossa função manipuladora de erros.

15.4 O Harbour e as suas estruturas de manipulação de erros

O Harbour possui uma estrutura que pode ser usada para manipular erros. Ela chama-se BEGIN SEQUENCE (criada na década de 1990). Nas próximas subseções nós aprenderemos como e quando usar a BEGIN SEQUENCE.

15.4.1 BEGIN SEQUENCE

A estrutura BEGIN SEQUENCE se enquadra em uma estrutura especial de sequencia. Ela não é uma estrutura de seleção pura nem um loop. Ela foi criada pelo antigo desenvolvedor da linguagem Clipper 5.0 que a define como *uma estrutura de controle frequentemente usada para controle de erros em tempo de execução* [Nantucket 1990, p. 1-18]. Imagine a seguinte situação ilustrada na listagem 15.7. Note que existe uma condição no interior do laço mais interno (o que incrementa o valor de y) que requer a saída dos dois laços. Porém o comando EXIT irá sair apenas do laço mais interno para o laço externo (o que incrementa o valor de x). Esse é um problema que de vez em quando ocorre quando nos deparamos com laços aninhados.

Listagem 15.7: Situação problema

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL x,y

   FOR x := 1 TO 15
      FOR y := 1 TO 15
         ? x , y
         IF x == 3 .AND. y == 10 // Condição de saída
            ? "DEVO SAIR DOS DOIS LAÇOS A PARTIR DAQUI."
            ? "MAS SÓ CONSIGO SAIR DO LAÇO MAIS INTERNO."
            EXIT
         ENDIF
      NEXT
   NEXT
   ? "Final do programa"
   ? " x vale " , x , " e y vale " , y

RETURN
```

A solução imediata para o nosso problema é botar uma segunda verificação na saída do laço externo (o que incrementa o valor de x), conforme a listagem 15.8.

Listagem 15.8: Situação problema: solução 1

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL x,y

   FOR x := 1 TO 15
      FOR y := 1 TO 15
         ? x , y
         IF x == 3 .AND. y == 10 // Condição de saída
            ? "DEVO SAIR DOS DOIS LAÇOS A PARTIR DAQUI."
            ? "MAS SÓ CONSIGO SAIR DO LAÇO MAIS INTERNO."
            EXIT
         ENDIF
      NEXT
      IF x == 3 .AND. y == 10  // Repare que repeti a condição
         ? "AGORA SIM. ESSA SOLUÇÃO SAI DOS DOIS LAÇOS."
         EXIT
      ENDIF
   NEXT
   ? "Final do programa"
   ? " x vale " , x , " e y vale " , y

RETURN
```

Agora que você entendeu o problema e uma possível solução, vamos agora implementar a solução ideal usando BEGIN SEQUENCE no código 15.9

Listagem 15.9: Situação problema: solução 2 (Usando BEGIN SEQUENCE)

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL x,y

   BEGIN SEQUENCE
      FOR x := 1 TO 15
         FOR y := 1 TO 15
            ? x , y
            IF x == 3 .AND. y == 10 // Condição de saída
               ? "DEVO SAIR DOS DOIS LAÇOS A PARTIR DAQUI."
               ? "COM O BEGIN SEQUENCE e BREAK ISSO É POSSÍVEL."
               BREAK // SAI DA SEQUÊNCIA E VAI PARA A LINHA 19
            ENDIF
         NEXT
      NEXT
   END SEQUENCE
   ? "Final do programa"
```

```
   ? " x vale " , x , " e y vale " , y

RETURN
```

Note que a estrutura BEGIN SEQUENCE deve possuir um comando BREAK interno (no nosso exemplo ele está na linha 14). O significado de BREAK é simples: quando o processamento encontra esse comando ele sai da sequência totalmente (no nosso exemplo ele vai para a linha 19). Essa estrutura especial de quebra de sequencia é poderosa e pode tornar um código mais claro, mas todo esse poder tem um preço: ela também pode gerar códigos confusos pois a sequencia de um fluxo pode ser quebrada de forma confusa. No exemplo da listagem 15.9 nós vimos que BEGIN SEQUENCE nos ajudou, mas se você abusar demais dessa estrutura ela pode tornar seu código confuso. Spence nos alerta para termos cuidado ao usar BEGIN SEQUENCE e BREAK; pois nós podemos criar uma codificação ilegível com o uso excessivo desse par [Spence 1994, p. 21].

Dica 82

Se você usar muito a estrutura de sequencia BEGIN SEQUENCE nos seus programas é melhor ficar atento: você pode estar escrevendo códigos difíceis de serem lidos posteriormente.

Dica 83

O uso básico de BEGIN SEQUENCE é manipular exceções [erros]. Ele fornece um local adequado para saltar quando ocorre um erro. Você pode usá-lo como um ponto de interrupção para a lógica profundamente aninhada.[Spence 1994, p. 21]

BEGIN SEQUENCE e RECOVER

Se você entendeu o exemplo inicial com BEGIN SEQUENCE deve compreender o uso do RECOVER. É o seguinte, vimos que após um BREAK o fluxo se desvia para a linha após o final da sequência (END SEQUENCE). Ocorre que, em alguns casos, você não quer que isso aconteça. É o caso de situações em que você quer dar um “tratamento especial” a rotinas que saíram da sequência através de um BREAK. Tudo bem, sabemos que ficou um pouco complicado, mas vamos explicar através de exemplos. O código a seguir (listagem 15.10) ilustra essa situação. Primeiramente vamos rever o uso de BEGIN SEQUENCE/BREAK sem o RECOVER.

Listagem 15.10: Listagem

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL cResp

   BEGIN SEQUENCE
       ? "Fluxo normal"
       ACCEPT "Deseja sair com o BREAK ? (S/N)" TO cResp
```

```
        IF cResp $ "Ss"
           BREAK
        ENDIF
        ? "Continua o fluxo"
        ? "O BREAK näo foi executado!"
    END SEQUENCE
    ? "Final do programa"

RETURN
```

O BREAK foi executado (o usuário teclou “S” como resposta) :

.:Resultado:.

```
Fluxo normal
Deseja sair com o BREAK ? (S/N)s
Final do programa
```

Nessa segunda execução o BREAK não foi executado.

.:Resultado:.

```
Fluxo normal
Deseja sair com o BREAK ? (S/N)n
Continua o fluxo
O BREAK não foi executado!
Final do programa
```

Note que as linhas 13 e 14 são executadas somente se o BREAK não foi processado. Note também que não existem linhas que são executadas somente quando o BREAK é processado. Agora iremos modificar o código para incluir o RECOVER. Veja se você consegue notar a diferença.

Listagem 15.11: Listagem

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL cResp

   BEGIN SEQUENCE
        ? "Fluxo normal"
        ACCEPT "Deseja sair com o BREAK ? (S/N)" TO cResp
        IF cResp $ "Ss"
           BREAK
        ENDIF
        ? "Continua o fluxo"
        ? "O BREAK näo foi executado!"
   RECOVER
        ? "Fluxo alternativo"
        ? "O BREAK foi executado"
   END SEQUENCE
   ? "Final do programa"

```

```
RETURN
```

Nessa execução o BREAK foi executado

.:Resultado:.

```
Fluxo normal
Deseja sair com o BREAK ? (S/N)s
Fluxo alternativo
O BREAK foi executado
Final do programa
```

Já nessa, o BREAK não foi executado

.:Resultado:.

```
Fluxo normal
Deseja sair com o BREAK ? (S/N)n
Continua o fluxo
O BREAK não foi executado!
Final do programa
```

Conclusão: o RECOVER serve para criar um fluxo extra que será executado somente se o BREAK foi executado.

BEGIN SEQUENCE e RECOVER com identificação através de variáveis

Existem casos em que, dentro da mesma sequência nós temos mais de um BREAK, conforme o exemplo abaixo:

Listagem 15.12: Listagem

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL cResp

   BEGIN SEQUENCE
       ? "Fluxo normal"
       ACCEPT "Deseja sair com o BREAK ? (S/N)" TO cResp
       IF cResp $ "Ss"
          BREAK
       ENDIF
       ? "Continua o fluxo"
       ? "O primeiro BREAK näo foi executado!"

       ACCEPT "Deseja sair com o BREAK ? (S/N)" TO cResp
       IF cResp $ "Ss"
          BREAK
       ENDIF
       ? "Continua o fluxo"
       ? "O segundo BREAK näo foi executado!"

   RECOVER
```

```
      ? "Fluxo alternativo"
      ? "O BREAK foi executado, mas qual dos dois ?"
   END SEQUENCE
   ? "Final do programa"

RETURN
```

Nessa execução o segundo BREAK foi executado, mas o programa não sabe dizer qual dos dois BREAKs causou a quebra de sequência.

.:Resultado:.

```
Fluxo normal
Deseja sair com o BREAK ? (S/N)n
Continua o fluxo
O primeiro BREAK não foi executado!
Deseja sair com o BREAK ? (S/N)s
Fluxo alternativo
O BREAK foi executado, mas qual dos dois ?
Final do programa
```

Pensando nisso, os desenvolvedores do antigo Clipper adicionaram um parâmetro que pode ser passado através do comando BREAK. Esse parâmetro (um valor ou expressão) será automaticamente transferido ao RECOVER através da cláusula USING. O exemplo a seguir modifica a listagem 15.12 para poder identificar qual BREAK causou a interrupção da sequência. Acompanhe a alteração a seguir.

Listagem 15.13: Listagem

```
/*
BEGIN SEQUENCE
*/
PROCEDURE Main
LOCAL cResp, nValBreak

   BEGIN SEQUENCE
      ? "Fluxo normal"
      ACCEPT "Deseja sair com o BREAK ? (S/N)" TO cResp
      IF cResp $ "Ss"
         BREAK 1
      ENDIF
      ? "Continua o fluxo"
      ? "O primeiro BREAK não foi executado!"

      ACCEPT "Deseja sair com o BREAK ? (S/N)" TO cResp
      IF cResp $ "Ss"
         BREAK 2
      ENDIF
      ? "Continua o fluxo"
      ? "O segundo BREAK não foi executado!"

   RECOVER USING nValBreak
      ? "Fluxo alternativo"
      ? "O BREAK número ", nValBreak , " foi executado!"
```

```
   END SEQUENCE                                                    26
   ? "Final do programa"                                           27
                                                                   28
RETURN                                                             29
```

.:Resultado:.

```
Fluxo normal
Deseja sair com o BREAK ? (S/N)n
Continua o fluxo
O primeiro BREAK não foi executado!
Deseja sair com o BREAK ? (S/N)s
Fluxo alternativo
O BREAK número           2  foi executado!
Final do programa
```

Note que a variável definida por RECOVER na linha 23 recebe o valor definido pelos BREAKs. Assim fica fácil identificar qual BREAK causou a interrupção.

BEGIN SEQUENCE com o BREAK em outra rotina

Cuidado! Esse é um exemplo onde o BEGIN SEQUENCE está em uma rotina mas o BREAK correspondente está em outra. Ele só deve ser tolerado em rotinas de recuperação de erros, portanto o exemplo a seguir serve apenas para ilustração.

Listagem 15.14: Listagem

```
/*                                                                 1
BEGIN SEQUENCE                                                     2
*/                                                                 3
PROCEDURE Main                                                     4
LOCAL cVar                                                         5
                                                                   6
                                                                   7
   BEGIN SEQUENCE                                                  8
                                                                   9
      ? "Ok"                                                       10
      MeuBreak()                                                   11
      ? "Essa linha näo será exibida"                              12
                                                                   13
   RECOVER USING cVar                                              14
                                                                   15
       ? "Valor passado : " , cVar                                 16
       // Operações locais de recuperação de erro                  17
                                                                   18
   END SEQUENCE                                                    19
                                                                   20
   ? "Final do programa"                                           21
                                                                   22
RETURN                                                             23
                                                                   24
FUNCTION MeuBreak()                                                25
                                                                   26
```

```
   BREAK TIME() // Retorno daqui, pessoal. Para a linha 14.

RETURN NIL
```

.:Resultado:.

```
Ok
Valor passado :  11:42:00
Final do programa
```

15.4.2 BEGIN SEQUENCE e o gerenciador de erros do Harbour

Como você deve ter percebido, o BEGIN SEQUENCE não está restrito apenas a manipulação de erros, mas como toda rotina de manipulação de erros requer uma quebra abrupta na sequência do programa, é natural que o BEGIN SEQUENCE seja utilizado para manipular tais eventos desagradáveis.

Agora nós iremos ver o uso do BEGIN SEQUENCE em conjunto com o gerenciador de erros do Harbour[4].

Listagem 15.15: BEGIN SEQUENCE + Gerenciador de erros

```
/*
Erro de linkedição
Adaptado de : http://www.pctoledo.com.br/forum/viewtopic.php?f=1&t=14710
*/
PROCEDURE Main

   LOCAL objLocal, bLastHandler, a := 0

   // Altera o error handler padrão...
   bLastHandler := ERRORBLOCK({ |objErr| Break(objErr) })

   BEGIN SEQUENCE

      a = a + "10" // <-- erro aqui

   RECOVER USING objLocal

      ? "Tratamento do erro"

   END


   // Restore previous error handler   // Restaura o error handler anterior (E

   ERRORBLOCK( bLastHandler )          // padrão do Harbour ou outro customiza

RETURN
```

[4]Exemplo foi retirado do fórum PC Toledo, e foi adaptado de uma explicação dada por Alexandre Santos (AlxSts) em http://www.pctoledo.com.br/forum/viewtopic.php?f=1&t=14710, On-Line: 14-Dez-2016

Onde está o BREAK ? Se você notar, ele está dentro do objeto gerenciador de erros, na linha 10. A novidade aqui é que ele está na forma de função. Temos, o comando BREAK e a função Break(), que faz a mesma coisa. A vantagem da função sobre o comando, nesse caso, é que ela (a função) pode fazer parte de uma expressão ou de um objeto, como aconteceu na linha 10 da listagem 15.15.

Dica 84

O BEGIN SEQUENCE só irá "quebrar" a sequência se ele encontrar um BREAK (comando ou função) durante o processamento. Se o BREAK estiver dentro do gerenciador de erros, então o BREAK só será ativado se algum erro ocorrer. Como nós estamos lidando com exemplos simples, talvez fique difícil para você reconhecer a importância dessa dupla poderosa, mas acredite: a forma mais eficaz de tratar erros é usando o BEGIN SEQUENCE em conjunto com o gerenciador de erros do Harbour (que dispara o comando ou função BREAK).

BEGIN SEQUENCE WITH __BreakBlock()

Finalmente chegamos ao modelo ideal para tratamento de erros. Trata-se de uma versão sintaticamente "enxuta" da dupla BEGIN SEQUENCE / Gerenciador de erros. Vamos tomar o codigo da listagem 15.15 e apresentar o equivalente usando essa nova versão. Acompanhe o exemplo da listagem 15.16.

Listagem 15.16: BEGIN SEQUENCE + Gerenciador de erros

```
/*
Erro de linkedição
Mais no fórum : http://www.pctoledo.com.br/forum/viewtopic.php?f=4&t=17311
*/
PROCEDURE Main
LOCAL a := 0

   BEGIN SEQUENCE WITH __BreakBlock()

      a = a + "10" // <-- erro aqui

   RECOVER

      ? "Tratamento do erro"

   END


RETURN
```

Note que conseguimos o mesmo resultado com uma sintaxe mais clara. Note também que a cláusula WITH invoca um bloco de gerenciamento de erros só da sequencia, por isso não precisamos restaurar o gerenciador padrão e erros depois.

Esse exemplo sintetiza tudo o que nós aprendemos sobre gerenciamento de erros e está presente em muitos códigos do projeto Harbour no github, de modo que nós recomendamos esse modelo como sendo simples e funcional.

15.5 Conclusão

Mais uma importante etapa vencida. Aprender sobre o gerenciamento de erros não trás apenas benefícios técnicos para o programador, mas é também um exercício de humildade. Saber que o erro é uma condição humana nos leva a programar defensivamente e faz com o nosso software resista ao teste do tempo. Ficam dois princípios :

1. Você precisa usar as boas práticas que nós vimos nos capítulos anteriores (rotinas, módulos, código claro, etc.) para que o seu software seja bem construído e fácil de manter.

2. Mesmo sendo construído com essas boas práticas o seu software, muito provavelmente, conterá erros. O ser humano é o elo fraco da corrente, não tem como fugir dessa realidade. Por isso você deve usar intensivamente estruturas de tratamento de erros durante a sua codificação.

Hoje em dia um terceiro princípio vem ganhando força e já está consolidado entre as equipes de desenvolvimento: o teste de software. O objetivo do teste automatizado de software é antecipar possíveis erros de execução. Assim, o provável erro deixa de ocorrer nas dependências do cliente para ocorrer durante a fase de desenvolvimento do software. Ou seja, a manipulação de erros ocorre após o erro acontecer, mas o teste de software ocorre sem haver erro algum (você não precisa constatar o erro para testar o seu software). Mas isso já é assunto para um outro capítulo (ou outro livro).

16 Tipos derivados

O sucesso nunca é definitivo.

Winston Churchill

Objetivos do capítulo

- Entender o que é um tipo derivado.
- Compreender o uso de arrays.
- Saber manipular um hash.
- Usar as principais funções manipuladoras de arrays e de hashs.

16.1 Introdução

Até agora nós vimos quatro tipos básicos de variáveis, são eles: numérico, caractere, data e lógico. Esses dados básicos possuem uma limitação: processam apenas um valor por vez, ou seja, se você declarar uma variável, ela só poderá conter um valor por vez. Existem casos, entretanto, em que você necessitará processar vários valores por variável. Para resolver esse problema, o Harbour possui dois tipos especiais que são chamados de tipos derivados de variáveis: o array[1] e o hash. O array é o tipo mais antigo, por isso mesmo nós o abordaremos primeiro. Em seguida nós estudaremos o hash, que é um tipo especial de array.

16.2 O que é um array ?

Um array é uma sequência de posições que se pode acessar diretamente através de uma variável e de um índice numérico. Aguilar define um array como sendo "um conjunto finito e ordenado de elementos" [Aguilar 2008, p. 228]. Essa definição é simples e abrange tudo o que precisamos saber sobre arrays. Dizer que é um conjunto ordenado significa dizer que cada elemento de um array possui uma posição numérica. Por exemplo: primeiro elemento, segundo elemento, etc.

Para que as coisas fiquem mais claras, vamos usar uma analogia adaptada de [Forbellone e Eberspacher 2005, p. 69]: imagine um prédio com um número finito de andares e imagine também que cada andar possui apenas um apartamento. Contudo, não basta saber qual apartamento desejamos chegar se não soubermos o endereço do prédio. O que precisamos de antemão é o endereço do prédio e só então nos preocuparmos para qual daqueles apartamentos queremos ir. Interessa chegar no apartamento, mas antes o morador precisa do :

- endereço do prédio;
- número do apartamento.

A mesma coisa acontece com os arrays: o prédio em questão equivale a variável de memória que armazenará o array, o número do apartamento é um identificador que aponta para o elemento do array e o conteúdo do apartamento é o valor armazenado nesse elemento. Quando o prédio possui um apartamento por andar isso quer dizer que o array tem apenas uma dimensão (quando isso acontece ele também recebe o nome de **vetor**. Se você se lembra das aulas de matemática do ensino médio basta recordar o conceito de "matriz coluna", que é um tipo especial de matriz. A figura 16.1 ilustra esse conceito.

[1] Iremos usar o termo em inglês por ser amplamente aceito na computação. Outros termos usados são : matriz e vetor (que é um tipo especial de matriz com apenas uma dimensão).

Figura 16.1: Matriz coluna

$$M_{3 \times 1} = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{pmatrix}$$

Quando o prédio possui mais de um apartamento por andar dizemos que o array tem duas dimensões. As figura a seguir ilustra um array bidimensional.

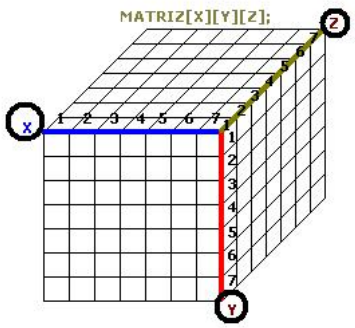
Figura 16.2: Array com duas dimensões

$$A = \begin{bmatrix} 2 & 3 & 5 & 6 \\ 4 & 2 & 1 & 1 \\ 5 & 1 & 2 & 3 \\ 6 & 1 & 3 & 2 \end{bmatrix}$$

A grande maioria dos conjuntos representados em computação possuem apenas duas dimensões. A programação voltada para sistemas de informação é, em sua maioria, composta de tais estruturas. Uma planilha eletrônica é um exemplo bem clássico.

Até agora podemos representar facilmente essas matrizes com lápis e papel, mas existem arrays que possuem três dimensões, cuja representação aproximada seria algo como o desenho a seguir :

Figura 16.3: Array com três dimensões



Existem também arrays cuja representação gráfica se torna impossível: são os arrays que possuem quatro ou mais dimensões. Tais estruturas são matematicamente possíveis, e as linguagens de programação podem representá-las, mas elas não podem

"ser visualizadas" por nós em toda a sua completude, pois o nosso cérebro só consegue representar três dimensões no máximo.

Dica 85

É importante não confundir o índice com o elemento. O índice é a posição do elemento no array (o número do apartamento), enquanto o elemento é o que está contido no elemento do array (o conteúdo do apartamento) [Forbellone e Eberspacher 2005, p. 71].

16.3 Entendo os arrays através de algoritmos

16.3.1 Um algoritmo básico usando arrays

Considere o problema a seguir:

Criar um programa que efetue a leitura dos nomes de 20 pessoas e em seguida apresente-os na mesma ordem em que foram informados.

O algoritmo exige os seguintes passos :

- Definir o array NOME com 20 elementos.
- Iniciar o programa, fazendo a leitura dos 20 nomes.
- Apresentar após a leitura, os 20 nomes.

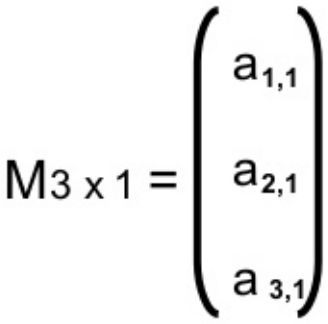
O algoritmo, em pseudo-código, seria assim:

Algoritmo 21: Coleta e exibição de 20 nomes

```
   Entrada:
   Saída:
1  início
2  |  NOME ← array( 20 ) // Array NOME
3  |  para contador de 1 até 20 faça
4  |  |  leia NOME[ contador ]
5  |  fim
6  |  para contador de 1 até 20 faça
7  |  |  escreva NOME[ contador ]
8  |  fim
9  fim
```

Se você está buscando entender através da representação de matriz que você viu no ensino médio, talvez você esteja estranhando o exemplo acima. Isso porque, mesmo uma matriz coluna (uma matriz unidimensional) tem os elementos da coluna representados, conforme a figura a seguir :

Figura 16.4: Array com uma dimensão (um vetor)



Já a representação de um elemento de uma matriz unidimensional (um vetor) possui apenas a representação da linha, porque representar a coluna seria desnecessário. Se nós fossemos imprimir os elementos da matriz da figura 16.4 usando um pseudo-código, a representação seria algo como :

```
? a[1] , a[2], a[3]
```

e não

```
? a[1][1], a[2][1], a[3][1]
```

A grande vantagem do array é que nós precisamos criar apenas uma variável que represente todo um conjunto de elementos. Note que é fácil recuperar os elementos gravados, basta percorrer o array através de um laço. Arrays e laços (estruturas de repetição) são dois elementos das linguagens de programação que trabalham juntos.

16.3.2 O problema da média da turma

Considere o seguinte problema:

Um colégio deseja saber quantos alunos, de um grupo de 10 alunos, tiraram uma nota superior a média da turma.

Primeiramente nós vamos tentar resolver esse problema **sem usar arrays**. Veja que a solução não é tão fácil quanto parece.

Algoritmo 22: Cálculo da média aritmética de 10 notas

```
   Entrada:
   Saída:
1  início
2      média ← 0 // Média da turma
3      nota ← 0 // Nota do aluno
4      contador ← 0 // Contador
5      acumulador ← 0 // Acumulador
6      enquanto contador < 10 faça
7          leia nota
8          contador ← contador + 1
9          acumulador ← acumulador + nota
10     fim
11     média ← acumulador / 10
12     escreva "A média anual da turma é" , média
13     .
14     .
15     // E agora ?????
16     // Como eu vou calcular quantos alunos tiraram a nota maior que
17     // a média da turma ?
18     // Eu não tenho mais a nota de cada aluno. Elas se perderam
19     // durante o processamento do laço "enquanto".
20 fim
```

Temos um problema: eu só posso contar quantos alunos obtiveram uma nota acima da média da turma logo após o cálculo da média (linha 11). Mas como eu posso comparar, se o valor da nota do aluno é sempre sobreposto pelo aluno seguinte ? (linha 7).

Quando surge um problema dessa natureza os arrays surgem como a única solução prática. Em vez de termos apenas uma variável chamada *nota* que seria sobreposta a cada iteração, nós teríamos um array chamado *nota* com dez posições, onde cada posição equivaleria a um aluno. Em vez de usar o laço *enquanto* (WHILE) eu passaria a usar o laço *para* (FOR).

Algoritmo 23: Cálculo da média aritmética de 10 notas usando vetores

```
   Entrada:
   Saída:
1  início
2      média ← 0 // Média da turma
3      nota ← array( 10 ) // Array nota do aluno
4      acumulador ← 0 // Acumulador
5      notaAcima ← 0 // Contador de notas acima da média
6      para contador de 1 até 10 faça
7          leia nota[ contador ]
8          acumulador ← acumulador + nota
9      fim
10     média ← acumulador / 10
11     escreva “A média anual da turma é” , média
12     .
13     // laço para verificar valores que estão acima da média
14     para contador de 1 até 10 faça
15         se nota[ contador ] > média então
16             notaAcima ← notaAcima + 1 //
17     fim
18     escreva “Número de valores acima da média” , notaAcima
19 fim
```

Agora que você já tem a definição de arrays através do algoritmo, vamos passar para a definição do array no Harbour. Ao final da próxima seção você estará apto a implementar o algoritmo acima usando Harbour.

16.4 Arrays em Harbour

As primeiras versões do array foram criadas pelo dBase, ainda na década de 1980. Quando o Clipper surgiu o conceito foi expandido para dar suporte a arrays de várias dimensões. Como muitos problemas de empresas que os computadores tratam dizem respeito a dados que podem ser agrupados para melhor compreensão, não é surpreendente que as características dos arrays no Clipper 5.0 tenham se tornado um dos mais poderosos conceitos da linguagem [Heimendinger 1994, p. 99].

Como nós já vimos no pseudo-código, os arrays são simplesmente conjuntos de variáveis de memória com um nome comum. Elementos individuais são endereçados usando índices. São números que aparecem entre colchetes após o nome do array. O fragmento a seguir ilustra o que foi dito:

```
LOCAL choices[ 10 ] // Declara um array de tamanho 10
```

Da mesma forma que as variáveis de memória, você precisa atribuir valores para os elementos antes de usá-los. O fragmento de código a seguir mostra como se faz isso:

```
choices[ 3 ] := 20
```

Quando queremos nos referir ao terceiro elemento desse array, podemos fazer conforme o fragmento a seguir:

```
? choices[ 3 ] // Imprime o terceiro elemento do array.
```

16.4.1 Declarando arrays

Portanto, um array é uma variável, igual a todas que estudamos anteriormente, por isso ela precisa ser declarada e inicializada. As regras de classe de variável são válidas para arrays tembém, e isso quer dizer que podemos ter os tipos LOCAL, STATIC, PRIVATE e PUBLIC.

Porém, diferente das outras variáveis, o array precisa obrigatoriamente ser inicializado antes de ser usado. Isso porque você precisa informar ao Harbour que a variável é um array e não uma variável comum. O exemplo a seguir (listagem 16.1) exemplifica algumas formas de se inicializar um array. Note que nós preferimos declarar os arrays como locais, mas as regras de declaração são as mesmas das variáveis comuns.

Listagem 16.1: Declarando e inicializando arrays

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aClientes := ARRAY( 2 )
LOCAL aSalarios := {}

RETURN
```

O exemplo acima declara e inicializa dois arrays[2]. O primeiro array (linha 5) possui dois elementos, ou seja, ele poderá conter dois valores de qualquer tipo (no caso específico, dois clientes). O segundo array (linha 6) não possui elementos e não pode receber nenhuma atribuição. A maioria das pessoas (principalmente programadores de outras linguagens) acham que a primeira forma é a mais usada, mas a maioria dos códigos em Harbour, que usam arrays, preferem a segunda forma de inicialização. Isso porque, geralmente, nós não sabemos de antemão quantos elementos o array irá possuir, então nós inicializamos um array vazio para, só depois, ir adicionando elementos.

Você também pode inicializar um array com valores pré-definidos, conforme o fragmento abaixo:

```
LOCAL aNotas := {  10 , 20 , 45 , 10 }
```

[2]De agora em diante nós chamaremos as “variáveis arrays” simplesmente de “arrays”.

Lembre-se, o fragmento a seguir também está correto. Porém, se você notar, o array só foi declarado, mas não inicializado.

```
LOCAL choices[ 10 ] // Declara um array de tamanho 10
```

Dica 86

Apesar de ser a forma preferida pelos programadores Harbour, inicializar um array com zero elementos nos trás um problema de performance: caso eu não saiba quantos elementos ele irá possuir, eu posso ir adicionando os elementos de um por um, a medida que eu vou precisando deles. É aí que está o problema: eu sempre vou ter que criar um elemento no final do array, e essa operação consome um recurso extra de processamento. Se você não souber quantos elementos o array conterá, você deve tentar descobrir o total logo de uma vez e usar o a função ARRAY, mas como isso geralmente não é possível, você terá que usar a função AADD para ir adicionando elemento por elemento. Veremos esses exemplos nas seções seguintes.

Outro fato digno de nota é a nomenclatura usada. Um array pode conter vários tipos diferentes de dados, por exemplo: o primeiro elemento de um array pode ser uma string, o segundo elemento pode ser uma data, etc. Portanto, não faz sentido prefixar um array com “c”, “n”, “l” ou “d”, já que um mesmo array pode conter variáveis de diversos tipos. A solução encontrada foi prefixar o array com a letra “a” de array.

Dica 87

As regras de nomenclatura de um array são as mesmas regras de qualquer variável, porém nós iremos prefixar um array com a letra “a”. Habitue-se a fazer isso nos seus programas, isso irá torná-los mais claros e fáceis de se entender.

16.4.2 Atribuindo dados a um array

Agora que nós já aprendemos a declarar e a inicializar um array vamos aprender a inserir dados nele. Gravar dados em um array é muito parecido com inicializar uma variável, a única diferença é que você deve informar um dado adicional chamado de índice. O índice é sempre informado entre colchetes. Veja no exemplo a seguir (listagem 16.2).

Listagem 16.2: Inicializando arrays

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aClientes := ARRAY( 2 )

    aClientes[ 1 ] := "Claudio Soto"
    aClientes[ 2 ] := "Roberto Linux"

```

```
    ? aClientes[ 1 ]
    ? aClientes[ 2 ]

RETURN
```

.:Resultado:.

```
Claudio Soto
Roberto Linux
```

O índice é aquele número que se encontra entre os colchetes logo após o nome da variável array. Ele identifica o elemento do array que esta sendo usado. O array foi declarado com o tamanho de dois elementos (linha 5), dessa forma, você precisa informar o índice do array para poder atribuir (linhas 7 e 8) ou acessar (linhas 10 e 11) os dados.

Dica 88

Os índices do array iniciam sempre com o valor 1. Nem todas as linguagens são assim. Na realidade, a maioria das linguagens iniciam o array com o valor zero. No Harbour, o índice do último elemento do array equivale a quantidade de elementos do array, mas em linguagens como C, C++, PHP, Java e C#, o tamanho do array sempre é o índice do último elemento mais um, pois os arrays nessas linguagens iniciam com o índice igual a 0.

Cuidado. Você deve ter cuidado quando for referenciar ou atribuir dados a um array, pois o índice precisa estar no intervalo aceitável. Por exemplo: na listagem 16.2 o array aClientes possui dois elementos. Assim, você só poderá trabalhar com dois índices (1 e 2). O exemplo a seguir (listagem 16.3) nos mostra o que acontece quando atribuímos o índice 3 a um array que só tem tamanho 2.

Listagem 16.3: Inicializando arrays (ERRO)

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aClientes := ARRAY( 2 )

    aClientes[ 1 ] := "Claudio Soto"
    aClientes[ 2 ] := "Roberto Linux"
    aClientes[ 3 ] := "Elemento fora do intervalo"

RETURN
```

.:Resultado:.

```
Error BASE/1133  Bound error: array assign
Called from MAIN(9)
```

Erros do tipo “array assign” ocorrem quando você tenta atribuir um valor a um elemento de array com um índice que está além dos seus limites. Tenha muito cuidado, pois esses erros acontecem sempre em tempo de execução e nem mesmo o compilador pode gerar algum aviso (warning) sobre eles.

Dica 89

O Harbour implementa um array de uma forma diferente da maioria das linguagens. Formalmente, as outras linguagens definem o array com um tamanho fixo, e também não permitem tipos de dados diferentes por array. O Harbour também define um array com um tamanho fixo, mas o seu tamanho pode ser aumentado (ou diminuído) dinamicamente. Durante a execução, o array pode crescer e encolher. E, o número de dimensões que um array pode ter é ilimitado. Além disso, o Harbour permite tipos diferentes de dados dentro de um mesmo array. Em resumo:

- Arrays em Harbour: são dinâmicos e permitem vários tipos de dados por array.
- Arrays na maioria das linguagens: são estáticos e permitem apenas um tipo de dado por array.

16.4.3 Algumas operações realizadas com arrays

Os exemplos a seguir abordarão algumas operações comuns que são realizadas com o array.

Percorrendo os itens de um array (Laço FOR)

Uma vez criado o array você pode usar o laço FOR para percorrer os seus elementos facilmente. O exemplo a seguir (listagem 16.4) calcula o quadrado dos cem primeiros números inteiros sem utilizar arrays.

Listagem 16.4: Obtendo o quadrado dos cem primeiros inteiros

```
/*
Arrays
*/
PROCEDURE Main
LOCAL x

    FOR x := 1 TO 100
        ? "Quadrado de ", x , " é " , x ^ 2
    NEXT

RETURN
```

De acordo com Spence, “FOR loops são uma maneira excelente de processar vetores.” [Spence 1991, p. 153]. Veja, a seguir, as últimas linhas do programa ao ser executado.

.:Resultado:.

```
Quadrado de          91  é        8281.00
Quadrado de          92  é        8464.00
Quadrado de          93  é        8649.00
Quadrado de          94  é        8836.00
Quadrado de          95  é        9025.00
Quadrado de          96  é        9216.00
Quadrado de          97  é        9409.00
Quadrado de          98  é        9604.00
Quadrado de          99  é        9801.00
Quadrado de         100  é       10000.00
```

Mas nós temos problema no nosso código (não é um erro de execução nem um erro de lógica). Imagine que você precise dos respectivos valores calculados para uma operação posterior. Note que você não tem mais esses valores após o laço finalizar. É para isso que servem os arrays: para manter na memória do computador os valores de um processamento anterior que envolva múltiplas ocorrências de uma mesma variável.

O programa a seguir (listagem 16.5) resolve isso com arrays.

Listagem 16.5: Armazenando valores para uma operação posterior

```
/*
Arrays
*/
#define ELEMENTOS 100
PROCEDURE Main
LOCAL aQuadrado := ARRAY( ELEMENTOS )
LOCAL x

    FOR x := 1 TO ELEMENTOS
        aQuadrado[ x ] := x ^ 2
    NEXT

    // Outras operações

    // Agora ainda tenho os 100 primeiros quadrados
    // graças ao array aQuadrado
    FOR x := 1 TO ELEMENTOS
        ? "Quadrado de ", x , " é " , aQuadrado[ x ]
    NEXT

RETURN
```

.:Resultado:.

```
Quadrado de          91  é        8281.00
Quadrado de          92  é        8464.00
Quadrado de          93  é        8649.00
Quadrado de          94  é        8836.00
Quadrado de          95  é        9025.00
Quadrado de          96  é        9216.00
Quadrado de          97  é        9409.00
```

```
Quadrado de          98  é        9604.00
Quadrado de          99  é        9801.00
Quadrado de         100  é       10000.00
```

Percorrendo os ítens de um array obtendo dinamicamente a sua quantidade de elementos (Função LEN)

Vamos continuar com o exemplo da listagem 16.5 para aprimorarmos o nosso conhecimento sobre os arrays. O programa já acumula o processamento na forma de arrays, mas ele possui mais uma limitação: ele processa uma quantidade fixa de elementos (sempre cem elementos). Caso você queira mudar isso você precisa modificar o programa e compilar de novo para que um novo executável seja gerado. O que é impraticável em situações da vida real. Vamos, então, realizar uma pequena mudança no nosso programa. Você, com certeza, deve estar lembrado da função LEN, que é usada para nos informar a quantidade de caracteres de uma string. Pois bem, a função LEN pode ser usada com arrays também, e ela retorna a quantidade de elementos do array. O programa a seguir aprimora o programa da listagem 16.5 para que ele processe uma quantidade informada em tempo de execução.

Listagem 16.6: Armazenando N valores para uma operação posterior

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aQuadrado
LOCAL x

    CLS
    INPUT "Informe a quantidade de elementos : " TO x

    aQuadrado := ARRAY( x )

    FOR x := 1 TO LEN( aQuadrado )
        aQuadrado[ x ] := x ^ 2
    NEXT

    FOR x := 1 TO LEN( aQuadrado )
        ? "Quadrado de ", x , " é " , aQuadrado[ x ]
    NEXT

RETURN
```

A seguir temos um exemplo quando o usuário digita 10.

.:Resultado:.

```
Informe a quantidade de elementos : 10
Quadrado de           1  é           1.00
Quadrado de           2  é           4.00
Quadrado de           3  é           9.00
```

```
Quadrado de           4  é          16.00
Quadrado de           5  é          25.00
Quadrado de           6  é          36.00
Quadrado de           7  é          49.00
Quadrado de           8  é          64.00
Quadrado de           9  é          81.00
Quadrado de          10  é         100.00
```

Adicionando itens a um array dinamicamente (Função AADD)

Vamos prosseguir do exemplo da seção anterior (listagem 16.6). Note que ele exige que eu saiba a quantidade de elementos do array previamente para, só depois, fazer o cálculo. Mas, nós já vimos quando estudamos os loops, que essa não é a única forma de processamento de dados em laços. Existe um caso em que você não sabe quantos elementos irá processar e a decisão precisa ser tomada no interior do laço. Ou seja, você precisa adicionar elementos ao seu array dinamicamente, pois você não sabe (de antemão) quantos elementos ele terá. Outra limitação é que os valores sempre começam com 1 e vão até um valor pré-determinado por você, sempre em ordem crescente. E se o usuário precisasse calcular uma quantidade de valores quaisquer ?

O exemplo da listagem 16.7 ilustra essa situação usando uma função chamada AADD. Essa função adiciona um elemento ao final do array.

Listagem 16.7: Armazenando N valores para uma operação posterior

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aQuadrado := {}
LOCAL x
LOCAL nValor


    DO WHILE .T.
       INPUT "Informe o elemento (999 FINALIZA) " TO nValor
       IF nValor == 999
          EXIT
       ENDIF
       AADD( aQuadrado, nValor )
    ENDDO

    FOR x := 1 TO LEN( aQuadrado )

   ? "O quadrado de " , aQuadrado[x] , " é " , ( aQuadrado[x] ^ 2 )
    NEXT

RETURN
```

A seguir temos um exemplo quando o usuário digita 3 valores quaisquer.

.:Resultado:.

```
Informe o elemento (999 FINALIZA) 6
Informe o elemento (999 FINALIZA) 100
Informe o elemento (999 FINALIZA) 549
Informe o elemento (999 FINALIZA) 999
O quadrado de           6  é          36.00
O quadrado de         100  é       10000.00
O quadrado de         549  é      301401.00
```

Note que o array é declarado e inicializado, na linha 5, como um array vazio. A função AADD é usada na linha 15. O primeiro parâmetro é o array e o segundo é o valor que o elemento que será inserido irá ter. A função AADD está descrita a seguir.

Descrição sintática 22

1. Nome : AADD()
2. Classificação : função.
3. Descrição : Adiciona um novo elemento ao final de um array.
4. Sintaxe

```
AADD( <aArray>, [<expValor>] ) -> Valor
```

Fonte : [Nantucket 1990, p. 5-1]

Note que AADD avalia <expValor> e retorna o seu valor. Caso <expValor> não seja especificado, AADD retorna NIL. Caso <expValor> seja um outro array, o novo elemento no array destino conterá uma referência[3] ao array especificado por <expValor> [Nantucket 1990, p. 5-1].

[3]Veremos o que é referência nas seções posteriores, ainda nesse capítulo. Fique atento.

Dica 90

Lembre-se:
Você pode formar arrays simplesmente escrevendo conforme o fragmento abaixo. Essa forma chama-se *array literal*.

```
LOCAL aEscolha := { "Adiciona", "Edita", "Relatório" }
```

É o mesmo que escrever :

```
LOCAL aEscolha[ 3 ]

  aEscolha[1]  := "Adiciona"
  aEscolha[2] :=  "Edita"
  aEscolha[3] :=  "Relatório"
```

Que equivale a

```
LOCAL aEscolha := ARRAY( 3 )

  aEscolha[1]  := "Adiciona"
  aEscolha[2] :=  "Edita"
  aEscolha[3] :=  "Relatório"
```

Que, também é a mesma coisa que :

```
LOCAL aEscolha := {}

  AADD( aEscolha,  "Adiciona" )
  AADD( aEscolha,  "Edita" )
  AADD( aEscolha,  "Relatório" )
```

Dica 91

Na listagem 16.7 nós usamos um laço FOR em conjunto com a função LEN para processar um array. Veja um fragmento de código semelhante:

```
FOR x := 1 TO LEN( aArray )
   ? aArray[ x ]
NEXT
```

Isso está correto, mas nós temos um pequeno problema de performance no laço FOR. Isso ocorre porque a função LEN() será processada várias vezes, obrigando o Harbour a reavaliar os limites do loop durante toda a iteração. O ideal seria evitar o uso de funções no cabeçalho de laços ou dentro de laços (nem sempre isso é possível, mas se for possível, lhe renderá ganhos de tempo). A função LEN(), nesse caso, poderia ser colocada fora do laço FOR.

```
nTamanho :=  LEN( aArray )
FOR x := 1 TO nTamanho
   ? aArray[ x ]
NEXT
```

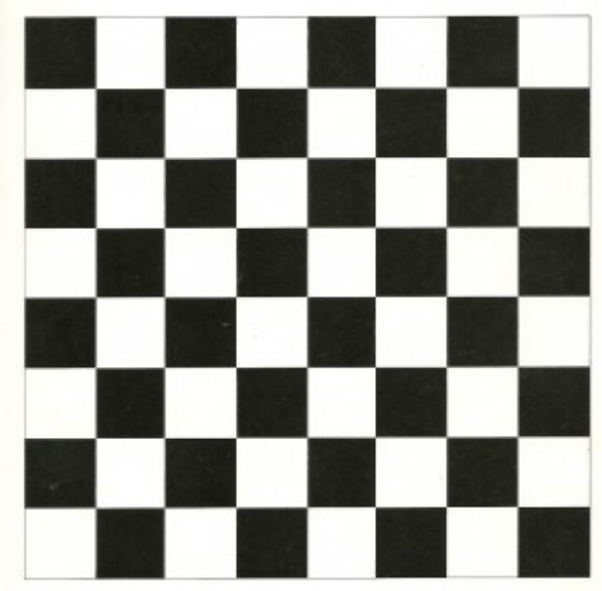
Atenção! Nem sempre a função de avaliação pode ser colocada fora do laço. Você precisa analisar caso a caso.

16.4.4 Arrays unidimensionais e multidimensionais

O equivalente matemático dos arrays são as matrizes que você viu durante o ensino médio. Até agora nós trabalhamos somente com arrays unidimensionais, cujo equivalente matemático seria uma matriz coluna.

Vimos que os arrays podem ser usados para representarem objetos com mais de uma dimensão. Por exemplo, a figura 16.5 possui duas dimensões.

Figura 16.5: Um array pode ser usado para representar um tabuleiro de xadrez



A figura 16.5 poderia ser representada através do seguinte array:

```
LOCAL aChess[ 8 ][ 8 ] // Tabuleiro de xadrez
```

Dica 92

Arrays bidimensionais são os mais usados em qualquer linguagem de programação e servem para representar uma variedade de objetos do mundo real: uma planilha eletrônica, um cardápio de lanchonete, assim como qualquer tabela com dados.

Figura 16.6: Uma tabela com dados é um ente bidimensional

O Harbour implementa o array aChess de uma forma bem incomum: é como se cada elemento do array principal (de 8 posições) fosse, ele também, um array de 8 posições. Quando um elemento de um array é um array também, podemos dizer que se trata de um array multidimensional.

O array aChess poderia ser representado da seguinte forma :

```
aChess := {{ "a11", "a12", "a13", "a14", "a15", "a16", "a17", "a18" },;
           { "a21", "a22", "a23", "a24", "a25", "a26", "a27", "a28" },;
           { "a31", "a32", "a33", "a34", "a35", "a36", "a37", "a38" },;
           { "a41", "a42", "a43", "a44", "a45", "a46", "a47", "a48" },;
           { "a51", "a52", "a53", "a54", "a55", "a56", "a57", "a58" },;
           { "a61", "a62", "a63", "a64", "a65", "a66", "a67", "a68" },;
           { "a71", "a72", "a73", "a74", "a75", "a76", "a77", "a78" },;
           { "a81", "a82", "a83", "a84", "a85", "a86", "a87", "a88" }}
```

Para inserir um valor em um elemento você também pode fazer assim :

```
aChess[ 5 ][ 2 ] := 12 // Substitui a string "a52" por 12
```

Note que tais representações acima se assemelham as matrizes estudadas no ensino médio. Se fossemos representar o elemento do array acima conforme a notação usada no ensino de matrizes, ela seria assim : $aChess_{5,2}$.

Dica 93

A representação de arrays multidimensionais obedece ao mesmo dos arrays bidimensionais vistos até agora. O fragmento a seguir declara um array com quatro dimensões.

```
LOCAL aProperty[ 4 ][ 3 ][ 2 ][ 7 ]
```

Você também pode usar a função AADD para ir montando o seu array multimensional, conforme o fragmento abaixo:

```
LOCAL aProperty := {}

AADD( aProperty , {} ) // O primeiro elemento é um array
AADD( aProperty , 23 ) // O segundo elemento é um número

AADD( aProperty[ 1 ] , 12 )
AADD( aProperty[ 1 ] , 13 )
```

O resultado final é algo como :

```
aProperty[ 1 ][ 1 ] é igual a 12.
aProperty[ 1 ][ 2 ] é igual a 13.
aProperty[ 2 ] é igual a 23.
```

Essa forma de “ir montando” um array confere ao Harbour uma flexibilidade maior, pois nós não precisamos criar arrays rigidamente retangulares.

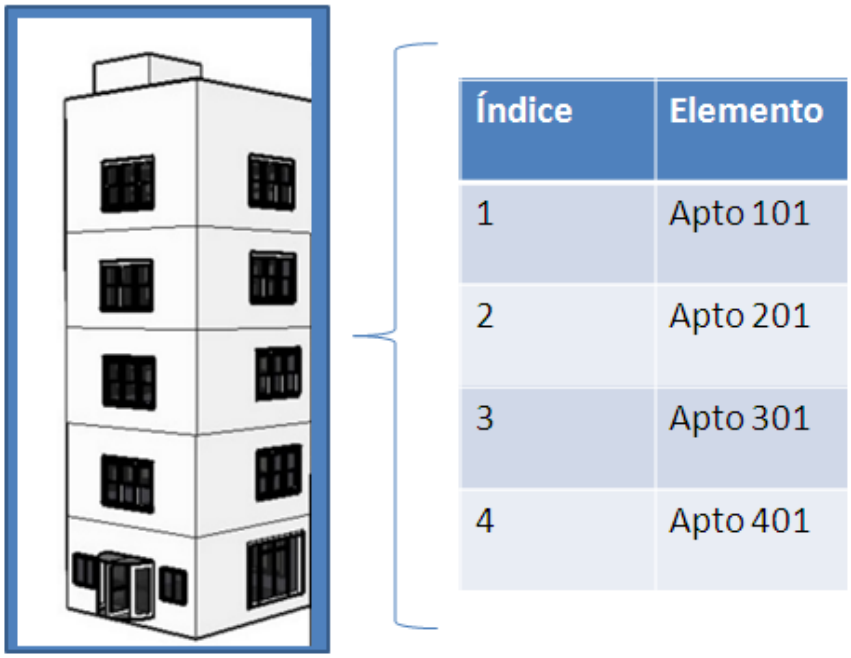
Você deveria anotar o fato de que muitas linguagens impõem uma rígida estrutura sobre os arrays. Cada elemento em uma dimensão deve ter o mesmo tamanho e tipo de todos os seus pares, e o número de elementos em qualquer dimensão é o mesmo para todo array. Em outras palavras, se você tem um array com 10 elementos na primeira dimensão, 20 na segunda e 5 na terceira, ele terá um array de 10x20x5 contendo 1000 elementos. [...]

O fato mais significativo sobre arrays [...] é que qualquer elemento do array pode ser ele próprio um array. **Esse conceito é muito mais poderoso que um array multidimensional**. [...] Você pode criar *arrays irregulares*, ou arrays que não são “quadrados” no sentido m x n x ... Como um exemplo, você pode ter um array com 10 elementos, o segundo dos quais é também um array com 6 elementos, e o terceiro elemento desse array é também um array...

Fonte : [Heimendinger 1994, p. 99]

Lembra do exemplo do apartamento que foi apresentado no início desse capítulo ? Pois bem, como no nosso suposto prédio existia apenas um apartamento por andar, por analogia, o array era unidimensional (um vetor).

Figura 16.7: Um array unidimensional (ou vetor)



Índice	Elemento
1	Apto 101
2	Apto 201
3	Apto 301
4	Apto 401

O fragmento a seguir ilustra como seria o array da figura 16.7 em Harbour:

```
aPredio := { "Apto 101" , "Apto 201" , "Apto 301" , "Apto 401" }

? aPredio[ 2 ] // Imprime "Apto 201"
```

Mas existem prédios com mais de um apartamento por andar (por exemplo dois apartamentos por andar), esse seria o caso de um array multidimensional (duas dimensões).

Figura 16.8: Um array multidimensional

Índice	Elemento	Índice	Elemento
1,1	Apto 101	1,2	Apto 102
2,1	Apto 201	2,2	Apto 202
3,1	Apto 301	3,2	Apto 302
4,1	Apto 401	4,2	Apto 402

O fragmento a seguir ilustra como seria o array da figura 16.8 em Harbour:

```
aPredio := { { "Apto 101" , "Apto 102" } ,;
             { "Apto 201" , "Apto 202" } ,;
             { "Apto 301" , "Apto 302" } ,;
             { "Apto 401" , "Apto 402" } }

? aPredio[ 2 ][ 1 ] // Imprime "Apto 201"
? aPredio[ 4 ][ 2 ] // Imprime "Apto 402"
```

A seguir a representação de um prédio com 4 apartamentos por andar.

Figura 16.9: Um array multidimensional (ou vetor)

Índice	Elemento	Índice	Elemento	Índice	Elemento	Índice	Elemento
1,1	Apto 101	1,2	Apto 102	1,3	Apto 103	1,4	Apto 104
2,1	Apto 201	2,2	Apto 202	2,3	Apto 203	2,4	Apto 204
3,1	Apto 301	3,2	Apto 302	3,3	Apto 303	3,4	Apto 304
4,1	Apto 401	4,2	Apto 402	4,3	Apto 403	4,4	Apto 404

O fragmento a seguir ilustra como seria o array da figura 16.9 em Harbour:

```
aPredio := {{ "Apto 101" , "Apto 102" , "Apto 103" , "Apto 104"},;
            { "Apto 201" , "Apto 202" , "Apto 203" , "Apto 204"},;
            { "Apto 301" , "Apto 302" , "Apto 303" , "Apto 304"},;
```

```
            { "Apto 401" , "Apto 402" , "Apto 403" , "Apto 404"}}

? aPredio[ 2 ][ 1 ] // Imprime "Apto 201"
? aPredio[ 3 ][ 3 ] // Imprime "Apto 303"
? aPredio[ 4 ][ 3 ] // Imprime "Apto 403"
```

O Harbour vai mais além: como cada elemento do array pode conter uma quantidade variada de arrays, então isso equivaleria a um prédio com uma quantidade variada de apartamentos por andares. Por exemplo:

Figura 16.10: Um array multidimensional do Harbour.

Índice	Elemento	Índice	Elemento	Índice	Elemento	Índice	Elemento
1,1	Apto 101	1,2	Apto 102				
2,1	Apto 201						
3,1	Apto 301	3,2	Apto 302				
4,1	Apto 401	4,2	Apto 402	4,3	Apto 403	4,4	Apto 404

O fragmento a seguir ilustra como seria o array da figura 16.9 em Harbour:

```
aPredio := { { "Apto 101" , "Apto 102"  } ,;
             { "Apto 201" } ,;
             { "Apto 301" , "Apto 302" } ,;
             { "Apto 401" , "Apto 402" , "Apto 403" , "Apto 404"} }

? aPredio[ 2 ][ 1 ] // Imprime "Apto 201"
? aPredio[ 4 ][ 3 ] // Imprime "Apto 403"

// O comando abaixo gera um erro de ``array asign''.
? aPredio[ 3 ][ 3 ] // Erro, pois o elemento não existe.
```

16.4.5 Arrays são referências quando igualadas

Se você usar um operador de atribuição para atribuir o conteúdo de um array em outro array você não esta criando um novo array, mas criando uma referência para o array antigo.

```
a1 := { 10, 20, 30 }
a2 := a1

a1[ 2 ] := 7
```

```
? a2[ 2 ] // Imprime 7 em vez de 20
```

Entendeu ? Quando eu uso o operador para atribuir um valor através de outra variável as duas variáveis apontam para o mesmo endereço de memória. Se eu mudar a2, a1 também mudará, e vice-verça. Isso porque as duas variáveis, apesar de diferentes, apontam para o mesmo local na memória (para o mesmo conteúdo).

A pegadinha da atribuição IN LINE

Uma consequência direta disso é a atribuição IN LINE. Até agora, com variáveis comuns, quando você inicializa diversas variáveis através de atribuição IN LINE você está criando variáveis independentes. Por exemplo:

```
LOCAL nVal1 := nVal2 := nVal3 := 0 // Cria 3 variáveis numéricas ind
```

Já com arrays isso não existe. Se você fizer uma atribuição IN LINE com arrays, você estará criando 3 referências para o mesmo array. Por exemplo:

```
LOCAL a1 := a2 := a3 := {}
```

Se você inserir um elemento em a2, automaticamente a1 e a3 receberão esse mesmo elemento. O Harbour cria referências para a mesma variável. O exemplo 16.8 ilustra esse comportamento.

Listagem 16.8: Atribuição IN LINE de arrays: CUIDADO!

```
/*
Atribuição IN LINE
*/
PROCEDURE Main
LOCAL a1 := a2 := a3 := {}
LOCAL x

   AADD( a1 , 10 )
   AADD( a1 , 20 )
   AADD( a1 , 30 )


   FOR x := 1 TO LEN( a3 )
       ? a2[ x ]
   NEXT


RETURN
```

.:Resultado:.

```
10
20
30
```

Note que apenas o array a1 recebeu elementos, mas como eles foram inicializados IN LINE, o array a2 e a3 viraram referências para o array a1. Programadores C que estão familiarizados com ponteiros sabem porque isso ocorreu, mas como nosso curso é introdutório, basta que você evite inicializar arrays IN LINE e também evitar atribuições.

Dica 94

Evite inicializar arrays IN LINE.

Dica 95

Da mesma forma que você deve evitar atribuições IN LINE você deve ter cuidado com atribuições de igualdade entre os arrays.

Dica 96

Caso você necessite criar um array independente de uma variável existente use a função ACLONE(). Essa função cria uma cópia independente do array.

```
a1 := { 10, 20, 30 }
a2 := ACLONE( a1 )

a1[ 2 ] := 7

? a2[ 2 ] // Imprime 20
```

A pegadinha do operador de comparação

De acordo com Ramalho

> uma matriz [array] só pode ser comparada com outra usando-se o duplo operador de igualdade "==". O operador retornará .t. apenas se elas fizerem referência à mesma localização de memória [Ramalho 1991, p. 397].

Listagem 16.9: Igualdade de arrays: CUIDADO!

```
/*
Atribuição IN LINE
*/
PROCEDURE Main
LOCAL a1 := a2 := {}  // Mesma referência
LOCAL a3 := {}
```

```
   ?                                                                    8
   ? "Vou preencher a1 e a3 com o mesmo conteúdo..."                    9
   ?                                                                    10
   AADD( a1 , 10 )                                                      11
   AADD( a1 , 20 )                                                      12
   AADD( a1 , 30 )                                                      13
                                                                        14
   AADD( a3 , 10 )                                                      15
   AADD( a3 , 20 )                                                      16
   AADD( a3 , 30 )                                                      17
                                                                        18
                                                                        19
   ? "a1 e a2 possuem a mesma referência, por isso säo iguais"          20
   ? "a1 == a2  :  ", a1 == a2                                          21
                                                                        22
   ? "a1 e a3 säo independentes, por isso näo säo iguais"               23
   ? "a1 == a3  :  ", a1 == a3                                          24
                                                                        25
                                                                        26
RETURN                                                                  27
```

Note que, apesar de a1 e a3 terem os mesmos elementos eles não são considerados iguais. Para serem iguais, os arrays precisam compartilhar o mesmo endereço.

.:Resultado:.

```
Vou preencher a1 e a3 com o mesmo conteúdo...

a1 e a2 possuem a mesma referência, por isso são iguais
a1 == a2  :   .T.
a1 e a3 são independentes, por isso não são iguais
a1 == a3  :   .F.
```

Caso você deseje verificar se dois arrays tem os mesmos elementos, você pode usar a função hb_valtoexp(). Essa função converte um valor qualquer para string[4].

Listagem 16.10: Igualdade entre elementos de arrays.

```
/*                                                                      1
Outras soluções em : http://www.pctoledo.com.br/forum/viewtopic.php?f=4&t=1747
*/                                                                      3
PROCEDURE Main                                                          4
LOCAL a1 := a2 := {}  // Mesma referência                               5
LOCAL a3 := {}                                                          6
                                                                        7
   ?                                                                    8
   ? "Vou preencher a1 e a3 com o mesmo conteúdo..."                    9
   ?                                                                    10
   AADD( a1 , 10 )                                                      11
   AADD( a1 , { 20 , 30 } )                                             12
```

[4]Outras soluções podem ser obtidas em `http://www.pctoledo.com.br/forum/viewtopic.php?f=4&t=17476`. Você deve ter cuidado quando for realizar essa verificação com arrays gigantescos, pois a conversão pode demandar muito tempo.

```
   AADD( a1 , 30 )

   AADD( a3 , 10 )
   AADD( a3 , { 20 , 30 } )
   AADD( a3 , 30 )


   ? "a1 e a2 possuem a mesma referência, por isso são iguais"
   ? "a1 == a2  :  ", a1 == a2

   ? "a1 e a3 são independentes, por isso não são iguais"
   ? "a1 == a3  :  ", a1 == a3


   ? "Agora vou comparar o conteúdo de a1 e a3 usando hb_valtoexp() ..."

   ? "hb_valtoexp( a1 ) == hb_valtoexp( a3 )  :  ", hb_valtoexp( a1 ) == hb_va
   ?
   ? "Veja o que faz hb_valtoexp()"
   ? hb_valtoexp( a1 )


RETURN
```

.:Resultado:.

```
Vou preencher a1 e a3 com o mesmo conteúdo...

a1 e a2 possuem a mesma referência, por isso são iguais
a1 == a2  :   .T.
a1 e a3 são independentes, por isso não são iguais
a1 == a3  :   .F.
Agora vou comparar o conteúdo de a1 e a3 usando hb_valtoexp() ...
hb_valtoexp( a1 ) == hb_valtoexp( a3 )  :   .T.

Veja o que faz hb_valtoexp()
{10, {20, 30}, 30}
```

16.4.6 Arrays na passagem de parâmetros

Quando nós estudamos as rotinas e a passagem de parâmetros, vimos que os argumentos são passados por valor. Se quisermos passar um argumento por referência devemos prefixar ele com um @. Os arrays também se comportam segundo essa regra, mas existem algumas mudanças significativas.

Vamos tentar explicar através de exemplos.

Primeiro, analise o código da listagem

Listagem 16.11: Arrays passados por valor

```
/*
Exemplo retirado de : Clipper 5.2 - Rick Spence , p.159
```

```
*/
PROCEDURE Main
LOCAL aNomes := { "Spence", "Thompson" }

   Test( aNomes )
   ? aNomes[ 1 ] // Spence


RETURN

PROCEDURE Test( aFormal )

   ? aFormal[1] // Spence
   aFormal := { "Rick" , "Malcolm" }

RETURN
```

.:Resultado:.

```
Spence
Spence
```

Até agora nada de novo com relação aos outros tipos de dados. O array foi passado por valor, conforme os outros tipos, e o valor não foi alterado fora da rotina.

Se quiséssemos, nós poderíamos passar o array por referência. Basta prefixar o argumento com um @. Também nada de novo aqui.

Listagem 16.12: Arrays passados por referência

```
/*
Exemplo retirado de : Clipper 5.2 - Rick Spence , p.159
*/
PROCEDURE Main
LOCAL aNomes := { "Spence", "Thompson" }

   Test( @aNomes )
   ? aNomes[ 1 ] // Rick


RETURN

PROCEDURE Test( aFormal )

   ? aFormal[1] // Spence
   aFormal := { "Rick" , "Malcolm" }

RETURN
```

.:Resultado:.

```
Spence
Rick
```

Agora muita atenção: Se eu passar um array por valor e for alterar **um elemento seu** dentro da rotina chamada, isso irá alterar o array na rotina chamadora. Acompanhe o código a seguir.

Listagem 16.13: Arrays sempre são passados por referência quando modificamos seus elementos internos.

```
PROCEDURE Main
LOCAL a1 := { 10, 20, 30 }

   ? "O primeiro elemento é ", a1[ 1 ]

   Muda( a1 )

   ? "O primeiro elemento foi alterado : ", a1[ 1 ]



RETURN

PROCEDURE Muda( aArray )

   aArray[ 1 ] := "Mudei aqui"

RETURN
```

.:Resultado:.

```
O primeiro elemento é          10
O primeiro elemento foi alterado :  Mudei aqui
```

Dica 97

Tenha cuidado quando for passar arrays como um argumento de uma rotina. Lembre-se de que os seus elementos podem sofrer alteração dentro dessa rotina e o array será alterado também na rotina chamadora.

16.4.7 Elementos de arrays na passagem de parâmetros

De acordo com Spence, “os elementos de um array são sempre enviados por valor, o que significa que a rotina não pode alterá-los” [Spence 1994, p. 162]

Listagem 16.14: Elementos de arrays na passagem de parâmetros

```

PROCEDURE Main
LOCAL aNomes := { "Spence", "Thompson" }

   Test( aNomes[ 1 ] )
   ? aNomes[ 1 ] // Spence
   Test( @aNomes[ 1 ] )
   ? aNomes[ 1 ] // Outro qualquer
```

```
                                                                    9
                                                                    10
RETURN                                                              11
                                                                    12
PROCEDURE Test( cNome )                                             13
                                                                    14
   cNome := "Outro qualquer"                                        15
                                                                    16
RETURN                                                              17
```

.:Resultado:.

```
Spence
Outro qualquer
```

Ou seja: elementos de arrays se comportam como se fossem variáveis comuns. Note que na linha 7 o elemento do array foi passado por referência, por isso ele foi modificado fora da rotina também.

16.4.8 Retornando arrays a partir de funções

Arrays podem ser retornados a partir de funções, inclusive o Harbour possui algumas funções que retornam arrays, como é o caso da função Directory(), que retorna um array com dados sobre uma pasta qualquer. Veremos essa e outras funções ainda nesse capítulo.

Dica 98

Retornar arrays de funções é útil quando você deseja retornar informações complexas.
Faça suas funções de retorno de arrays ao invés de passar muitos parâmetros por referência [Spence 1991, p. 156].

Listagem 16.15: Retornando arrays de funções

```
                                                                    1
PROCEDURE Main                                                      2
LOCAL aNomes1, aNomes2                                              3
                                                                    4
   aNomes1 := Test( "Gilberto" , "Silvério" )                       5
   aNomes2 := Test( "William" , "Manesco" )                         6
                                                                    7
   ? "Nome 1 : " , aNomes1[ 1 ] , aNomes1[ 2 ]                      8
   ? "Nome 2 : " , aNomes2[ 1 ] , aNomes2[ 2 ]                      9
                                                                    10
RETURN                                                              11
                                                                    12
FUNCTION Test( cFirst, cLast )                                      13
                                                                    14
RETURN { cFirst , cLast }                                           15
```

.:Resultado:.

```
Nome 1 :  Gilberto Silvério
Nome 2 :  William Manesco
```

Note que, a cada retorno da função vem um array com uma nova referência. Ou seja, eles não tem relação entre si e podem ser usados livremente.

16.4.9 Lembra do laço FOR EACH ?

O laço FOR EACH foi apresentado quando nós estudávamos as estruturas de repetição. Como ainda não havíamos visto arrays e hashs, os exemplos dados foram apenas com strings. A listagem a seguir ilustra como devemos fazer para percorrer todos os elementos de um array unidimensional.

Listagem 16.16: Laço FOR EACH com arrays

```
PROCEDURE Main
LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" , "F" , "G" }
LOCAL cElemento

   FOR EACH cElemento IN aLetras
       ? cElemento
   NEXT

RETURN
```

.:Resultado:.

```
A
B
C
D
E
F
G
```

O enumerador do laço FOR EACH tem propriedades especiais que podem ser acessadas usando o seguinte formato abaixo:

```
<elemento>:<propriedade>
```

Como estamos estudando os arrays, vamos demonstrar no fragmento a seguir as propriedades especiais do enumerador aplicada ao array.

```
<elemento>:__enumindex // O índice do elemento corrente
<elemento>:__enumvalue // O valor do elemento corrente
<elemento>:__enumbase // O array
```

A listagem 16.17 ilustra isso:

Listagem 16.17: Laço FOR EACH com arrays

```
PROCEDURE Main
LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" , "F" , "G" }
LOCAL cElemento

   FOR EACH cElemento IN aLetras
       ? cElemento:__enumindex , " ==> "
       ?? cElemento:__enumvalue, " = " ,  cElemento
   NEXT

RETURN
```

.:Resultado:.

```
        1  ==>  A  =  A
        2  ==>  B  =  B
        3  ==>  C  =  C
        4  ==>  D  =  D
        5  ==>  E  =  E
        6  ==>  F  =  F
        7  ==>  G  =  G
```

Note que a propriedade __enumvalue é dispensável (linha 8), já que digitar cElemento é a mesma coisa que digitar cElemento:__enumvalue.

Temos também a propriedade __enumbase, que retorna o array completo que gerou a consulta. A listagem 16.18 ilustra isso:

Listagem 16.18: Laço FOR EACH com arrays

```
PROCEDURE Main
LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" , "F" , "G" }
LOCAL cElemento

   FOR EACH cElemento IN aLetras
       ? hb_valtoexp( cElemento:__enumbase )
   NEXT

RETURN
```

.:Resultado:.

```
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
```

```
{"A", "B", "C", "D", "E", "F", "G"}
```

16.4.10 Algumas funções que operam sobre arrays

O Harbour possui inúmeras funções que operam sobre os arrays, mas como nosso curso é introdutório nós veremos apenas as principais funções. Gostaríamos de enfatizar também que a teoria sobre os arrays encerra-se na seção anterior, portanto, você pode passar para a seção seguinte (que trata de hashs). De qualquer maneira é bom dar, pelo menos, uma olhada nessa seção, pois ela servirá como material para consultas futuras.

Dica 99

A lista oficial com as funções do Harbour e outras informações importantes você encontra em `https://harbour.github.io/doc/index.html`.

AADD()

A função AADD() foi vista na seção 16.4.3.

ACLONE()

A função ACLONE() já foi vista na seção 16.4.5. Ela serve para copiar um array para outro array sem que os mesmos referenciem a mesma posição na memória. O fragmento a seguir ilustra o uso da função ACLONE().

```
a1 := { 10, 20, 30 }
a2 := ACLONE( a1 )

a1[ 2 ] := 7

? a2[ 2 ] // Imprime 20
```

Caso você tenha alguma dúvida sobre o problema que a função ACLONE() resolve, consulte a seção 16.4.5.

ASIZE()

Descrição sintática 23

1. Nome : ASIZE()
2. Classificação : função.
3. Descrição : Aumenta ou diminui um array
4. Sintaxe

```
AADD( <aArray>, [<nTamanho>] ) -> aDestino

   <aArray> : Array de origem
   <nTamanho> : Tamanho desejado

   <aDestino> : Array destino
```

Fonte : [Nantucket 1990, p. 5-27]

Listagem 16.19: Função ASIZE()

```
PROCEDURE Main
LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" }
LOCAL cElemento

   ?
   ? "Os elementos originais"
   FOR EACH cElemento IN aLetras
       ? cElemento
   NEXT
   ?
   ? "Diminuindo o tamanho para 3 elementos"
   aLetras := ASIZE( aLetras, 3 )
   FOR EACH cElemento IN aLetras
       ? cElemento
   NEXT
   ?
   ? "Aumentando o tamanho para 5 elementos"
   aLetras := ASIZE( aLetras, 5 )
   FOR EACH cElemento IN aLetras
       ? cElemento
   NEXT


RETURN
```

.:Resultado:.

```
Os elementos originais
A
B
C
D
E

Diminuindo o tamanho para 3 elementos
A
B
C

Aumentando o tamanho para 5 elementos
A
B
C
NIL
NIL
```

Note que quando nós aumentamos o tamanho de um array, se o tamanho final for maior do que o tamanho do array original os espaços adicionais receberão o valor NIL.

ADEL()

A função ADEL() exclui um elemento de um array mas preserva o tamanho final do array. Essa função faz parte do Clipper 5.

Descrição sintática 24

1. Nome : ADEL()
2. Classificação : função.
3. Descrição : Elimina um elemento do array.
4. Sintaxe

```
ADEL( <aArray>, <nPosição> ) -> aDestino

   <aArray> : Array que contém um elemento a ser eliminado.
   <nPosição> : Posição do elemento que será eliminado.

   <aDestino> : Array destino
```

Fonte : [Nantucket 1990, p. 5-11]

O fragmento a seguir mostra como ADEL() se comporta.

```
LOCAL aArray := { "Pêra", "Uva", "Maçã" }

ADEL( aArray , 2 ) // { "Pêra", "Maçã" , NIL }
```

Note que ela exclui o elemento **e todos os elementos a partir daquela posição até o final** perdem uma posição. O último elemento torna-se NIL.

HB_ADEL()

Essa função foi desenvolvida pela equipe do Harbour, e aprimora a função ADEL(). Ela funciona da mesma forma que a função ADEL(), mas ela tem um terceiro parâmetro opcional. Se esse parâmetro for true (.t.) então o último elemento deixa de ser NIL e é excluído definitivamente.

Acompanhe o fragmento de código a seguir :

```
LOCAL aArray := { "Harbour", "is", "Power" }
hb_ADel( aArray, 2 ) // Result: aArray is { "Harbour", "Power", NIL } -
hb_ADel( aArray, 2, .T. ) // Result: aArray is { "Harbour", "Power" } -
```

Fonte : `https://harbour.github.io/doc/harbour.html#hb_adel` - On line: 29-Dez-2016.

AINS()

A função AINS() insere um elemento NIL em um array e (da mesma forma que ADEL()) preserva o seu tamanho final. Essa função faz parte do Clipper 5.

Descrição sintática 25

1. Nome : AINS()
2. Classificação : função.
3. Descrição : Insere um elemento NIL em um array. O elemento recém inserido é NIL até que um novo valor seja atribuído a ele. Após a inserção **o último elemento é descartado**.
4. Sintaxe

```
AINS( <aArray>, <nPosição> ) -> aDestino

   <aArray> : Array que contém um elemento a ser inserido.
   <nPosição> : Posição do novo elemento.

   <aDestino> : Array destino
```

Fonte : [Nantucket 1990, p. 5-19]

O fragmento a seguir mostra como AINS() se comporta.

```
LOCAL aArray := { "Pêra", "Uva", "Maçã" }

AINS( aArray , 2 ) // { "Pêra", NIL, "Maçã"  }
```

Note que ela exclui o último elemento.

HB_AINS()

Essa função foi desenvolvida pela equipe do Harbour, e aprimora a função AINS(). Ela funciona da mesma forma que a função AINS() mas ela pode inserir um valor no array sem perder o último elemento.

Acompanhe o fragmento de código a seguir :

```
LOCAL aArray := { "Harbour", "Power!" }
hb_AIns( aArray, 2,  "is", .F. ) // --> { "Harbour", "is" }
hb_AIns( aArray, 2,  "is", .T. ) // --> { "Harbour", "is", "Power!" }
```

Fonte : `https://harbour.github.io/doc/harbour.html#hb_ains` - On line: 29-Dez-2016.

Em resumo, os adicionais da função HB_AINS() são :

1. O valor adicionado, não necessariamente, é NIL. Isso é determinado pelo terceiro parâmetro.
2. O último valor, não necessariamente, irá se perder. Isso é determinado pelo quarto parâmetro (.t.).

16.5 O Hash

Array associativo ou Hash é um recurso oriundo de linguagens interpretadas, como o PHP, Perl, Python e Ruby. O Clipper não tinha essa estrutura de dados. Com o surgimento do Harbor e as constantes melhorias decorrentes da cultura de software-livre, o Harbour acabou implementando essa estrutura de dados também. Grosso modo, um Hash é um tipo especial de array, cujo ponteiro pode ser uma string também. O fragmento de código abaixo ilustra essa diferença:

```
aCliente[ 10 ] := "Daniel  Crocciari'' // Array
hCliente[ "nome" ] := "Daniel  Crocciari'' // Hash
```

16.5.1 Criando um Hash

Um hash novo é criado com a função HB_HASH() ou sem o uso dessa função.

Com a função HB_HASH()

Primeira forma: sem parâmetros. Usada sem parâmetros, a função criará um hash vazio (sem elementos). Apenas declara uma variável e informa que é um hash.

```
LOCAL hClientes := HB_HASH()
```

Segunda forma: com parâmetros separado por vírgulas.

```
LOCAL hClientes := HB_HASH( "nome" , "Alexandre" , "sobrenome" , "Sa

 ? hClientes[ "nome" ] // Alexandre
 ? hClientes[ "sobrenome" ] // Santos
```

Sem a função HB_HASH()

Primeira forma: Um Hash vazio.

```
LOCAL hClientes := { => }
```

Segunda forma: Semelhante a um array mas com o par chave/valor separado por setas (“=>”). Esse formato é semelhante ao das outras linguagens que tem hash.

```
LOCAL hClientes := { "nome" => "Alexandre" , "sobrenome" => "Santos"
```

16.5.2 Percorrendo um Hash com FOR EACH

Um Hash pode ser percorrido facilmente com o laço FOR EACH, conforme a listagem 16.20.

Listagem 16.20: Laço FOR EACH com hashs

```
PROCEDURE Main
LOCAL hLetras := { "A" => "Letra A" , "B" => "Letra B" , "C" => "Letra C" }
LOCAL cElemento

   FOR EACH cElemento IN hLetras
       ? cElemento
   NEXT

RETURN
```

.:Resultado:.

```
Letra A
Letra B
Letra C
```

O laço FOR EACH também tem possui suas próprias propriedades especiais para os enumeradores.

```
<elemento>:__enumindex // A ordem numérica do elemento corrente
<elemento>:__enumvalue // O valor do elemento corrente
<elemento>:__enumbase // O hash
<elemento>:__enumkey // A chave do hash
```

A listagem 16.21 ilustra isso:

Listagem 16.21: Laço FOR EACH com hashs

```
1
2
PROCEDURE Main
3
LOCAL hLetras := { "A" => "Letra A" , "B" => "Letra B" , "C" => "Letra C"  }
4
LOCAL cElemento
5
6
   FOR EACH cElemento IN hLetras
7
       ? cElemento:__enumindex , cElemento:__enumkey , " ==> "
8
       ?? cElemento:__enumvalue, " = " ,  cElemento
9
   NEXT
10
11
RETURN
```

.:Resultado:.

```
        1 A  ==> Letra A  =  Letra A
        2 B  ==> Letra B  =  Letra B
        3 C  ==> Letra C  =  Letra C
```

A novidade aqui é a propriedade __enumkey, que retorna a chave do hash. As demais propriedades também estão presentes nos arrays.

Note que a propriedade __enumindex serve tanto para arrays quanto para hashs. Note também que a propriedade __enumvalue é dispensável (linha 8), já que digitar cElemento é a mesma coisa que digitar cElemento:__enumvalue.

Temos também a propriedade __enumbase, que retorna o hash completo que gerou a consulta. A listagem 16.22 ilustra isso:

Listagem 16.22: Laço FOR EACH com hashs

```
1
2
PROCEDURE Main
3
LOCAL hLetras := { "A" => "Letra A" , "B" => "Letra B" , "C" => "Letra C"  }
4
LOCAL cElemento
```

```
   FOR EACH cElemento IN hLetras
       ? hb_valtoexp( cElemento:__enumbase )
   NEXT

RETURN
```

.:Resultado:.

```
{"A"=>"Letra A", "B"=>"Letra B", "C"=>"Letra C"}
{"A"=>"Letra A", "B"=>"Letra B", "C"=>"Letra C"}
{"A"=>"Letra A", "B"=>"Letra B", "C"=>"Letra C"}
```

Mais sobre o laço FOR EACH pode ser obtido em `http://pctoledo.com.br/forum/viewtopic.php?f=4&t=17591&hilit=hash`.

Dica 100

Note que um hash possui três componentes, mas apenas dois deles são visíveis ao programador:

1. Chave : é a string que identifica o elemento. Ela fica entre colchetes.
2. Valor : é o valor do elemento.
3. Índice : é a ordem numérica onde o elemento se encontra. Esse componente não é visível ao programador, mas pode ser obtido através da função HB_hPos(), que será vista na próxima seção.

16.5.3 O hash e as atribuições in line

Os hashs assemelham-se aos arrays também na forma de atribuição. Lembra dos arrays e as atribuições in-line ? Pois bem, com os hashs acontece a mesma coisa. A listagem a seguir ilustra a situação.

Listagem 16.23: O hash e as atribuições in line

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL hCopia
LOCAL cElemento

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "á" ] := "A minúscula"
   hLetras[ "Z" ] := "Letra Z"



   ? "-------------------------------"
   ? "Valores de hLetra"
```

```
   ? "-------------------------------"
   FOR EACH cElemento IN hLetras
      ? cElemento
   NEXT
   // Uma atribuição e uma alteração
   hCopia := hLetras
   hCopia[ "novo" ] := "Nova letra"

   ? "-------------------------------"
   ? "Valores de hLetra após hCopia receber um novo elemento"
   ? "-------------------------------"

   FOR EACH cElemento IN hLetras
      ? cElemento
   NEXT


RETURN
```

.:Resultado:.

```
-------------------------------
Valores de hLetra
-------------------------------
Letra K
Letra A
A minúscula
Letra Z
-------------------------------
Valores de hLetra após hCopia receber um novo elemento
-------------------------------
Letra K
Letra A
A minúscula
Letra Z
Nova letra
```

Note que o hash hLetra e o hash hCopia apontam para a mesma área de memória, e consequentemente, possuem os mesmos elementos. O que acontece com um, automaticamente, acontece com o outro também.

16.5.4 As características de um hash

Sempre que você cria um hash, ele já vem configurado com algumas características básicas que definem o seu comportamento. Essas propriedades assumem dois valores: ligada (.t.) ou desligada (.f.). As propriedade estão listadas nas próximas subseções:

AutoAdd

Essa propriedade determina se um elemento de um hash será adicionado automaticamente logo após a sua definição. Essa propriedade pode ser entendida se você

recordar dos arrays. Acompanhe o fragmento abaixo :

```
LOCAL aClientes := {}

     aClientes[ 1 ] := "Rick Spence" // Erro de array assign
```

Como você já deve ter percebido, o fragmento acima gerará um erro de execução pois o array foi inicializado com zero elementos, mas nós tentamos inserir um elemento que, é claro, está fora do intervalo válido.

O código a seguir (listagem 16.24) foi implementado usando um hash e faz uma atribuição semelhante.

Listagem 16.24: Propriedades do hash

```
1
2
3
4
5
6
7
8
```

```
PROCEDURE Main
LOCAL hLetras := {  =>   }

   hLetras[ "A" ] := "Letra A"
   ? hLetras[ "A" ]

RETURN
```

Porém, ao contrário do array, o programa fez exatamente o desejado.

.:Resultado:.

```
Letra A
```

Isso só foi possível porque a propriedade *AutoAdd* do hash é setada como verdadeira (.t.) por default. Mas pode existir um caso onde você quer que o hash se comporte como um array, ou seja, que ele só aceite atribuições em chaves previamente criadas. Para que isso aconteça você deve setar a propriedade *AutoAdd* como falsa (.t.), e isso é feito através da função HB_hAutoAdd(). No código da listagem 16.25 nós tentamos fazer a mesma coisa que o codigo da listagem 16.24, mas agora nós obtemos um erro.

Listagem 16.25: Propriedades do hash - AutoAdd

```
1
2
3
4
5
6
7
8
9
10
```

```
PROCEDURE Main
LOCAL hLetras := {  =>   }

   hb_hAutoAdd( hLetras , .f. )

   hLetras[ "A" ] := "Letra A"
   ? hLetras[ "A" ]

RETURN
```

Na linha 5 nós alteramos a propriedade *AutoAdd* do hash. Note que o erro ocorreu na linha 7, onde eu tentei fazer a atribuição. Veja que a mensagem de erro é a mesma gerada para um array.

.:Resultado:.

```
Error BASE/1133  Bound error: array assign
Called from MAIN(7)
```

Binary

Determina a forma como o hash será ordenado, se no padrão binário ou se de acordo com o codepage usado. Talvez você não se sinta muito a vontade com essa propriedade, porque ela necessita do entendimento de dois conceitos ainda não vistos completamente: ordenação de hash e codepage. Mesmo assim vamos tentar explicar essa propriedade. Primeiramente vamos entender como um array é ordenado, acompanhe a listagem 16.26.

Listagem 16.26: Propriedades do hash - Binary

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL hOrdenado
LOCAL cElemento

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "á" ] := "A minúscula"
   hLetras[ "Z" ] := "Letra Z"

   hOrdenado := HB_hSort( hLetras )

   FOR EACH cElemento IN hOrdenado
      ? cElemento
   NEXT

RETURN
```

Observe que a ordenação é feita através da função HB_hSort(), e que ela ordena pela chave do hash, e não pelo seu valor.

.:Resultado:.

```

Letra A
Letra K
Letra Z
A minúscula
```

Note que a chave de um hash pode ter caracteres acentuados também caracteres que não pertencem ao nosso alfabeto, como mandarim, árabe, etc. Tudo irá depender do codepage ativo. Pois bem, quando a propriedade binary está ativa (.t.), que é o default, o hash irá desprezar a ordenação de caracteres especiais e irá ordenar apenas levando em consideração o código binário. Caso contrário, ela irá levar em conta os caracteres especiais de cada idioma.

Dica 101

Deixe o padrão de busca Binary com o seu valor default (.t.), pois o algoritmo de ordenação é mais eficiente.

Dica 102

Evite colocar acentos ou caracteres especiais na chave de um hash.

Para alterar a forma como a ordenação é feita use a função HB_hBinary(). Veja um exemplo logo a seguir.

Listagem 16.27: Propriedades do hash - Binary

```
1
2  PROCEDURE Main
3  LOCAL hLetras := {  =>   }
4  LOCAL hOrdenado
5  LOCAL cElemento
6
7     hb_hBinary( hLetras , .f. ) // Leva em consideração o codepage
8
9     hLetras[ "K" ] := "Letra K"
10    hLetras[ "A" ] := "Letra A"
11    hLetras[ "á" ] := "A minúscula"
12    hLetras[ "Z" ] := "Letra Z"
13
14    hOrdenado := HB_hSort( hLetras )
15
16    FOR EACH cElemento IN hOrdenado
17       ? cElemento
18    NEXT
19
20 RETURN
```

Note que a mudança no padrão de ordenamento foi mudado na linha 7. Acompanhe o resultado a seguir:

.:Resultado:.

```
Letra A
Letra K
Letra Z
A minúscula
```

Veja que o resultado é exatamente igual a busca binária, porém, de acordo com o codepage e do símbolo usados, essa ordenação poderá se alterar. Por isso nós recomendamos ordenar sempre pelo padrão binário, pois não mudará a forma com que as chaves são ordenadas, independente de qual codepage estiver ativo, além do mais, como já foi dito, a busca binária é mais rápida.

CaseMatch

CaseMatch determina se a chave do hash deverá ser sensível a maiúsculas/minúsculas ou não. Acompanhe um exemplo na listagem 16.28.

Listagem 16.28: Propriedades do hash - CaseMatch

```
PROCEDURE Main
LOCAL hClientes := {  =>   }
LOCAL cElemento


   hClientes[ "Cliente" ] := "Robert Plant"
   hClientes[ "CLIENTE" ] := "Jimmy Page"
   hClientes[ "cliente" ] := "Jonh Bonham"
   hClientes[ "CLIente" ] := "Jonh Paul Jones"


   FOR EACH cElemento IN hClientes
      ? cElemento
   NEXT

RETURN
```

Esse exemplo serve para ilustrar que o comportamento padrão **leva em consideração maiúsculas e minúsculas**. Acompanhe o resultado a seguir:

.:Resultado:.

```
Robert Plant
Jimmy Page
Jonh Bonham
Jonh Paul Jones
```

Esse é o comportamento padrão. Para alterar esse comportamento você deve alterar a propriedade CaseMatch através da função HB_hCaseMatch(). Veja a alteração na listagem a seguir.

Listagem 16.29: Propriedades do hash - CaseMatch

```
PROCEDURE Main
LOCAL hClientes := {  =>   }
LOCAL cElemento

   HB_hCaseMatch( hClientes, .f. )

   hClientes[ "Cliente" ] := "Robert Plant"
   hClientes[ "CLIENTE" ] := "Jimmy Page"
   hClientes[ "cliente" ] := "Jonh Bonham"
   hClientes[ "CLIente" ] := "Jonh Paul Jones"
```

```
   FOR EACH cElemento IN hClientes
      ? cElemento
   NEXT

RETURN
```

Esse exemplo serve para ilustrar duas coisas:

1. Quando a propriedade CaseMatch é setada com o valor falso, o Harbour passa a desconsiderar as diferenças entre maiúsculas e minúsculas nas chaves dos hashs.
2. Já que o Hash considera “CLIENTE” = “cliente”, por exemplo, ele não vai criar um novo elemento para o hash (nas linhas 10,11 e 12), mas vai sobrepor o elemento anterior.

Acompanhe o resultado a seguir:

.:Resultado:.

```
Jonh Paul Jones
```

Note que o Hash possui apenas um valor. Os valores anteriores foram sobrepostos pois a chave foi considerada a mesma.

KeepOrder

KeepOrder significa “Manter a ordem”. Mas manter a ordem de que ? A resposta é : a ordem natural com que os elementos foram inseridos. Vamos explicar melhor: quando o hash é criado, a ordem com que os seus elementos são dispostos é a ordem com que foram adicionados. Essa é a propriedade padrão do hash.

Listagem 16.30: Propriedades do hash - KeepOrder

```

PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL cElemento

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "B" ] := "Letra B"
   hLetras[ "Z" ] := "Letra Z"

   FOR EACH cElemento IN hLetras
      ? cElemento
   NEXT

RETURN
```

.:Resultado:.

```
Letra K
Letra A
Letra B
Letra Z
```

Para desativar essa propriedade use a função HB_hKeepOrder(). Quando essa propriedade está desativada o hash é ordenado automaticamente pelos valores das chaves. Acompanhe a mudança:

Listagem 16.31: Propriedades do hash - KeepOrder

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL cElemento

   HB_hKeepOrder( hLetras , .f. )

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "B" ] := "Letra B"
   hLetras[ "Z" ] := "Letra Z"

   FOR EACH cElemento IN hLetras
      ? cElemento
   NEXT

RETURN
```

.:Resultado:.

```
Letra A
Letra B
Letra K
Letra Z
```

A função de ordenamento HB_hSort() não é mais necessária, pois o hash é ordenado automaticamente. É bom lembrar que, se essa propriedade for desativada o Harbour terá um trabalho extra de processamento (para manter o hash sempre indexado pela chave). Portanto, caso você não vá ordenar o seu Hash, evite desativar essa propriedade.

16.5.5 Funções que facilitam o manuseio de Hashs

A seguir nós temos algumas funções que facilitam o manuseio de Hashs. Você pode pular esse trecho e passar para a seção seguinte se desejar, mas nós aconselhamos que você dê, pelo menos, uma lida nessa seção. Isso porque esse material é mais para consulta posterior e não irá acrescentar nada a teoria dos hashs que nós acabamos de ver.

Você pode considerar essa seção como sendo um adicional ao material que nós vimos. É um material importante, pois lhe dá um leque de soluções e também lhe dá uma noção, através de exemplos, do que pode ser feito com um hash, porém, como já foi dito, esse material não acrescenta nada a teoria já vista.

HB_hHasKey()

Descrição sintática 26

1. Nome : HB_hHasKey()
2. Classificação : função.
3. Descrição : retorna um valor lógico (falso ou verdadeiro) indicando a existência ou não da chave informada no hash pesquisado.
4. Sintaxe

```
HB_hHasKey( <hHash>, <xKey> ) -> lExist

     <hHash> : Hash onde a chave será pesquisada.
     <xKey> : O valor da chave a ser buscado.

     lExist : Verdadeiro se o valor existir ou falso se o valo
```

Fonte : `http://www.kresin.ru/en/hrbfaq.html`

Listagem 16.32: Função HB_hHasKey()

```
PROCEDURE Main
LOCAL hLetras := {  =>   }

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "B" ] := "Letra B"
   hLetras[ "Z" ] := "Letra Z"

   ? "Buscando K maiúsculo : ", HB_hHasKey( hLetras , "K" ) // Achou

   ? "Buscando k minúsculo (CaseMatch verdadeiro) : ", HB_hHasKey( hLetras , "

   HB_hCaseMatch( hLetras , .f. ) // Mudei a propriedade Case Match

   ? "Buscando k minúsculo (CaseMatch falso) : ", HB_hHasKey( hLetras , "k" )


RETURN
```

Note que a propriedade CaseMatch influencia na busca do valor. Quando CaseMatch está ativo (padrão) a função retornou falso ao buscar o valor “k” minúsculo na linha 11, mas quando a propriedade CaseMatch foi desativada a mesma busca retornou verdadeiro (na linha 14).

.:Resultado:.

```
Buscando K maiúsculo :  .T.
Buscando k minúsculo (CaseMatch verdadeiro) :  .F.
Buscando k minúsculo (CaseMatch falso) :  .T.
```

HB_hGet()

A função HB_hGet() obtém o valor de um elemento de Hash através da sua chave. O fragmento a seguir exemplifica o seu uso.

```
hClientes := { => }
hClientes[ "2345" ] := "Marcos"
hClientes[ "123456" ] := "Claudia"

xChave := HB_hGet( hClientes, "123456" )  // Atribui ``Claudia'' a
```

Note que essa função equivale a uma atribuição normal já vista anteriormente. Portanto, as duas formas abaixo são equivalentes.

```
xChave := HB_hGet( hClientes, "123456" )

é a mesma coisa que

xChave := hClientes[ "123456" ]
```

HB_hSet()

A função HB_hSet() atribui um valor a um elemento de Hash através da sua chave. É a mesma coisa que uma atribuição normal, já vista. Portanto, as duas formas a seguir são equivalentes:

```
HB_hSet( hClientes, "123456" , "Claudia" )

é a mesma coisa que

hClientes[ "123456" ] := "Claudia"
```

HB_hGetDef()

Essa função é usada quando você deseja atribuir um valor default, caso o valor da chave buscada não exista. Iremos ilustrar o uso de HB_hGetDef() através de dois exemplos. O primeiro exemplo resolve um problema clássico de busca sem usar essa função, já o segundo exemplo irá usar a função HB_hGetDef().

Primeiramente o problema: o usuário deve informar uma chave de hash e o programa irá verificar se essa chave existe. Caso ela exista o programa deve atribuir a uma variável o valor do elemento correspondente, mas caso ela não exista, o programa deve atribuir um valor default a essa variável. Primeiramente veja o programa sem HB_hGetDef().

Listagem 16.33: Sem HB_hGetDef()

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL cNomeLetra, cChave

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "B" ] := "Letra B"
   hLetras[ "Z" ] := "Letra Z"

   ACCEPT "Informe a chave a ser buscada : " TO cChave
   IF HB_hHasKey( hLetras , cChave ) // Achou
      cNomeLetra := hLetras[ cChave ]
   ELSE
      cNomeLetra := "Näo encontrada"
   ENDIF

   ? "A variável cNomeLetra contém o valor : " , cNomeLetra

RETURN
```

Veja duas execuções desse programa, primeiro eu achei a chave :

.:Resultado:.

```
Informe a chave a ser buscada : K
A variável cNomeLetra contém o valor :  Letra K
```

Agora eu não achei a chave:

.:Resultado:.

```
Informe a chave a ser buscada : V
A variável cNomeLetra contém o valor :  Não encontrada
```

Agora vamos resolver esse mesmo problema usando a função HB_hGetDef(), repare como o código fica mais “enxuto”.

Listagem 16.34: Com HB_hGetDef()

```
```

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL cNomeLetra, cChave

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "B" ] := "Letra B"
   hLetras[ "Z" ] := "Letra Z"

   ACCEPT "Informe a chave a ser buscada : " TO cChave

   cNomeLetra := HB_hGetDef( hLetras, cChave, "Näo encontrada" )

   ? "A variável cNomeLetra contém o valor : " , cNomeLetra

RETURN
```

HB_hDel()

Essa função exclui um elemento de um Hash.

Listagem 16.35: Excluindo um elemento de um Hash

```

PROCEDURE Main
LOCAL hLedZeppelin:= {  =>   }
LOCAL cElemento

   hLedZeppelin[ "Vocalista" ] := "Robert Plant"
   hLedZeppelin[ "Guitarrista" ] := "Jimmy Page"
   hLedZeppelin[ "Baterista" ] := "Jonh Bonham"
   hLedZeppelin[ "Baixista" ] := "Jonh Paul Jones"

   FOR EACH cElemento IN hLedZeppelin
      ? cElemento
   NEXT
   ? "O hash possui " , LEN( hLedZeppelin ) , " elementos."

   hb_hDel( hLedZeppelin , "Baterista" )
   ? "O hash possui agora " , LEN( hLedZeppelin ) , " elementos."

RETURN
```

.:Resultado:.

```
Robert Plant
Jimmy Page
Jonh Bonham
Jonh Paul Jones
O hash possui           4  elementos.
O hash possui agora           3  elementos.
```

É bom ressaltar que o elemento realmente foi excluído. Não confunda o comportamento dessa função com o comportamento da função ADEL(), que exclui elementos de um array (a função ADEL() deixa o tamanho final do array inalterado). Portanto, HB_hDel() faz o serviço completo.

Se por acaso a chave informada não existir, nenhum erro é gerado, e o hash (é claro) ficará inalterado.

HB_hPos()

A função HB_hPos() retorna um valor numérico referente a posição que a chave ocupa no hash.

Descrição sintática 27

1. Nome : HB_hPos()
2. Classificação : função.
3. Descrição : retorna um valor numérico referente a posição que a chave ocupa no hash.
4. Sintaxe

```
HB_hPos( <hHash>, <xKey> ) -> nPosicao

     <hHash> : Hash onde a chave será pesquisada.
     <xKey> : O valor da chave a ser buscado.

     nPosicao : A posição que o elemento ocupa no hash.
```

Fonte : `http://www.t-gtk.org/index.php/pt/harbour-mnu-es-es/harbour-reference-guide/21-hrg-en-api/hb-en-hash-table/57-hb-en-hb-hpos`

Essa função é muito usada em conjunto com as funções HB_hValueAt() e HB_hKeyAt() que nós veremos detalhadamente a seguir.

HB_hValueAt()

A função HB_hValueAt() retorna o valor de um elemento de um hash a partir da sua posição. O exemplo a seguir (listagem 16.36) ilustra o seu uso em conjunto com a função HB_hPos().

Listagem 16.36: Achando o valor de um elemento a partir da sua posição

```
                                                                    1
PROCEDURE Main                                                      2
LOCAL hPinkFloyd := {  =>   }                                       3
LOCAL GetList := {}                                                 4
LOCAL cNome := SPACE( 30 )                                          5
```

```
LOCAL nPosicao                                                                  6
                                                                                7
   CLS                                                                          8
                                                                                9
                                                                                10
   hPinkFloyd[ "Guitarrista" ] := "David Gilmour"                               11
   hPinkFloyd[ "Baixista" ] := "Roger Waters"                                   12
   hPinkFloyd[ "Tecladista" ] := "Richard Wright"                               13
   hPinkFloyd[ "Baterista" ] := "Nick Mason"                                    14
                                                                                15
   @ 3, 4 SAY "Selecione abaixo para ver o nome do integrante : "               16
   @ 4, 4, 12, 30 GET cNome LISTBOX { "Guitarrista",;                           17
                                      "Baixista",;                              18
                                       "Tecladista",;                           19
                                       "Baterista" }                            20
   READ                                                                         21
                                                                                22
   nPosicao := HB_hPos( hPinkFloyd, cNome )                                     23
                                                                                24
   @ 16, 4 SAY "O nome do " + cNome + " é " + HB_hValueAt( hPinkFloyd, nPosica
                                                                                25
RETURN                                                                          26
```

Abaixo uma simulação.

.:Resultado:.

```
    Selecione abaixo para ver o nome do integrante :

   Guitarrista
   Baixista
   Tecladista
   Baterista




    O nome do Tecladista é Richard Wright
```

Note que, na linha 23, eu obtive a posição numérica com HB_hPos() para, só depois, usar HB_hValueAt().

A função HB_hValueAt() também possui um terceiro parâmetro, que corresponde ao novo valor que eu quero atribuir ao elemento. O seu uso está ilustrado no código a seguir:

Listagem 16.37: Obtendo ou trocando um valor

```
                                                                                1
PROCEDURE Main                                                                  2
LOCAL hLetras := {  =>   }                                                      3
LOCAL cElemento                                                                 4
LOCAL cAntigo                                                                   5
```

```
   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "B" ] := "Letra B"
   hLetras[ "Z" ] := "Letra Z"

   ? "O conjunto atual é : "

   FOR EACH cElemento IN hLetras
      ? cElemento:__enumkey, " => " , cElemento
   NEXT

   cAntigo := HB_hValueAt( hLetras, 2 , "A nova Letra A" )
   ? "O novo valor inserido foi: ", cAntigo
   ? "O conjunto AGORA é : "
   FOR EACH cElemento IN hLetras
      ? cElemento:__enumkey, " => " , cElemento
   NEXT

RETURN
```

Note que ela retorna o novo valor.

.:Resultado:.

```
O conjunto atual é :
K  =>  Letra K
A  =>  Letra A
B  =>  Letra B
Z  =>  Letra Z
O novo valor inserido foi:  A nova Letra A
O conjunto AGORA é :
K  =>  Letra K
A  =>  A nova Letra A
B  =>  Letra B
Z  =>  Letra Z
```

HB_hKeyAt()

Essa função é muito semelhante a HB_hValueAt(), só que em vez de retornar o valor ela retorna o nome da chave do elemento.

Listagem 16.38: Obtendo a chave do hash a partir da posição

```
PROCEDURE Main
LOCAL hPinkFloyd := {  =>   }
LOCAL nPosicao

   hPinkFloyd[ "Guitarrista" ] := "David Gilmour"
   hPinkFloyd[ "Baixista" ] := "Roger Waters"
   hPinkFloyd[ "Tecladista" ] := "Richard Wright"
```

```
   hPinkFloyd[ "Baterista" ] := "Nick Mason"                                    9
                                                                                10
   INPUT "Digite um valor entre 1 e 4 : " TO nPosicao                           11
                                                                                12
   ? "O nome da chave selecionada é : " , HB_hKeyAt( hPinkFloyd, nPosicao )     13
                                                                                14
RETURN
```

Note que ela retorna o novo valor.

.:Resultado:.

```
Digite um valor entre 1 e 4 : 1
O nome da chave selecionada é :  Guitarrista
```

Caso você informe um valor inexistente para a chave, por exemplo 7, um erro de execução será gerado:

.:Resultado:.

```
Digite um valor entre 1 e 4 : 7
Error BASE/1187  Bound error: HB_HKEYAT
Called from HB_HKEYAT(0)
Called from MAIN(12)
```

Importante: essa função não possui o terceiro parâmetro que HB_hValueAt() possui.

HB_hPairAt()

Essa função retorna um array de dois elementos (um par[5]). O primeiro elemento é a chave do hash e o segundo elemento é o valor do hash.

[5] A tradução da palavra “pair” é par, daí o nome da função conter a expressão “PairAt”.

Descrição sintática 28

1. Nome : HB_hPairAt()
2. Classificação : função.
3. Descrição : Retorna um array com dois elementos correspondendo a chave e ao valor do hash na posição informada.
4. Sintaxe

```
HB_hPairAt( <hHash>, <nPosicao> ) -> aArray

     <hHash> : Hash onde a chave será pesquisada.
     <nPosicao> : A posição do elemento.

     <aArray> : Array de dois elementos.
      O primeiro é a chave do elemento  hash e o
      segundo é o valor do elemento hash.
```

Listagem 16.39: Obtendo o par chave/valor do hash a partir da posição

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL aResult
LOCAL cElemento


   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "á" ] := "A minúscula"
   hLetras[ "Z" ] := "Letra Z"

   aResult := HB_hPairAt( hLetras , 1 )

   FOR EACH cElemento IN aResult
      ? cElemento
   NEXT

RETURN
```

.:Resultado:.

```
K
Letra K
```

HB_hDelAt()

O funcionamento dessa função é semelhante as funções anteriores. Só que ele exclui um elemento do hash a partir da posição, conforme a sintaxe a seguir:

Descrição sintática 29

1. Nome : HB_hDelAt()
2. Classificação : função.
3. Descrição : Exclui um elemento do hash a partir da sua posição.
4. Sintaxe

```
HB_hDelAt( <hHash>, <nPosicao> ) -> NIL

     <hHash> : Hash onde a chave será pesquisada.
     <nPosicao> : A posição do elemento.
```

Dica 103

A preposição **At** (cuja tradução significa “em”) caracteriza as seguintes funções:

1. HB_hKeyAt() : acha a chave a partir da posição.
2. HB_hValueAt() : acha o valor a partir da posição.
3. HB_hPairAt() : retorna o par chave/valor (em forma de array) a partir da posição.
4. HB_hDelAt() : exclui um elemento a partir da posição (não confunda com a função HB_hDel(), que já foi apresentada anteriormente.).

Em todas elas você deve passar a posição do elemento (ordem em que ele se encontra). Por isso você pode associar a preposição **At** com esse tipo de pesquisa (fica melhor para memorizar).
IMPORTANTE!
A função HB_hPos(hHash, Key) retorna a posição (índice) da chave de um hash. Ela deve ser usada para que você descubra a posição. Portanto, ela atua em conjunto com as 4 funções apresentadas.

HB_hKeys() e HB_hValues()

Essas duas funções retornam, respectivamente, as chaves e os valores de um hash, respectivamente. O retorno se dá em forma de array, conforme o exemplo abaixo:

Listagem 16.40: Obtendo o as chaves e os valores de um hash na forma de array

1

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL aChaves, aValores
LOCAL cElemento


   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "á" ] := "A minúscula"
   hLetras[ "Z" ] := "Letra Z"

   aChaves := HB_hKeys( hLetras )
   aValores := HB_hValues( hLetras )
   ? "------------------------------"
   ? "Array de chaves"
   ? "------------------------------"
   FOR EACH cElemento IN aChaves
      ? cElemento
   NEXT
   ? "------------------------------"
   ? "Array de valores"
   ? "------------------------------"
   FOR EACH cElemento IN aValores
      ? cElemento
   NEXT


RETURN
```

.:Resultado:.

```

------------------------------
Array de chaves
------------------------------
K
A
á
Z
------------------------------
Array de valores
------------------------------
Letra K
Letra A
A minúscula
Letra Z
```

HB_hFill()

A palavra “fill”, em português significa “preencher”, e é isso que essa função faz. Ela preenche um hash com um valor informado qualquer.

Listagem 16.41: Preenchendo um hash.

```
PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL cElemento

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "á" ] := "A minúscula"
   hLetras[ "Z" ] := "Letra Z"

   ? "------------------------------"
   ? "Valores originais"
   ? "------------------------------"
   FOR EACH cElemento IN hLetras
      ? cElemento
   NEXT
   ? "------------------------------"
   ? "Valores atribuidos por HB_hFill()"
   ? "------------------------------"
   HB_hFill( hLetras , "Harbour Project" )
   FOR EACH cElemento IN hLetras
      ? cElemento
   NEXT


RETURN
```

.:Resultado:.

```
------------------------------
Valores originais
------------------------------
Letra K
Letra A
A minúscula
Letra Z
------------------------------
Valores atribuidos por HB_hFill()
------------------------------
Harbour Project
Harbour Project
Harbour Project
Harbour Project
```

17 Conclusão

> O saber ensoberbece, mas o amor edifica. Se alguém julga saber alguma coisa, com efeito, não aprendeu ainda como convém saber.
>
> I Coríntios 8: 1 e 2

17.1 Umas palavras finais

Quando eu era estudante eu tive dois tipos de professores. O primeiro tipo me ensinou o passo a passo em um ambiente controlado, livre de imprevistos e de surpresas. Já o segundo tipo me mostrou, logo de início, que você pode facilmente perder o controle em uma situação corriqueira do dia-a-dia. Os professores do primeiro grupo me fizeram ter o grande prazer de ver a minha solução funcionando, e os professores do segundo grupo me ensinaram a ter cautela. Na minha opinião, os dois tipos são necessários: o primeiro tipo me incentivava a seguir em frente, a tentar e a correr riscos. Afinal de contas, as coisas não eram tão difíceis assim. O segundo tipo me alertava que até mesmo situações simples poderiam guardar surpresas desagradáveis. O primeiro tipo me ensinou a ter ousadia e a não temer obstáculos. O segundo tipo me ensinou a ter humildade e cautela. O primeiro tipo me fez um otimista, mas o segundo tipo me fez pessimista. Quando usar as qualidades do primeiro tipo, e quando usar as qualidades do segundo tipo é uma questão que exige discernimento e paciência. Não existem respostas prontas nesse dilema.

Se você sonha em ter o seu próprio negócio e a sua própria equipe de desenvolvedores, ou se você sonha em trabalhar em uma grande empresa de desenvolvimento, não perca tempo planejando demais. Esse tipo de coisa não se aprende na faculdade nem com a leitura de livros. A capacitação contínua é um aspecto importante, mas somente ela não fará de você um profissional de primeira. Seja ousado e diga sim, eu posso. Mas não caia ante a sua própria soberba e nem dê aos problemas uma dimensão maior do que eles tem na realidade.

As seções a seguir contém informações sobre alguns assuntos que você deve estudar caso deseje continuar com o estudo do Harbour (ou qualquer linguagem de programação).

17.2 Dicas para um programa de estudo

17.2.1 Programação orientada a objeto

O que nós vimos foi apenas a ponta do iceberg de programação orientada a objeto. O Harbour possui suporte para outras características desse paradigma, como : métodos privados, métodos protegidos, herança múltipla, etc.

17.2.2 Acesso a base de dados SQL

Eleja um banco de dados para estudar: MySQL (MariaDb), PostgreSQL, Oracle, Firebird (Interbase), Sybase, MS Access, SQLite, etc. e passe a estudar as formas de conexão que o Harbour dispõe para manusear o seu banco de dados.

17.2.3 Programação GUI

O Harbour possui várias bibliotecas de terceiros que trabalham com componentes gráficos: MiniGui, HwGui, HbQT, MiniGui extended, etc. Eleja uma delas para trabalhar.

17.3 Conecte-se a um grupo de estudos

Procure se cadastrar em fóruns e listas de discussões. Abaixo temos os dois principais fóruns para o usuário brasileiro :

1. PC Toledo : O principal fórum em português sobre Harbour, Clipper e tecnologia xBase. O pessoal é bastante atencioso e lá tem inúmeras soluções interessantes. O endereço é `http://www.pctoledo.com.br/forum`.
2. Harbour users : O principal fórum em inglês sobre Harbour. O endereço é `http://https://groups.google.com/d/forum/harbour-users`.

Importante: antes de perguntar algo, procure ver se o assunto já não foi respondido. Faça uma pequena pesquisa dentro dos assuntos discutidos dentro do fórum, isso evitará posts repetidos.

17.4 Conclusão

Esse livro ainda não está terminado (hoje é 13 de fevereiro de 2017), mas estou satisfeito em conseguir concluir a primeira versão dele. Quando eu iniciei a sua escrita, em agosto de 2016, eu não imaginava que ele ficaria tão extenso e nem que demoraria tanto para concluir a sua versão inicial. O código fonte desse livro está disponível em `http://www.pctoledo.com.br/forum/download/file.php?id=4442`.

Parte II

Tópicos específicos do Harbour

18 O pré-processador do Harbour

Você não pode ter sinal maior de orgulho crônico que no momento em que se julga bastante humilde.

Willian Law - Cit. por C.S. Lewis

Objetivos do capítulo

- Entender o que é o pré-processador.
- Entender as regras de tradução para criação de comandos
- Saber a diferença entre #command e #translate
- Saber a diferença entre #command e #xcommand, e entre #translate e #xtranslate
- Aprender a criar seus próprios comandos e pseudo-funções.

18.1 Introdução

Nós já estudamos as fases de geração de um programa, vamos rever rapidamente essas fases logo a seguir:

1. Pré-processamento
2. Compilação
3. Linkedição
4. Geração do executável

Vimos também que a fase de pré-processamento é a primeira fase de todas, antes mesmo da compilação. Durante essa fase o Harbour irá realizar a substituição de alguns símbolos especiais e os converterá em funções e expressões literais. Aplicamos alguns exemplos quando estudamos as constantes manifestas, no capítulo 4, através das diretivas #define e #include. Ocorre que, essas não são as únicas diretivas de pré-processamento que existem. Na realidade o Harbour possui algumas outras diretivas bem mais poderosas que uma simples substituição literal feita por #define. O presente capítulo irá estudar detalhadamente esses símbolos, que são conhecidos como “diretivas do pré-processador”.

18.2 Construindo a sua própria linguagem

Quando nós utilizamos qualquer linguagem de programação nós manipulamos variáveis, seus respectivos operadores e expressões diversas. Essa manipulação se dá através das intruções da linguagem. O Harbour possui as seguintes intruções :

```
                              ⎧ Funções
Instruções do Harbour         ⎨ Comandos
                              ⎩ Declarações
```

Já vimos, no capítulo referente as funções, que uma função não faz parte de uma linguagem de programação. Na verdade, toda linguagem é criada primeiro e as funções em um momento posterior. A principal consequência prática disso é que nós podemos construir as nossas próprias funções.

Agora, temos mais uma novidade: os comandos da linguagem Harbour também foram construídos em um momento posterior a criação da linguagem. Na verdade, os comandos do Harbour são funções disfarçadas. O disfarce ocorre na fase de pré-processamento através de algumas diretivas que veremos nesse capítulo. Como já era de se esperar, a consequência prática disso é que nós podemos construir os nossos próprios comandos. O comando SET DATE BRITISH, por exemplo, equivale a função _DFSET(). Isso quer dizer que, quando você está usando o comando SET DATE BRITISH você, na realidade, está usando essa função. A seguir temos o fragmento que “cria” o comando SET DATE BRITISH

```
#command SET DATE [TO] BRITISH  => _DFSET( "dd/mm/yyyy", "dd/mm/yy" )
```

Esse código está em um arquivo chamado “std.ch” que fica na pasta *include* do Harbour. Esse arquivo é incluído automaticamente na fase de pré-processamento para “traduzir” os comandos do Harbour em funções.

Por que, então, adotar os comandos ? Não seria melhor explicar somente as funções ? Os motivos que justificam o uso de comando são basicamente três :

1. A antiga linguagem dBase e as primeiras versões do Clipper possuíam comandos, não apenas funções. As últimas versões do Clipper, e agora o Harbour, nasceram com esse compromisso de compatibilidade com esses códigos antigos. Esse é um motivo puramente histórico.
2. Um comando é mais claro do que uma função em muitas ocasiões. Por exemplo, é bem melhor decorar “SET DATE BRITISH” do que decorar a função *_DFSET("dd/mm/yyyy", "dd/mm/yy")*.
3. Quando você esquece um parâmetro obrigatório em uma função isso irá gerar um erro de execução. Por outro lado, se você esquecer esse mesmo parâmetro em um comando, isso gerará um erro de compilação. Lembre-se que erros de compilação são “melhores” do que os famigerados erros de execução.

Durante esse capítulo nós estudaremos mais um pouco sobre a diretiva #define e #include e também veremos como criar os nossos próprios comandos com as diretivas #command, #xcommand, #translate e #xtranslate.

18.3 A diretiva #define

A diretiva #define já foi abordada anteriormente no capítulo sobre constantes e variáveis. Ela foi inspirada na diretiva #define usada na linguagem C, cujo funcionamento é exatamente o mesmo. Até agora nós sabemos que essa diretiva é usada na criação de constantes manifestas. Sabemos também que ela é sensível a maiúscula e minúscula. Nas próximas subseções nós iremos revisar esse tópico e acrescentar outros sobre a diretiva #define.

18.3.1 #define como um identificador sem texto de substituição

O uso mais básico da diretiva #define é como um simples identificador, conforme o fragmento abaixo:

```
#define VERSAO_DE_TESTE
```

Ele é usado quando houver a necessidade de testar a existência de um identificador com as diretivas #ifdef ou #ifndef (que veremos logo a seguir). Isso é útil para a compilação condicional, ou seja, para a inclusão ou exclusão de código durante a compilação. Iremos abordar alguns exemplos nas próximas subseções imediatamente a seguir.

#ifdef

A diretiva #ifdef compila uma seção de código se um identificador estiver definido. Dessa forma, o desenvolvedor pode ter várias versões do mesmo sistema sem precisar manter os códigos separados. Por exemplo: você pode ter uma versão de produção,

uma versão exclusiva para testes e uma versão de demonstração em apenas um fonte. O fragmento de código a seguir demonstra um esqueleto geral para a compilação com #ifdef.

```
#define DEMO

.
.  Declarações diversas : rotinas, etc.
.

#ifdef DEMO
     ? "Essa é uma versão de demonstração."
     // Comandos e instruções exclusivos de uma versão demo.
#endif
```

A sintaxe dessa diretiva também contempla um #else, cuja função é similar a sintaxe das estruturas de decisões já vistas. A listagem 18.1 ilustra o uso do uso de #ifdef com o #else.

Listagem 18.1: Macros

```
#define TESTE
PROCEDURE Main

   #ifdef TESTE
      ? "Essa é uma versão de testes"
   #else
      ? "Essa é uma versão de produção"
   #endif

RETURN
```

Se você simplesmente gerar o programa e executar você obterá.

.:Resultado:.

```
Essa é uma versão de testes
```

Agora, se eu retirar a definição de TESTE e compilar, conforme a listagem 18.2 eu irei obter a versão de produção.

Listagem 18.2: Macros

```
PROCEDURE Main

   #ifdef TESTE
      ? "Essa é uma versão de testes"
   #else
      ? "Essa é uma versão de produção"
```

```
   #endif                                                                    8
                                                                             9
RETURN                                                                       10
```

O resultado é o seguinte:

.:Resultado:.

```
Essa é uma versão de produção
```

Agora, uma novidade. Você pode definir um identificador sem usar necessariamente a diretiva #define. Para fazer isso você tem que gerar o programa (através do hbmk2) com o símbolo -d seguido do nome do identificador que você deseja criar. No nosso exemplo o nome do identificador é TESTE. Portanto, você teria que fazer assim :

.:Resultado:.

```
    hbmk2 seuprograma -dTESTE
```

O resultado da execução do código 18.2 agora é :

.:Resultado:.

```
Essa é uma versão de testes
```

Dica 104

Procure automatizar a criação de versões do seu sistema. Crie, pelo menos, duas versões: versão de testes e a versão de produção. O uso da diretiva -d para a definição do identificador é mais prático do que a diretiva #define.

Dica 105

Vimos, nos capítulos iniciais, que um comentário de múltiplas linhas não pode aninhar outro comentário de múltiplas linhas. Portanto, o código a seguir está errado:

```
/*

   IF lAchou
       lValida := ProcessaNotas()
   ENDIF

   /*
       Se a nota tiver um ítem inválido então sai do sistema
   */
   IF .NOT. lValida
       QUIT
   ENDIF

*/
```

Ocorre, que existem ocasiões em que você deseja comentar um extenso trecho de um código. Isso acontece porque você pode suspeitar que esse trecho contém um erro. Nada mais prático do que comentar esse trecho problemático e recompilar de novo o programa. Mas, e se esse trecho problemático contiver comentários de múltiplas linhas ? Nesse caso você pode esquecer esse comentário e recorrer a uma diretiva para eliminar esse trecho. Acompanhe o mesmo trecho acima com essa diretiva, logo a seguir:

```
#ifdef SIMBOLO_INDEFINIDO

   IF lAchou
       lValida := ProcessaNotas()
   ENDIF

   /*
       Se a nota tiver um ítem inválido então sai do sistema
   */
   IF .NOT. lValida
       QUIT
   ENDIF
#endif
```

O SIMBOLO_INDEFINIDO realmente não deve estar definido mesmo. Somente assim o trecho #ifdef ... #endif não será compilado.

#ifndef

A diretiva #ifndef é o oposto da diretiva #ifdef. Enquanto a diretiva #ifdef testa se o símbolo existe, a diretiva #ifndef testa se o símbolo **não existe**. Essa diretiva também possui o opcional #else.

#undef

A diretiva #undef remove uma macro ou constante manifesta definido por #define ou pela diretiva de compilação -d.

18.3.2 #define como constante manifesta

As constantes manifestas já foram abordadas na primeira parte do capítulo 4, que trata do assunto das constantes.

18.3.3 #define como pseudo-função

Uma pseudo-função[1] é um identificador seguido de abre parênteses, e seguindo, sem espaços, por uma lista de argumentos e de um fecha parênteses e da expressão de substituição [Nantucket 1990, p. 3-12]. Os fragmentos a seguir nos mostram alguns exemplos:

```
#define AREA(nTam, nLarg)      (nTam * nLarg)
#define SETVAR( x, y )     (x := y)
```

.:Resultado:.

```
? AREA( 2, 4 )  // Exibe 8
SETVAR( x, 12 ) // Atribui o valor 12 a variável x
SETVAR( y, AREA( 3, 3 ) ) // Atribui o valor 9 a variável y
```

Quando a expressão de substituição é trocada pela pseudo-função, os nomes correspondentes na expressão são expandidos como texto do argumento. É importante ressaltar que **todos** os argumentos devem ser especificados. Se não o forem, a chamada de função não é expandida e a saída fica como foi encontrado [Nantucket 1990, p. 3-12].

[1] Pseudo-funções tiveram a sua origem na Linguagem C, lá elas recebem o nome de “macros”. Aqui os desenvolvedores do antigo Clipper optaram pelo termo “pseudo-função” para não causar confusão com o operador de macro &.

Dica 106

Se uma pseudo-função não é uma função de verdade qual a sua abrangência ?

Como uma pseudo-função é definida através da diretiva #define, a sua abrangência é a mesma dos outros símbolos. Ela passa a valer a partir da linha em que foi definida e é visível até o final do código fonte. Para facilitar você deve criar o hábito de definir as suas pseudo-funções logo nas primeiras linhas do seu programa. Se fizer isso basta considerar a sua abrangência igual a de uma variável estática externa (que é o próprio PRG).

A vantagem das pseudo-funções

Pseudo-funções não possuem a perda de tempo da chamada de uma rotina. Todas as funções, ao serem chamadas, pressupõe um desvio na sequência de execução do programa. Essa sequência novamente é quebrada quando o programa encontra uma instrução RETURN. As Pseudo-funções não possuem essa perda de tempo pois a quebra na sequência não existe, já que a função não existe. Isso causa ganhos na velocidade de execução do programa.

Dica 107

É importante incluir a expressão resultante de uma pseudo-função entre parênteses para forçar a ordem correta de avaliação.

As desvantagem das pseudo-funções

Pseudo-funções possuem várias desvantagens que podem fazer você considerar se vale mesmo a pena o seu uso. Eis a lista de desvantagens:

1. São difíceis de depurar dentro do Debugger.
2. Não permitem saltar argumentos.
3. Tem uma abrangência diferente de funções e procedures declarados.
4. São sensíveis a maiúsculas e minúsculas.

Kernighan e Pike acrescentam que o uso de pseudo-funções só se justifica através do argumento de que elas trazem ganhos de desempenho. Esse argumento, contudo, é fraco já que com as máquinas e compiladores modernos isso é irrelevante [Kernighan e Pike 2000, p. 19].

Dica 108

Uma precaução extra é necessária quando se vai usar uma pseudo-função. Evite incrementar uma variável quando ela for um argumento. O exemplo a seguir ilustra esse problema.

Listagem 18.3: Macros

```
1
2 #define POTENCIA_2( b )  ( b * b )
3 PROCEDURE Main
4 LOCAL x := 1
5
6     ? POTENCIA_2( ++x ) // 2 elevado a 2 é 4
7
8 RETURN
```

.:Resultado:.

```
          6
```

O que, obviamente, está errado. Isso aconteceu porque a pseudo-função foi trocada pela expressão que a definiu (na linha 2). Acompanhe a substituição no fragmento abaixo:

```
LOCAL x := 1
? ( ++x * ++x ) // 2 * 3
```

18.4 As diretivas #command e #translate

De acordo com o desenvolvedor do Clipper,

> #command e #translate são diretivas de tradução para definir comandos e pseudo-funções. Cada diretiva especifica uma regra de tradução. A regra consiste de duas partes: um modelo de entrada e um modelo resultante. [Nantucket 1990, p. 3-1].

A diferença ente #command e #translate é que a diretiva #command considera equivalente somente se o texto de entrada for uma declaração completa, ao passo que uma diretiva #translate considera equivalente um texto que não é uma declaração completa.

Dica 109

Prefira usar #command mais para definições gerais e #translate para os casos especiais.

A abrangência de #(x)command e #(x)translate é a mesma de #define. A definição #undef não se aplica a essas definições, sendo restrita somente a #define.

Essas diretivas (#command e #translate) são bem mais complexas do que a diretiva #define. Elas possuem várias normas que controlam a forma com que a substituição é feita e também não são sensíveis a maiúsculas/minúsculas. De fato, como você já deve saber, o comando CLS pode ser escrito assim : cls, Cls, cLS, cLs ou CLs. Nós optamos por escrever os comandos com letras maiúsculas porque isso torna os programas mais fáceis de se ler. Nas próximas subseções iremos estudar essas normas uma a uma. As normas de substituição são as mesmas, tanto para #command quanto para #translate, por isso elas serão tratadas ao mesmo tempo.

Dica 110

Só para ilustrar a diferença, caso fossemos definir uma pseudo-função, de modo a abranger os vários tipos de variações na escrita, deveríamos fazer assim :

```
#DEFINE Max( a , b ) IIF( a > b , a , b )
#DEFINE MAx( a , b ) IIF( a > b , a , b )
#DEFINE MAX( a , b ) IIF( a > b , a , b )
#DEFINE mAX( a , b ) IIF( a > b , a , b )
#DEFINE maX( a , b ) IIF( a > b , a , b )
#DEFINE max( a , b ) IIF( a > b , a , b )
#DEFINE MaX( a , b ) IIF( a > b , a , b )
#DEFINE mAx( a , b ) IIF( a > b , a , b )
```

Com o #translate bastaria apenas

```
#tranaslate Max( <a> , <b> ) => IIF( <a> > <b> , <a> , <b> )
```

Assim, podemos digitar Max(), MAX(), max(), etc.
Fonte : http://www.blacktdn.com.br/2010/09/protheus-diretivas-do-pre-processador.html#ixzz4XqC48b1p

18.4.1 Marcadores de equivalência

Palavras

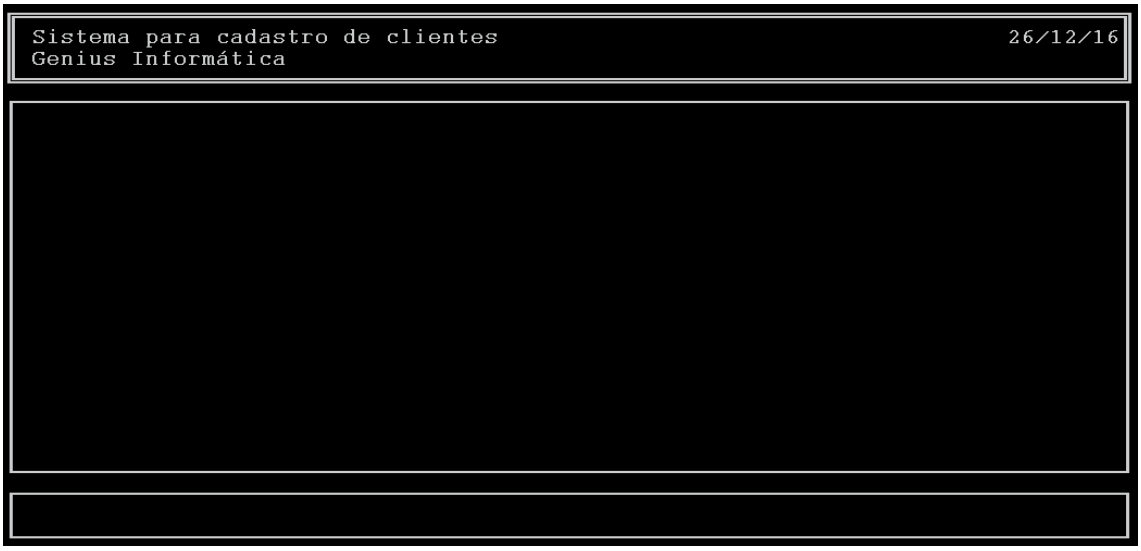
Realiza uma substituição simples de acordo com o padrão do antigo dBase, ou seja, sensível a maiúsculas e minusculas, e somente os quatro primeiros caracteres precisam coincidir.

Listagem 18.4: Macros

```
#command MOSTRA A TELA INICIAL => MostraTela()
PROCEDURE Main

    MOSTRA A TELA INICIAL

RETURN

PROCEDURE MostraTela()

   SET DATE BRITISH
```

```
   CLS                                                                  12
   @ 0,0 TO 3,79 DOUBLE                                                 13
   @ 1,2 SAY "Sistema para cadastro de clientes"                        14
   @ 1,71 SAY DATE()                                                    15
   @ 2,2 SAY "Genius Informática"                                       16
                                                                        17
   @ 4,0 TO 21,79                                                       18
   @ 22,0 TO 24,79                                                      19
   ? ""                                                                 20
                                                                        21
                                                                        22
RETURN                                                                  23
                                                                        24
```

Figura 18.1: Tela inicial em forma de comando



O comando CLS é outro exemplo de substituição simples de palavras. Ele está definido, em std.ch[2] e a definição nós transcrevemos a seguir :

```
#command CLS                          => Scroll() ; SetPos( 0, 0 )
```

Veja que o comando CLS é composto de uma sequência de duas funções separadas por um ponto e vírgula, que indica uma quebra de linha.

Outro fato interessante é que nós podemos criar um comando a partir de outro pré-definido no arquivo. Por exemplo, logo após a definição de CLS nós temos a definição de CLEAR SCREEN (velho conhecido do dBase).

```
#command CLEAR SCREEN                 => CLS
```

Em resumo, as diretivas #command, #xcommand, #translate e #xtranslate obedecem a mesma regra sintática, conforme o quadro abaixo :

[2]Já vimos, mas é bom repetir, que todo comando é definido em std.ch e é incluído automaticamente durante a compilação do programa.

```
#command <padrão de entrada> => <padrão de saída>
#xcommand <padrão de entrada> => <padrão de saída>
#translate  <padrão de entrada> => <padrão de saída>
#xtranslate  <padrão de entrada> => <padrão de saída>
```

O símbolo =>, posicionado entre o padrão de entrada e o padrão de saída faz parte da sintaxe. Esse símbolo é composto de um sinal de igual seguido por um sinal de maior, sem espaços em branco entre si.

Marcadores-padrão

Vamos agora passar para um modelo um pouco mais complexo de comando, que são os comandos que possuem valores passados pelo usuário. Por exemplo:

```
KEYBOARD " "
```

A passagem de valores informados pelo usuário é feito pelo marcador-padrão de equivalência. Para criar um marcador assim basta **colocar uma variável entre os símbolos <>**. Os comandos acima são representados da seguinte forma:

```
#command KEYBOARD <x>                    => __Keyboard( <x>  )
```

O comando GOTO também foi definido com o marcador padrão. Lembre-se que ele pode se escrito assim também :

```
GO 12
```

Note que a sintaxe padrão de comandos dBase aceita apenas os quatro caracteres significativos, como então o comando GOTO foi escrito com apenas dois ? Isso aconteceu porque ele está definido assim no std.ch :

```
#command GOTO <x>                 => dbGoto( <x> )
#command GO <x>                   => dbGoto( <x> )
```

Cláusulas opcionais

Você com certeza já deve ter notado que alguns comandos exigem cláusulas opcionais. Por exemplo, o comando @ ... PROMPT pode ter uma cláusula de mensagem de rodapé ou não:

```
  @ 10,10 PROMPT "Cadastro"
  @ 12,10 PROMPT "Relatório" MESSAGE "Relatórios gerenciais do sistema"
```

Para representar cláusulas opcionais envolva o conteúdo com colchetes. No arquivo std.ch o comando @ ... PROMPT está representado assim:

```
#command @ <row>, <col> PROMPT <prompt> [MESSAGE <msg>];
  =>;
  _AtPrompt( <row>, <col>, <prompt>, <msg> )
```

Marcadores de expressões extendidas

Quando nós estudamos as macros, vimos que um efeito semelhante pode ser conseguido envolvendo os argumentos de comandos entre parênteses. Dessa forma :

```
USE bancos
```

tem o mesmo efeito que

```
c := "bancos"
USE &c
```

que, por sua vez, tem o mesmo efeito que

```
USE ( "bancos" ) // os parênteses funcionam como se fossem macros
```

Os marcadores estendidos são amplamente usados, porque permitem uma flexibilidade maior que apenas seria conseguido com as macros. Veremos o porquê disso nas próximas subseções.

A seguir temos outro exemplo do marcador estendido com o comando ERASE (responsável por apagar um arquivo). A definição desse comando está logo a seguir

```
#command ERASE <(f)>                    => FErase( <(f)> )
```

Dessa forma podemos escrever :

```
ERASE log.txt
```

ou

```
ERASE  ("log.txt")
```

Note que o formato desse marcador requer parênteses no seu interior:

```
 <(marcador)>
```

Marcador de listas

Existem alguns comandos cujos argumentos podem variar em número, por exemplo, o nosso velho conhecido comando “?” :

```
? a , b , c , 1234
```

Para traduzir uma lista de argumentos usamos o marcador de equivalência de listas, conforme abaixo:

```
#command ?  [<explist,...>]          => QOut( <explist> )
```

Ou seja, você deve especificar um nome seguido **por uma vírgula e três pontos**, entre os símbolos de maior e menor.

Marcador de correspondência restrito

A forma mais fácil de se entender esse tipo de marcador é estudando como o comando USE é definido. Você deve se recordar que o comando USE é usado para abrir um arquivo. Por exemplo:

```
USE paciente
```

A função que equivale a esse comando é chamada dbUseArea(). Essa função possui 6 parâmetros, sendo apenas um deles obrigatório (o nome do arquivo que você deseja abrir), conforme a definição abaixo:

```
 DBUSEAREA( [<lNewArea>], [<cDriver>], <cName>, [<xcAlias>],
       [<lShared>], [<lReadonly>]) --> NIL
```

Vamos compreender por partes. Primeiramente, se o comando USE tivesse apenas um parâmetro, então ele poderia ser definido assim:

```
#command USE <(dbf)>  => dbUseArea( NIL, NIL, <(dbf)> )
```

Mas, sabemos que esse comando possui um tipo especial de parâmetro que informa se ele deve ser aberto na área de trabalho atual ou se deverá ocupar a proxima área disponível. Por exemplo:

```
USE paciente NEW
```

Pois bem, a cláusula NEW é criada através de um marcador de correspondência restrito. Primeiramente vamos usar a função dbUseArea() com a cláusula NEW. Observe:

```
dbUseArea( .t. , NIL, "paciente" )
```

Ou seja, o primeiro parâmetro da função dbUseArea() é um valor lógico que, se for verdadeiro, irá abrir o arquivo na próxima área disponível (o que equivale a cláusula NEW do comando USE).

O comando USE, somente na forma acima, poderia ser definido assim:

```
#comand USE <(dbf)> [<nw: NEW>] =>;
    <dbUseArea( <.nw.>, NIL , <(dbf)> )
```

O marcador de correspondência funciona assim: se o programador digitou “NEW”, então a variável “nw” irá receber o valor lógico verdadeiro, caso contrário (se o programador omitiu a cláusula), essa variável (“nw”) irá receber o valor falso. Note, no nosso exemplo acima, que a variável nw foi atrelada a cláusula NEW através da correspondência [<nw: NEW>] e que ela foi usada no lado direito dessa forma <.nw.>.

De acordo com Spence,

> Um marcador de correspondência restrito é usado numa cláusula opcional para corresponder a uma entrada. Se a correspondência for bem sucedida, uma variável lógica é estabelecida, a qual você usa no lado direito [Spence 1991, p. 116].

Spence complementa que se a correspondência NEW for encontrada, a variável “nw” é verdadeira, caso contrário “nw” é falsa [Spence 1991, p. 116].

Vamos para um segundo exemplo dessa cláusula, dessa vez definindo o comando SET MESSAGE, que diz em que linha a mensagem do comando @ ... PROMPT deverá ser exibida :

```
#command SET MESSAGE TO <n> [<cent: CENTER, CENTRE>] => ;
            Set( _SET_MESSAGE, <n> ) ; Set( _SET_MCENTER, <.cent.>
```

Note que a variável lógica “cent” é verdadeira caso o programador digite CENTER ou CENTRE. Portanto, podemos criar uma cláusula opcional que pode ser escrita de diferentes formas.

É bom ressaltar que, apesar da clausula ser definida com letras maiúsculas, o programador não é obrigado a digitar com letras maiúsculas. Os seguintes formatos estão corretos:

```
use paciente new
set message to 23 center
```

Marcador coringa

Existem comandos que eram do dBase III e foram extintos nas versões posteriores do Clipper e do Harbour. Como, então, garantir a compatibilidade entre os dois códigos ? Por exemplo: no dBase existia um comando chamado SET TALK. As linguagens posteriores não tem mais esse comando pois ele é restrito ao ambiente interpretativo do dBase. Foi para casos assim que o marcador coringa foi criado. Ele simplesmente corresponde a qualquer texto do ponto corrente até o final de uma declaração. Por exemplo:

```
#command SET TALK <*x*> =>
```

Dessa forma, nós criamos um comando que não serve para nada. Qualquer coisa pode ser colocado no lugar de <*x*>. O código a seguir funciona perfeitamente no Harbour, mas estaria errado se fosse interpretado pelo interpretador dBase:

```
PROCEDURE Main

SET TALK QUALQUER COISA

? "OI"

RETURN
```

O marcador coringa também é ideal para se criar comentários. Você se lembra que nós dissemos, no capítulo sobre loops, que podemos escrever comentários após o final de um bloco ? A seguir temos as definições retiradas de std.ch :

```
#command ENDDO <*x*>        => enddo
#command ENDIF <*x*>        => endif
#command ENDCASE <*x*>      => endcase
```

Isso explica o comentário abaixo (linha 11):

```
/*

*/
PROCEDURE Main
LOCAL x

   INPUT "Informe o valor : " TO x
   DO CASE
      CASE x > 0 .AND. x < 10
          ? "Valor inválido"
   ENDCASE ESSE É UM TIPO DE COMENTÁRIO QUE NÓS DESACONSELHAMOS


RETURN
```

Dica 111

Nós dissemos que o marcador coringa é ideal para se criar comentários, mas não dissemos que essa é a sua única função. Fique atento pois ele voltará na próxima seção, quando formos estudar o marcador de resultado caractere simples (através do comando RUN).

18.4.2 Marcador de resultado

Agora que nós encerramos o "lado esquerdo" (equivalência) iremos passar para o lado direito (os resultados). Algo que, particularmente, sempre me confundia era o fato dos autores abordarem os marcadores de equivalência e os marcadores de resultado separadamente. Para mim cada marcador de equivalência tinha o seu respectivo marcador de resultado, mas veremos que essa regra nem sempre é verdadeira, de modo que faz todo sentido abordar os marcadores de resultado separadamente.

Marcador caractere simples

Esse marcado simplesmente converte a entrada em uma cadeia de caracteres. Por exemplo, o comando RUN serve para executar um comando do sistema operacional:

```
PROCEDURE Main

   RUN "DIR C:" // Lista os arquivos

RETURN
```

Note que não foi preciso colocar os parênteses para expandir a string, como foi feito com o comando USE. A definição do comando está logo a seguir :

```
#command RUN <*cmd*>    => __Run( #<cmd> )
```

Note que o **marcador de resultado** é definido com um *hashtag* [3] :

[3] Não diga que é "jogo-da-velha" pois você vai entregar a sua idade!

```
#<cmd>
```

Outro fato interessante é que o nome da variável não depende do marcador que a antecede (ou que a envolve). A variável que foi usada chama-se *cmd* e ela aparece assim no lado esquerdo :

```
<*cmd*>
```

e assim no lado direito :

```
#<cmd>
```

Dica 112

É importante ressaltar que um comando pode ser definido mais de uma vez. O comando RUN, por exemplo, também está definido de outra forma no arquivo std.ch, conforme vemos no fragmento a seguir :

```
#command RUN ( <cmd> )   => __Run( <cmd> )
```

Essa definição torna a sintaxe do comando RUN semelhante a do comando USE, assim nós também podemos fazer :

```
RUN DIR C:  // sem as aspas
```

ou

```
RUN ("DIR C:")  // com as aspas mais parênteses para expandir o conte
```

Marcador caractere condicional

Quando estudávamos o lado esquerdo (as entradas) nós vimos um marcador chamado “marcador de caractere extendido”. Esse marcador foi usado para permitir a expansão de expressões entre parênteses, conforme os exemplos a seguir :

```
RUN ("DIR C:")  // com as aspas mais parênteses para expandir o conteúd
```

e

```
USE ("bancos")  // com as aspas mais parênteses para expandir o conteúd
```

Um marcador de caractere condicional é simplesmente o equivalente a esse marcador no lado direito da expressão. Ele funciona segundo as regras a seguir :

1. Coloca aspas no texto de entrada. Por exemplo: nomes é convertido para “nomes”.

   ```
   USE bancos
   ```

   é convertido para

   ```
   dbUSEArea( NIL, NIL,  "bancos" )
   ```

2. Se o texto de entrada vier envolto com parênteses, então não acrescenta nada. Por exemplo: ("nomes") permanece uma string : ("nomes").

   ```
   USE ("bancos")
   ```

   é convertido para

   ```
   dbUSEArea( NIL, NIL,  "bancos" )
   ```

Dica 113

Quando nós estudamos o marcador caractere condicional nós entendemos porque o parênteses envolvendo argumentos de comandos funcionam como se fossem macros. Simplesmente a função que define o comando não aceita literais, apenas strings. Observe o exemplo abaixo :

```
USE ("bancos")
```

Serve para informar que a expressão que está dentro dos parênteses deverá ser passada sem alterações para a função.

```
dbUSEArea( NIL, NIL,  "bancos" )
```

É bem diferente de uma macro, e mais aconselhável, pois é resolvido em tempo de compilação. Spence acrescenta que é assim que as expressões estendidas são implementadas. Certos comandos que esperam uma literal aceitarão expressões colocadas entre parênteses [Spence 1991, p. 119].

Marcador tipo lógico

Marcadores do tipo lógico convertem entradas em uma constante lógica. Nós já vimos esse marcador quando estudávamos o marcador de correspondência restrito.

```
#comand USE <(dbf)> [<nw: NEW>] =>;
    <dbUseArea( <.nw.>, NIL , <(dbf)> )
```

O marcador do tipo lógico é circundado por pontos e é colocado entre sinais de menor que e maior que. Esse marcador geralmente é usado em conjunto com o marcador de correspondência restrito.

Dica 114

Use o marcador tipo lógico quando sua string de resultado precisar saber se apareceu uma cláusula opcional no arquivo fonte.

Marcador tipo bloco

O marcador do tipo bloco converte uma entrada em um bloco de código. Por exemplo, considere o comando LOCATE.

```
#command LOCATE [FOR <for>] [WHILE <while>] [NEXT <next>] ;
                [RECORD <rec>] [<rest:REST>] [ALL] => ;
         __dbLocate( <{for}>, <{while}>, <next>, <rec>, <.rest.> )
```

As cláusulas FOR e WHILE serão transformadas em blocos no fluxo da saída. Portanto, quando você escrever :

```
LOCATE FOR user = "Microvolution" WHILE forum =  "PCToledo"
```

isso será traduzido para

```
__dbLocate( {|| user = "Microvolution" }, {|| forum =  "PCToledo" } ,,,
```

Note que a cláusulá REST não foi passada, por isso o marcador opcional de correspondência restrito <rest:REST> gerou um .F. no último parâmetro da função [Spence 1994, p. 91].

Marcador-padrão de caractere

Cláusulas repetitivas

Vários comandos permitem-lhe repetir cláusulas de palavras-chaves. Por exemplo, podemos escrever:

```
REPLACE valor WITH 10, pedido WITH 200, valorbruto WITH 300
```

Essa repetição de cláusulas pode ser conseguida colocando a cláusula a ser repetida, com a vírgula, entre colchetes ([]).

```
#command REPLACE <f1> WITH <v1>[, <fN> WITH <vN>] => ;
         _FIELD-><f1> := <v1> [; _FIELD-><fN> := <vN>]
```

Note que, nem sempre um comando é definido através de funções. Existem casos, como esse do REPLACE, onde um comando é uma expressão formada por diversos operadores.

Vamos apresentar outro exemplo através do comando SET RELATION. Ele possui esse formato:

```
SET RELATION TO NumFat INTO Faturas,;
       NumPeca INTO Pecas,;
       NumForn INTO Fornecedores
```

Veja, a seguir, que ele é definido através de funções:

```
#command SET RELATION [<add:ADDITIVE>] ;
               [TO  <exp1> INTO <(alias1)> [<scp1:SCOPED>];
            [, [TO] <expN> INTO <(aliasN)> [<scpN:SCOPED>]]] => ;
         if ( ! <.add.> ) ; dbClearRelation() ; end ;
           ; dbSetRelation( <(alias1)>, <{exp1}>, <"exp1">, <.scp1.> )
         [ ; dbSetRelation( <(aliasN)>, <{expN}>, <"expN">, <.scpN.> )
```

Não precisa entender a definição toda, apenas atente para as duas últimas linhas. Note que a última linha define uma possível repetição de cláusula.

Dica 115

Evite usar a repetição de cláusulas sempre que puder. No comando REPLACE, por exemplo, é preferível a forma a seguir :

```
REPLACE valor WITH 10
REPLACE pedido WITH 200
REPLACE valorbruto WITH 300
```

do que essa forma:

```
REPLACE valor WITH 10, pedido WITH 200, valorbruto WITH 300
```

Isso porque, quando um erro de compilação ocorre, fica mais fácil achar o erro usando a primeira forma. **Por que é mais fácil ?**
Porque, na primeira forma temos um comando por linha, e na segunda forma temos três comandos por linha. Lembre-se que, quando um erro ocorre, ele virá com o número da linha. Dessa forma, quanto menor for o número de instruções por linha melhor para você detectar o erro.

18.5 A diretiva #translate

Segundo Heimendinger,

> a diretiva #command aceita uma instrução completa, enquanto que a diretiva #translate aceita strings de código fonte que não são instruções completas. A diretiva #translate encontrará o seu uso em casos mais específicos [Heimendinger 1994, p. 120].

Observe o exemplo a seguir :

```
#translate run => qout
PROCEDURE Main

   RUN( "DIR C:" )

RETURN
```

RUN agora se comporta como a função QOut (que é a função que define o comando "?").

.:Resultado:.

```
DIR C:
```

O #translate se parece muito com #define, só que é bem mais avançado, já que o #define é sensível a maiúsculas e minúsculas. #translate também se parece muito com #command, mas para definição de comandos completos o #command é preferível, inclusive, o arquivo std.ch usa #command.

Outra diferença entre as duas diretivas é que #command restrige-se somente a comandos, enquanto #translate pode ser usado para a criação de pseudo-funções sem as limitações de maiúsculas/minúsculas de #define. Por exemplo:

```
#translate Maximo( <a> , <b> ) => IIF( <a> > <b> , <a> , <b> )
PROCEDURE Main

   ? Maximo( 2 , 3 )

RETURN
```

O mesmo código acima não funcionaria com #command, já que não foi definido um comando, mas apenas uma pseudo-função.

18.6 As diretivas #xcommand e #xtranslate

As diretivas #xcommand e #xtranslate funcionam como #command e #translate exceto que eles não obedecem ao padrão dBase que considera apenas os quatro primeiros caracteres significativos. Ou seja, eu posso criar comandos ou pseudo-funções com uma variedade maior de caracteres. Todas as outras regras se aplicam. Fonte : https://vivaclipper.wordpress.com/tag/xtranslate/.

18.7 Criando os seus próprios comandos

Até agora nós abordamos os diversos tipos de tradução tendo como base o arquivo std.ch, agora nós iremos lhe apresentar alguns comandos definidos pelo usuário. Para que as coisas fiquem mais fáceis para você, procure sempre estudar os comandos definidos no arquivo std.ch.

18.7.1 O laço REPITA ... ATE ou REPEAT ... UNTIL

Lembra do laço REPITA ... ATÉ que foi abordado no capítulo sobre loops ? Nós dissemos que esse laço não existe formalmente como uma instrução, mas que poderia ser simulado através do laço DO WHILE ... ENDDO, conforme abaixo :

```
PROCEDURE Main
LOCAL  x := -10

  DO WHILE .T.
      ++x
      IF x > 0
         EXIT
      ENDIF
  ENDDO

RETURN
```

No exemplo acima, o laço se repetirá até que x seja maior do que zero. Ou seja, “repita até x > 0”.

Esse laço pode ser definido através dos seguintes comandos :

```
#command REPEAT => DO WHILE .T.                                          1
#command UNTIL <exp> => IF ( <exp> ); EXIT; ENDIF; ENDDO                 2
PROCEDURE Main                                                           3
LOCAL x := -10                                                           4
                                                                         5
  REPEAT                                                                 6
    ++x                                                                  7
  UNTIL x > 0                                                            8
                                                                         9
RETURN                                                                   10
```

18.7.2 O arquivo common.ch

O Harbour dispõe de um arquivo chamado common.ch. Esse arquivo agrupa algumas pseudo-funções e comandos que não foram incorporados a lista de comandos originais do Harbour (em std.ch). Vamos analisar a construção do comando DEFAULT TO. Ele está definido a seguir:

```
#xcommand DEFAULT <v1> TO <x1> [, <vn> TO <xn> ] => ;
                                IF <v1> == NIL ; <v1> := <x1> ; END ;
                                [; IF <vn> == NIL ; <vn> := <xn> ; END
```

A função desse comando é atribuir um valor padrão caso um parâmetro não tenha sido definido. Note que existe uma cláusula de repetição para testar múltiplos parâmetros, dessa forma poderíamos escrever :

```
#include "common.ch"                                                    1
FUNCTION MyFunction( x, y, z )                                          2
DEFAULT x TO 1, y TO 2, z TO 3                                          3
                                                                        4
RETURN x + y + z                                                        5
```

Note que, como esse comando não foi definido em std.ch, ele teve que ser incluído com #include “common.ch”.

18.8 Como ver o resultado de um pré-processamento ?

Você pode saber o que está sendo gerado na fase de pré-processamento. Para conseguir isso basta compilar o seu arquivo com a opção -p. Um detalhe importante: se você estiver usando o arquivo hbmk.hbm coloque a opção -p nele, caso contrário faça na linha de comando da seguinte forma: **hbmk2 seuarquivo.prg -p**. Vamos compilar o seguinte arquivo :

Listagem 18.5: List
Fonte: codigos/dbf07.prg

```

PROCEDURE Main                                                          2
```

```
                                                                    3
   USE paciente                                                     4
   LIST upper(nome), nascimento, altura, peso                       5
                                                                    6
                                                                    7
RETURN                                                              8
```

.:Resultado:.

```
> hbmk2 dbf07 -p
Harbour 3.2.0dev (r1507030922)
Copyright (c) 1999-2015, http://harbour-project.org/
Compiling 'dbf07.prg' and generating preprocessed output to
   'dbf07.ppo'...
Lines 11, Functions/Procedures 1
```

Note que o arquivo dbf07.ppo foi gerado. O seu conteúdo está listado a seguir:

Listagem 18.6: List PPO
Fonte: codigos/dbf07.ppo

```
                                                                    1
PROCEDURE Main                                                      2
                                                                    3
                                                                    4
   dbUseArea( .F.,, "paciente",, iif( .F. .OR. .F., ! .F., NIL ), .F. )
                                                                    5
   __dbList( .F., { {|| upper(nome)}, {|| nascimento}, {|| altura}, {|| peso}
                                                                    6
                                                                    7
RETURN                                                              8
```

Bem mais complicado, mas é assim que o compilador receberá o arquivo. Essa opção pode ser útil caso você queira ver o que está sendo gerado quando você estiver criando os seus próprios comandos.

18.9 Os benefícios dos comandos

Os comandos possuem os seguintes benefícios :

1. a grande maioria dos erros é reportada em tempo de compilação;
2. o código é mais claro com comandos;
3. você pode imitar os comandos de uma linguagem similar. Dessa maneira, você pode usar um arquivo de código-fonte, que operará tanto em Harbour quanto no dialeto;
4. caso algum comando seja excluído no futuro, você mesmo poderá criar a versão desse comando;

5. você pode implementar comandos de uso comercial e usá-los para desenvolver aplicações. Se você está desenvolvendo aplicações financeiras poderá criar comandos como DEBIT, CREDIT, SETUP ACCOUNT ou até mesmo em português, como BAIXAR NOTA, EXCLUIR PEDIDO, etc.

Dica 116

Geralmente, quando uma função possuir muitos parâmetros, é melhor para você criar diversas versões de comandos baseados nessa mesma função.

19 Acesso a arquivos em baixo nível

> Pedi, e dar-se-vos-á; buscai e encontrareis; batei, e abrir-se-vos-á. Porque aquele que pede recebe; e o que busca encontra; e, ao que bate, se abre.
>
> Mateus 6:7-8

Objetivos do capítulo

- Obter conhecimentos básicos de Windows

19.1 Introdução

O Harbour possui suporte a acesso a arquivos e dispositivos em baixo nível. Esse capítulo é destinado ao estudo das funções e comandos que providenciam um acesso em baixo nível a arquivos diversos. Vamos começar definindo o que vem a ser um "acesso em baixo nível". Imagine as seguintes situações abaixo:

- Paulo chegou no escritório de contabilidade cedo para poder revisar a sua planilha no MS Excel.
- Rita está retocando a foto do seu aniversário usando o Photoshop.
- Clara prefere editar o seu arquivo em latex usando o TexStudio.
- João está usando o antigo Dbase III para editar seus arquivos DBFs, ele detesta o hbrun.

O que essas quatro situações tem em comum ?

A resposta: essas quatro situações nos mostram pessoas editando arquivos. A partir dessa simples constatação vamos desenvolver o nosso aprendizado, até chegarmos em conceitos mais complexos.

No nosso exemplo, cada arquivo possui um programa associado. Por exemplo, eu não vou poder trabalhar com uma planilha do MS Excel usando o Photoshop. Da mesma forma, o TexStudio não consegue trabalhar com um arquivo no formato DBF. Ou seja, embora todos os exemplos envolvam arquivos, eles não podem ser usados corretamente por qualquer programa.

Quando nós conseguimos abrir qualquer arquivo, e trabalhar com ele, independente dele ser uma planilha, um banco de dados ou uma imagem, dizemos que essa forma de acesso é "em baixo nível".

A expressão "acesso a arquivos em baixo nível"não é uma referência ao tipo de arquivo, mas a forma com a qual eles são acessados. Todas as linguagens de programação possuem formas para acessar qualquer arquivo, independente dele ser uma planilha, uma imagem ou um banco de dados. Essa forma genérica de acesso, recebe o nome de "acesso a arquivos em baixo nível"[1].

Você pode erroneamente achar que tal forma de acesso é a melhor de todas, pois ela consegue abrir qualquer arquivo, mas as coisas não são bem assim. Apesar de abrir qualquer arquivo, essas funções não conseguem identificar facilmente, dentro do arquivo aberto, uma imagem, uma célula ou um registro. Eles só conseguem "enxergar"uma "fileira gigante de bytes". Dar um sentido a esses bytes é uma tarefa que é realizada por um programa especialmente desenvolvido para trabalhar com esse arquivo. Nas próximas seções iremos aprender a usar as funções que manipulam qualquer arquivo em baixo nível.

19.1.1 O que todos os arquivos tem em comum ?

Todos os arquivos possuem características que são válidas para qualquer formato, seja ele uma planilha, uma imagem ou qualquer dado. Essas características recebem o

[1] O termo vem do original : "Low level file access". Essa expressão também é correntemente traduzida como "Acesso a arquivos **de** baixo nível", porém a tradução mais acertada seria : "Acesso a arquivos **em um** baixo nível", já que o termo "baixo nível"refere-se a forma de acesso, e não ao arquivo em si.

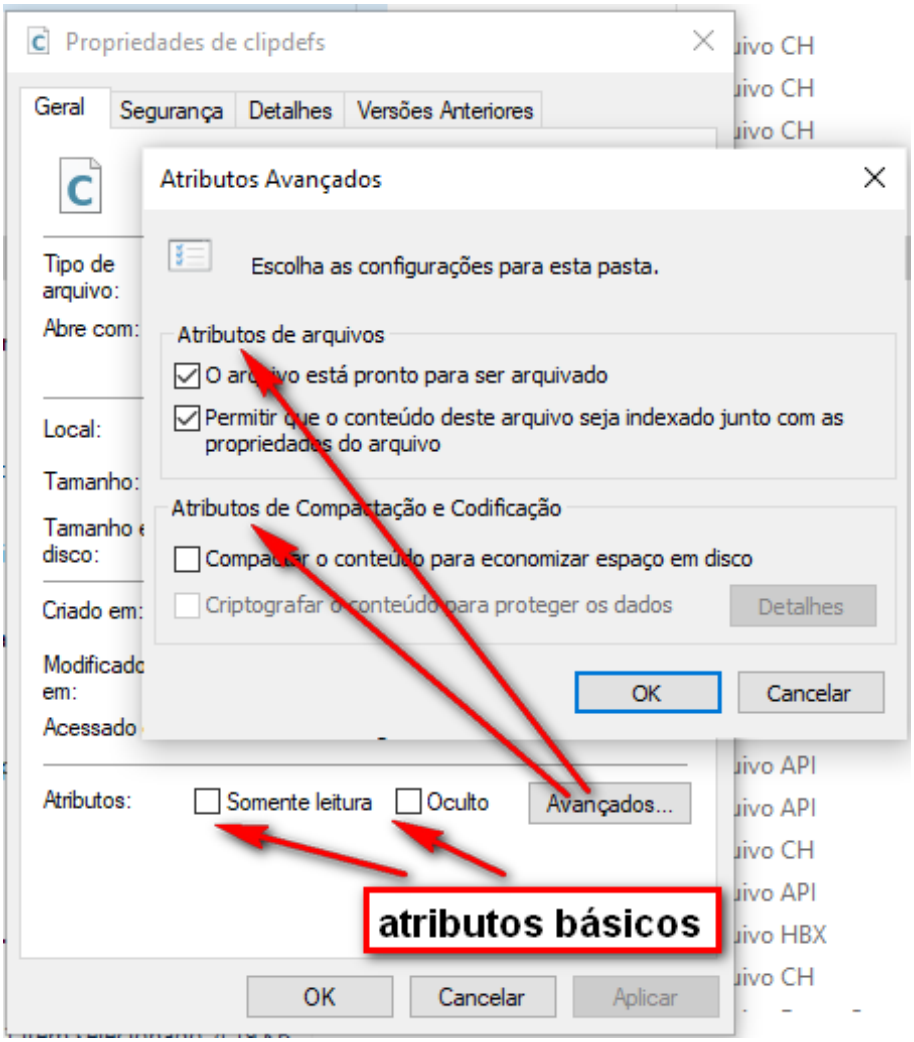
termo técnico de "atributo". Um atributo são dados que informam algo sobre o arquivo em questão. Esses dados não pertencem ao arquivo e nem ficam armazenados "dentro"dele[2]. Quem detém os dados dos atributos é o próprio sistema operacional. Alguns exemplos de atributos:

1. somente leitura
2. escrita x leitura
3. arquivo oculto
4. diretório/pasta
5. sistema

A função principal do atributo é instruir o sistema operacional a como lidar com o arquivo. É bom ressaltar que a lista acima não é fixa e pode variar de um sistema operacional para outro. Por exemplo, no Windows, um arquivo é oculto porque teve o seu atributo definido como tal, mas no Linux um arquivo é oculto, não porque tem um atributo, mas porque ele começa com um "."(ponto). Outro exemplo: o Windows considera executável todo arquivo que tem uma extensão ".exe", já o Linux tem um atributo especial que define um dado arquivo como executável.

[2]arquivos sem conteúdo (zero byte) também tem atributos, por isso tais atributos não "ficam"no arquivo.

Figura 19.1: Para ver os atributos de um arquivo no Windows clique com o botão direito sobre o nome do arquivo e selecione propriedades.



Nas subseções a seguir, veremos os atributos que podem ser manipulados pelo Harbour.

Forma de abertura

Um arquivo pode ser aberto basicamente de duas formas:

1. leitura/gravação : o arquivo pode ser lido e também pode sofrer alterações;
2. somente leitura : o arquivo só pode ser lido.

Muita gente acredita que "leitura/gravação"e "somente leitura"são as formas usadas para se abrir um arquivo. Esse conceito está certo, mas não é completo. Você pode abrir um arquivo que tem o atributo "leitura/gravação"em um modo somente leitura, dessa forma esse arquivo poderá ser aberto por mais de um usuário na rede. Mas você também pode dizer para o sistema operacional que determinado arquivo é "somente leitura", e dessa forma ele não poderá ter o seu conteúdo modificado. Nesse caso, se alguém tentar abrir um arquivo com o atributo "somente leitura"em modo de "leitura/gravação", obterá um erro do sistema operacional.

Dica 117

Uma coisa é um arquivo com o atributo "somente leitura", e outra coisa é um arquivo que foi aberto em modo "somente leitura".

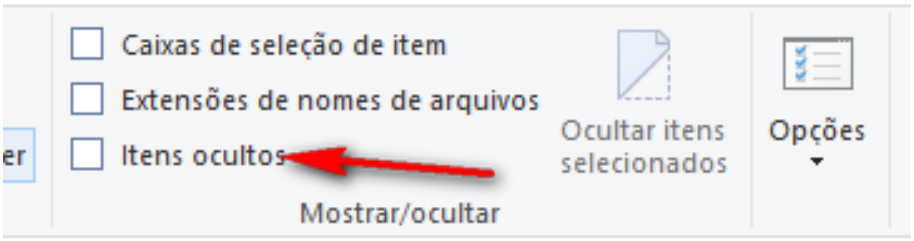
1. um arquivo com o atributo "somente leitura"só pode ser aberto em modo "somente leitura";
2. um arquivo com o atributo "leitura/gravação"pode ser aberto em modo "somente leitura"ou em modo "leitura/gravação".

Uma coisa é o atributo do arquivo, outra coisa é a forma que usamos para abrir esse arquivo. Essa seção trata somente de atributos.

Arquivos ocultos

Um arquivo oculto não pode ser visto em uma listagem normal. Para você conseguir "enxergar"esse arquivo, você precisa usar um flag especial que possibilite a sua visualização. Na figura 19.2, usamos o gerenciador de arquivos do Windows para ver os arquivos ocultos.

Figura 19.2: Marque o check box para poder ver os arquivos ocultos.



Arquivos de sistema

Um arquivo de sistema é um arquivo que faz parte do sistema operacional, geralmente quando vamos realizar uma operação sobre um arquivo com esse atributo, o sistema operacional nos dá um aviso.

Diretório

Um diretório (ou pasta) também é um arquivo como qualquer outro, mas um arquivo que tem o atributo de diretório. As funções que iremos ver não conseguem ler esse atributo, dessa forma, não poderemos criar diretórios usando as funções desse capítulo.

19.1.2 Handle

Quando você abre um arquivo usando as funções do Harbour (ou de qualquer outra linguagem), o sistema operacional devolve um identificador chamado "Handle". Use o Handle para poder referenciar o arquivo em operações futuras. Veja no exemplo abaixo:

```
// Criei um arquivo chamado novo.txt
hHandle := FCreate( "novo.txt" )
FWrite( hHandle , "Inserindo uma linha no arquivo novo.txt")
```

Iremos ver com detalhes esse e outros casos mais adiante. Por enquanto basta você saber que para realizar qualquer operação em um arquivo ele deve estar aberto e você deve saber o handle do mesmo.

19.2 Funções de acesso a arquivos de baixo nível

19.2.1 Criando arquivos

Para criar um arquivo use a função FCreate(), o funcionamento dela é simples, o primeiro parâmetro é o nome do arquivo a ser criado, e o segundo parâmetro é o atributo do arquivo criado. Se você não informar o segundo parâmetro, o arquivo criado terá os seguintes atributos :

- normal (não será de sistema)
- visível
- leitura/gravação

Listagem 19.1: Versão inicial do nosso teste de travamento.
Fonte: codigos/arquivo00.prg

```
#include "fileio.ch"                                                    1
PROCEDURE Main                                                          2
                                                                        3
    LOCAL hHandle                                                       4
                                                                        5
    ? "Criando fsociety : " , hHandle := FCreate( "fsociety" )          6
    IF hHandle == F_ERROR                                               7
        ? "Erro durante a abertura do arquivo"                          8
    ELSE                                                                9
        ? "O arquivo foi aberto. E continua aberto nesse instante."     10
        ? "Tecle algo para encerrar"                                    11
        Inkey(0)                                                        12
    ENDIF                                                               13
                                                                        14
RETURN                                                                  15
```

Algumas observações sobre a listagem :

1. o Harbour providencia um arquivo chamado "fileio.ch"com constantes simbólicas que tornam o seu código mais claro. No nosso exemplo, a função FCreate() retorna -1 se houver uma falha na criação do arquivo. Note que usamos a constante simbólica F_ERROR para tornar o código mais claro;
2. se o arquivo já existir a função FCreate() irá criar um novo e o conteúdo do anterior irá se perder.

Para conferir, faça uma pesquisa na sua pasta em busca do arquivo "fsociety". Se tudo deu certo ele está lá.

Dica 118

No Harbour, o Handle de um arquivo é um valor numérico. Vimos que variáveis numéricas recebem o prefixo "n", mas no caso específico de Handles de arquivos, use o prefixo "h". Essa é prática aconselhada, contudo você pode usar a notação que preferir para as suas variáveis.

Vamos fazer um pequeno teste

O objetivo do nosso teste é simular um caso onde o arquivo não pode ser aberto. Você lembra que o arquivo criado possui o atributo "leitura/gravação"? Isso ocorre porque não passamos o segundo parâmetro, aquele que define o atributo do arquivo. Nesse caso, o atributo padrão é "leitura/gravação". Execute o programa gerado e não saia dele.

A tela deve se parecer com :

.:Resultado:.

```
Criando fsociety :         292
O arquivo foi aberto. E continua aberto nesse instante.
Tecle algo para encerrar
```

Não saia do programa. Agora abra outro console e execute o programa novamente. Note que ele não conseguiu criar o arquivo.

.:Resultado:.

```
Criando fsociety :          -1
Erro durante a abertura do arquivo
```

Isso aconteceu porque **eu não posso criar um arquivo que está sendo usado no momento**.

Dica 119

Após uma operação de criação de arquivo, verifique o valor do Handle.

19.2.2 Criando arquivos

19.3 Conclusão

20 Funções que trabalham com arquivos em alto nível

A perfeição é alcançada quando não há mais nada para se tirar, e não quando não há mais nada para se colocar.

Antoine de St. Exupéry

Objetivos do capítulo

- Trabalhar com arquivos

20.1 Introdução

Já vimos como trabalhar com arquivos manipulando diretamente seus bytes. Aprendemos também alguns conceitos básicos sobre arquivos, que valem para qualquer sistema operacional. Nesse capítulo continuaremos a estudar os arquivos, mas dessa vez iremos trabalhar com um outro tipo de função. Essas funções realizam tarefas em um nível mais alto, de forma que elas são mais fáceis de se trabalhar.

20.2 Funções de arquivos

20.2.1 Verificando a existência de um arquivo

20.2.2 Obtendo informações sobre os arquivos

20.3 Funções auxiliares

As funções auxiliares não trabalham diretamente com arquivos, mas auxiliam na extração de informações.

20.3.1 Funções hb_FName

Listagem 20.1: Funções hbFName
Fonte: codigos/fileaux01.prg

```
PROCEDURE Main                                                  1
                                                                2
    LOCAL cFile := "/home/test.dbf"                             3
                                                                4
       ? "Path : " ,  HB_FNameDir( cFile )                      5
       ? "Extension : " , HB_FNameExt( cFile )                  6
       ? "Single name : " , HB_FNameName( cFile )               7
                                                                8
 RETURN                                                         9
```

20.4 Formatos especiais de arquivos

Todos os arquivos que serão vistos nessa seção são arquivos em formato texto, cujo conteúdo pode ser visto por qualquer editor de textos, como o Bloco de Notas do Windows. O que diferencia um do outro é a forma como o dado está estruturado. O que será visto, portanto, é apenas a forma como os dados se estruturam para serem lidos pelo Harbour. Lembrando sempre que tais arquivos são bastante conhecidos dos profissionais de desenvolvimento de todas as linguagens e tecnologias.

20.4.1 Formato de arquivos CSV

CSV é a sigla para Comma-Separated Values[1] Ou seja, tratam-se de arquivos de texto que separam valores através de vírgulas.

Uso típico de um arquivo no formato CSV

O formato CSV foi um dos primeiros a ser usado para intercambiar dados entre aplicações. Bastante usado por bancos desde os primórdios da automação nessas instituições, o CSV é bem simples. É somente um arquivo texto onde cada registro equivale a uma linha, e os campos são separados, geralmente por uma vírgula, conforme o exemplo a seguir:

```
QuotaAmount,StartDate,OwnerName,Username
150000,2016-01-01,Chris Riley,trailhead9.ub20k5i9t8ou@example.com
150000,2016-02-01,Chris Riley,trailhead9.ub20k5i9t8ou@example.com
150000,2016-03-01,Chris Riley,trailhead9.ub20k5i9t8ou@example.com
150000,2016-01-01,Harold Campbell,trailhead14.jibpbwvuy67t@example.com
150000,2016-02-01,Harold Campbell,trailhead14.jibpbwvuy67t@example.com
150000,2016-03-01,Harold Campbell,trailhead14.jibpbwvuy67t@example.com
150000,2016-01-01,Jessica Nichols,trailhead19.d1fxj2goytkp@example.com
150000,2016-02-01,Jessica Nichols,trailhead19.d1fxj2goytkp@example.com
150000,2016-03-01,Jessica Nichols,trailhead19.d1fxj2goytkp@example.com
```

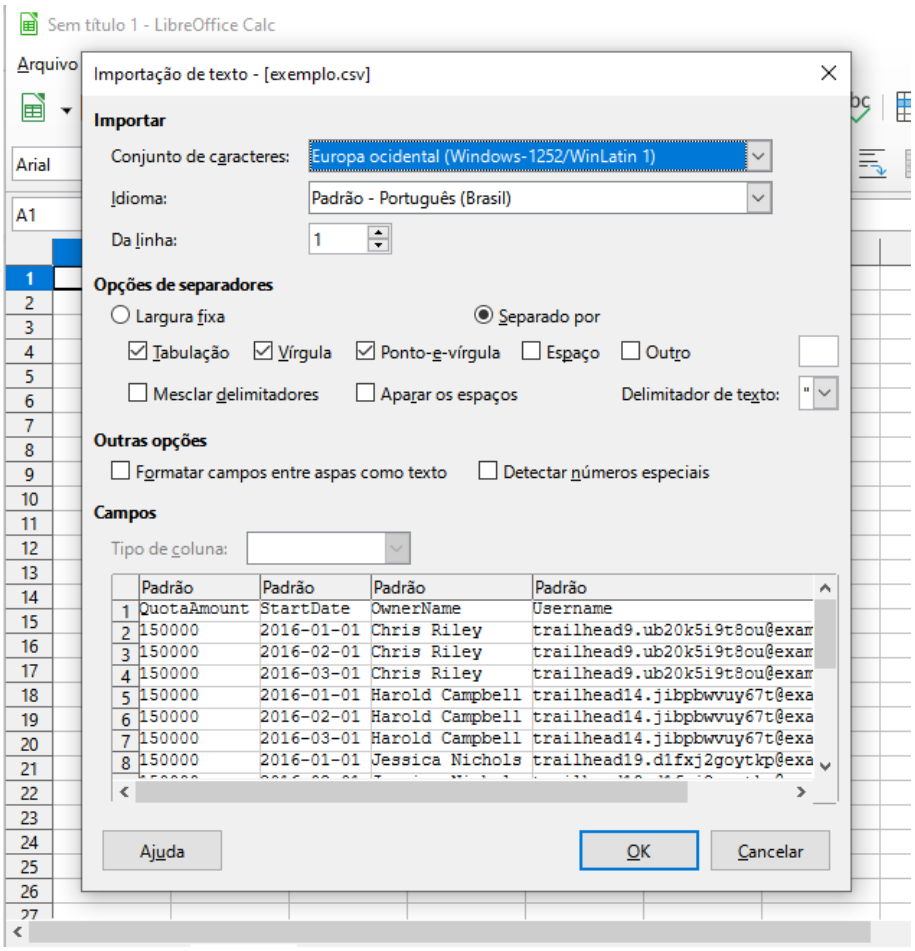
As planilhas eletrônicas são perfeitas para abrirem tais arquivos, inclusive elas possuem assistentes que facilitam a importação desse formato, conforme a figura 20.1.

Dica 120

Quando se trata de arquivos para intercambiar dados entre aplicações, a primeira coisa que você deve buscar é o manual de uso que é disponibilizado pelo orgão responsável pelos dados. Isso é particularmente verdadeiro quando trata-se de um arquivo no formato CSV, já que nós até podemos saber onde começa e onde termina cada campo, mas nem sempre o conteúdo do campo está claro. Existem casos, não raros, onde o arquivo não está no formato CSV embora se pareça muito com ele. É o caso de alguns arquivos usados por instituições bancárias, onde não existem delimitadores, mas sim posições a serem respeitadas. Por exemplo: "da coluna 15 até a coluna 20, informe o código da operação com zeros a esquerda".
De qualquer forma, independente do formato do arquivo, você precisa ter acesso ao seu manual de procedimentos, para que a importação e a exportação tenham êxito. Geralmente esse manual está no site do fornecedor de dados, se tiver dificuldade, solicite ao profissional que estiver associado ao procedimento a ser realizado, pode ser um gerente de banco, o contador da organização ou o seu próprio cliente que irá lhe indicar a fonte correta.

[1] Algumas publicações trazem CSV significando Character-Separated Value, já que não necessariamente o caractere separador é uma vírgula.

Figura 20.1: O assistente do LibreOffice Calc para importação de arquivos CSVs.



Como o Harbour cria arquivos no formato CSV

Os arquivos no formato CSV são arquivos textos com uma estrutura bem simples, diferente dos arquivos no formato INI, XML e JSON. Por esse motivo o Harbour não tem nenhuma função especial para

Lendo dados de um arquivo CSV (Método I)

Listagem 20.2: Criando um arquivo no formato CSV (Parte I).
Fonte: codigos/csv01.prg

```
PROCEDURE MAIN

    LOCAL aCsv := hb_ATokens( hb_MemoRead("exemplo.csv"), .t. )
    LOCAL x

    FOR x := 1 TO LEN( aCsv )
        ? aCsv[x]
    NEXT

RETURN
```

As primeiras linhas do resultado :

.:Resultado:.

```
QuotaAmount,StartDate,OwnerName,Username
150000,2016-01-01,Chris Riley,trailhead9.ub20k5i9t8ou@example.com
150000,2016-02-01,Chris Riley,trailhead9.ub20k5i9t8ou@example.com
150000,2016-03-01,Chris Riley,trailhead9.ub20k5i9t8ou@example.com
```

Assim, cada elemento do array equivale a uma linha do arquivo. Ainda não é o ideal porque temos que separar os campos através das virgulas. Isso é feito novamente com hb_aTokens(), conforme a listagem 20.3 :

Listagem 20.3: Criando um arquivo no formato CSV (Parte II).
Fonte: codigos/csv02.prg

```
PROCEDURE MAIN

    LOCAL aCsv := hb_ATokens( hb_MemoRead("exemplo.csv"), .t. )
    LOCAL aLinha
    LOCAL x,y

    FOR x := 2 TO LEN( aCsv )
        aLinha := hb_ATokens( aCsv[x] , "," )
        ?
        FOR y := 1 TO LEN( aLinha )
            ?? aLinha[y] , "  "
        NEXT
    NEXT

RETURN
```

Lendo dados de um arquivo CSV (Método II)

O Harbour possui comandos que possibilitam a importação de dados de um arquivo CSV para um DBF. Isso é feito através do comando APPEND FROM.

Descrição sintática 30

1. Nome : APPEND FROM
2. Classificação : comando.
3. Descrição : Importa registros a partir de um arquivo (.dbf) ou ASCII.
4. Sintaxe

```
APPEND FROM <xcArquivo> ;
[FIELDS <idLista Campos>];
[<abrangência>];
[WHILE <lCondição>];
[FOR <lCondição>];
[SDF|DELIMITED ;
[WITH BLANK|<xcDelimitador>]]
```

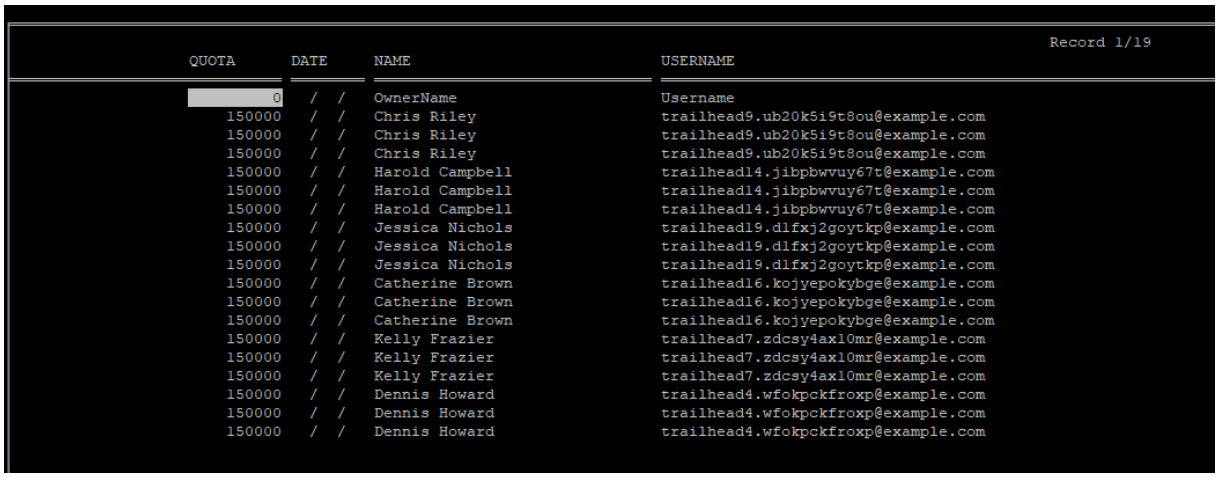
5. Parâmetros
   - FROM <xcArquivo> : Nome do arquivo fonte. Se a extensão não for especificada, (.dbf) será assumido. Se SDF ou DELIMITED é especificado, a extensão de arquivo assumida é (.txt) a menos que outra seja informada.
   - FIELDS <idLista Campos> : especifica a lista de campos que serão copiados de <xcArquivo>. O assumido é todos os campos.
   - <abrangência> : é a porção do arquivo fonte que será usado.
     a) NEXT <n> importa os primeiros <n> registros
     b) RECORD <n> importa somente o registro número <n>
     c) ALL importa todos (esse é o padrão)
   - WHILE <lCondição> : especifica o conjunto de registros que obedecem à condição a partir do primeiro elemento do arquivo fonte até que a condição seja falsa. Maiores detalhes sobre essa cláusula na página 555.
6. Fonte : [Nantucket 1990, p. 4-18]

Listagem 20.4: Criando um arquivo no formato CSV através do comando APPEND FROM.
Fonte: codigos/csvappend01.prg

```
PROCEDURE MAIN                                                  1
                                                                2
    LOCAL aStru := { {"QUOTA","N",10,0},;                       3
                     {"DATE","D",8,0},;                         4
                     {"NAME","C",30,0},;                        5
                     {"USERNAME","C",50,0} }                    6
                                                                7
    DBCREATE( "csvquota" , aStru )                              8
```

```
    USE csvquota EXCLUSIVE
    APPEND FROM exemplo.csv DELIMITED WITH ","
    GO TOP
    BROWSE()

RETURN
```

Figura 20.2: Criando um arquivo no formato CSV através do comando APPEND FROM.



Record 1/19

QUOTA	DATE	NAME	USERNAME
0	/ /	OwnerName	Username
150000	/ /	Chris Riley	trailhead9.ub20k5i9t8ou@example.com
150000	/ /	Chris Riley	trailhead9.ub20k5i9t8ou@example.com
150000	/ /	Chris Riley	trailhead9.ub20k5i9t8ou@example.com
150000	/ /	Harold Campbell	trailhead14.jibpbwvuy67t@example.com
150000	/ /	Harold Campbell	trailhead14.jibpbwvuy67t@example.com
150000	/ /	Harold Campbell	trailhead14.jibpbwvuy67t@example.com
150000	/ /	Jessica Nichols	trailhead19.dlfxj2goytkp@example.com
150000	/ /	Jessica Nichols	trailhead19.dlfxj2goytkp@example.com
150000	/ /	Jessica Nichols	trailhead19.dlfxj2goytkp@example.com
150000	/ /	Catherine Brown	trailhead16.kojyepokybge@example.com
150000	/ /	Catherine Brown	trailhead16.kojyepokybge@example.com
150000	/ /	Catherine Brown	trailhead16.kojyepokybge@example.com
150000	/ /	Kelly Frazier	trailhead7.zdcsy4axl0mr@example.com
150000	/ /	Kelly Frazier	trailhead7.zdcsy4axl0mr@example.com
150000	/ /	Kelly Frazier	trailhead7.zdcsy4axl0mr@example.com
150000	/ /	Dennis Howard	trailhead4.wfokpckfroxp@example.com
150000	/ /	Dennis Howard	trailhead4.wfokpckfroxp@example.com
150000	/ /	Dennis Howard	trailhead4.wfokpckfroxp@example.com

O comando APPEND FROM, nesse exemplo, atendeu parcialmente a nossa necessidade. Dois problemas foram encontrados :

1. **Problema:** A importação de campos data não funcionou.
   Solução : Para que uma data seja reconhecida como tal, ela deve estar no formato aaaammdd (ano,mês e dia) sem qualquer tipo de separador. Reconhecemos que essa não é uma solução ideal para a importação de campos do tipo data. Uma solução paliativa é manter o campo data como caractere e criar um campo extra do tipo data[2]. Posteriormente, após a importação, você deve percorrer o arquivo novamente realizando as devidas conversões. O exemplo da listagem 20.5 ilustra o caso.

   Listagem 20.5: Paliativo para importação de campos do tipo data através do comando APPEND FROM.
   Fonte: codigos/csvappend04.prg

```
PROCEDURE MAIN

    LOCAL aStru := { {"QUOTA","N",10,0},;
                     {"DATESTR","C",10,0},; //recebe a data
                     {"DATE","D",8,0},; //para conversão depois
                     {"NAME","C",30,0},;
                     {"USERNAME","C",50,0} }
```

[2]Essa solução não é ideal porque frequentemente nós não temos o controle sobre um arquivo CSV de terceiros. Essa solução funciona quando estamos trocando dados em um ambiente controlado.

```
    DBCREATE( "csvquota" , aStru )                                          9
    USE csvquota EXCLUSIVE                                                  10
    APPEND FROM exemplo.csv FIELDS QUOTA,;                                  11
                                   DATESTR,;                                12
                                   NAME,;                                   13
                                   USERNAME;                                14
                              DELIMITED WITH ","                            15
    GO TOP                                                                  16
    // laço que faz a conversão para data                                   17
    DO WHILE .NOT. EOF()                                                    18
        REPLACE DATE WITH ;                                                 19
          hb_STOD( hb_StrReplace( FIELD->DATESTR , "-" , "" )  )            20
        SKIP                                                                21
    ENDDO                                                                   22
    GO TOP                                                                  23
    BROWSE()                                                                24
                                                                            25
RETURN                                                                      26
```

2. **Problema:** O comando importou os cabeçalhos do arquivo.
   Solução : Para importar apenas determinadas linhas do arquivo CSV, use a cláusula FOR do comando APPEND FROM. A cláusula FOR funciona como uma espécie de um poderoso filtro, que pode ter condições diversas. No nosso caso, como não queremos importar a primeira linha do arquivo, basta fazer FOR RECNO() > 1 (RECNO() é uma função que retorna sempre número da linha correspondente).

   Listagem 20.6: Comando APPEND FROM com a cláusula FOR.
   Fonte: codigos/csvappend02.prg

```
    APPEND FROM exemplo.csv FOR RECNO() > 1 DELIMITED WITH ","              1
```

   A cláusula FOR pode ser usada para importar apenas as linhas que satisfazem a determinadas condições de um campo. Como por exemplo :

```
APPEND FROM exemplo.csv FOR NAME = "Jessica"
```

   Irá resultar :

Figura 20.3: Filtrando com APPEND FROM.

Juntando os dois problemas acima, o APPEND FROM seria :

```
APPEND FROM exemplo.csv FIELDS QUOTA,;
                               DATESTR,;
```

```
                                NAME,;
                                USERNAME;
                          FOR RECNO() > 1;
                          DELIMITED WITH ","
```

Seguido do laço de gravação de datas

```
DO WHILE .NOT. EOF()
   REPLACE DATE WITH ;
       hb_STOD( hb_StrReplace( FIELD->DATESTR , "-" , "" )  )
   SKIP
ENDDO
```

Importante: Essa é uma solução dada como paliativo para o problema na importação de datas, com certeza existem formas mais eficientes. Tal paliativo deve ser usado sempre que a data **não estiver** no formato aaaammdd. Outras soluções existem para o mesmo problema, essa foi apenas uma ilustração de como se contornar um resultado indesejado. Cada situação requer uma atenção especial do programador. Existirão casos em que essa conversão posterior nem será necessária, e o valor data em formato texto já resolve o problema.

20.4.2 Formato de arquivo INI

O formato de arquivo INI são arquivos de texto simples com uma estrutura básica composta de "seções"e "propriedades". Por exemplo:

```
[Seção]
propriedade1=valor
propriedade2=valor
[Seção2]
; Comentários são iniciados com ";"
propriedade3=valor
propriedade4=valor
```

O nome da seção aparece em uma linha, entre colchetes ([e]). Todas as chaves após a declaração de seção são associadas com aquela seção. Não há um delimitador explícito de "final de seção". Elas terminam na próxima declaração de seção ou no final do arquivo e não podem ser aninhadas. Fonte: `https://pt.wikipedia.org/wiki/INI_(formato_de_arquivo)`. Acessado em 29-Set-2021. É bom ressaltar que o nome de uma propriedade pode se repetir, desde que em seções diferentes, e também que os comentários são sempre iniciados por um ";".

O arquivos INI são bastante comuns porque são práticos e fáceis de manusear. Faça um pequeno teste. Vá para a raiz do seu sistema de arquivos e execute o comando abaixo. Você pode se surpreender com a quantidade de arquivos nesse formato gravados no seu computador.

.:Resultado:.

```
cd c:\
dir *.ini /s
```

Uso típico de um arquivo no formato INI

O uso típico de um arquivo no formato INI é na parametrização de sistemas. O exemplo abaixo é de um arquivo INI usado na configuração de um conhecido jogo :

```
[CONFIG]

; This mod is only for single player games
SinglePlayerOnly = 0

; Set to 1 to disallow use of WorldBuilder scenarios
IgnoreWorldBuilderScenarios = 0

; Allow public maps to be used with this mod
AllowPublicMaps = 0

; Mod Image file
ImageFile = 0

; Name of Mod
Name = TXT_KEY_AMREV_TITLE

; Description of Mod
Description = TXT_KEY_AMREV_DESCRIPTION
```

Da mesma forma, podemos criar arquivos no formato INI para gravarmos configurações do nosso sistema, por exemplo:

```
[GERAL]
nome=Editora Nascimento
cnpj=1234567898
email=nascimento@dominio.net
[DATABASE]
ip=192.168.0.1
porta=5432
usuario=novolivro
senha=%$#@!@%&()))()
```

Como o Harbour cria arquivos no formato INI

O Harbour usa funções específicas para manipular arquivos no formato INI. A listagem 20.7 mostra como criar um arquivo no formato INI.

Listagem 20.7: Criando um arquivo no formato INI.
Fonte: codigos/ini01.prg

```
PROCEDURE MAIN
```

```
    LOCAL hIni := { => }

    hIni["BANCO01"] := { => }
    hIni["BANCO01"]["USUARIO"] := "USER01"
    hIni["BANCO01"]["PASSWORD"] := "PASS"
    hb_IniWrite( "test.ini" , hIni )

    ? hb_MemoRead( "test.ini" )

    RETURN
```

.:Resultado:.

```
[BANCO01]
USUARIO=USER01
PASSWORD=PASS
```

Note que existe uma relação entre o formato INI e o Hash, conforme a lista a seguir:

1. Primeiramente iniciamos o Hash. Esse Hash representa o arquivo inteiro.

   ```
   hHash := { => }
   ```

2. A primeira chave do Hash corresponde a "seção"do arquivo INI a ser gerado, ela sempre armazena um outro Hash.

   ```
   hHash["BANCO01"] := { => }
   ```

3. O Hash que está contido equivale as propriedades e seus respectivos valores dentro da seção.

   ```
   hHash["BANCO01"]["USUARIO"] := "USER01"
   ```

Concluindo: para criar um arquivo INI, primeiramente crie um Hash obedecendo as regras já estabelecidas e grave o arquivo através da função hb_IniWrite, conforme o formato abaixo :

```
hb_IniWrite( "nomedoarquivo.ini" , hMeuHash )
```

A função hb_IniWriteStr() funciona de forma semelhante, só que ela não grava em um arquivo, apenas retorna o conteúdo o INI em forma de string. Por exemplo, a listagem 20.8 não cria um arquivo, ela apenas retorna o conteúdo.

Listagem 20.8: Retornando uma string no formato INI.
Fonte: codigos/ini02.prg

```
PROCEDURE MAIN

    LOCAL hIni := { => }

    hIni["BANCO01"] := { => }
```

```
hIni["BANCO01"]["USUARIO"] := "USER01"
hIni["BANCO01"]["PASSWORD"] := "PASS"
? hb_IniWriteStr( hIni )

RETURN
```

Note que a linha 11 retorna uma string, ou seja, nenhum arquivo foi gerado.

.:Resultado:.

```
[BANCO01]
USUARIO=USER01
PASSWORD=PASS
```

Dica 121

Você deve estar achando que a função hb_IniWriteStr() é inútil, já que hb_IniWrite() já faz o trabalho todo, inclusive já grava no arquivo. Contudo, existem casos onde você vai desejar apenas uma string no formato do arquivo a ser gravado. Esses casos ocorrem, por exemplo, quando você quer criptografar a string antes de gravar em um arquivo.

```
cStr := hb_IniWriteStr( hMeuIni )
hb_MemoWrite( "config.ini" , hb_crypt( cStr,"minhachave" ))
```

Assim, temos um arquivo ini totalmente criptografado, que pode ser descriptografado assim :

```
cStr := hb_decrypt( hb_MemoRead("config.ini"),"minhachave" )
hMeuIni := hb_IniReadStr( cStr )
```

Lendo dados de um arquivo INI

20.4.3 Formato de arquivos XML

20.4.4 Formato de arquivos JSON

20.5 Arquivos compactados

20.6 Funções de diretório

20.7 Conclusão

21 Impressão

> Pedi, e dar-se-vos-á; buscai e encontrareis; batei, e abrir-se-vos-á. Porque aquele que pede recebe; e o que busca encontra; e, ao que bate, se abre.0
>
> Mateus 6:7-8

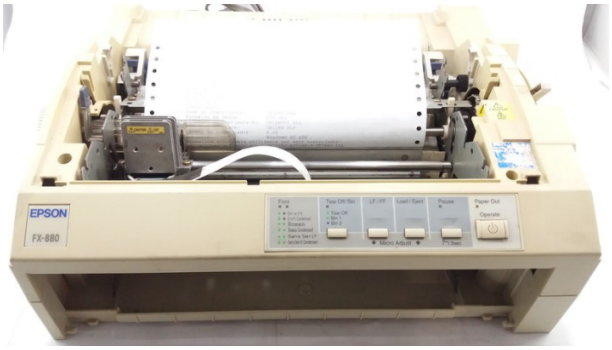
Objetivos do capítulo

- Entender como o Harbour realiza impressões

21.1 Introdução

A tecnologia de impressão mudou muito nos últimos quarenta anos. Na década de 1980, com a popularização dos micro-computadores, a impressão de documentos também cresceu. Naquela época, era muito comum reportagens falando sobre o escritório do futuro, ou o escritório sem papel. Contudo o consumo de papel aumentou em vez de diminuir. Depois que uma pessoa, ou uma organização, comprava um computador, o segundo equipamento a se adquirir era quase sempre uma impressora. Atualmente a impressora ainda é um ítem indispensável nas organizações, mas vem perdendo espaço nos lares, apesar das vendas só crescerem em termos absolutos. A tecnologia predominante naquelas décadas era a matricial (figura 21.1), cuja impressão era através de uma fita (semelhante a das antigas máquinas de datilografar). O conteúdo a ser impresso também era bem simples, em sua maioria apenas texto, sem imagens ou desenhos, no máximo um gráfico simples.

Figura 21.1: Uma impressora matricial.



Atualmente ainda existem impressoras matriciais a venda, mas elas já perderam espaço para as impressoras jato-de-tinta e laser. Além dessas tecnologias citadas temos também impressoras térmicas, jato de cera, cera térmica, dye-sublimation e tinta sólida.

21.2 A primeira forma de impressão

Não é nosso intuito aqui descrever todas as formas que um programa usa para imprimir um documento, afinal de contas algumas formas já nem existem mais. Contudo, é importante entender como a impressão era realizada na década de 1980, porque essa forma ainda é usada, embora não seja a forma mais utilizada. Vamos partir de um exemplo conhecido.

21.3 Programação multiusuário

O comando SET PRINTER altera o

Descrição sintática 31

1. Nome : SET PRINTER
2. Classificação : comando.
3. Descrição : Comuta o eco de saída do console para a impressora ou arquivo.
4. Sintaxe

```
SET PRINTER on|off|<lComuta>

SET PRINTER TO [<xcDispositivo>|<xcArquivo>]
```

5. Parâmetros
   - <cArqDados> : O nome do arquivo
6. Fonte : [Nantucket 1990, p. 5-240]

21.4 Conclusão

22 Integração com o MS Windows

> Muitos usuários devem estar cientes de que estão roubando softwares. Eles pagam pelo hardware, mas software é algo que podem compartilhar. Quem se importa com o fato de que as pessoas que trabalharam em um software tenham que ser pagas por isso ? Isso é justo ?
>
> Bill Gates

Objetivos do capítulo

- Obter conhecimentos básicos de Windows

22.1 Introdução

O Harbour possui suporte a programação Windows.

22.2 MS Excel

22.2.1 Verificando se tem suporte ao MS Excel

Listagem 22.1: Verificando se tem suporte ao MS Excel.
Fonte: codigos/excel01.prg

```
PROCEDURE Main()                                                    1
                                                                    2
                                                                    3
    IF win_oleCreateObject( "Excel.Application" )  != NIL           4
       ? "Esse computador tem suporte ao software MS Excel"         5
    ELSE                                                            6
       ? "Erro: Computador sem suporte ao software MS Excel: [" + ; 7
                win_oleErrorText() + "]"                            8
    ENDIF                                                           9
                                                                    10
RETURN                                                              11
```

22.2.2 Gravando um valor em uma célula

Listagem 22.2: Gravando um valor em uma célula.
Fonte: codigos/excel02.prg

```
REQUEST HB_CODEPAGE_UTF8 // Para a acentuação correta               1
PROCEDURE Main()                                                    2
                                                                    3
    LOCAL oExcel := win_oleCreateObject( "Excel.Application" )      4
    LOCAL oSheet                                                    5
                                                                    6
    hb_CdpSelect("UTF8") // Seleciono UTF-8                         7
    IF oExcel  != NIL                                               8
        oExcel:Visible := .T.                                       9
        oExcel:WorkBooks:Add()                                      10
        oSheet := oExcel:ActiveSheet()                              11
        oSheet:Cells( 1, 1 ):Value := "Olá, pessoal"                12
    ELSE                                                            13
       ? "Erro: MS Excel indisponível. [" + win_oleErrorText() + "]" 14
    ENDIF                                                           15
                                                                    16
RETURN                                                              17
```

22.2.3 Obtendo as planilhas

Listagem 22.3: Obtendo as planilhas.
Fonte: codigos/excel03.prg

```
REQUEST HB_CODEPAGE_UTF8 // Para a acentuação correta
PROCEDURE Main()

    LOCAL oExcel := win_oleCreateObject( "Excel.Application" )
    LOCAL oWorkBook
    LOCAL oSheet
    LOCAL nI, nCount

    hb_CdpSelect("UTF8") // Seleciono UTF-8
    IF oExcel  != NIL
        oExcel:Visible := .T.
        ? "Primera forma"
        oWorkBook := oExcel:WorkBooks:Add()
        FOR EACH oSheet IN oWorkBook:WorkSheets()
            ? oSheet:Name()
        NEXT
        ? "Segunda forma"
        nCount := oWorkBook:WorkSheets:Count()
        FOR nI := 1 TO nCount
            ? oWorkBook:WorkSheets:Item( nI ):Name
        NEXT

    ELSE
       ? "Erro: MS Excel indisponível. [" + win_oleErrorText() + "]"
    ENDIF

RETURN
```

22.2.4 Configuração geral

Listagem 22.4: Configurando a fonte para todas as células.
Fonte: codigos/excel04.prg

```
REQUEST HB_CODEPAGE_UTF8 // Para a acentuação correta
PROCEDURE Main()

    LOCAL oExcel := win_oleCreateObject( "Excel.Application" )
    LOCAL oSheet

    hb_CdpSelect("UTF8") // Seleciono UTF-8
    IF oExcel  != NIL
        oExcel:Visible := .T.
        oExcel:WorkBooks:Add()
        oSheet := oExcel:ActiveSheet()
        // Configura a fonte para todas as células
        oSheet:Cells:Font:Name := "Courier"
        oSheet:Cells:Font:Size := 18

```

```
    ELSE
       ? "Erro: MS Excel indisponível. [" + win_oleErrorText() + "]"
    ENDIF

RETURN
```

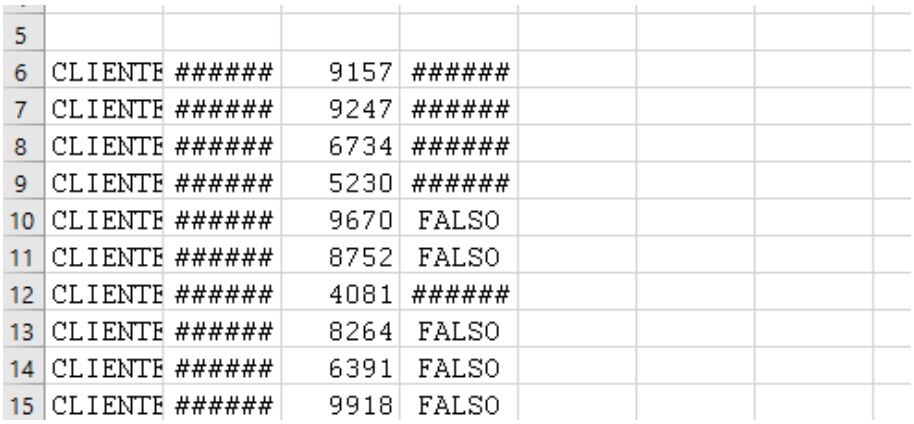
22.3 Lendo dados de um arquivo

22.3.1 Lendo dados de um arquivo I

Listagem 22.5: Lendo dados de um arquivo I.
Fonte: codigos/excel0501.prg

```
REQUEST HB_CODEPAGE_UTF8 // Para a acentuação correta
#define COLUNA1 1
#define COLUNA2 COLUNA1 + 1
#define COLUNA3 COLUNA2 + 1
#define COLUNA4 COLUNA3 + 1
PROCEDURE Main()

    LOCAL oExcel := win_oleCreateObject( "Excel.Application" )
    LOCAL oSheet
    LOCAL nLinha

    CriaArquivoParaTestes()
    USE excel
    hb_CdpSelect("UTF8") // Seleciono UTF-8
    IF oExcel  != NIL
        oExcel:Visible := .T.
        oExcel:WorkBooks:Add()
        oSheet := oExcel:ActiveSheet()
        // Configura a fonte para todas as células
        oSheet:Cells:Font:Name := "Courier"
        oSheet:Cells:Font:Size := 12
        nLinha := 6 // Linha inicial
        DO WHILE .NOT. EOF()
            oSheet:Cells( nLinha, COLUNA1 ):Value := ;
                                          ALLTRIM(FIELD->NOME)
            oSheet:Cells( nLinha, COLUNA2 ):Value := FIELD->NASC
            oSheet:Cells( nLinha, COLUNA3 ):Value := FIELD->COMPRAS
            oSheet:Cells( nLinha, COLUNA4 ):Value := FIELD->VIP
            ++nLinha
            SKIP
        ENDDO
    ELSE
       ? "Erro: MS Excel indisponível. [" + win_oleErrorText() + "]"
    ENDIF

RETURN
```

Figura 22.1: Lendo dados de um arquivo I.



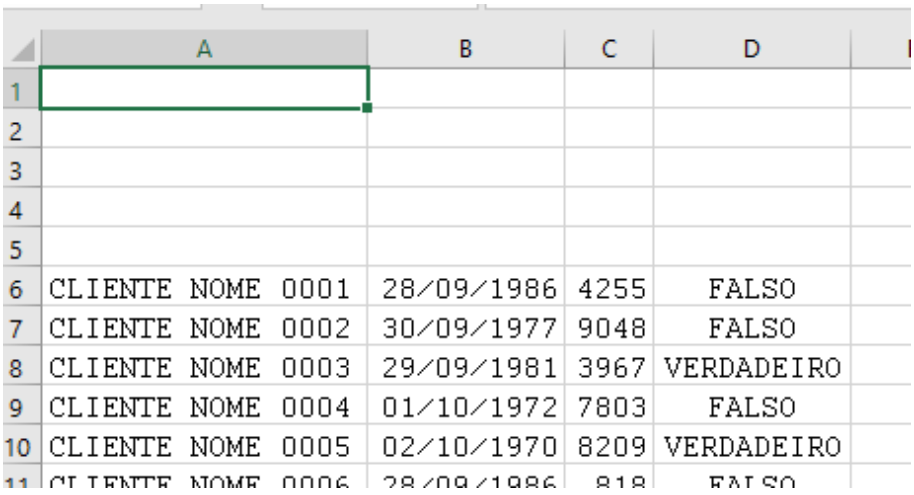
5				
6	CLIENTE	######	9157	######
7	CLIENTE	######	9247	######
8	CLIENTE	######	6734	######
9	CLIENTE	######	5230	######
10	CLIENTE	######	9670	FALSO
11	CLIENTE	######	8752	FALSO
12	CLIENTE	######	4081	######
13	CLIENTE	######	8264	FALSO
14	CLIENTE	######	6391	FALSO
15	CLIENTE	######	9918	FALSO

22.3.2 Lendo dados de um arquivo II: Ajustando a largura

Listagem 22.6: Lendo dados de um arquivo II: Ajustando a largura.
Fonte: codigos/excel0502.prg

```
nLinha := 6 // Linha inicial
DO WHILE .NOT. EOF()
    oSheet:Cells( nLinha, COLUNA1 ):Value := ;
                                        ALLTRIM(FIELD->NOME)
    oSheet:Cells( nLinha, COLUNA2 ):Value := FIELD->NASC
    oSheet:Cells( nLinha, COLUNA3 ):Value := FIELD->COMPRAS
    oSheet:Cells( nLinha, COLUNA4 ):Value := FIELD->VIP
    ++nLinha
    SKIP
ENDDO
oSheet:Columns( COLUNA1 ):AutoFit()
oSheet:Columns( COLUNA2 ):AutoFit()
oSheet:Columns( COLUNA3 ):AutoFit()
oSheet:Columns( COLUNA4 ):AutoFit()
```

Figura 22.2: Lendo dados de um arquivo II: Ajustando a largura.



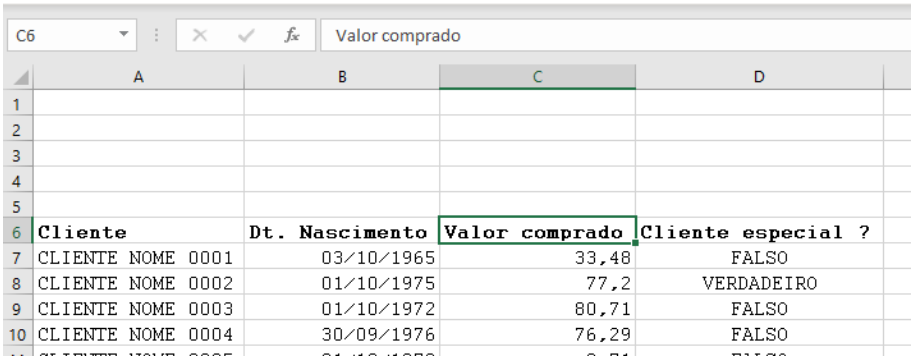
	A	B	C	D
1				
2				
3				
4				
5				
6	CLIENTE NOME 0001	28/09/1986	4255	FALSO
7	CLIENTE NOME 0002	30/09/1977	9048	FALSO
8	CLIENTE NOME 0003	29/09/1981	3967	VERDADEIRO
9	CLIENTE NOME 0004	01/10/1972	7803	FALSO
10	CLIENTE NOME 0005	02/10/1970	8209	VERDADEIRO
11	CLIENTE NOME 0006	28/09/1986	818	FALSO

22.3.3 Lendo dados de um arquivo III: Formatando cabeçalhos

Listagem 22.7: Lendo dados de um arquivo III: Formatando cabeçalhos.
Fonte: codigos/excel0503.prg

```
        /* Formatando cabeçalhos*/
        oSheet:Cells( ++nLinha, COLUNA1 )
        oSheet:Cells( nLinha, COLUNA1 ):Value := "Cliente"
        oSheet:Cells( nLinha, COLUNA2 ):Value := "Dt. Nascimento"
        oSheet:Cells( nLinha, COLUNA3 ):Value := "Valor comprado"
        oSheet:Cells( nLinha, COLUNA4 ):Value := "Cliente especial ?"
        oSheet:Rows( nLinha ):Font:Bold := .t. // Negrito
        /* Iniciando a exportação para o Excel */
        DO WHILE .NOT. EOF()
            ++nLinha
            oSheet:Cells( nLinha, COLUNA1 ):Value := ;
                                            ALLTRIM(FIELD->NOME)
            oSheet:Cells( nLinha, COLUNA2 ):Value := FIELD->NASC
            oSheet:Cells( nLinha, COLUNA3 ):Value := FIELD->COMPRAS
            oSheet:Cells( nLinha, COLUNA4 ):Value := FIELD->VIP
            SKIP
        ENDDO
```

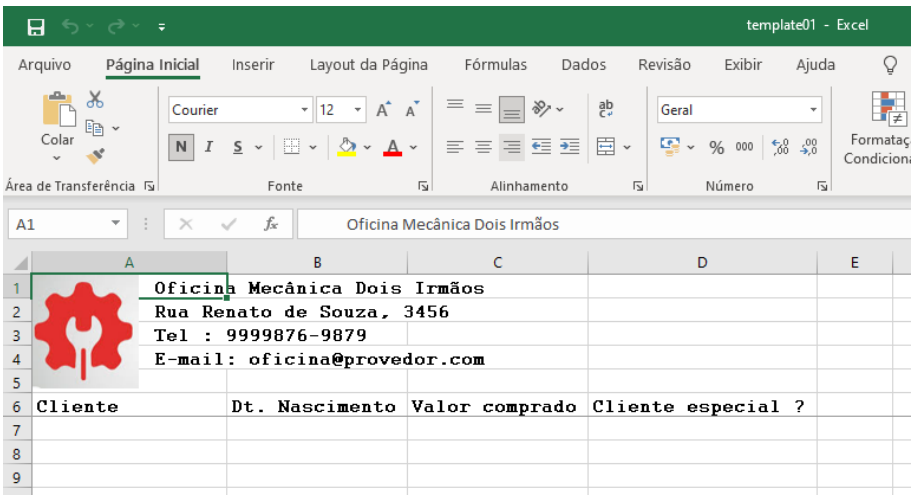
Figura 22.3: Lendo dados de um arquivo III: Formatando cabeçalhos.



22.4 Usando uma planilha pré-formatada como modelo

22.4.1 Modelo inicial

Figura 22.4: Modelo inicial: arquivo template01.xlsx.



Basicamente apenas temos que inserir o comando de abrir a planilha. Ou seja, troque a linha

```
oExcel:WorkBooks:Add()
```

pela linha da listagem 22.8

Listagem 22.8: Importando dados para a planilha aberta.
Fonte: codigos/excel06.prg

```
oExcel:WorkBooks:Open( Hb_DirBase()  +  "template01.xlsx")
```

Dica 122

Um detalhe interessante. O comando abaixo

```
oExcel:Visible := .T.
```

torna a planilha visível ao usuário. Nos exemplos passados, nós colocamos esse comando no início do processamento para que você acompanhe as alterações feitas na planilha. **Contudo, é aconselhável que você torne a planilha visível somente após todo o processo ter terminado**. Se por acaso você executar esse comando antes das alterações, isso pode levar a resultados indesejáveis. Esse problema costuma ocorrer quando o usuário realiza alguma operação de interação com a planilha enquanto ela está sendo preenchida.
Conclusão: deixe o `oExcel:Visible := .T.` para o final.

22.5 Conclusão

23 O Harbour e a Filosofia do UNIX

> Falar é fácil, me mostre o código.
>
> Linus Torvalds

Objetivos do capítulo

- Obter conhecimentos básicos de Windows

23.1 Introdução

Usar o Harbour no ambiente *NIX é relativamente fácil. Na verdade não há muito a ser feito, o que você precisa é ter um conhecimento básico do *NIX, o que na prática significa saber usar o Linux em modo texto. No entanto, você será mais produtivo se aprender um pouco sobre a Filosofia do UNIX, que é o sistema que originou o Linux e os BSDs. Nesse capítulo veremos uma versão curta dessa Filosofia, e em seguida iremos aplicar esses princípios em um programa Harbour. Usaremos o Linux para a grande maioria dos exemplos, mas veremos na prática que esses princípios, apesar de nascerem no mundo UNIX, podem ser aplicados também no mundo Windows, através do Prompt de Comando e Powershell.

23.2 A filosofia do UNIX

A Filosofia do UNIX começa com a idéia de uma coleção de ferramentas bem projetadas e que funcionam como um time. A primeira versão dessa filosofia foi criada por Ken Thompson na década de 1970, mas ao longo do tempo ela sofreu diversos "desdobramentos". Abordaremos uma versão resumida em três princípios listados por Peter Salus [Salus 1994, p. 53] :

1. Escreva programas que façam uma coisa e a façam bem.
2. Escreva programas para trabalharem juntos.
3. Escreva programas para lidar com fluxos de texto, porque essa é uma interface universal.

23.2.1 Princípio I

Podemos dividir os programas em duas categorias: a primeira são comandos de único propósito, e a segunda categoria é composta por programas interativos. A maioria dos programas desenvolvidos em Harbour pertencem a segunda categoria: são programas para controle de estoque, contas a pagar, contabilidade, etc. Apesar de terem um único propósito eles são bem abrangentes em suas tarefas. Por exemplo, um controle de estoque tem rotinas de "baixa"de mercadorias, relatórios e cadastros diversos. Tudo isso em apenas um único executável. Já a grande maioria dos comandos do UNIX/Linux são de único propósito. Isso tem tudo a ver com o primeiro princípio da filosofia do UNIX. De acordo com Norton e Hahn:

> Cada comando do UNIX deve fazer uma coisa apenas, e deve fazê-la bem. Pense nos comandos do UNIX como ferramentas. Uma vez que você aprenda um comando, poderá acrescentá-lo à sua caixa de ferramentas pessoal. Em vez de ter um programa que tente fazer tudo, como um canivete suíço, o UNIX concede a você uma ampla seleção de ferramentas de único propósito, cada uma das quais realiza bem um serviço. Se precisar executar tarefas complexas, o UNIX fornece meios de combinar as ferramentas para atender às suas necessidades.[Norton e Hahn 1992, p. 114]

Em resumo: faça apenas uma coisa. Por exemplo, o comando "ls"do *NIX faz apenas uma única tarefa: listar arquivos e diretórios. Ele é sucinto e direto ao ponto: se não há nada a listar o ls não exibirá mensagem alguma, nem sequer algo do tipo "não há arquivos". Esse princípio não é exclusivo do UNIX, você pode aplicá-lo em outras plataformas e linguagens, mas ele é particularmente verdadeiro em ambientes UNIX. Eric Raymond resume esse princípio através do acrônimo KISS[1].

23.2.2 Princípio II

O UNIX oferece ferramentas simples que podem ser combinadas, quando você solicitar, para desempenhar tarefas mais complexas. Ou seja, escreva programas para trabalharem juntos. Os programas criados de acordo com o princípio I não serão realmente produtivos, se eles não trabalharem em conjunto. Mas como conseguir esse "trabalho em equipe"?

23.2.3 Princípio III

Escreva programas que manipulem streams[2] de texto, pois esta é uma interface universal. Esse terceiro princípio é a "cola"que faz com que o princípio II seja posto em prática. Veremos como esses princípios são colocados em prática na seção a seguir.

23.3 O Harbour e o Linux

23.3.1 Integrando o stream de saída do programa

Vamos começar, como sempre, com um exemplo simples:

Listagem 23.1:
Fonte: codigos/unix01.prg

```
PROCEDURE MAIN

    hb_cdpSelect("UTF8")
    ?? "O peito do pé do Pedro é preto."
    ? "O rato roeu a roupa do rei de Roma."

RETURN
```

Ao executar, no shell Bash, temos a seguinte saída :

[1] "Keep It Simple", ou "Mantenha Simples"em português, é um princípio conhecido do projeto de qualquer produto. Raymond acrescentou um "S"a mais. KISS, acrônimo em inglês de: "Keep It Simple, Stupid", ou seja, "Mantenha Simples, Estúpido"é um princípio geral que valoriza a simplicidade do projeto e defende que toda a complexidade desnecessária seja descartada.https://pt.wikipedia.org/wiki/Princ%C3%ADpio_KISS Acessada em 21-Set-2021

[2] em português fluxo, é uma sequência de elementos de dados disponibilizados ao longo do tempo. Um fluxo pode ser considerado como itens em uma esteira transportadora sendo processados um por vez, em vez de em grandes lotes. Fonte: https://pt.wikipedia.org/wiki/Stream_(computa%C3%A7%C3%A3o) Acessado em 21-Set-2021.

.:Resultado:.

```
O peito do pé do Pedro é preto.
O rato roeu a roupa do rei de Roma.
```

Vamos agora verificar se esse programa está seguindo os princípios abordados pela Filosofia UNIX. Para fazer o teste usaremos o comando grep para receber o stream de dados gerado pelo nosso programa.

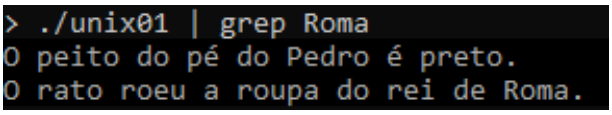
Digite :

.:Resultado:.

```
./unix01 | grep Roma
```

O resultado é :

Figura 23.1: grep lendo a saída do nosso programa (sem sucesso).



Ou seja, o programa **não** está interagindo com outros comandos do UNIX. O comando grep não está conseguindo realizar a busca porque ele não está recebendo nenhuma stream. O resultado esperado seria a segunda string, pois contém a palavra "Roma".

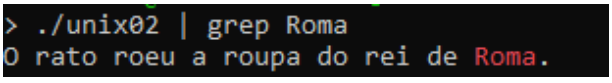
Para conseguir que um programa leia a saída do pipe, algumas modificações são mínimas são necessárias, conforme a listagem 23.2.

Listagem 23.2:
Fonte: codigos/unix02.prg

```
PROCEDURE MAIN

    hb_cdpSelect("UTF8")
    outstd( "O peito do pé do Pedro é preto." + hb_eol() )
    outstd( "O rato roeu a roupa do rei de Roma." )

RETURN
```

Agora sim, o comando grep está conseguindo ler o stream de saída.

Figura 23.2: grep lendo a saída do nosso programa.



Dica 123

Essa técnica também vale para o Windows, basta substituir o grep pelo comando equivalente.
No Prompt de Comando use findstr :

```
unix02 | findstr Roma
```

No Powershell use Select-String -Pattern :

```
.\unix02 | Select-String -Pattern Roma
```

23.4 Conclusão

24 Expressões regulares

Na teoria, a prática é simples.

Trygve Reenskaug

Objetivos do capítulo

- Entender o funcionamento do banco de dados do Harbour.

24.1 Introdução

24.2 Classificação das expressões regulares

24.3 O básico

Vamos começar com um exemplo simples: você quer buscar a string "Você"dentro de uma string.

Listagem 24.1: Busca simples
Fonte: codigos/regexp01.prg

```
/*

Busca simples

*/
REQUEST HB_CODEPAGE_UTF8
PROCEDURE MAIN
    LOCAL cRegex  := "Você" // O que estou buscando

   LOCAL cString := "Você vai vencer. Você não vai desistir." // Onde busco
    LOCAL aResult // O resultado vem aqui

    HB_CDPSELECT( "UTF8" )
    ? "Buscando  : " ,  cRegex
    ? "Na string : " , cString
    aResult := hb_Regex( cRegex, cString  )
    ? hb_ValToExp( aResult )

RETURN
```

A saída do programa.

.:Resultado:.

```
Buscando  :  Você
Na string :  Você vai vencer. Você não vai desistir.
{"Você"}
```

Algumas considerações iniciais:

1. Usei o codepage UTF-8 porque o editor que estou usando (VS Code) está configurado para UTF-8, mas outra codificação poderia ser usada.
2. A função hb_RegExp() é quem faz todo o serviço.
3. O primeiro argumento é **o que** vou buscar.
4. O segundo argumento é **onde** vou buscar.
5. O retorno é um array com todas as ocorrências.

Descrição sintática 32

1. Nome : hb_RegExp()
2. Classificação : função.
3. Descrição : Avalia uma expressão regular
4. Sintaxe

```
hb_RegExp( <cRegExp>|<hRegExp> ,;
           <cBusca>;
           [, <lCase> [, <lMultiLine> ]];
           ) -> aMatch

aMatch : array com as correspondências
```

5. Parâmetros
   - <cRegExp>|<hRegExp> : <cRegExp> é uma string com a busca a ser realizada, <hRegExp> é uma string compilada com hb_RegExpComp().
   - <cBusca> : String onde a busca será realizada.
   - [<lCase>]: busca fará distinção entre maiúsculas e minúsculas ? (padrão .T.)
   - [<lMultiLine>] : especifica se a expressão usará o modo multilinhas para correspondência. (padrão .F.)
6. Fonte : `https://github.com/Petewg/harbour-core/wiki/Regular-Expressions`

Faz sentido você questionar o porquê da função retornar um array. Mas, conforme já vimos, uma expressão regular descreve um conjunto de cadeia de caracteres.

Importante : durante o nosso aprendizado usaremos a função hb_RegExp(), mas o Harbour possui outras funções. Nós usaremos apenas essa função porque ela é suficiente para aprendermos expressões regulares. Quando você aprender expressões regulares, você poderá usar os conceitos aprendidos em qualquer linguagem que suporte essas expressões[1].

24.4 Metacaracteres tipo Âncora

24.4.1 A metáfora da âncora.

Às vezes queremos realizar a busca em apenas uma determinada área do texto. É isso que os metacaracteres tipo âncora fazem, eles restringem a área da busca. É

[1] Existe um padrão (POSIX) que torna a sintaxe das expressões regulares homogêneas, independente da linguagem de programação. O que você vai aprender pode ser usado em Shell Script, Javascript, Java, C++, Object Pascal, Python, etc.

como se o seu buscador fosse uma embarcação que irá buscar algo em apenas uma região do lago. Para que isso aconteça, ela precisa estar ancorada nessa região.

Figura 24.1: A âncora restringe a embarcação a uma determinada área.

Temos os seguintes tipos de âncoras : de linha e de palavra. As âncoras de linha são :

1. ^ : realiza a busca no início da linha
2. $: realiza a busca no final da linha

As âncoras de palavra são :

1. \b : realiza a busca no início da palavra (ou no final dela)
2. \B : é uma negação de \b

24.4.2 Início da linha

Para pesquisar no início da linha use o metacaractere ^. Nesse exemplo, da listagem 24.2, buscamos uma string que inicie com a palavra "você".

Listagem 24.2: Buscando uma ocorrência no início da linha.
Fonte: codigos/regexp03.prg

```
REQUEST HB_CODEPAGE_UTF8
PROCEDURE MAIN
    LOCAL cRegex  := "^você" // O que estou buscando
    LOCAL cString  // Onde busco
    LOCAL aResult // O resultado vem aqui
    LOCAL aString := { "você vai vencer, você não vai desistir" ,;
                       "gosto muito de você" }

```

```
    HB_CDPSELECT( "UTF8" )                                              10
    ? "Buscando  : " ,  cRegex                                          11
    FOR EACH cString IN aString                                         12
        ? "Na string : " , cString                                      13
        aResult := hb_Regex( cRegex, cString  )                         14
        ? hb_ValToExp( aResult )                                        15
    NEXT                                                                16
                                                                        17
RETURN                                                                  18
```

O resultado do nosso teste:

.:Resultado:.

```
Buscando  :  ^você
Na string :  você vai vencer, você não vai desistir
{"você"}
Na string :  Gosto muito de você
{}
```

Observe que a segunda busca não encontrou a string porque ela não está no início da linha.

24.4.3 Fim da linha

Para pesquisar no final da linha use o metacaractere `$`. Faça uma pequena modificação no programa da listagem 24.2. Na linha 8 troque o conteúdo da variável cRegExp por "você$".

.:Resultado:.

```
    Buscando  :  você$
    Na string :  você vai vencer, você não vai desistir
    {}
    Na string :  Gosto muito de você
    {"você"}
```

24.4.4 A string existe no início ou no final da palavra

Agora a coisa mudou. A busca agora é por uma determinada string no início (ou no final) de uma palavra. Vamos realizar alguns testes buscando várias situações envolvendo a string "bem"dentro de cada uma das strings abaixo:

```
o bem-te-vi chegou cedo
e não nos cansemos de fazer o bem
o bem vence o mal
os marinheiros bebem muito
use o bemol para baixar um semitom
```

Para isso nós usaremos a expressão regular \b ("b"de borda).

Buscando palavras que começam com uma string : \b

Para pesquisarmos todas as palavras começadas com "bem", use a expressão "\bbem".

Listagem 24.3: Buscando uma ocorrência no início da palavra.
Fonte: codigos/regexp05.prg

```
REQUEST HB_CODEPAGE_UTF8
PROCEDURE MAIN
    LOCAL cRegex  := "\bbem" // O que estou buscando
    LOCAL cString, aString := {} // Onde busco
    LOCAL aResult // O resultado vem aqui

    HB_CDPSELECT( "UTF8" )
    ? "Buscando  : " ,  cRegex
    AADD( aString , "o bem-te-vi chegou cedo"  )
    AADD( aString , "e não nos cansemos de fazer o bem"  )
    AADD( aString , "o bem vence o mal"  )
    AADD( aString , "os marinheiros bebem muito")
    AADD( aString , "use o bemol para baixar um semitom")
    FOR EACH cString IN aString
        ? "Na string : " , cString
        aResult := hb_Regex( cRegex, cString  )
        ? hb_ValToExp( aResult )
    NEXT
RETURN
```

As strings encontradas :

```
o bem-te-vi chegou cedo
e não nos cansemos de fazer o bem
o bem vence o mal
use o bemol para baixar um semitom
```

Buscando palavras que terminam com uma string : \b

Para pesquisarmos todas as palavras começadas com "bem", use a expressão "bem\b".

Prática número 28

Modifique o programa da listagem 24.3 e realize a busca por strings terminadas em "bem".
As strings encontradas :

```
o bem-te-vi chegou cedo
e não nos cansemos de fazer o bem
o bem vence o mal
os marinheiros bebem muito
```

Você deve achar estranho "bem-te-vi"terminar com a string "bem". Realmente é estranho, mas esse comportamento não é do Harbour. Ele está presente em todas as linguagens que usam o padrão POSIX de expressões regulares. De acordo com esse padrão, o sinal -"tem o mesmo efeito que um espaço em branco.

Buscando palavras que começam e terminam com uma string : \b

Para pesquisarmos todas as palavras começadas e terminadas com "bem", use a expressão "\bbem\b". Isso é útil para listar uma palavra isolada dentro de um texto.

Prática número 29

Modifique o programa da listagem 24.3 e realize a busca por strings começadas e terminadas em "bem".
As strings encontradas :

```
o bem-te-vi chegou cedo
e não nos cansemos de fazer o bem
o bem vence o mal
```

24.4.5 A string NÃO existe no início ou no final da palavra

Para **negar** as expressões começadas ou finalizadas por uma string, use \B ("B"maiúsculo).

Não iremos nos estender nos exemplos pois se você aprendeu a seção anterior, com certeza não terá dificuldade nessa. Vamos apenas listar os casos e os resultados baseado nos nossos exemplos. Um exercício interessante é pegar o código da listagem 24.3 e adaptar para usarmos nos casos a seguir. Lembre-se, agora é "B"maiúsculo (\B) porque estamos negando.

Strings que contém somente palavras que NÃO começam com "bem"

```
os marinheiros bebem muito
```

Strings que contém somente palavras que NÃO terminam com "bem"

```
use o bemol para baixar um semitom
```

Strings que contém somente palavras que NÃO começam e NÃO terminam com "bem"

Nenhuma ocorrência aqui.

24.5 Metacaracteres Representantes

24.6 Conclusão

Pontos importantes que merecem atenção especial

1. bla

25 Integração com a Linguagem C

[Deus] tudo faz para o melhor e nada poderá prejudicar a quem o ama. Conhecer, porém, em particular, as razões que puderam movê-lo a escolher esta ordem do universo, tolerar os pecados e dispensar as suas graças salutares de uma determinada forma, eis o que ultrapassa a força de um espírito finito.

Leibniz - Discurso de Metafísica

Objetivos do capítulo

- conhecer um pouco da história da linguagem C.
- aprender a desenvolver um pequeno programa em C.
- chamar rotinas em C através do Harbour.
- aprender a passar parâmetros para rotinas em C.
- obter o retorno de uma função em C a partir do Harbour.
- chamando uma rotina Harbour a partir do da linguagem C.

25.1 Um pouco de história

A linguagem C foi criada em 1970

O objetivo desse capítulo é estudar esses dois recursos e fazer o melhor uso deles nos nossos códigos.

26 Programação GUI

> É difícil escrever um software que seja executado correta e eficientemente. Assim sendo, depois que um programa está funcionando em um ambiente, você não quer repetir todo o esforço quando tiver que movê-lo para um compilador, processador ou sistema operacional diferentes. Idealmente, ele não deve precisar de nenhuma alteração.
>
> Brian W. Kernighan e Rob Pike

Objetivos do capítulo

- Entender o que é portabilidade.
- Funções de portabilidade
- SETs de portabilidade

26.1 Introdução

27 Segurança

.

Objetivos do capítulo

- Entender o que é portabilidade.
- Funções de segurança

27.1 Introdução

28 Ponteiros

.

Objetivos do capítulo

- Entender o que são ponteiros.
- Funções de segurança

28.1 Introdução

Parte III

Banco de dados

29 Arquivos DBFs : Modo interativo

Na teoria, a prática é simples.

Trygve Reenskaug

Objetivos do capítulo

- Entender o funcionamento do banco de dados do Harbour.
- Aprender a criar um arquivo e a realizar operações básicas sobre ele.
- Entender como funciona o ponteiro dos registros de um arquivo.
- Conhecer os comandos básicos de manipulação de arquivos e suas respectivas cláusulas.
- Entender como funciona o mecanismo de exclusão de dados.
- Aprender a indexar um arquivo.

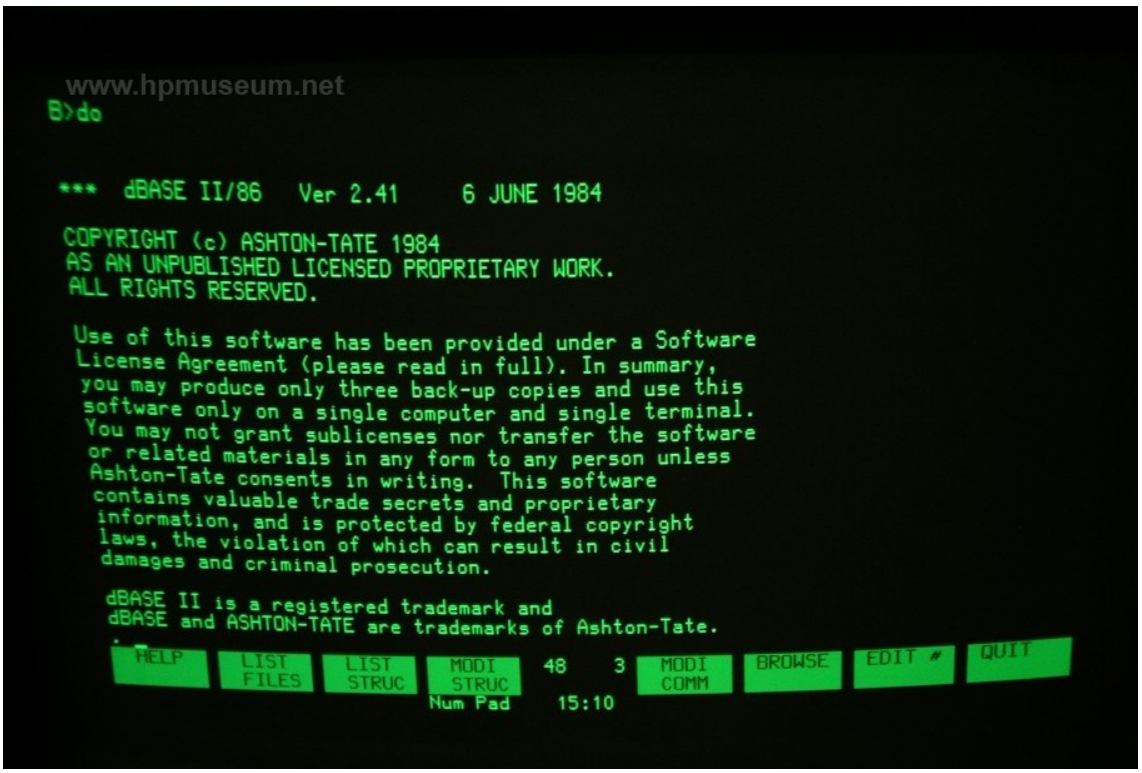
29.1 Introdução

Vimos que o ancestral mais antigo da linguagem Harbour é uma linguagem chamada dBase. O dBase foi um interpretador que surgiu durante a década de 1980, e que fez bastante sucesso[1]. O sucesso foi tanto que acabou originando diversos outros produtos concorrentes baseados na sua estrutura: Clipper (um compilador para código dBase, criado pela Nantucket), FoxPro (versão do dBase desenvolvido pela Microsoft), dbFast, Joiner (versão brasileira semelhante ao Clipper, que permitia que o mesmo código fosse compilado para DOS e para Unix), xBase++ , Flagship e muitos outros. Com tamanha quantidade de produtos, um termo novo acabou surgindo: o xBase. Portanto, xBase significa algo mais do que uma linguagem. O termo remete a uma teia de produtos e serviços baseados na tecnologia dBase.

Quando nós falamos em xBase nós queremos dizer três coisas basicamente:

1. Um formato de armazenamento de dados que revolucionou a indústria da informática. Esse formato caracteriza-se por arquivos com a extensão DBF[2].
2. Uma linguagem de programação que permite a manipulação dos arquivos DBFs.
3. Uma cultura que envolve diversos produtos de diferentes criadores, mas semelhantes na sua estrutura básica: a manipulação de arquivos DBF.

Figura 29.1: Tela do dBase II (1984)



[1] O dBase, por sua vez, é baseado em um produto chamado Vulcan (uma referência ao Sr. Spock, de Star Trek), que surgiu no final da década de 1970.

[2] DBF significa “database file”, algo como “Arquivo de banco de dados”

29.1.1 DBF x SQL

Os arquivos no formato DBF foram os responsáveis pelo imenso sucesso do dBase e dos produtos subsequentes, porém o mercado da informática deu uma guinada importante durante a década de 1990. Essa mudança pode ser resumida em uma expressão bem usada na época: “Computação Cliente x Servidor”. Essa tecnologia nova popularizou um formato bem antigo, que remonta a década de 1960, mas bastante robusto e com fortes raízes acadêmicas: o SQL. Quando o padrão SQL se popularizou (se tornou economicamente acessível) as aplicações baseadas no antigo dBase perderam mercado rapidamente.

29.1.2 Quais as diferenças entre os arquivos DBFs e um banco de dados SQL ?

Vamos tentar responder a essa pergunta nas próximas subseções logo a seguir, mas você só compreenderá realmente quando passar a praticar os exemplos.

A farmácia e a loja de departamentos

Quando você chega em uma farmácia (dessas tradicionais que vendem apenas remédio), você não vai na prateleira pegar o seu remédio. Você primeiro faz o pedido à balconista, para só depois pegar o medicamento.

Figura 29.2: Você faz o pedido, mas o trabalho de obter o produto é feito pela balconista

Já em uma loja de departamentos o procedimento é diferente: você mesmo faz o trabalho que antes era do balconista.

Figura 29.3: Você mesmo faz todo o trabalho

Da mesma forma, quando você trabalha acessando arquivos DBF, você mesmo é quem tem todo o trabalho de montar a consulta e colher os resultados (os dados). O algoritmo de busca é desenvolvido por você, e toda a lógica de programação é sua responsabilidade. Já quando você utiliza um banco de dados SQL, a coisa muda de figura: você "diz" (através da linguagem SQL) o que deseja, e fica aguardando que o banco de dados lhe retorne o resultado, daí a expressão "Computação Cliente x Servidor".

Os arquivos DBFs equivalem a compra em um supermercado e a consulta SQL equivale a uma compra em uma farmácia (que precisa de um balconista, que equivale ao servidor de banco de dados).

Vamos passar para o próximo passo no nosso aprendizado. Imagine agora que você desenvolveu um sistema para uma empresa que possui uma rede local com 400 computadores e 1 servidor de banco de dados. Vamos raciocinar juntos na resolução do seguinte problema: *eu tenho um arquivo com 1 milhão de produtos e eu quero selecionar apenas os produtos que contém a string "cremoso superquick moranguinho".*

- Usando arquivos DBFs : eu teria que **trazer todo o arquivo (1 milhão de registros) para o meu computador, através da rede local ou internet, e montar a minha consulta.**
- Já usando o padrão SQL: eu teria que **perguntar ao banco de dados e ficar aguardando o retorno da pesquisa, que seria bem inferior a 1 milhão de registros**[3].

[3]Na realidade não temos como ter a absoluta certeza de que o número de registros retornados seria substancialmente inferior, mas podemos arriscar um "chute calculado", baseado na crença de que a string "cremoso superquick moranguinho" está presente em um número reduzido de produtos.

Uma rede de computadores possui limitações de tráfego de dados e de velocidade. Qual dos dois métodos acima você acha que é o mais eficiente: o primeiro ou o segundo ?

Se você respondeu que o segundo método é o mais eficiente você acertou. Isso porque o tráfego de informações na rede é bem inferior e isso é melhor para todos os usuários [4]. É bom ressaltar que o Harbour tem uma série de características que minimizam, e muito, a lentidão de uma consulta a arquivos em uma rede local, mas mesmo assim os arquivos DBFs são inferiores aos bancos de dados SQL.

Dica 124

Existem formas de eliminar o problema do tráfico de dados na rede local usando arquivos DBFs ?
Sim, de três formas. **A primeira delas** é recorrendo a produtos adicionais que transformam a tradicional forma de acesso aos arquivos em uma requisição cliente x servidor. Existe um produto chamado ADS (Advantag Database Server) que nada mais é do que uma versão cliente x servidor de arquivos DBFs. É um excelente produto, há alguns anos foi adquirido pela multinacional SAP[a]. O ponto negativo é que trata-se de um produto caro para a nossa realidade. Existe um outro produto chamado de LETOdb, que é uma extensão (o termo técnico é RDD), para Harbour, que realiza a mesma coisa que o ADS. Esse produto é open source e foi desenvolvido especialmente para Harbour[b].
A segunda alternativa é executar a aplicação em um servidor LINUX (preferencialmente) usando um servidor de SSH. Dessa forma, os clientes podem ser windows, linux, mac ou qualquer outro sistema, mas o servidor deverá ser , preferencialmente, um LINUX[c]. Essa forma de acesso é bem antiga e ainda é usada por bancos e também por grandes corporações com diversas filiais em pontos distantes uns dos outros. O único programa que necessita ser instalado na estação do usuário é um cliente de ssh[d], pois todo o processamento é feito no servidor.
Existe ainda uma terceira alternativa que é através de um cliente RDP. Essa tecnologia é muito usada por máquinas Windows e também é bastante usada para distribuir aplicações Harbour.
Existem outras formas não abordadas nessa dica, mas são menos usadas.

[a] Ver um produto, baseado em tecnologia xBase, ser adquirido por uma multinacional pioneira em sistemas de informação é uma prova do vigor das bases de dados xBase.

[b] Maiores informações em `http://www.kresin.ru/en/letodb.html`. O produto pode ser baixado em `https://sourceforge.net/projects/letodb/`. E no fórum PCToledo tem muita informação em português, basta acessar: `http://www.pctoledo.com.br/forum/viewtopic.php?f=33&t=8167`

[c] Existem servidores de SSH para windows, mas a maioria deles possui restrições de licença. O mundo LINUX, que é derivado do UNIX não cobra nada pelo servidor de SSH e já tem décadas de experiência nessa forma de acesso.

[d] O cliente mais conhecido para windows chama-se putty.

Outro quesito importante é a segurança: um arquivo DBF pode ser facilmente copiado e aberto em outro local sem senha alguma de proteção, já o SQL possui

[4] Se você compartilha a internet com muitas pessoas e duas delas baixam arquivos gigantescos, toda a rede fica lenta durante esse processo. A mesma coisa acontece quando você usa arquivos DBFs, com milhares de registros, em aplicações corporativas com muitos usuários concorrentes.

mecanismos adicionais de segurança que protegem o seu acesso (como usuários e senhas). Devemos ser cautelosos em afirmar que um servidor SQL é mais seguro que uma base DBF. É verdade que o SQL é mais seguro pois tem politicas de usuários, senhas, etc. Mas se o invasor conseguir acesso ao servidor SQL e obter privilégios de administrador, então a base SQL poderá ser acessada remotamente de locais externos. Não é bom subestimar o quesito segurança quando for implantar um sistema de informação.

Dica 125

Vale a pena conhecer o banco de dados do Harbour ?
A resposta é “Sim, vale a pena conhecer o funcionamento dos arquivos DBFs”, pelos seguintes motivos:

1. O Harbour possui dezenas de funções que facilitam a utilização do seu banco de dados, o que lhe dará uma maior flexibilidade no desenvolvimento de soluções. Estudar o Harbour e não conhecer o seu banco de dados é desprezar uma gama de recursos que a linguagem pode lhe oferecer.

2. O conhecimento adquirido com o estudo do banco de dados do Harbour (dicionário de dados, índices, etc.) lhe ajudará na compreensão do funcionamento interno de um banco de dados SQL. Isso lhe será de grande ajuda no desenvolvimento de soluções. Você entenderá os princípios que regem o funcionamento interno de uma base de dados. Se você é estudante de sistemas de informação, então você verá na prática o funcionamento interno de alguns recursos que você só viu em sala de aula.

3. Outros aplicativos (MS Excel) e linguagens de programação (PHP) possuem funções que manipulam diretamente os arquivos DBFs. Entender o funcionamento de um banco de dados DBF é um diferencial importante para o domínio de outros programas e linguagens.

Aprender o funcionamento do banco de dados do Harbour vale a pena tanto para lhe dar mais recursos quanto para lhe dar mais entendimento do funcionamento de um banco de dados SQL. Além do mais, não é difícil e o tempo de aprendizado é curto.

29.2 O banco de dados padrão do Harbour

Os arquivos DBFs são adotados pelo Harbour como o formato padrão de banco de dados. Se você leu o parágrafo anterior pode achar que ninguém mais usa esse formato de dados, mas isso não é verdade. Na verdade a seção anterior serviu para lhe esclarecer que existem padrões superiores ao padrão DBF, mas não dissemos que uma aplicação corporativa é impossível de ser desenvolvida usando esse formato. Na verdade, se você fizer uma pesquisa entre os aplicativos comerciais que atualmente estão no mercado, pode até se surpreender com a quantidade de aplicações que utilizam o clássico padrão dBase.

Nas próximas subseções nós aprenderemos mais sobre os arquivos DBFs através de pequenas tarefas práticas, por enquanto iremos listar os arquivos utilizados pelo

banco de dados padrão do Harbour.

- **Arquivos com extensão DBF :** São os arquivos de dados. Esses arquivos contém a estrutura dos registros (veremos o que é isso nas subseções seguintes) e os dados de cada registro.
- **Arquivos com extensão DBT :** São arquivos auxiliares dos arquivos de dados. Contém todos os campos memo do arquivo de dado associado. Se o seu arquivo de dado (DBF) não tiver um campo memo, então não existirão arquivos DBT. Os campos memo são os campos memorandos, e são usados para armazenar extensas porções de strings. Uma carta, por exemplo, pode ser armazenada em um arquivo DBT. [5]
- **Arquivos com extensão NTX :** São arquivos auxiliares criados para permitir acesso ordenado ou direto aos registros de um arquivo de dados. Um único arquivo de dados (DBF) pode ter vários arquivos de índices associados. O Harbour permite vários formatos de arquivos de índices, como arquivos com a extensão NSX e CDX, mas os arquivos com extensão NTX são o formato padrão do Harbour, embora não seja o mais eficiente. Veremos mais sobre índices nas seções seguinte.

29.2.1 A estrutura dos arquivos de dados do Harbour

O banco de dados do Harbour pode ser classificado como um sistema do tipo relacional. Um banco de dados relacional trata os dados como se estivessem organizados em tabelas (algo como uma planilha bem simples). Nessas tabelas as colunas são identificadas por nomes descritivos (os campos) e as linhas são os registros. A tabela fica armazenada em um arquivo com a extensão DBF. O arquivo da figura 29.4 poderia se chamar “clientes.dbf”.

Figura 29.4: Estrutura dos arquivos de dados do Harbour

[5] Os arquivos DBTs ou arquivos Memo não serão abordados nesse livro.

Por exemplo, ainda na figura 29.4, o terceiro **registro** do **arquivo** clientes.dbf tem um **campo** chamado RG, cujo valor é 27489.

Temos, portanto, três conceitos fundamentais :

1. Arquivo de dados : que é um arquivo com a extensão DBF que contém a estrutura e os dados. Internamente um arquivo tem o formato de uma tabela.
2. Registro : É uma linha da tabela. Os registros são numerados automaticamente em ordem crescente.
3. Campo : É a coluna da tabela. O conceito de campo é interessante, pois um campo em particular possui um nome (RG, Nome, Cidade, Telef, etc.), possui um tipo de dado e possui também um tamanho. A totalidade dos campos de um arquivo, com o seu nome, tipo de dado e tamanho, é chamado de "estrutura do registro".

29.2.2 Campos Memo

Um campo tipo Memo foi criado para armazenar longas cadeias de strings. Você pode usar ele para armazenar cartas, memorandos e toda espécie de strings.

Segundo Vidal ,

> um campo Memo pode conter textos longos, como por exemplo , observações ou comentários, específicos para cada registro de dados. Estes textos serão armazenados em um arquivo separado, identificado pelo mesmo nome do arquivo de dados a ele associado e pela extensão (.dbt). [Vidal 1989, p. 50]

29.3 Primeiro passo: criando um arquivo DBF

29.3.1 Criando um arquivo DBF

Arquivos DBFs são acessados diretamente pela sua aplicação. Isso quer dizer que você deve saber a localização exata desses arquivos. É bom criar um local específico para colocar esses arquivos, mas eles podem ficar em qualquer pasta visível, até mesmo na própria pasta onde a sua aplicação está. Nos nossos exemplos, nós iremos assumir que o arquivo DBF ficará no mesmo local onde está a sua aplicação.

Para criar um arquivo DBF você precisa passar as seguintes informações :

1. Nome do campo : os nomes dos campos podem conter letras, números ou símbolo sublinhado (underline). A primeira posição do nome de um campo deve ser uma letra. Outro fato importante: nomes de campos não podem ter espaços em branco. Além disso, o tamanho do nome do campo não pode exceder a 10 caracteres.
2. Tipo de dado : Um campo pode ser do tipo caractere, numérico, data, lógico ou memo.

3. Tamanho do campo : Se o campo for caractere ou numérico o seu tamanho é determinado pelo programador no ato da criação do arquivo DBF. Campos lógicos sempre tem tamanho 1, campos data sempre tem tamanho 8 e campos Memo (ainda não vistos) ficam armazenados em um arquivo externo com a extensão DBT mas sempre ocupam o espaço de 10 caracteres no arquivo DBF.

O Harbour possui uma função chamada de DBCREATE() que é usada para criar um arquivo de dados DBF. Ela possui dois parâmetros, o primeiro é o nome do arquivo e o segundo é um array com as definições dos campos.

Descrição sintática 33

1. Nome : DBCREATE()
2. Classificação : função.
3. Descrição : Cria um arquivo de dados a partir de um array que possui a estrutura do arquivo.
4. Sintaxe

```
DBCREATE( <cArqDados>, <aEstrut> ,;
          [<cDriver>], [<lOpen>],;
          [<cAlias>]) -> NIL
```

5. Parâmetros
   - <cArqDados> : O nome do arquivo
   - <aEstrut> : Uma matriz com a estrutura do arquivo a ser criado
   - [<cDriver>] : Driver RDD
   - [<lOpen>] : Parâmetros de abertura[a]
   - [<cAlias>] : alias do arquivo

Fonte : https://vivaclipper.wordpress.com/2014/01/17/dbcreate/

[a]Se NIL o arquivo não é aberto, mas apenas criado (esse é o padrão), .T. se for para criar e abrir em nova área e .F. se for para criar e abrir na área corrente

O exemplo a seguir (listagem 29.1) lhe ajudará a entender como isso funciona.

Listagem 29.1: Criando um arquivo DBF
Fonte: codigos/dbf01.prg

```
PROCEDURE Main
LOCAL aStruct := {  { "NOME" , "C" , 50, 0 },;
                    { "NASCIMENTO" , "D" , 8 , 0 },;
                    { "ALTURA" , "N" , 4 , 2 },;
                    { "PESO" , "N" , 6 , 2 } }
```

```
    IF .NOT. FILE( "paciente.dbf" )
       DBCREATE( "paciente" , aStruct )
       ? "Arquivo de pacientes criado."
    ENDIF

RETURN
```

.:Resultado:.

```
Arquivo de pacientes criado.
```

IMPORTANTE : é essencial que você digite e execute esse programa para que o arquivo pacientes.dbf seja criado. Esse arquivo será usado na próxima seção, que utilizará o utilitário HBRUN.

O array passado para DBCREATE() deve conter a estrutura do arquivo a ser criado. Note que o array é multidimensional (duas dimensões). Cada elemento do array é também um array com os atributos de cada campo. Os atributos são :

1. Nome do campo : segundo as regras já vista.
2. Tipo de dado : Os valores válidos são “C” para caractere, “N” para numérico, “L” para lógico, “D” para data ou “M” para memo.
3. Tamanho : Se o tipo de dado for lógico, sempre coloque 1. Se o tipo for data sempre coloque 8 e se o tipo for memo, sempre coloque 10. Quando o campo for caractere ou numérico você deve informar o tamanho do campo de acordo com as suas necessidades.
4. Decimais : Se o tipo for numérico, informe o número de casas decimais. Para os demais tipos sempre informe 0 (zero).

Se por acaso o arquivo informado já existir então ele será sobreposto pela nova estrutura que você informou a DBCREATE(). Portanto tenha cuidado, se por acaso você tiver um arquivo com 1000 registros, por exemplo, e for usar a função DBCREATE() informando o mesmo nome desse tal arquivo, então um novo arquivo será criado e os 1000 registros serão perdidos.

Dica 126

Para evitar que a função DBCREATE() apague arquivos existentes, você pode usar a função FILE() para verificar a existência, ou não, do arquivo a ser criado.

```
    IF .NOT. FILE( "clientes.dbf" ) // Se o arquivo não existir
        // Crie o arquivo com DBCREATE
        DBCREATE( "clientes.dbf" , aEstrutura )
    ENDIF
```

29.4 O utilitário HBRUN

O Harbour disponibiliza para você um utilitário chamado HBRUN para a execução de operações no modo interativo. Até agora, nós trabalhamos o modo programável da linguagem Harbour, pois esse é o modo usado para a criação de programas. Todavia, o Harbour pode ser usado também no modo interativo, que é uma forma de se obter imediatamente o resultado de um comando ou operação. O modo interativo necessita de um “ambiente” ou local onde esses comandos possam ser executados, e o nome do utilitário chama-se HBRUN. Vamos, através de pequenos exemplos, lhe mostrar o funcionamento desse aplicativo, acompanhe nas subseções a seguir.

29.4.1 Executando o utilitário HBRUN

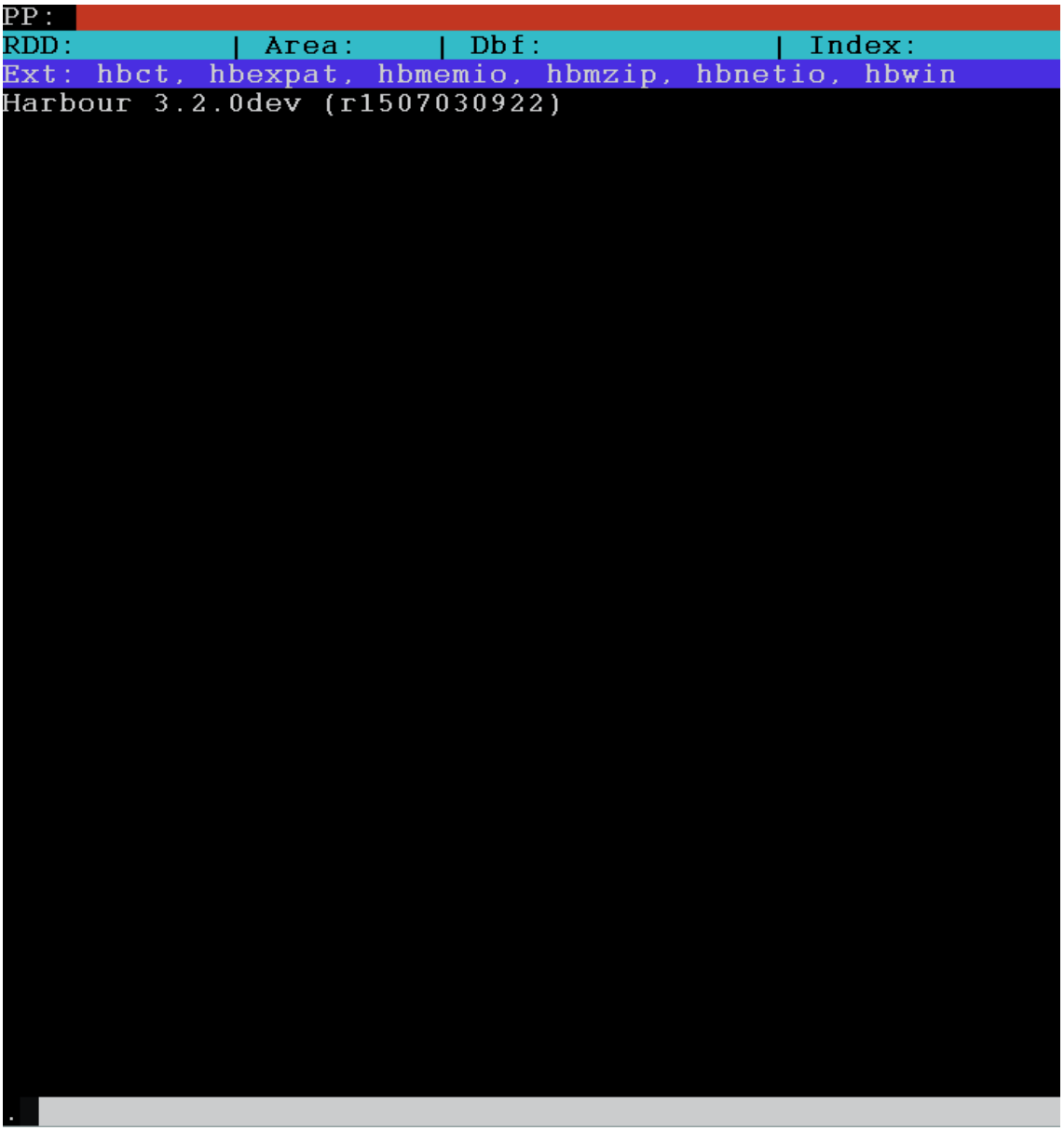
Para executar o utilitário HBRUN simplesmente digite e tecle enter :

.:Resultado:.

```
hbrun
```

O programa hbrun irá ocupar a tela inteira, conforme a figura 29.5 a seguir :

Figura 29.5: A tela inicial do HBRUN



Qual a importância do utilitário HBRUN para o nosso aprendizado ?

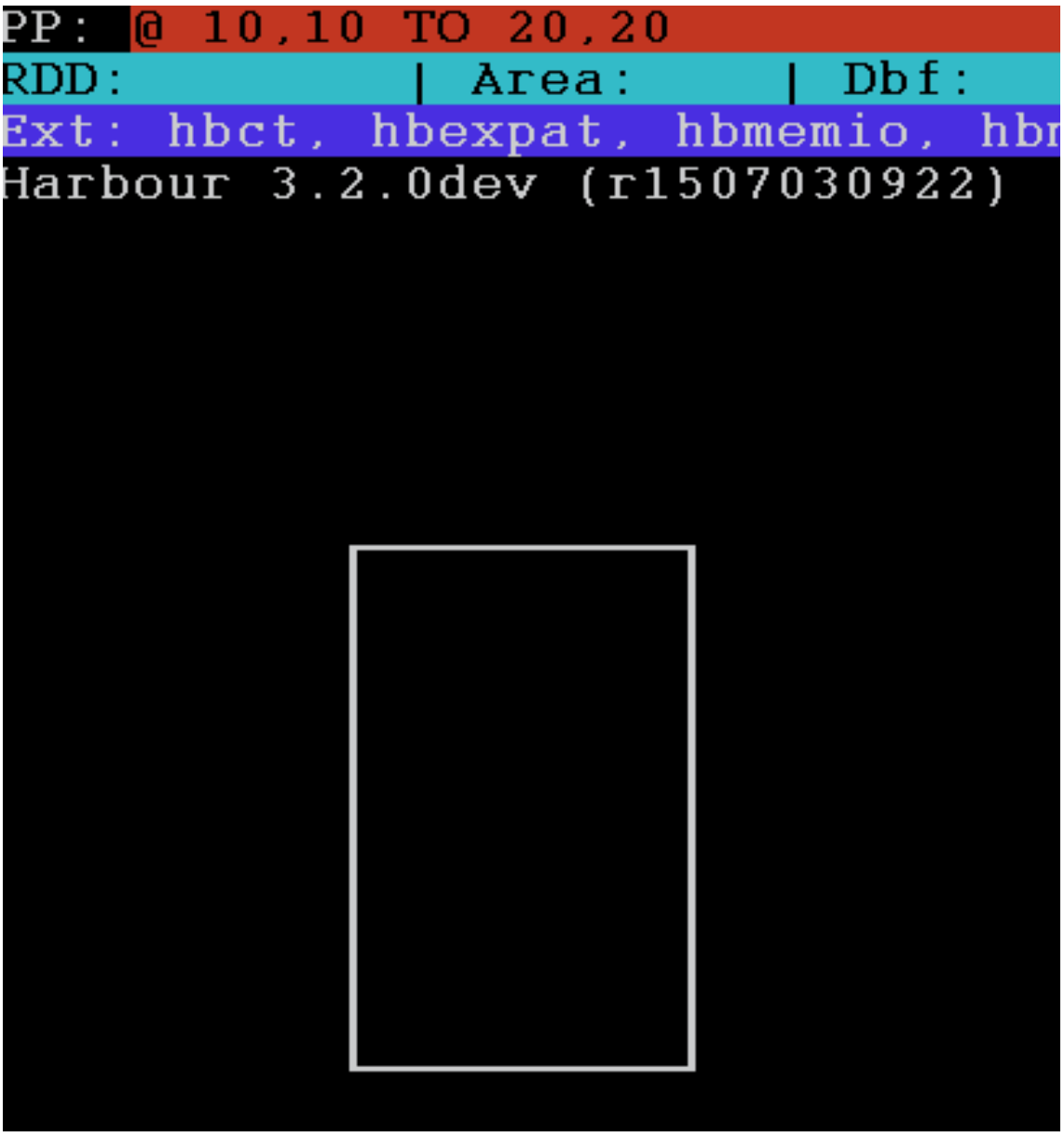
O Harbour é uma linguagem de programação que possui o próprio banco de dados, e um utilitário como o HBRUN é muito importante para o aprendizado dos comandos que operam sobre os arquivos DBFs, pois possibilita que o resultado de um comando seja visualizado imediatamente após a sua execução. Nós poderíamos ter usado o HBRUN como uma ferramenta para o aprendizado nos capítulos anteriores. mas preferimos usar somente agora com o banco de dados do Harbour, pois ele proporciona uma velocidade bem maior no aprendizado, além do mais, o HBRUN pode ser usado no seu dia-a-dia de programador, na manipulação direta de arquivos DBFs, sem precisar ficar digitando programas para realizar operações diversas no banco de dados.

Note que na parte inferior do utilitário HBRUN existe uma barra horizontal branca. Quando você for digitar os seus comandos, observe que eles aparecem nessa barra horizontal. Por exemplo, digite o comando abaixo:

```
@ 10,10 TO 20,20
```

Esse comando foi visto no capítulo que estuda a interface modo texto do Harbour. Note que após você teclar ENTER, o resultado aparece imediatamente.

Figura 29.6: Executando um comando no HBRUN



Agora nós usaremos o HBRUN para aprender a manipular arquivos DBFs. Primeiramente é importante que o arquivo “paciente.dbf” (criado através da listagem 29.1) esteja na pasta corrente.

29.4.2 Abrindo o arquivo “paciente.dbf”

Para executar qualquer operação em um arquivo de dados nós precisamos primeiro “abrir” o arquivo. Uma das formas de se abrir um arquivo é através do comando USE,

conforme o fragmento abaixo. Note que você não precisa informar a extensão (.dbf) para o comando, pois ele já pressupõe isso.

```
USE paciente // Abre o arquivo paciente.dbf
```

Digite, no HBRUN, o comando “USE paciente”. Note que a barra horizontal superior, de cor ciano (ou azul claro) , passa a exibir algumas informações sobre o arquivo que está aberto.

Figura 29.7: A barra de informações do arquivo DBF aberto.

Vamos analisar essas informações uma por uma:

1. RDD: DBFNTX - Informa o tipo de índice que será criado, caso a gente venha a indexar o arquivo. Não se preocupe com isso, veremos os índices mais adiante com detalhes.
2. Área : 1 - Quando você abre um arquivo, automaticamente o Harbour atribui a ele um identificador chamado de “identificador de área de trabalho”. Cada arquivo aberto ocupa uma área de trabalho. Você pode estar pensando que a área de trabalho é criada logo após a abertura do arquivo, mas isso não é verdade. A área de trabalho existe antes da abertura do arquivo, ou seja, existem áreas de trabalho que estão ocupadas e áreas de trabalho que estão desocupadas. Outra observação importante é que uma área de trabalho, quando ocupada, pode conter apenas um arquivo. A consequência prática disso é que você deve selecionar a área antes do arquivo ser aberto. A única exceção é quando for abrir o primeiro arquivo do seu programa, pois ele sempre ocupará a primeira área de trabalho (a área número 1).
3. Dbf: PACIENTE - É o nome do arquivo DBF.
4. Index: (Não foi preenchido) - É o índice que está ativo no momento. Não se preocupe com isso, veremos os índices mais adiante com detalhes.
5. `1/ 0` - Todo arquivo de dados possui um “ponteiro” que indica qual o registro que está ativo no momento. O número 1 indica que o “ponteiro” está no primeiro registro. O número zero indica a quantidade de registros do arquivo. Como ele acabou de ser criado, nós temos zero registro no total. Isso é realmente confuso: como um arquivo tem zero registros no total e o ponteiro indica que estamos no primeiro registro ? Isso acontece porque o arquivo DBF possui um registro chamado de “registro-fantasma”. Ele não contém dados, mas é o responsável pela geração dos “registros reais” a medida que eles forem surgindo.

29.4.3 Criando um registro em branco no arquivo “paciente.dbf”

Para podermos gravar dados em um arquivo DBF nós precisamos, primeiramente, criar um registro em branco. Para criar esse registro em branco digite o comando do fragmento abaixo:

```
APPEND BLANK
```

O comando APPEND BLANK está descrito logo abaixo.

Descrição sintática 34

1. Nome : APPEND BLANK.
2. Classificação : comando.
3. Descrição : Adiciona um registro vazio ao arquivo de dados corrente.
4. Sintaxe

```
APPEND BLANK
```

Fonte : [Nantucket 1990, p. 4-17]

Logo após você digitar ENTER um registro em branco é criado. Note que a barra de status do arquivo DBF (que nós vimos na seção anterior) tem uma sutil mudança. Antes ele mostrava `1/ 0` e agora ele mostra `1/ 1`.

Figura 29.8: Barra de status do DBF após APPEND BLANK

Os novos valores de campos são inicializados com valores vazios para cada tipo de dado. A campos caractere são atribuídos espaços, campos numéricos são inicializados com zero, campos lógicos são inicializados com falso (.f.), a campos data são atribuídos CTOD(”’), e campos memo são deixados vazios.

29.4.4 Gravando dados no arquivo “paciente.dbf”

Para gravarmos dados usamos o comando REPLACE, conforme o exemplo do fragmento abaixo :

```
REPLACE NOME WITH "Samuel Armstrong"
```

Pronto, o nosso primeiro registro criado com APPEND BLANK tem o seu campo nome preenchido com a string "Samuel Armstrong".

Resumindo: comando APPEND BLANK cria um registro com todos os campos em branco, e o comando REPLACE serve para gravar os dados nesse registro.

Descrição sintática 35

1. Nome : REPLACE.
2. Classificação : comando.
3. Descrição : Atribui novos valores a variáveis campo.
4. Sintaxe

```
REPLACE <idCampo> WITH <exp> [, <idCampo> WITH <exp2] ...
     [<abrangência>] [WHILE <lCondição>] [FOR <lCondição]
```

Fonte : [Nantucket 1990, p. 4-83]

29.4.5 Exibindo dados do arquivo “paciente.dbf”

Quando você digitou o comando REPLACE você não obteve retorno algum. Nem mesmo a barra horizontal que mostra informações sobre o DBF exibiu algo novo. Agora nós iremos exibir o que foi inserido.

Existem várias formas para mostrar o conteúdo de um campo em um banco de dados, mas todas elas possuem um ponto em comum: “um campo do banco de dados comporta-se como se fosse uma variável”. Portanto, uma forma simples de exibir o conteúdo do que foi gravado é através do comando “?”. Portando, digite o seguinte fragmento no HBRUN.

```
? NOME
```

Note que no canto superior direito da tela, logo abaixo das barras horizontais de informação aparece a string:

.:Resultado:.

```
Samuel Armstrong
```

Você também pode usar o comando “?” para imprimir os demais campos, conforme o fragmento a seguir.

```
? NOME, NASCIMENTO, PESO, ALTURA
```

O HBRUN irá exibir:

.:Resultado:.

```
Samuel Armstrong                          -  -     0.00 0.00
```

Vamos atribuir um peso ao paciente. Para isso faça:

```
REPLACE PESO WITH 90
```

Dica 127

O HBRUN guarda a lista dos últimos comandos digitados, assim você não precisa ficar redigitando comandos. Apenas digite a seta para cima e o último comando que você digitou irá aparecer. Repita o procedimento até que o comando que você deseja repetir apareça. Você pode alterar o comando (usando as setas de movimentação) se quiser.

Use a seta para cima para que o fragmento abaixo reapareça, depois tecle ENTER.

```
? NOME, NASCIMENTO, PESO, ALTURA
```

Note que o resultado agora é outro.

.:Resultado:.

```
Samuel Armstrong                          -  -    90.00 0.00
```

Para inserir a altura faça da mesma forma:

```
REPLACE ALTURA WITH 1.9
```

A data possui um detalhe especial, você precisa alterar o formato da data (no nosso exemplo nós usamos a função CTOD(), mas nós já vimos no capítulo sobre tipos de dados que existem outras formas de se inicializar uma data).

```
SET DATE BRITISH
REPLACE NASCIMENTO WITH CTOD("26/08/1970")
```

É bom ressaltar que o comando “SET DATE BRITISH” só precisa ser executado uma vez durante a seção com o HBRUN. Agore, de novo, use a seta para cima para que o fragmento abaixo reapareça, depois tecle ENTER.

```
? NOME, NASCIMENTO, PESO, ALTURA
```

.:Resultado:.

```
Samuel Armstrong                       26/08/1970  90.00 1.90
```

Pronto, acabamos de inserir o nosso primeiro registro em um banco de dados padrão dBase. Note que eu posso alterar a informação que foi digitada usando o comando REPLACE. Se eu digitar o fragmento abaixo :

```
REPLACE NOME WITH "Stevenson da Silva"
```

Eu estarei alterando o campo NOME. A string "Samuel Armstrong"não está mais lá, agora o campo NOME contém a string "Stevenson da Silva".

Novamente, seta para cima para confirmar:

```
? NOME, NASCIMENTO, PESO, ALTURA
```

Prática número 30

Insira mais alguns registros no arquivo “paciente.dbf”[a]. Para inserir um novo registro não se esqueça de usar o comando “APPEND BLANK” antes de usar a sequência de comandos REPLACEs. Coloque nomes variados, pesos e alturas diversas. Quando for inserir a data de nascimento utilize outras formas de inserção: função STOD() e o padrão 0d, além da nossa velha conhecida função CTOD().

[a]Não precisa inserir muitos registros, basta inserir uns dois a mais.

29.4.6 Vendo o conteúdo de um arquivo DBF com o comando LIST

Agora vamos abrir o programa HBRUN e listar todos os seus registros.

.:Resultado:.

```
hbrun
```

Dentro do utilitário, abra o arquivo de pacientes através do seguinte comando:

.:Resultado:.

```
use paciente
```

Supondo que você inseriu alguns registros, conforme nós aconselhamos na prática passada, o seu arquivo de pacientes deve estar com alguns registros. Para listar o conteúdo do arquivo use o comando LIST seguido dos campos que você deseja ver.

.:Resultado:.

```
list NOME, NASCIMENTO, PESO, ALTURA
```

A listagem deve se parecer com o fragmento a seguir, só que conterá os seus pacientes (é claro) :

```
    1  Samuel Armstrong                1970-08-26  90.00 1.90
    2  Charles Haddon Spurgeon         1934-12-12  87.00 1.50
```

O comando LIST, como o seu nome sugere, é usado para listar rapidamente os dados de um arquivo DBF.

Descrição sintática 36

1. Nome : LIST
2. Classificação : comando.
3. Descrição : Lista os registros para o console.
4. Sintaxe

```
LIST <lista de expressões> [TO PRINTER]
       [TO FILE <xcArquivo>]
       [<abrangência>]
       [WHILE <lCondição>]
       [FOR <lCondição>]
       [OFF]
```

Fonte : [Nantucket 1990, p. 4-66]

Supondo que você realmente está tendo o seu primeiro contato com os arquivos DBFs, é natural que haja um certo desconforto ao ler a descrição do comando LIST logo acima, contudo não se preocupe, pois nós iremos detalhar todas cláusulas do comando uma por uma. Nós só não fizemos isso ainda porque precisamos deixar claro que todas essas cláusulas existem em muitos outros comandos do Harbour, e é interessante aprender esses comandos e treinar com alguns exemplos. No final, essas cláusulas já não serão estranhas para você.

Vamos começar, então: o Harbour possui um grupo de comandos que foram derivados do antigo dBase. Esses comandos manipulam arquivos DBFs usando todos uma sintaxe muito parecida. **Todos eles** possuem, pelo menos, alguma das seguintes cláusulas abaixo[6] :

1. OFF : É uma cláusula opcional para apresentação ou não do número dos registros na tela.
2. <lista de expressões> : É uma lista obrigatória que pode conter qualquer expressão (funções, expressões numéricas e tipos de dados literais) ou **nomes de campos** do arquivo em uso, separados por vírgula.
3. FOR e WHILE : São cláusulas especiais, para a seleção de registros a serem processados.
4. <lCondição> : Note que ela pode aparecer na cláusula FOR ou na cláusula WHILE. É uma expressão lógica usada para especificar a seleção dos registros a serem processados.

[6] Adaptado de [Vidal 1989, p. 65]

5. TO PRINT/TO FILE : São cláusulas opcionais alternativas para direcionar a saída do comando para a impressora ou para um arquivo em disco, ao invés da tela.
6. <xcArquivo> : Caso a cláusula TO FILE seja utilizada o nome do arquivo a ser gravado deverá ser especificado pelo usuário.
7. <Abrangência> : Como operamos sobre um conjunto de dados (chamado arquivo), nós podemos restringir a abrangência da ação desse comando. Os tipos de abrangência são :
   - ALL : Todos os registros.
   - NEXT <n> : Apenas os <n> registros seguintes, a partir do atual.
   - RECORD <n> : Apenas o registro <n>.
   - REST : Os registros restantes.

Agora nós iremos aplicar essas cláusulas em exemplos usando o comando LIST que nós já conhecemos, e quando nós formos abordar os demais comandos, você não sentirá dificuldade em entender as suas cláusulas, pois estão presentes no comando LIST. Vamos utilizar o arquivo "paciente.dbf" para ilustrarmos o comando LIST, mas como nós não temos muitos dados dentro dele, vamos ter que inserir vários dados. Na pasta prática você pode encontrar um arquivo chamado paciente.prg. Esse arquivo irá inserir vários dados fictícios no nosso arquivo de pacientes.

Prática número 31

Vá para a pasta prática.

1. Se tiver um arquivo chamado paciente**.dbf**, você deve apagá-lo[a].
2. Agora compile o fonte que está lá, chamado paciente**.prg**.
3. Quando terminar execute o programa paciente**.exe**.
4. Execute o **hbrun**
5. Abra o arquivo paciente.
6. Liste todos os registros desse arquivo.

[a] É importante que você apague o seu arquivo, caso ele exista para que a gente possa ter a certeza de que tanto o meu arquivo quanto o seu arquivo tenham os mesmos dados. Para apagar um arquivo estando dentro do Prompt de Comando faça : *del paciente.dbf* (estamos pressupondo que a pasta onde você executou o comando é a mesma onde o arquivo está.

Vamos agora efetuar algumas operações com o comando LIST. **É importante ressaltar que você deve estar na mesma pasta onde o seu arquivo paciente.dbf foi criado.**

O conteúdo do arquivo paciente.prg que esta na pasta codigos está reproduzido abaixo :

Listagem 29.2: Nosso arquivo para prática e aprendizado (ele está na pasta codigos, portanto você não precisar digitar nada. É só compilar e executar esse arquivo para

gerar os dados para nossos testes.)
Fonte: codigos/paciente.prg

```
PROCEDURE Main
LOCAL aStruct := {  { "NOME" , "C" , 50, 0 },;
                    { "NASCIMENTO" , "D" , 8 , 0 },;
                    { "ALTURA" , "N" , 4 , 2 },;
                    { "PESO" , "N" , 6 , 2 } }


    IF .NOT. FILE( "paciente.dbf" )
       DBCREATE( "paciente" , aStruct )
       ? "Arquivo de pacientes criado."
    ENDIF

    USE paciente // Abrindo o arquivo paciente.dbf
    IF .NOT. USED()
       ? "O arquivo 'paciente.dbf' não foi aberto com sucesso."
       ? "A operaçäo será abortada"
       QUIT
    ENDIF

    ? "Inserindo dados no arquivo paciente.dbf"
      APPEND BLANK
      REPLACE NOME WITH 'Armor'
      REPLACE NASCIMENTO WITH STOD('19941216')
      REPLACE ALTURA WITH 1.55
      REPLACE PESO WITH 73
      APPEND BLANK
      REPLACE NOME WITH 'Arneles'
      REPLACE NASCIMENTO WITH STOD('19811215')
      REPLACE ALTURA WITH 1.51
      REPLACE PESO WITH 63
      APPEND BLANK
      REPLACE NOME WITH 'Artemisa'
      REPLACE NASCIMENTO WITH STOD('19921213')
      REPLACE ALTURA WITH 1.70
      REPLACE PESO WITH 114
      APPEND BLANK
      REPLACE NOME WITH 'Artemisia'
      REPLACE NASCIMENTO WITH STOD('19721212')
      REPLACE ALTURA WITH 1.95
      REPLACE PESO WITH 60
      APPEND BLANK
      REPLACE NOME WITH 'Ashley'
      REPLACE NASCIMENTO WITH STOD('19881123')
      REPLACE ALTURA WITH 1.52
      REPLACE PESO WITH 88
      APPEND BLANK
      REPLACE NOME WITH 'Kesia'
      REPLACE NASCIMENTO WITH STOD('19581223')
```

```
REPLACE ALTURA WITH 1.86
REPLACE PESO WITH 108
APPEND BLANK
REPLACE NOME WITH 'Astride'
REPLACE NASCIMENTO WITH STOD('19861230')
REPLACE ALTURA WITH 1.58
REPLACE PESO WITH 65
APPEND BLANK
REPLACE NOME WITH 'Gertrudes'
REPLACE NASCIMENTO WITH STOD('19921018')
REPLACE ALTURA WITH 1.73
REPLACE PESO WITH 96
APPEND BLANK
REPLACE NOME WITH 'Atlante'
REPLACE NASCIMENTO WITH STOD('19871028')
REPLACE ALTURA WITH 1.88
REPLACE PESO WITH 112
APPEND BLANK
REPLACE NOME WITH 'Atlantida'
REPLACE NASCIMENTO WITH STOD('19881026')
REPLACE ALTURA WITH 1.99
REPLACE PESO WITH 109
APPEND BLANK
REPLACE NOME WITH 'Audria'
REPLACE NASCIMENTO WITH STOD('19921223')
REPLACE ALTURA WITH 1.98
REPLACE PESO WITH 91
APPEND BLANK
REPLACE NOME WITH 'Augustos'
REPLACE NASCIMENTO WITH STOD('19681217')
REPLACE ALTURA WITH 1.60
REPLACE PESO WITH 104
APPEND BLANK
REPLACE NOME WITH 'Aurete'
REPLACE NASCIMENTO WITH STOD('19641030')
REPLACE ALTURA WITH 1.99
REPLACE PESO WITH 76
APPEND BLANK
REPLACE NOME WITH 'Aurelio'
REPLACE NASCIMENTO WITH STOD('19591130')
REPLACE ALTURA WITH 1.71
REPLACE PESO WITH 64
APPEND BLANK
REPLACE NOME WITH 'Bianca'
REPLACE NASCIMENTO WITH STOD('19561211')
REPLACE ALTURA WITH 1.65
REPLACE PESO WITH 65
APPEND BLANK
REPLACE NOME WITH 'Bianor'
REPLACE NASCIMENTO WITH STOD('19681030')
REPLACE ALTURA WITH 1.61
```

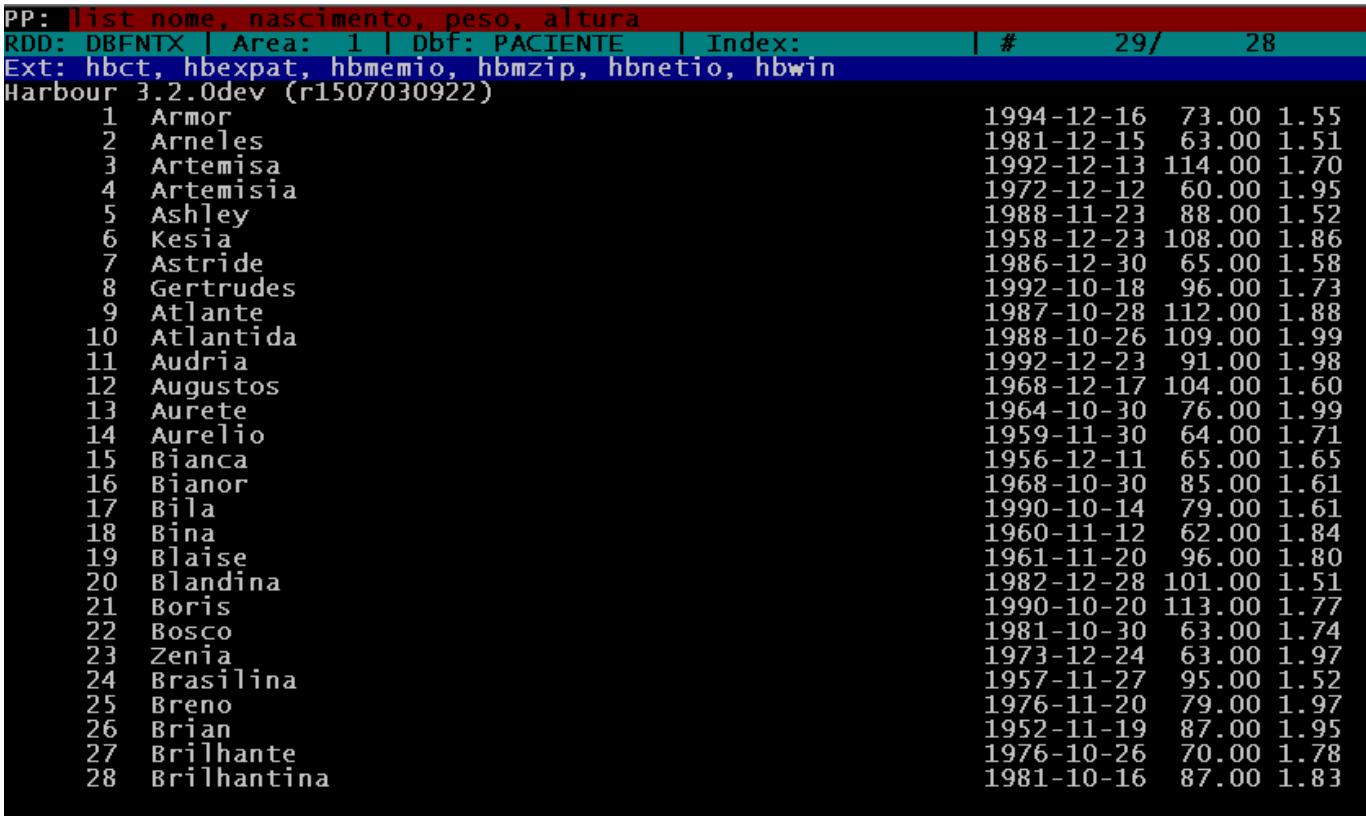
```
REPLACE PESO WITH 85
APPEND BLANK
REPLACE NOME WITH 'Bila'
REPLACE NASCIMENTO WITH STOD('19901014')
REPLACE ALTURA WITH 1.61
REPLACE PESO WITH 79
APPEND BLANK
REPLACE NOME WITH 'Bina'
REPLACE NASCIMENTO WITH STOD('19601112')
REPLACE ALTURA WITH 1.84
REPLACE PESO WITH 62
APPEND BLANK
REPLACE NOME WITH 'Blaise'
REPLACE NASCIMENTO WITH STOD('19611120')
REPLACE ALTURA WITH 1.80
REPLACE PESO WITH 96
APPEND BLANK
REPLACE NOME WITH 'Blandina'
REPLACE NASCIMENTO WITH STOD('19821228')
REPLACE ALTURA WITH 1.51
REPLACE PESO WITH 101
APPEND BLANK
REPLACE NOME WITH 'Boris'
REPLACE NASCIMENTO WITH STOD('19901020')
REPLACE ALTURA WITH 1.77
REPLACE PESO WITH 113
APPEND BLANK
REPLACE NOME WITH 'Bosco'
REPLACE NASCIMENTO WITH STOD('19811030')
REPLACE ALTURA WITH 1.74
REPLACE PESO WITH 63
APPEND BLANK
REPLACE NOME WITH 'Zenia'
REPLACE NASCIMENTO WITH STOD('19731224')
REPLACE ALTURA WITH 1.97
REPLACE PESO WITH 63
APPEND BLANK
REPLACE NOME WITH 'Brasilina'
REPLACE NASCIMENTO WITH STOD('19571127')
REPLACE ALTURA WITH 1.52
REPLACE PESO WITH 95
APPEND BLANK
REPLACE NOME WITH 'Breno'
REPLACE NASCIMENTO WITH STOD('19761120')
REPLACE ALTURA WITH 1.97
REPLACE PESO WITH 79
APPEND BLANK
REPLACE NOME WITH 'Brian'
REPLACE NASCIMENTO WITH STOD('19521119')
REPLACE ALTURA WITH 1.95
REPLACE PESO WITH 87
```

```
      APPEND BLANK                                                    152
      REPLACE NOME WITH 'Brilhante'                                   153
      REPLACE NASCIMENTO WITH STOD('19761026')                        154
      REPLACE ALTURA WITH 1.78                                        155
      REPLACE PESO WITH 70                                            156
      APPEND BLANK                                                    157
      REPLACE NOME WITH 'Brilhantina'                                 158
      REPLACE NASCIMENTO WITH STOD('19811016')                        159
      REPLACE ALTURA WITH 1.83                                        160
      REPLACE PESO WITH 87                                            161
                                                                      162
                                                                      163
RETURN                                                                164
```

Listando o conteúdo

Essa é fácil (nós já fizemos em outra ocasião). Execute o utilitário HBRUN, abra o arquivo "paciente.dbf" e execute o comando LIST NOME, NASCIMENTO, PESO, ALTURA. Confira na imagem abaixo:

Figura 29.9: Listagem do conteúdo de um arquivo



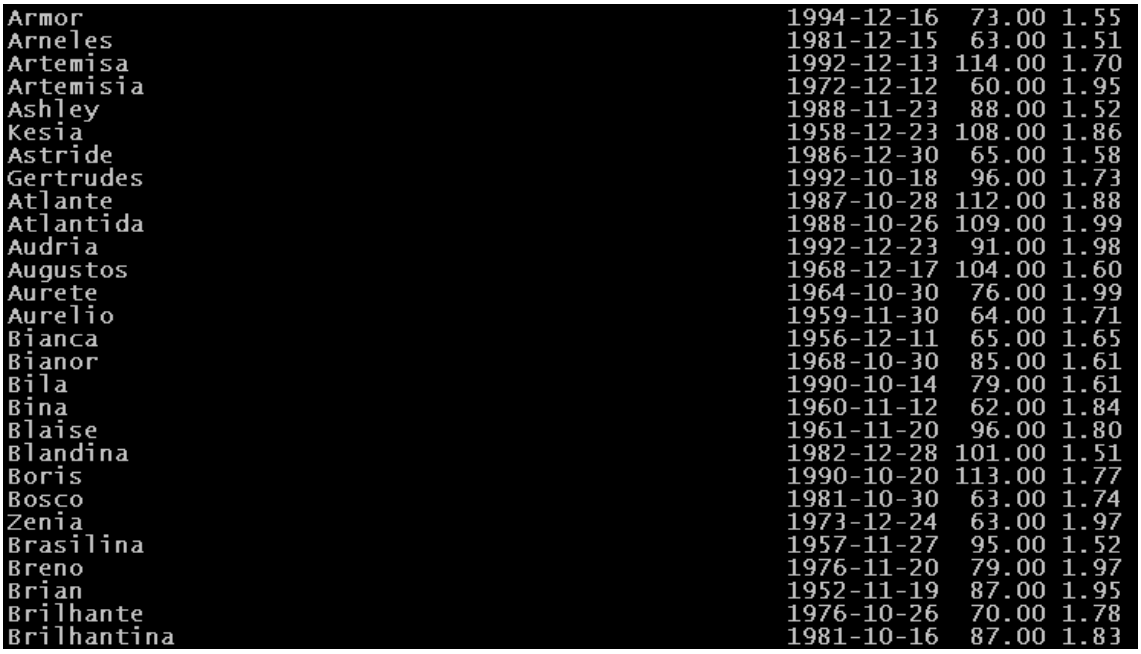
Os comandos digitados foram:

```
use paciente
list nome, nascimento, peso, altura
```

A cláusula OFF : listando o conteúdo sem exibir o número do registro

Agora, vamos listar o mesmo conteúdo, mas sem exibir o número do registro. Note que usamos a cláusula OFF. Acompanhe na imagem:

Figura 29.10: Listagem do conteúdo de um arquivo sem o número do registro



Pressupondo que você já executou (no exemplo anterior) o comando USE para abrir o arquivo, o comando que foi digitado está reproduzido a seguir:

```
list nome, nascimento, peso, altura OFF
```

Alterando a abrangência do comando

Existem casos em que o comando LIST, como usado até agora, produz resultados indesejáveis. Imagine que o nosso arquivo de pacientes possua dez mil registros. Será que exibir todos os dez mil registros de uma vez só é uma boa alternativa ? Claro que não. Por isso você pode alterar a abrangência do comando LIST para que ele possa operar apenas em um subconjunto de dados. Vamos repetir os tipos de abrangências :

- ALL : Todos os registros.
- NEXT <n> : Apenas os <n> registros seguintes, a partir do atual.
- RECORD <n> : Apenas o registro <n>.
- REST : Os registros restantes.

Existe uma abrangência que é a padrão (default) para o comando LIST, que é a abrangência ALL. Isso quer dizer que se eu não digitar a abrangência, o comando LIST irá exibir todos os registros do arquivo. Nos próximas exemplos nós iremos alterar a abrangência do comando LIST **e iremos aprender na prática a importância do "ponteiro do registro"**, não perca.

Abrangência NEXT

Vamos exibir apenas os cinco primeiros registros do arquivo paciente.dbf. Mas antes vamos limpar a tela com o comando CLS.

```
CLS
```

Agora use a "seta para cima" para recuperar o último comando LIST digitado e altere-o, deixando conforme o fragmento a seguir:

```
list nome, nascimento, peso, altura NEXT 5
```

O comando acima deve exibir os 5 registros seguintes.

Mas, para a nossa surpresa, nada aconteceu[7]. Engraçado... nós temos certeza que no arquivo tem 28 registros gravados, porque eles não apareceram ? O que aconteceu ?

Você está lembrado do ponteiro do arquivo ? Olhe para a última informação, a barra de informações do DBF, veja onde está o ponteiro do arquivo.

Figura 29.11: Ponteiro logo após o comando *"LIST nome, nascimento, peso, altura OFF"*

Dica 128

Não esqueça que o arquivo DBF possui um registro a mais (com as definições dos campos) chamado de "registro fantasma", por isso quando você "ultrapassa" o final do arquivo você "se posiciona" automaticamente nesse dito registro. A consequência prática disso é que você pode receber informações estranhas do aplicativo HBRUN, como essa que você recebeu, informando que você está no registro 29 de um total de 28. Esse é o "registro fantasma".

É importante entender como isso foi acontecer. Vamos explicar passo-a-passo: quando você digitou o comando *"LIST nome, nascimento, peso, altura OFF"* na seção anterior (cláusula OFF, sem a cláusula NEXT), as seguintes operações foram realizadas:

1. O ponteiro do arquivo foi deslocado para o primeiro registro.
2. Os campos que você solicitou que fossem exibidos foram mostrados na linha de baixo: *nome, nascimento, peso, altura*.
3. O ponteiro do arquivo foi deslocado para o próximo registro.
4. As operações dos passos 2 e 3 se repetiram até que o registro fantasma foi encontrado. Lembre-se que o registro fantasma serve também para lhe dizer que não existem mais dados para serem exibidos (chegou no final do arquivo).

[7]Nós estamos pressupondo que você acabou de digitar o comando *"LIST nome, nascimento, peso, altura OFF"* que foi visto na seção anterior.

5. Quando o registro fantasma é encontrado o comando LIST encerra a sua operação, mas ele não altera a posição do ponteiro do arquivo. Ou seja, eu continuo no tal registro fantasma.

Quando eu fui tentar exibir os próximos cinco registros, eu não consegui, pois não havia mais registros para serem exibidos.

E agora ? Como eu faço para exibir apenas os cinco primeiros registros ? A resposta é simples: você precisa deslocar o ponteiro para o primeiro registro antes de exibir os cinco primeiros. E como eu faço isso ? Através de um comando chamado GO ou GOTO[8]. O comando GO desloca o ponteiro do registro para um determinado registro. Veja a sua sintaxe a seguir:

Descrição sintática 37

1. Nome : GO[TO]
2. Classificação : comando.
3. Descrição : move o ponteiro de registro para um registro específico.
4. Sintaxe

```
 GO[TO] <nRegistro>|BOTTOM|TOP

<nRegistro> especifica o número do registro destino.
BOTTOM especifica o último registro na área de trabalho.
TOP especifica o último registro na área de trabalho.

     Lembre-se: A barra ``|'' na nossa sintaxe serve para
     informar  que você deve usar
     uma, e apenas uma, das três alternativas.
     Exemplos de uso seriam :

     GO 3 // Vai para o terceiro registro
     GO TOP // Vai para o primeiro registro
     GO BOTTOM // Vai para o último registro
```

Fonte : [Nantucket 1990, p. 4-54]

Então, para que o comando passe a exibir arquivos você tem que fazer conforme o fragmento abaixo :

```
GO TOP // ou GO 1
list nome, nascimento, peso, altura NEXT 5
```

[8]Não confunda essa comando com o "famigerado" GOTO que foi abordado no capítulo sobre estruturas de repetição. Esse comando do Harbour é diferente.

Dica 129

GO TOP é a mesma coisa que GO 1 ? Nesse caso sim, mas existem casos (ainda não vistos) em que GO TOP não é a mesma coisa que GO 1. Quando nós formos estudar os índices nós veremos porque GO TOP é diferente de GO 1. **A dica é:** se você quiser ir para o início do arquivo **use GO TOP sempre**. Não use GO 1.

Dica 130

Para ver o efeito desse comando sem se confundir com outras saídas anteriores, habitue-se a usar o comando CLS para "limpar a tela"

Figura 29.12: A saída do comando no HBRUN

Olhe agora para a barra de registros do HBRUN:

Figura 29.13: Ponteiro após o comando LIST com a cláusula NEXT 5

Dica 131

O comando LIST é bom porque ele ajuda no aprendizado, mas ele é muito pouco usado na prática. Quando você for programar profissionalmente, talvez você queira usar os laços vistos anteriormente juntamente com outros comandos de navegação, como GOTO, SKIP, etc. Tudo vai depender da situação.

Vamos agora aprender outro comando: o SKIP. Vamos partir dos nossos exemplos. Até agora estudamos a abrangência NEXT, mas note que essa abrangência possui uma limitação. Observe o fragmento a seguir:

```
CLS
GO TOP
LIST nome, nascimento, peso, altura NEXT 5
```

Figura 29.14: A saída do comando no HBRUN

Agora, se eu quiser exibir os próximos 5, basta que eu faça (de novo):

```
LIST nome, nascimento, peso, altura NEXT 5
```

É aí que surge o problema: como a abrangência NEXT começa a exibir do registro atual (e não do próximo) o quinto registro (paciente Ashley) se repetirá na próxima listagem:

Figura 29.15: O quinto registro (Ashley) se repete na próxima listagem

A solução para esse problema é deslocar o ponteiro um registro para frente antes de executar o próximo LIST ... NEXT 5. O deslocamento do ponteiro é feito através do comando SKIP. O resultado completo seria :

```
CLS // Limpa a tela (só para facilitar para você)
GO TOP // Garante que eu estou no início do arquivo
LIST nome, nascimento, peso, altura NEXT 5 // Lista do 1 ao 5
SKIP // Pula do 5 para o 6 (agora eu estou no sexto registro)
LIST nome, nascimento, peso, altura NEXT 5 // Lista do 6 ao 10
```

Agora sim, o quinto registro não se repetiu porque eu usei o comando SKIP antes de executar o segundo LIST. Veja o resultado do fragmento acima na imagem a seguir:

Figura 29.16: O quinto registro **não** se repete na próxima listagem

O comando SKIP está descrito logo a seguir :

Descrição sintática 38

1. Nome : SKIP
2. Classificação : comando.
3. Descrição : move o ponteiro de registro para uma nova posição.
4. Sintaxe

```
SKIP [<nRegistros>] [ALIAS [<idAlias>|<nAreaDeTrabalho>]

[<nRegistros>] : Quantidade de registros. O valor padrão é 1.

Eu posso mover o registro de um arquivo em outra área de trabal
basta usar a cláusula ALIAS seguido de um identificador de área
trabalho.
```

Fonte : [Nantucket 1990, p. 4-153]

Note que, eu posso especificar quantos registros eu quero “pular” [9]. Inclusive eu posso voltar uma quantidade de registros através de um número negativo.

Dica 132

Se o meu arquivo tem 10 registros e eu estou no primeiro registro e digito o comando SKIP 1000. O que acontecerá ? Se você respondeu que o ponteiro irá para o registro fantasma, então você acertou. Da mesma forma, se eu estou no registro 10 do referido arquivo e digito SKIP -9999999 o ponteiro irá para o primeiro registro. A consequência prática disso é: o comando SKIP não reporta nenhum erro caso ele receba uma ordem sua para se mover além dos limites do arquivo. **A dica é** : cuidado com a posição do ponteiro do registro após o uso de um comando SKIP, você precisa sempre se certificar se o final do arquivo já chegou (isso é feito através de uma função chamada EOF()[a]).

[a] Veremos o uso da função EOF() mais na frente.

Abrangência REST

A abrangência REST se parece muito com a abrangência NEXT, a diferença é que REST opera do registro onde está o ponteiro até o final do arquivo, ou seja REST significa “o restante dos registros”. A consequência prática disso é que o ponteiro irá se situar no registro fantasma sempre que você executar a abrangência REST.

Abrangência RECORD

A abrangência RECORD sempre atua sobre um registro, que é o registro especificado por você. Por exemplo, no fragmento a seguir:

```
LIST nome, nascimento, peso, altura RECORD 5
```

Irá exibir apenas o quinto registro, e o ponteiro do registro ficará sobre o referido registro.

[9] O significado de SKIP é “PULAR”.

A cláusula WHILE e FOR

O comando LIST possui duas cláusulas que podem "filtrar" os dados apresentados de acordo com o retorno de uma expressão lógica (cujo retorno é verdadeiro (.t.) ou falso (.f.)). A cláusula FOR é a mais usada das duas (quando o arquivo é pequeno), pois ela percorre todo o arquivo **e sempre inicia do primeiro registro**, sem precisar de um comando GO TOP para posicionar o ponteiro no início do registro. No exemplo a seguir nós listamos apenas os pacientes que nasceram no mês de dezembro:

```
LIST nome, nascimento, altura, peso FOR MONTH( nascimento ) = 12
```

O resultado do comando :

Figura 29.17: LIST com a cláusula FOR

```
 1  Armor                                1994-12-16 1.55  73.00
 2  Arneles                              1981-12-15 1.51  63.00
 3  Artemisa                             1992-12-13 1.70 114.00
 4  Artemisia                            1972-12-12 1.95  60.00
 6  Kesia                                1958-12-23 1.86 108.00
 7  Astride                              1986-12-30 1.58  65.00
11  Audria                               1992-12-23 1.98  91.00
12  Augustos                             1968-12-17 1.60 104.00
15  Bianca                               1956-12-11 1.65  65.00
20  Blandina                             1982-12-28 1.51 101.00
23  Zenia                                1973-12-24 1.97  63.00
```

Note que todo o arquivo foi percorrido e todos os registros foram analisados e 11 registros satisfazem a expressão lógica. Observe que existe uma "quebra" na sequência dos números dos registros (do registro 4 pula para o 6, do 7 pula para o 11, etc.), já que os demais foram omitidos pois são pacientes que nasceram em outro mês.

A cláusula WHILE funciona de forma semelhante, mas ela **não posiciona o registro no início do arquivo** e também deixa de processar quando a condição deixa de ser satisfeita. Observe:

```
CLS // Limpa a tela (NÃO É OBRIGATÓRIO)
GO TOP // Vai para o início (para sair do registro fantasma do nosso LI
LIST nome, nascimento, altura, peso WHILE MONTH( nascimento ) = 12
```

Figura 29.18: LIST com a cláusula WHILE

```
1  Armor                                1994-12-16 1.55  73.00
2  Arneles                              1981-12-15 1.51  63.00
3  Artemisa                             1992-12-13 1.70 114.00
4  Artemisia                            1972-12-12 1.95  60.00
```

Note que ele exibiu somente quatro registros, já que o quinto registro não satisfaz a condição lógica (o mês não é dezembro).

Dica 133

Você deve estar achando que nunca vai usar a cláusula WHILE, já que a cláusula FOR traz resultados mais precisos, mas isso não é verdade. A cláusula WHILE é bastante usada quando o arquivo é muito grande (possui milhares de registros) e está indexado. Nas próximas seções nós aprenderemos mais sobre isso.

Outro detalhe interessante sobre a cláusula WHILE é que, caso o ponteiro de registro esteja em um registro que não satisfaça a cláusula WHILE, o processamento não será realizado. Por exemplo, faça o seguinte teste:

```
GO 5 // Vai para o registro 5 (Ashley) que nasceu em novembro
LIST nome, nascimento, altura, peso WHILE MONTH( nascimento ) = 12
```

Nenhum dado será apresentado, pois a data de nascimento do registro desse paciente é 23/11/1988 (mês de novembro). Então, quando for usar a cláusula WHILE não se esqueça :

1. Lembre-se de que a cláusula WHILE não “meche” no ponteiro do registro antes de entrar em ação.
2. Certifique-se de que o registro inicial satisfaça a expressão lógica.
3. Certifique-se de que após o registro final não tenha, “lá mais na frente”, outro registro que satisfaça a condição, pois ele não aparecerá no processamento.

Mais uma vez repetimos: você só irá entender a utilidade da cláusula WHILE quando aprender o que significa ordenação (ou indexação) de arquivos. Veremos o que é isso mais adiante.

A cláusula TO PRINTER

Essa cláusula envia o resultado do comando para a impressora, mas como o sistema de impressão não foi explicado, nós não iremos abordar essa cláusula.

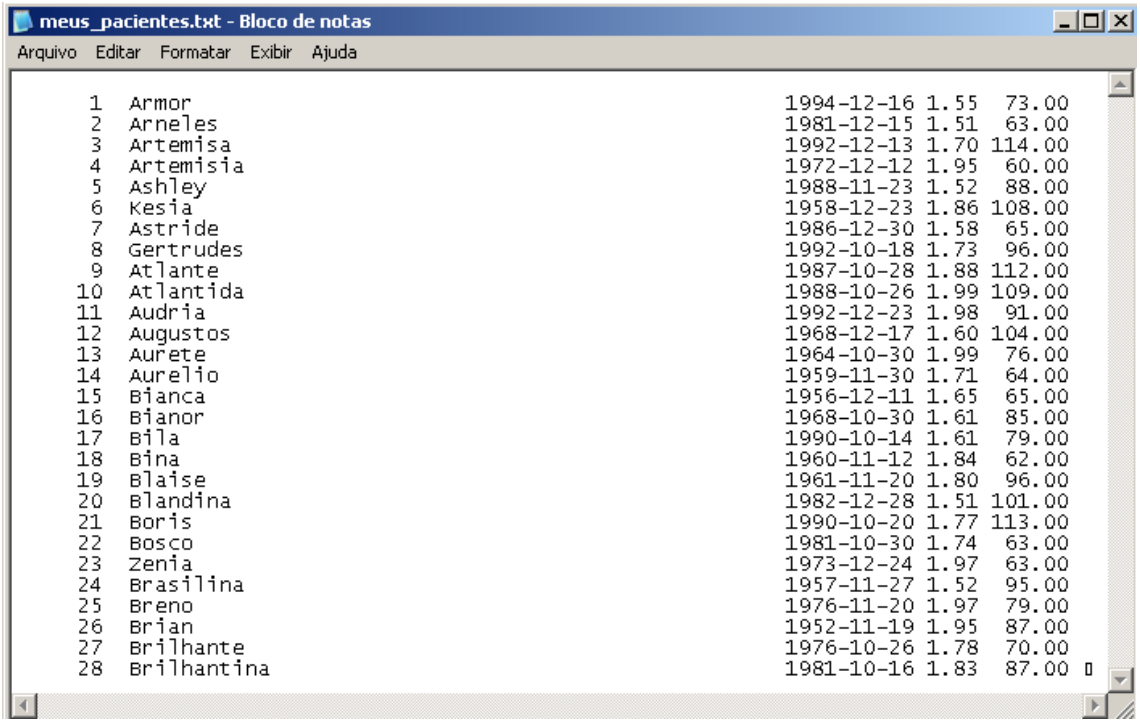
A cláusula TO FILE

A cláusula TO FILE envia a saída do comando para um arquivo no formato texto simples (pode ser aberto pelo programa NOTEPAD do windows). Ele funciona segundo o fragmento abaixo :

```
LIST nome, nascimento, altura, peso TO FILE meus_pacientes
```

Note que o comando LIST adiciona automaticamente uma extensão (.txt) padrão dos arquivos textos.

Figura 29.19: Saida do comando LIST em texto



Caso você não queira essa extensão, você mesmo pode especificar a extensão que desejar quando for informar o nome do arquivo, por exemplo:

```
LIST nome, nascimento, altura, peso TO FILE meus_pacientes.prn
```

Dica 134

Note que, quando você envia o comando para um arquivo o resultado também é exibido na tela. Isso nem sempre é o desejado. Existem situações em que você não quer ver o resultado na tela, mas apenas quer o resultado no arquivo. Para evitar que a saída do comando LIST "ecoe" para a tela use o SET abaixo :

```
SET CONSOLE OFF
```

Para retornar para a situação anterior faça :

```
SET CONSOLE ON
```

IMPORTANTE!!
O comando *SET CONSOLE OFF* desativa também outras saídas da tela, como a dos comandos "?", ACCEPT, INPUT e WAIT. Por esse motivo você deve retornar a situação anterior logo após gerar o seu arquivo.

Combinando o comando LIST com as suas funções

O comando LIST (assim como os demais comandos de manipulação de arquivos) podem ser usados em conjunto com expressões e funções. Por exemplo, o programa da listagem 29.3 ilustra o uso de uma função criada por você, sendo chamada de dentro de um comando LIST.

Listagem 29.3: O uso do comando LIST com as suas funções.
Fonte: codigos/list01.prg

```
PROCEDURE Main

    CLS
    SET DATE BRITISH

    USE paciente
    IF .NOT. USED()
       ? "Problemas na abertura do arquivo"
       QUIT
    ENDIF

    ? "Clínica Genesis"
    ? "Diagnóstico de pacientes"
    ? REPLICATE( "-" , 80 ) // 80 símbolos "-"
    LIST LEFT(nome,15), nascimento,;
         peso, altura,;
         DIAGNOSTICO_IMC( peso, altura )


RETURN


FUNCTION DIAGNOSTICO_IMC( nPeso, nAltura )
LOCAL nImc
LOCAL cDiagnostico := ""

   nImc := nPeso / ( nAltura ^ 2 )

   DO CASE
      CASE nImc <= 16.99
         cDiagnostico := "Baixo peso grave"
      CASE nImc >= 17 .AND. nImc <= 18.49
         cDiagnostico := "Baixo peso"
      CASE nImc >= 18.5 .AND. nImc <= 24.99
         cDiagnostico := "Peso normal"
      CASE nImc >= 25 .AND. nImc <= 29.99
         cDiagnostico := "Sobrepeso"
      CASE nImc >= 30 .AND. nImc <= 34.99
         cDiagnostico := "Obesidade grau I"
      CASE nImc >= 35 .AND. nImc <= 39.99
         cDiagnostico := "Obesidade grau II"
      CASE nImc >= 40
         cDiagnostico := "Obesidade grau III (Obesidade mórbida)"
   ENDCASE

RETURN cDiagnostico
```

.:Resultado:.

```
Clínica Genesis
Diagnóstico de pacientes
------------------------------------------------------------------------

      1  Armor           16/12/94  73.00 1.55 Obesidade grau I
      2  Arneles         15/12/81  63.00 1.51 Sobrepeso
      3  Artemisa        13/12/92 114.00 1.70 Obesidade grau II
      4  Artemisia       12/12/72  60.00 1.95 Baixo peso grave
      5  Ashley          23/11/88  88.00 1.52 Obesidade grau II
      6  Kesia           23/12/58 108.00 1.86 Obesidade grau I
      7  Astride         30/12/86  65.00 1.58 Sobrepeso
      8  Gertrudes       18/10/92  96.00 1.73 Obesidade grau I
      9  Atlante         28/10/87 112.00 1.88 Obesidade grau I
     10  Atlantida       26/10/88 109.00 1.99 Sobrepeso
     11  Audria          23/12/92  91.00 1.98 Peso normal
     12  Augustos        17/12/68 104.00 1.60 Obesidade grau III
   (Obesidade mórbida)
     13  Aurete          30/10/64  76.00 1.99 Peso normal
     14  Aurelio         30/11/59  64.00 1.71 Peso normal
     15  Bianca          11/12/56  65.00 1.65 Peso normal
     16  Bianor          30/10/68  85.00 1.61 Obesidade grau I
     17  Bila            14/10/90  79.00 1.61 Obesidade grau I
     18  Bina            12/11/60  62.00 1.84 Baixo peso
     19  Blaise          20/11/61  96.00 1.80 Sobrepeso
     20  Blandina        28/12/82 101.00 1.51 Obesidade grau III
   (Obesidade mórbida)
     21  Boris           20/10/90 113.00 1.77 Obesidade grau II
     22  Bosco           30/10/81  63.00 1.74 Peso normal
     23  Zenia           24/12/73  63.00 1.97 Baixo peso grave
     24  Brasilina       27/11/57  95.00 1.52 Obesidade grau III
   (Obesidade mórbida)
     25  Breno           20/11/76  79.00 1.97 Peso normal
     26  Brian           19/11/52  87.00 1.95 Peso normal
     27  Brilhante       26/10/76  70.00 1.78 Peso normal
     28  Brilhantina     16/10/81  87.00 1.83 Sobrepeso
```

Só para encerrar...

Você deve ter reparado que quando gerou o programa alguns símbolos de “warnings” apareceram.

.:Resultado:.

```
list01.prg(18) Warning W0001  Ambiguous reference 'NOME'
list01.prg(18) Warning W0001  Ambiguous reference 'NASCIMENTO'
list01.prg(18) Warning W0001  Ambiguous reference 'PESO'
list01.prg(18) Warning W0001  Ambiguous reference 'ALTURA'
list01.prg(18) Warning W0001  Ambiguous reference 'PESO'
list01.prg(18) Warning W0001  Ambiguous reference 'ALTURA'
```

Isso acontece porque o Harbour encontrou o nome dos campos do arquivo e não soube identificar o que eles eram. Lembra que é aconselhável declarar as variáveis ? O Harbour detectou uma “ambiguidade”, ou seja, um símbolo que, tanto pode ser uma variável quanto pode ser um campo. Como ele não pode ter a “certeza” do que esses

símbolos são ele emitiu um aviso de “cuidado”. Para evitar esses avisos, você tem que informar para o Harbour que esses campos pertencem a um arquivo DBF. Como fazer isso ? Simples, basta declarar esses campos como se faz com qualquer variável, só que para isso você precisa usar a instrução FIELD. O código da listagem 29.4 faz isso na linha 3.

Listagem 29.4: Evitando ambiguidades no seu programa com a instrução FIELD.
Fonte: codigos/list02.prg

```
1
2  PROCEDURE Main
3  FIELD nome, nascimento, peso, altura
4
5      CLS
6      SET DATE BRITISH
7
8      USE paciente
9      IF .NOT. USED()
10        ? "Problemas na abertura do arquivo"
11        QUIT
12     ENDIF
13
14     ? "Clínica Genesis"
15     ? "Diagnóstico de pacientes"
16     ? REPLICATE( "-" , 80 ) // 80 símbolos "-"
17     LIST LEFT(nome,15), nascimento,;
18          peso, altura,;
19          DIAGNOSTICO_IMC( peso, altura )
20
21
22 RETURN
23
24
25 FUNCTION DIAGNOSTICO_IMC( nPeso, nAltura )
26 LOCAL nImc
27 LOCAL cDiagnostico := ""
28
29    nImc := nPeso / ( nAltura ^ 2 )
30
31    DO CASE
32       CASE nImc <= 16.99
33          cDiagnostico := "Baixo peso grave"
34       CASE nImc >= 17 .AND. nImc <= 18.49
35          cDiagnostico := "Baixo peso"
36       CASE nImc >= 18.5 .AND. nImc <= 24.99
37          cDiagnostico := "Peso normal"
38       CASE nImc >= 25 .AND. nImc <= 29.99
39          cDiagnostico := "Sobrepeso"
40       CASE nImc >= 30 .AND. nImc <= 34.99
41          cDiagnostico := "Obesidade grau I"
42       CASE nImc >= 35 .AND. nImc <= 39.99
43          cDiagnostico := "Obesidade grau II"
```

```
      CASE nImc >= 40                                                    44
         cDiagnostico := "Obesidade grau III (Obesidade mórbida)"        45
   ENDCASE                                                              46
                                                                        47
RETURN cDiagnostico                                                     48
```

Dica 135

É sempre bom declarar através da instrução FIELD que determinados símbolos são campos de arquivos. Isso tornará os seus programas mais seguros e claros. Note que a instrução FIELD deve preceder qualquer outra, assim, ela deve estar no início da rotina (junto com a declaração de variáveis LOCAL e STATIC internas).

Dica 136

Quando a instrução FIELD é usada antes de qualquer rotina então ela vale para todo o arquivo PRG .

Dica 137

Se o arquivo PRG do seu programa é muito extenso e você usa um campo de um arquivo apenas uma vez, você pode prefixar esse campo com FIELD, em vez de “declarar” no início do seu fonte. Isso é feito mediante o operador “->”.

```
? FIELD->NOME, FIELD->NASCIMENTO
```

Dica 138

Outra forma (a mais popular) de declarar que um símbolo é um campo é prefixar esse campo com o seu alias. Assim :

```
? PACIENTE->NOME, PACIENTE->NASCIMENTO
```

Tanto essa forma quanto o prefixo com FIELD são válidas.

Na próxima seção, nós iremos estudar os demais comandos de manipulação de DBFs herdados do dBase.

29.4.7 Posicionando-se sobre um registro com através do comando LOCATE

Até agora vimos três formas de nos movimentarmos por entre os registros de um arquivo: os comandos GO, SKIP e LIST. Vamos agora aprender mais um comando muito útil para a navegação entre os registros : o comando LOCATE. O interessante do comando LOCATE é que ele posiciona o ponteiro em um determinado registro, de acordo com um critério de pesquisa (cláusulas WHILE ou FOR). O comando LOCATE

possui uma estreita relação com uma função chamada FOUND(). Essa função retorna um valor lógico de acordo com o resultado do comando LOCATE. Se o comando LOCATE “achou” o registro buscado essa função retorna verdade (.t.), caso contrário ela retorna falso (.f.).

O comando LOCATE está descrito a seguir :

Descrição sintática 39

1. Nome : ?
2. Classificação : comando.
3. Descrição : exibe um ou mais valores no console (tela).
4. Sintaxe

```
? [ <lista de expressões>]
```

Fonte : [Nantucket 1990, p. 4-1]

Vamos fazer uma busca no nosso arquivo paciente.dbf :

```
CLS // Apenas para limpar a tela
USE paciente // Caso o arquivo não esteja aberto
LOCATE FOR nome = "Ashley"
```

Você deve estar sobre o quinto registro, mas como ter certeza de que você encontrou esse registro ? Dentro do HBRUN, uma forma seria usar o comando “?”, seguido do campo NOME.

```
? NOME
```

A saída do HBRUN deve mostrar :

.:Resultado:.

```
Ashley
```

Essa forma de verificação pode funcionar bem dentro de utilitários de linha de comando, como o HBRUN, mas não é o ideal dentro de um código Harbour. É nessas horas que a função FOUND() entra em cena. Faça o seguinte, digite o seguinte fragmento no HBRUN:

```
? FOUND()
```

A saída do HBRUN deve mostrar :

.:Resultado:.

```
.T.
```

Note que esse valor lógico verdadeiro (.t.) ainda foi o resultado do comando LOCATE executado recentemente.

Quando você estiver programando códigos de busca em um banco de dados DBF, você pode fazer algo como :

```
LOCATE FOR nome = "Ashley"                                              1
IF  FOUND()                                                             2
    // Aqui eu achei o registro, posso realizar algumas                 3
    // operações nele.                                                  4
ELSE                                                                    5
    // Aqui eu não achei, uma mensagem para o                           6
    // usuário seria bom                                                7
    ?  "Registro não encontrado!"                                       8
ENDIF                                                                   9
```

Dica 139

Lembra das regras de igualdade de strings vistas no capítulo de operadores ?
Se você usar o operador "=" e a string da esquerda conter a string da direita, então eles são considerados iguais.
Agora você deve estar entendendo porque essa forma de igualdade foi usada: para facilitar buscas em banco de dados DBFs. Faça a seguinte experiência : tente fazer uma busca com o operador de duplo igual ("==").

```
LOCATE FOR nome == "Ashley"
```

Note que o registro não foi encontrado dessa vez. A função FOUND() retorna falso (.f.) e o ponteiro do arquivo foi para o tal registro fantasma.
Você sabe explicar porque ?

Resposta: primeiramente, vamos entender como o comando LOCATE funciona.
O comando LOCATE (**com a cláusula FOR**) faz uma "varredura" no arquivo e visita registro a registro, avaliando a expressão informada por você
(1) Vamos analisar a primeira busca (LOCATE FOR nome = "Ashley").
Como o campo NOME tem um tamanho de 50 caracteres, ele irá realizar, para cada elemento do arquivo a seguinte comparação:

```
"Armor                "    = "Ashley"   (Não é igual)
"Arneles              "    = "Ashley"   (Não é igual)
"Artemisa             "    = "Ashley"   (Não é igual)
"Artemisia            "    = "Ashley"   (Não é igual)
"Ashley               "    = "Ashley"   (É igual)

É igual porque o valor da esquerda "contém" o valor da direita.

FOUND() retorna verdadeiro e o ponteiro deve parar no quinto registro
```

(2) Já no segundo caso temos LOCATE FOR nome == "Ashley".
Como o campo NOME tem um tamanho de 50 caracteres e o operador duplo igual[a] faz uma comparação exata , ele irá realizar, para cada elemento do arquivo a seguinte comparação:

```
"Armor                "  == "Ashley"   (Não é igual)
"Arneles              "  == "Ashley"   (Não é igual)
"Artemisa             "  == "Ashley"   (Não é igual)
"Artemisia            "  == "Ashley"   (Não é igual)
"Ashley               "  == "Ashley"   (Não é igual) <--

... e assim sucessivamente, até chegar no final do arquivo.
FOUND() retorna falso e o ponteiro irá para o registro fantasma,
que é mesma coisa que o final do arquivo.
```

Conclusão : quando estiver buscando valores em banco de dados DBFs, use o operador "=" e não o operador "exatamente igual".

[a] Também chamado de "exatamente igual"

29.4.8 CONTINUE: o complemento do comando LOCATE

O comando LOCATE não atua em conjunto apenas com a função FOUND(). Ele também possui um comando que complementa o seu uso. O nome desse comando é CONTINUE. Esse comando pesquisa a partir do registro corrente até encontrar o próximo registro que satisfaça a condição do LOCATE mais recente executado na área corrente. Ele encerra-se quando a condição é obedecida ou atinge o fim do arquivo. Se o comando CONTINUE tem sucesso, o registro encontrado torna-se o atual e FOUND() retorna verdadeiro (.t.), se não houve sucesso FOUND() retorna falso (.f.) [Nantucket 1990, p. 4-31].

Descrição sintática 40

1. Nome : CONTINUE
2. Classificação : comando.
3. Descrição : Reassume um LOCATE pendente.
4. Sintaxe

```
CONTINUE
```

Fonte : [Nantucket 1990, p. 4-31]

Dica 140

O comando CONTINUE só funciona se o LOCATE pendente foi construído com a cláusula FOR. Ou seja, a abrangência e a condição WHILE são ignoradas. Caso você esteja usando um LOCATE com WHILE e necessita continuar a busca por um registro, repita o LOCATE adicionando a abrangência REST.

29.4.9 Excluindo um registro com o comando DELETE

Já vimos que o comando APPEND BLANK serve para incluir um registro em branco e que o comando REPLACE serve para alterar o valor de um ou mais campos de um registro. Agora chegou a hora de aprender a excluir um registro através do comando DELETE.

Para excluir um registro você primeiro precisa estar com o ponteiro posicionado no registro que você deseja excluir. Uma das formas de se fazer isso é usar o comando LOCATE previamente e logo em seguida o comando DELETE, conforme o fragmento a seguir:

```
LOCATE FOR nome = "Marx"
IF FOUND()
    DELETE // Agora excluo o registro.
ENDIF
```

Quando você usa o comando DELETE você não está excluindo um registro, mas apenas marcando-o para uma futura exclusão. O registro continua a existir, com todos os dados, só que ele agora recebeu uma marca especial. Essa marca indica que ele foi "marcado" para uma operação futura que, de fato, realizará a exclusão desse registro. A consequência prática disso é : **"um registro marcado para a exclusão com o comando DELETE pode ser recuperado, já que os seus dados não foram excluídos na realidade."**

Vamos fazer uma pequena experiência: no HBRUN, abra o arquivo paciente.dbf e busque o paciente cujo nome é "Zenia".

.:Resultado:.

```
LOCATE FOR nome = "Zenia"
```

Para se certificar que encontrou o paciente digite :

.:Resultado:.

```
? FOUND()
.T.
```

Agora exclua esse comando com o comando DELETE.

.:Resultado:.

```
DELETE
```

Uma forma mais compacta de se fazer isso é usando o comando DELETE com a cláusula FOR, conforme o fragmento a seguir:

.:Resultado:.

```
DELETE FOR nome = "Zenia"
```

Use agora o comando LIST para ver todos os campos NOME do arquivo paciente.dbf :

.:Resultado:.

```
LIST nome
```

A saída do comando é :

Figura 29.20: Listagem após uma exclusão

Note que o comando LIST "colocou" um asterisco no registro que foi excluído. Um registro "excluído" por DELETE, portanto é apenas "marcado para uma exclusão futura".

Vamos continuar com os nossos testes usando o HBRUN: agora nós vamos deixar de mostrar todos os registros que foram marcados para a exclusão, criando uma ilusão no usuário de que eles não existem mais. Para fazer isso, basta usar o comando *SET DELETE ON*.

.:Resultado:.

```
CLS // Só para limpar a tela e ficar mais claro para você
SET DELETE ON
LIST nome
```

Note que houve um "salto" na numeração dos registros:

Figura 29.21: Listagem após uma exclusão

Agora vamos "excluir" o paciente cujo nome é Bosco. Nós vamos usar o comando DELETE com a cláusula FOR (a cláusula FOR funciona de forma semelhante para

todos os comandos, caso você tenha alguma dúvida, dê uma olhada no comando LIST).

.:Resultado:.

```
CLS // Só para limpar a tela e ficar mais claro para você
DELETE FOR nome = "Bosco"
LIST nome
```

Figura 29.22: Listagem após uma exclusão

Note que, novamente, houve uma quebra na sequência. Essas “quebras” indicam que o registro ainda “está lá”, apenas nós não estamos vendo por causa do estado de SET DELETE (feito nos fragmentos anteriores). Para exibir os registros excluídos faça :

.:Resultado:.

```
SET DELETE OFF
CLS
LIST nome
```

Os pacientes “Zenia” e “Bosco” voltaram, mas foram exibidos com um asterisco. Isso indica, como nós já vimos, que eles estão “marcados para exclusão”.

Figura 29.23: Registros marcados para a exclusão

Vamos continuar com nossas experiências. Agora nós vamos tentar localizar um registro que foi marcado para a exclusão.

.:Resultado:.

```
LOCATE FOR nome = "Zenia"
```

Vamos ver o resultado da função FOUND() :

.:Resultado:.

```
? FOUND()
.T.
```

Agora vamos alterar o SET DELETE :

.:Resultado:.

```
SET DELETE ON
LOCATE FOR nome = "Zenia"
```

Vamos ver o resultado da função FOUND() :

.:Resultado:.

```
? FOUND()
.T.
```

Conclusão : o resultado de comandos de busca dependem do estado de SET DELETE. Se ele estiver ON então o registro, apesar de apenas ter sido "marcado para a exclusão" não é mais encontrado pelo comando LOCATE.

Vamos fazer outro pequeno teste no HBRUN. Vá para o registro de número 21:

.:Resultado:.

```
GO 21
```

A função RECNO() retorna o número do registro corrente:

.:Resultado:.

```
? RECNO()
21
```

Agora "pule", com o comando SKIP, para o próximo registro.

.:Resultado:.

```
SKIP
```

Para qual registro você acha que foi ? Para o registro 22 (que está marcado para a exclusão) ou para o próximo registro não marcado (no caso o registro 24) ?

Tudo depende do estado de SET DELETE. Como ele, atualmente, está ON, então o registro corrente agora é o de número 24.

.:Resultado:.

```
? RECNO()
24
```

Dica 141

O comando GO **independe** do estado de SET DELETE. Por exemplo, GO 22, no nosso exemplo, irá para o paciente "Bosco", independente dele ter sido marcado ou não para a exclusão.

Dica 142

Se o comando SET DELETE não for definido, o seu valor padrão é OFF. Ou seja, ele irá exibir os registros marcados para a exclusão. **Isso pode confundir o usuário final**, portanto, no início do seu programa defina SET DELETE ON.

O comando DELETE está descrito sintaticamente a seguir :

Descrição sintática 41

1. Nome : DELETE
2. Classificação : comando.
3. Descrição : marca registros para eliminação.
4. Sintaxe

```
DELETE [<abrangência>]
   [FOR <lCondição>]
   [WHILE <lCondição>]
```

Fonte : [Nantucket 1990, p. 4-44]

Dica 143

Lembre-se: caso você esqueça o significado das cláusulas desses comandos que operam sobre banco de dados, basta consultar o nosso estudo sobre o comando LIST. **As cláusulas desses comandos operam da mesma forma.** A única diferença é a quantidade de cláusulas, por exemplo: o comando DELETE[a] possui a abrangência (RECORD (padrão), NEXT, ALL e REST), e as cláusulas FOR e WHILE. Já o comando LIST, além dessas cláusulas possui a lista de campos, a cláusula OFF e as cláusulas TO PRINTER e TO FILE. Para saber quais cláusulas estão disponíveis, basta consultar o nosso quadro de sintaxe.

[a] E o comando RECALL, que será visto a seguir.

Dica 144

Cuidado!!
O comando :

```
DELETE FOR nome = "A"
```

exclui todos os registros que iniciam com a letra “A”.
O comando :

```
DELETE FOR nome = "Adriana"
```

exclui todos os registros que iniciam com a string “Adriana”. Os motivos já foram explicados exaustivamente[a], mas não custa nada insistir nesse ponto.

[a] Se o operador “=” for usado para realizar a comparação entre strings, e a string da esquerda conter a da direita, então elas são consideradas iguais.

29.4.10 A função DELETED()

A função DELETED() serve para informar ao programador se o registro corrente está marcado para a exclusão (deletado) ou não. Acompanhe:

Descrição sintática 42

1. Nome : DELETED()
2. Classificação : função.
3. Descrição : retorna o status do registro corrente apontando se está marcado para a eliminação ou não.
4. Sintaxe

```
DELETED() -> lMarcado

DELETED() retorna verdadeiro (.t.) se o registro corrente
 está marcado para a eliminação e falso (.f.) caso contrário.
```

Fonte : [Nantucket 1990, p. 5-66]

A vantagem de uma função sobre um comando (nós já vimos) é que a função pode ser "embutida" dentro de uma expressão. Vamos, fazer a seguir uma pequena demonstração do comando DELETED() juntamente com a função IF().

No HBRUN faça :

.:Resultado:.

```
CLS
SET DELETE OFF
LIST nome, IF( DELETED() , "Excluído", "Normal" )
```

Você poderia perguntar : "Os dados que foram marcados para a exclusão são exibidos com um '*', qual a importância da função DELETED(), nesse caso ?"

A resposta: A exibição do '*' é uma característica do comando LIST ou seja, nem sempre um registro excluído aparece com o asterisco (o comando "?", por exemplo, não exibe asteriscos caso o campo seja de um registro marcado para exclusão). Um outro motivo é que a função DELETED() poder ser usada para uma tomada de decisão dentro de um processamento, conforme o fragmento abaixo:

```
SET DELETE ON                                                    1
... // Várias instruções                                         2
GO nRec // Vou para o registro de número armazenado em nRec      3
IF DELETED() // Se ele está marcado                              4
    SKIP // Pula para o seguinte, que com certeza não estará     5
            // marcado para exclusão, pois o SET DELETE está ON  6
                                                                 7
// e o comando SKIP ignora os registros marcados, nesse caso.
ENDIF                                                            8
```

29.4.11 Recuperando um registro com o comando RECALL

Agora nós vamos recuperar esses registros que foram marcados para a exclusão. Isso é feito com o comando RECALL. O comando RECALL é o inverso do comando DELETE. Veja a seguir a sintaxe do comando RECALL:

Descrição sintática 43

1. Nome : RECALL
2. Classificação : comando.
3. Descrição : recupera registros marcados para eliminação.
4. Sintaxe

```
RECALL [<abrangência>]
   [FOR <lCondição>]
   [WHILE <lCondição>]
```

Fonte : [Nantucket 1990, p. 4-78]

Vamos agora aplicar o comando RECALL. No HBRUN digite:

.:Resultado:.

```
CLS
SET DELETE OFF
RECALL FOR nome = "Zenia"
```

Note que eu só posso recuperar um registro se eu puder “vê-lo”, ou seja, o SET DELETE deve estar OFF. Se o comando SET DELETE estivesse ON, então o registro da paciente “Zenia” não poderia ser localizado, e portanto não poderia ser recuperado. Uma exceção a essa regra é o comando GO. Lembre-se que o SET DELETE ON não altera o comportamento do comando GO <número do registro>. Vamos testar logo a seguir.

.:Resultado:.

```
CLS
SET DELETE ON
GO 22 // O registro que contém o paciente Bosco (QUE ESTÁ MARCADO)
RECALL // Como eu já "estou no registro" basta chamar RECALL
```

Para conferir faça :

.:Resultado:.

```
CLS
LIST nome
```

Note que os registros foram recuperados e agora você deve ter um arquivo com 28 registros (nenhum deles está marcado para exclusão).

29.4.12 Eliminando de vez os registros marcados para a exclusão através do comando PACK

Quando você tiver a certeza de que os registros marcados para a exclusão não serão mais necessários, use o comando PACK para se livrar deles. Quando você usa o comando PACK, todos os registros marcados para eliminação são removidos do arquivo corrente e o espaço em disco é recuperado. Após o término o ponteiro é posicionado no primeiro registro lógico na área de trabalho corrente (equivale a um GO TOP).

Dica 145

O comando PACK pode demorar para ser concluído caso o arquivo tenha milhares de registros. O ideal é reservar uma rotina só para usar o comando PACK. É perda de tempo usar o comando PACK imediatamente após um DELETE. Evite também colocar o PACK dentro de estruturas de repetição. O uso mais comum (e ideal) do comando PACK está ilustrado a seguir:

```
USE clientes
? "Aguarde enquanto o seu arquivo é otimizado" // Aviso ao usuário
PACK
```

Um cenário bem comum também é usar o PACK e, logo em seguida, indexar os seus arquivos[a].

[a] Veremos o que é isso nas seções seguintes.

29.4.13 SET FILTER TO: Trabalhando com subconjuntos de dados.

O comando SET FILTER TO serve para ocultar uma quantidade de registros que não atendam a uma determinada condição lógica.

Descrição sintática 44

1. Nome : SET FILTER TO
2. Classificação : comando.
3. Descrição : esconde registros que não atendam uma condição.
4. Sintaxe

```
SET FILTER TO [<lCondição>]

Onde <lCondição> é o resultado de uma expressão lógica
 (verdadeiro ou falso).
```

Fonte : [Nantucket 1990, p. 4-125]

Quando a condição está ativa, a área de trabalho corrente age como se contivesse somente os registros que atendem a condição especificada. Uma condição FILTER é uma das propriedades de uma área de trabalho. Uma vez ativada, a condição pode ser retornada na forma de uma cadeia de caracteres usando-se a função DBFILTER() [Nantucket 1990, p. 4-125].

Importante: uma vez estabelecido o filtro ainda não é ativado até que o ponteiro de registro seja movido de sua posição corrente. Nós aconselhamos o uso do GO TOP para isso.

Vamos a um exemplo através do HBRUN. No fragmento a seguir nós exibimos apenas os pacientes que possuem altura inferior a 1,60 m :

.:Resultado:.

```
CLS
USE paciente
SET FILTER TO altura < 1.6 // Defino o filtro
GO TOP // Movo o ponteiro para ativar o filtro
LIST nome, altura
```

Figura 29.24: Resultado com SET FILTER TO

```
 1  Armor                                                    1.55
 2  Arneles                                                  1.51
 5  Ashley                                                   1.52
 7  Astride                                                  1.58
20  Blandina                                                 1.51
24  Brasilina                                                1.52
```

Vantagens do SET FILTER TO

O SET FILTER torna as operações com arquivos mais práticas e mais claras quando você quer realizar uma série de operações sobre um subconjunto de dados. Com o SET FILTER, eu preciso definir a condição apenas uma vez, enquanto que, com os demais comandos eu preciso sempre especificar a abrangência da ação (com FOR, ALL, WHILE, etc.). Você só não deve esquecer de desativar o filtro quando não precisar mais dele. Para fazer isso chame SET FILTER TO sem argumentos.

Acompanhe no fragmento a seguir :

```
// Listar apenas os pacientes com altura inferior a 1.6
USE paciente
DO WHILE .NOT. EOF() // Faça enquanto não for final do arquivo
     IF paciente->altura < 1.6
        ? paciente->nome
     ENDIF
     SKIP
ENDDO
```

Produz o mesmo efeito que :

```
// Listar apenas os pacientes com altura inferior a 1.6
USE paciente
```

```
SET FILTER TO paciente->altura < 1.6
GO TOP
DO WHILE .NOT. EOF() // Faça enquanto não for final do arquivo
     ? paciente->nome
     SKIP
ENDDO
SET FILTER TO // Desativa o filtro
```

Quando o filtro de uma área de trabalho está ativo, **todos os comandos e funções** que operam sobre arquivos obedecem a condição informada no filtro.

Dica 146

Não esqueça de desativar o FILTRO quando você não precisar mais dele. Para desativar a condição do SET FILTER simplesmente faça :

```
SET FILTER TO
```

Desvantagens do SET FILTER TO

Nós vimos que o SET FILTER TO atua na área de trabalho corrente e faz com que ela **pareça conter** um subconjunto de dados. Não se esqueça de que este comando processa sequencialmente todos os registros na área de trabalho. Por esta razão, o tempo necessário para o processamento de uma área de trabalho filtrada será o mesmo que numa área de trabalho normal. Dessa forma, quando você for operar com arquivos que contenham milhares de registros, o tempo de processamento de uma área filtrada é igual ao tempo de processamento dessa mesma área sem o filtro, pois todos os registros (mesmo os ocultos) deverão entrar na condição de filtro.

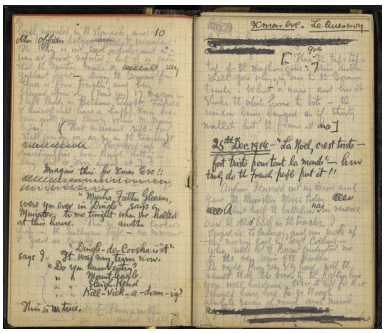
Dica 147

Se você tem certeza de que o arquivo não irá armazenar milhares de registros durante a vida útil da sua aplicação você pode usar o filtro sem problemas. Caso você note que o arquivo irá conter milhares de registros, então é melhor passar para outras formas de busca, como a busca através de índices.
Uma aplicação que usa filtros em arquivos que podem vir a crescer muito no futuro apresenta a seguinte característica: **nos primeiros dias (ou meses) a aplicação se comporta bem, mas quando o arquivo começar a ficar com muitos registros, a sua aplicação se tornará lenta**.

29.5 Os índices.

Antes de entender os índices do Harbour, vamos entender o problema que eles se propõem a resolver, que é **"a demora na busca de valores dentro de um arquivo com uma quantidade muito grande de registros"**. Vamos partir de uma situação corriqueira do dia-a-dia: vamos supor que você precisa buscar o telefone de uma pessoa em uma agenda. Vamos supor também que a agenda se pareça com a da figura a seguir:

Figura 29.25: Uma agenda bagunçada



Os nomes estão fora de ordem e a única solução é ir percorrendo folha por folha até encontrar o telefone dessa pessoa. Agora compare essa agenda com essa outra a seguir:

Figura 29.26: Agenda ordenada por ordem alfabética.

Nessa segunda agenda os nomes estão dispostos segundo uma ordem alfabética, e também temos pequenas subdivisões, uma para cada letra. Dessa forma a busca se tornará mais rápida porque não precisamos mais percorrer folha por folha até encontrar o nome que queremos. Basta você "saltar" para o ponto onde inicia os nomes com uma determinada inicial e, a partir desse ponto, ir realizando uma busca alfabética[10].

No nosso primeiro exemplo, o arquivo está desordenado, e no nosso segundo exemplo o arquivo encontra-se ordenado por nome. A cláusula FOR, que nós estudamos em inúmeros comandos, é usada para buscas em arquivos desordenados, pois ela sempre inicia a sua busca do primeiro registro e vai até o final. A busca usando a cláusula FOR "varre" o arquivo todo, por isso ela é mais lenta.

.:Resultado:.

[10]Da mesma forma com que você busca um nome em um dicionário convencional com centenas de páginas.

```
USE paciente
LIST nome, altura FOR nome = "Berenice" // Percorro o arquivo todo!!!
```

Agora, imagine essa mesma busca com a cláusula WHILE. Vamos rever um pouco essa cláusula porque ela foi pouco usada até agora. A cláusula WHILE requer dois pré-requisitos para que funcione corretamente :

1. O ponteiro precisa estar sobre o primeiro registro que satisfaça a condição.
2. O arquivo precisa estar ordenado, pois a cláusula WHILE irá continuar enquanto a condição estiver satisfeita.

Por exemplo, considere a lista do fragmento abaixo, :

```
Ana Maria
Beatriz Nogueira
Bianca Vasconcelos
Francisca Miranda
Berenice Assunção
Zenia Araújo
```

Vamos listar apenas os registros que começam com a letra "B".
Usando a cláusula FOR :

.:Resultado:.

```
USE paciente
LIST nome, altura FOR nome = "B" // Percorro o arquivo todo!!!
```

O meu retorno será :

```
Beatriz Nogueira
Bianca Vasconcelos
Berenice Assunção
```

Agora, vamos fazer a mesma operação usando a cláusula WHILE. Como eu preciso “estar” sobre o primeiro registro que satisfaça a condição eu preciso, primeiramente, localizar esse registro, vamos usar o comando LOCATE para isso e logo em seguida usar o comando LIST com a cláusula WHILE, assim:

.:Resultado:.

```
USE paciente
LOCATE FOR nome = "B"
LIST nome, altura WHILE nome = "B" // Percorro enquanto (while).
```

O meu retorno será :

```
Beatriz Nogueira
Bianca Vasconcelos
```

Note que dessa vez veio um registro a menos que a mesma operação com o comando FOR. Por que isso aconteceu ? Vamos mostrar (de novo) a lista completa para você tentar descobrir. Caso você não consiga descobrir a resposta, basta seguir a leitura, pois ela está logo após a exibição da lista.

```
Ana Maria
Beatriz Nogueira
Bianca Vasconcelos
Francisca Miranda
Berenice Assunção
Zenia Araújo
```

Vamos analisar a busca em detalhes:

1. Primeiro, sabemos que o ponteiro deve estar sobre o primeiro registro que satisfaça a condição. Usamos o comando LOCATE para “achar” esse registro e posicionar o ponteiro.
2. Passamos a exibir os registros "enquanto"(while) o campo NOME começar com a letra "B".
3. Encontramos um nome que começa com "F"(Francisca Miranda), e a cláusula WHILE abandona a busca. Note que o nome "Berenice Assunção"não é exibido, pois encontra-se após "Francisca Miranda".

Conclusão: para que a cláusula WHILE funcione corretamente, ela precisa atuar em um arquivo ordenado. Se os registros estivessem conforme a lista a seguir (ordem alfabética por nome), a cláusula WHILE funcionaria corretamente e com um detalhe adicional: ela seria muito mais rápida do que uma busca com a clausula FOR, que percorre o arquivo completamente sem necessidade.

```
Ana Maria
Beatriz Nogueira
Berenice Assunção
Bianca Vasconcelos
Francisca Miranda
Zenia Araújo
```

Conclusão parcial: a ordenação dos registros obedecendo a uma determinada ordem aumenta a velocidade das buscas. Na próxima seção nós aprenderemos uma forma (ainda não é a ideal) de se ordenar seus registros.

29.5.1 Uma solução parcial para o problema: a ordenação de registros.

Os registros de um banco de dados podem ser ordenados obedecendo a um determinado padrão estabelecido por você: ordem por data de nascimento, ordem alfabética de nome, etc. O Harbour faz isso através do comando SORT, mas esse método possui alguns inconvenientes:

1. Eu não posso ordenar o arquivo que está aberto. O que eu posso fazer é usar o arquivo aberto e criar um outro arquivo com a ordenação que eu desejo dar.
2. Quanto maior o tamanho do arquivo, maior o tempo necessário para a ordenação.
3. Se eu incluir um novo registro no meu arquivo original eu vou ter que realizar o processo de ordenamento todo de novo para poder gerar um novo arquivo com o novo registro recém-criado.

Vamos fazer um teste usando o HBRUN:

.:Resultado:.

```
USE PACIENTE
SORT TO PACIENTE2 ON NOME
USE PACIENTE2
LIST NOME, NASCIMENTO, PESO, ALTURA
```

O resultado está listado logo a seguir :

Figura 29.27: Ordenando um arquivo com SORT

```
  1  Armor                1994-12-16  73.00 1.55
  2  Arneles              1981-12-15  63.00 1.51
  3  Artemisa             1992-12-13 114.00 1.70
  4  Artemisia            1972-12-12  60.00 1.95
  5  Ashley               1988-11-23  88.00 1.52
  6  Astride              1986-12-30  65.00 1.58
  7  Atlante              1987-10-28 112.00 1.88
  8  Atlantida            1988-10-26 109.00 1.99
  9  Audria               1992-12-23  91.00 1.98
 10  Augustos             1968-12-17 104.00 1.60
 11  Aurelio              1959-11-30  64.00 1.71
 12  Aurete               1964-10-30  76.00 1.99
 13  Bianca               1956-12-11  65.00 1.65
 14  Bianor               1968-10-30  85.00 1.61
 15  Bila                 1990-10-14  79.00 1.61
 16  Bina                 1960-11-12  62.00 1.84
 17  Blaise               1961-11-20  96.00 1.80
 18  Blandina             1982-12-28 101.00 1.51
 19  Boris                1990-10-20 113.00 1.77
 20  Bosco                1981-10-30  63.00 1.74
 21  Brasilina            1957-11-27  95.00 1.52
 22  Breno                1976-11-20  79.00 1.97
 23  Brian                1952-11-19  87.00 1.95
 24  Brilhante            1976-10-26  70.00 1.78
 25  Brilhantina          1981-10-16  87.00 1.83
 26  Gertrudes            1992-10-18  96.00 1.73
 27  Kesia                1958-12-23 108.00 1.86
 28  Zenia                1973-12-24  63.00 1.97
```

Lembre-se, que o arquivo original não foi alterado. Esse arquivo listado na figura 29.27 é o arquivo “paciente2.dbf” que foi gerado pelo comando SORT.

Dica 148

Use o comando SORT para criar “fotografias” do seu banco de dados de tempos em tempos. Use uma planilha eletrônica (como o Excel[a]) para abrir o seu arquivo recém-ordenado e analisar os seus dados mais rapidamente.

[a] O Microsoft Excel consegue abrir diretamente um arquivo DBF. Não abra seu arquivo original usando aplicativos de terceiros (como o Excel ou o Calc), o ideal é criar uma cópia ordenada para analisar com os recursos da planilha.

Ao agrupar os registros em uma ordem (geralmente alfabética) o comando SORT resolve o antigo problema gerado pelas buscas com a cláusula WHILE. Mas vimos que ele ainda tem muitos inconvenientes. Por exemplo: você ainda precisa percorrer sequencialmente o arquivo (com o comando LOCATE ou SKIP) para poder achar o ponto onde o subconjunto desejado tem inicio. Por exemplo, no arquivo “paciente2.dbf”, caso eu queira listar os pacientes que iniciam com a letra “G” eu terei que percorrer sequencialmente o arquivo até chegar no registro 26 (paciente “Gertrudes”). A vantagem sobre o outro método é que, quando o nome do próximo paciente não começar mais pela letra “G” eu terei a certeza de que a minha busca terminou e não preciso mais

percorrer o restante do arquivo. **Conclusão:** arquivos ordenados ajudam a lhe informar quando um determinado subconjunto de dados termina, mas não lhe informa quando esse mesmo subconjunto se inicia. Eu sempre terei que percorrer o arquivo desde o início para achar o ponto onde o subconjunto se inicia.

29.5.2 Uma solução definitiva para o problema: a indexação de registros.

A solução ideal para as buscas em um arquivo chama-se "indexação". Um índice é um outro tipo de arquivo (não é um DBF), cuja função é manter um registro de onde os dados no arquivo original se encontram. Vamos voltar para o HBRUN e fazer alguns testes.

Primeiramente abra o arquivo "paciente.dbf" e liste o seu conteúdo:

.:Resultado:.

```
CLS
USE PACIENTE
LIST NOME, NASCIMENTO, PESO, ALTURA
```

Vamos reproduzir o resultado a seguir :

Figura 29.28: Listando o conteúdo de paciente.dbf

```
 1  Armor                1994-12-16  73.00 1.55
 2  Arneles              1981-12-15  63.00 1.51
 3  Artemisa             1992-12-13 114.00 1.70
 4  Artemisia            1972-12-12  60.00 1.95
 5  Ashley               1988-11-23  88.00 1.52
 6  Kesia                1958-12-23 108.00 1.86
 7  Astride              1986-12-30  65.00 1.58
 8  Gertrudes            1992-10-18  96.00 1.73
 9  Atlante              1987-10-28 112.00 1.88
10  Atlantida            1988-10-26 109.00 1.99
11  Audria               1992-12-23  91.00 1.98
12  Augustos             1968-12-17 104.00 1.60
13  Aurete               1964-10-30  76.00 1.99
14  Aurelio              1959-11-30  64.00 1.71
15  Bianca               1956-12-11  65.00 1.65
16  Bianor               1968-10-30  85.00 1.61
17  Bila                 1990-10-14  79.00 1.61
18  Bina                 1960-11-12  62.00 1.84
19  Blaise               1961-11-20  96.00 1.80
20  Blandina             1982-12-28 101.00 1.51
21  Boris                1990-10-20 113.00 1.77
22  Bosco                1981-10-30  63.00 1.74
23  Zenia                1973-12-24  63.00 1.97
24  Brasilina            1957-11-27  95.00 1.52
25  Breno                1976-11-20  79.00 1.97
26  Brian                1952-11-19  87.00 1.95
27  Brilhante            1976-10-26  70.00 1.78
28  Brilhantina          1981-10-16  87.00 1.83
```

Note que a paciente "Zenia" ocupa a posição 23 de um total de 28 registros e que o último registro é a paciente "Brilhantina". Isso indica que o arquivo não está ordenado por nome.

Vamos agora criar um índice para esse arquivo chamado de paciente01.ntx[11]

Para criarmos um índice nós precisamos informar qual campo irá servir de critério para ordenação. No nosso caso, o campo será o NOME. Esse campo recebe o nome

[11] Os índices que nós iremos estudar possuem a extensão NTX, mas existem outros padrões. Usaremos esse porque ele é o padrão do Harbour.

de “chave do índice”. No exemplo a seguir o arquivo “paciente01.ntx” possui a sua chave formada apenas pelo campo NOME[12].

.:Resultado:.

```
INDEX ON NOME TO PACIENTE01
LIST NOME, NASCIMENTO, PESO, ALTURA
```

A imagem abaixo mostra os últimos registros exibidos pelo comando LIST.

Figura 29.29: Listando o conteúdo de paciente.dbf com a chave de índices criada

Note que agora os registros estão ordenados. Na verdade o termo correto é “indexado”, pois os registros ainda estão na ordem natural em que foram inseridos no nosso arquivo. Note, na figura 29.29, que o número que indica a posição do registro não se alterou (“Zenia” continua na posição 23 e “Brilhantina” continua na posição 28). Isso serve para mostrar que a ordem não foi alterada.

Os registros foram postos em uma ordem apenas porque o arquivo de índices que você criou (com o comando INDEX) está ativo. Faça o seguinte teste: feche o arquivo “paciente.dbf” e abra ele de novo, conforme abaixo :

```
CLS
USE PACIENTE // Abre o arquivo, mas agora sem o índice
LIST NOME
```

Note que os registros agora aparecem na ordem anterior. Ou seja, eles não estão mais indexados. Isso acontece porque você precisa abrir o arquivo de índices juntamente com o arquivo original de dados sempre que for trabalhar com ele. Para fazer isso sem ter sempre que recriar o arquivo de índices, use o comando SET INDEX TO, conforme o fragmento a seguir:

```
CLS
USE // Fecha o arquivo aberto
USE PACIENTE // Abre o arquivo
SET INDEX TO PACIENTE01 //  PACIENTE01.NTX
LIST NOME
```

Agora sim, eles estão em ordem alfabética de novo. Essa é o básico que nós devemos saber sobre os índices. Vamos agora, nas próximas subseções aprender mais sobre indexação.

[12]Existem índices cujas chaves são compostas por vários campos, como por exemplo: FIRSTNAME+LASTNAME, BAIRRO+ENDERECO, etc. Veremos os índices compostos ainda nesse capítulo.

Dica 149

Abrir um arquivo com o seu respectivo índice (usando SET INDEX TO) é muito mais rápido do que ter que criar sempre o índice (usando INDEX ON). O comando INDEX ON deve ser usado esporadicamente.

29.5.3 A importância de manter o índice atualizado.

Trabalhar com índices requer cuidados adicionais. O mais importante deles é abrir o arquivo de índices sempre que você for abrir o arquivo DBF que contém os dados. Vamos realizar uma pequena experiência. Usando o HBRUN abra o arquivo "paciente.dbf" sem abrir o seu índice "paciente01.ntx".

.:Resultado:.

```
USE PACIENTE
```

Agora insira um registro novo com a paciente "Ana Maria".

.:Resultado:.

```
APPEND BLANK
REPLACE NOME WITH ``Ana Maria''
```

Agora feche o arquivo e abra ele de novo com o índice criado anteriormente em "paciente01.ntx".

.:Resultado:.

```
CLS
USE PACIENTE // Abre o arquivo na área do anterior(fecha o anterior)
SET INDEX TO PACIENTE01 //  PACIENTE01.NTX
LIST NOME
```

Note que o registro recém incluso não apareceu na listagem:

Figura 29.30: Onde está a paciente Ana Maria ?

Isso aconteceu porque você abriu o arquivo juntamente com um índice desatualizado.

Dica 150

Quando abrir um arquivo, não esqueça de abrir os seus índices também.

E agora ? Como nós resolvemos isso ?
Para trazer a paciente “Ana Maria” de volta, você precisa indexar de novo o arquivo :

.:Resultado:.

```
CLS
USE PACIENTE // Abre o arquivo na área do anterior(fecha o anterior)
```

```
INDEX ON NOME TO PACIENTE01
LIST NOME
```

Pronto. Ana Maria está de volta.

29.5.4 O comando SEEK

Lembra da comparação que nós fizemos entre o comando LIST com a cláusula FOR e esse mesmo comando com a cláusula WHILE ? Acompanhe o fragmento a seguir para recordar onde nós paramos.

.:Resultado:.

```
USE paciente
LOCATE FOR nome = "B"
LIST nome, altura WHILE nome = "B" // Percorro enquanto (while).
```

Vamos recordar as conclusões tiradas :

1. Para que a cláusula WHILE funcione corretamente, ela precisa atuar em um arquivo ordenado.
2. Arquivos ordenados ajudam a lhe informar quando um determinado subconjunto de dados termina, mas não lhe informam quando esse mesmo subconjunto se inicia. Eu sempre terei que percorrer o arquivo desde o início para achar o ponto onde o subconjunto se inicia.

Pois bem, nós já vimos a diferença entre um arquivo ordenado e um arquivo indexado, agora vamos a mais uma diferença importantíssima: **arquivos indexados possuem a facilidade de informar ao usuário onde o subconjunto se inicia.** Dessa forma, eu posso encontrar rapidamente o início do subconjunto sem ter que percorrer o arquivo desde o início como havíamos fazendo. O comando SEEK faz essa busca conforme o fragmento a seguir:

.:Resultado:.

```
USE paciente
SET INDEX TO paciente01
SEEK "B"
? nome
```

Compare esse fragmento com o fragmento anterior. Veja que fizemos duas coisas: a primeira delas foi abrir o índice que possui a chave de indexação pelo campo NOME e a segunda foi trocarmos o comando LOCATE pelo comando SEEK. Note que nós não precisamos informar o campo no comando SEEK pois isso é desnecessário, já que o SEEK busca usando a chave do índice que está aberto. Como a chave do arquivo "paciente01.ntx" é o campo NOME do arquivo "paciente.dbf", eu só preciso informar "SEEK" seguido da string correspondente a chave.

Descrição sintática 45

1. Nome : SEEK
2. Classificação : comando.
3. Descrição : Pesquisa um índice através de um valor chave especificado.
4. Sintaxe

```
SEEK <expPesquisa>

<expPesquisa> : é uma expressão à qual a
chave de indexação deverá corresponder.
```

Fonte : [Nantucket 1990, p. 4-97]

Existem duas instruções que operam em conjunto com o comando SEEK. O primeiro deles é a função FOUND(), já vista anteriormente em conjunto com o comando LOCATE. Com o SEEK ela funciona da mesma maneira: se o comando SEEK achar o valor descrito na chave, então FOUND() retorna verdadeiro (.t.), caso contrário FOUND() retorna falso (.f.).

Existe também um comando que determina se a busca deve ser exata ou aproximada. Atualmente estamos buscando conforme o critério de igualdade de strings, mas existe um outro critério de busca que é determinado pelo comando SET SOFTSEEK. O valor padrão de SET SOFTSEEK é OFF. Mas se o valor de SET SOFTSEEK estiver ON então o comando SEEK irá se posicionar em um registro que mais se aproxime do valor buscado. Por exemplo, acompanhe o fragmento abaixo :

.:Resultado:.

```
USE paciente
SET INDEX TO paciente01
SEEK "C"
? FOUND()
```

O valor de FOUND() é falso, pois não existe paciente cujo nome comece com a letra "C". Agora faça o seguinte teste com SET SOFTSEEK ON:

.:Resultado:.

```
USE paciente
SET INDEX TO paciente01
SET SOFTSEEK ON  // Ativa a busca relativa
SEEK "C"
? FOUND()
```

Agora o valor de FOUND() retornou verdadeiro. Como SET SOFTSEEK está ON a busca será realizada mas, caso o valor não seja encontrado, o ponteiro **não vai** para o final do arquivo, mas sim para o próximo registro que “mais se aproxime” da chave buscada. O próximo registro, nesse caso, é um registro cujo nome comece com uma letra qualquer “maior do que” a letra “C”. No nosso caso é a paciente “Gertrudes”.

.:Resultado:.

```
? NOME // Imprime "Gertrudes"
```

Agora, para encerrar, vamos voltar ao fragmento do comando LIST com a cláusula WHILE que nós iremos repetir, mais uma vez, a seguir :

.:Resultado:.

```
USE paciente
LOCATE FOR nome = "B"
LIST nome, altura WHILE nome = "B" // Percorro enquanto (while).
```

Agora vamos resolver o problema da busca do primeiro elemento do subconjunto através do comando SEEK.

.:Resultado:.

```
USE paciente
SET INDEX TO paciente01
SEEK "B"
LIST nome, altura WHILE nome = "B" // Percorro enquanto (while).
```

Esse modelo de busca é muito mais rápido do que os modelos anteriores e é amplamente utilizado pelos programadores Harbour.

29.5.5 Funções na chave do índice

Você também pode indexar um arquivo usando uma função do Harbour ou até mesmo uma função criada por você. O fragmento a seguir ilustra uma criação de uma chave composta por uma função criada por você.

```
USE PACIENTE
INDEX ON IMC( NOME ) TO TEMP
```

Agora, muita atenção! Quando você fizer isso você deve buscar o dado usando essa função também juntamente com o comando SEEK.

```
SEEK IMC( cNome )
```

Dica 151

Quando for usar uma função criada por você como chave de um índice não esqueça de linkar essa mesma função com a aplicação.

Indexando datas

Quando você for indexar um campo data você deve, necessariamente, usar uma função, conforme o fragmento a seguir:

```
USE PACIENTE
INDEX ON DTOS( NASCIMENTO ) TO PACIENTE02
```

A função DTOS() retorna uma data no formato ano-mês-dia. Esse formato é ideal para ordenações.

Da mesma forma que o exemplo anterior, quando for buscar um valor use, preferivelmente, a mesma função com o comando SEEK, conforme o fragmento a seguir:

```
SEEK DTOS( dData )
```

Dica 152

Na verdade você não é obrigado a usar a função DTOS() para realizar buscas no índice. O que você tem que saber é que a string de busca deve ser no mesmo formato do retorno da função usada na indexação. Por exemplo, se eu quero buscar a data 01/12/2012 eu posso fazer assim :

```
SEEK "20121201"
```

Uma chave de um índice NÃO DEVE possuir uma função que retorna uma string com tamanhos variados.

Muito cuidado quando for usar funções que alteram o tamanho de um campo, pois o Harbour não irá trabalhar corretamente com elas. Por exemplo, as funções ALLTRIM(), TRIM(), RTRIM() E LTRIM() enquadram-se nessa categoria, portanto, evite-as.

Dica 153

Um banco de dados padrão DBF inclui os espaços em branco como componentes de uma chave de índice e leva em consideração o tamanho final desse campo. Se ele for variável, erros estranhos acontecerão na sua aplicação. Você pode usar funções para indexar campos caracteres, mas o retorno dessas funções deve ser uma string de tamanho uniforme. Por exemplo : se o campo NOME possui tamanho 50, você pode indexar usando a função LEFT(NOME, 30), que levará em consideração apenas os 30 primeiros caracteres do campo. Mas você não pode indexar usando a função ALLTRIM(NOME), pois isso gerará registros de tamanhos variáveis.

29.5.6 Indexando campos lógicos

O Harbour não aceita valores lógicos na chave do índice. Para contornar essa limitação você pode usar a função IIF() conforme o fragmento a seguir.

```
INDEX ON IIF( CAMPOLOGICO , "T" , "F" ) TO TEMP
```

Para buscar os valores use o comando SEEK conforme os fragmentos abaixo:

```
SEEK "T"
```

ou

```
SEEK "F"
```

Isso porque são esses os retornos da função IIF() usada na indexação.

29.5.7 Criando índices compostos

As regras para a criação de índices compostos são as mesmas usadas na concatenação de variáveis. Lembre-se, que quando for concatenar vários tipos diferentes você deve converter os tipos todos para caractere para depois converter. O exemplo abaixo mostra a criação de uma chave composta:

```
USE PACIENTE
INDEX ON NOME + DTOS( NASCIMENTO ) + STRZERO( ALTURA , 6 , 2 ) TO TEMP
```

A busca com SEEK deve incluir essas funções também:

```
SEEK cNome + DTOS( dNascimento ) + STRZERO( nAltura , 6 , 2 )
```

ou então deve obedecer ao formato de retorno dessas funções, conforme o fragmento a seguir:

```
SEEK "Marcia          " + "20120109" + "000001.60"
```

O fragmento a seguir mostra uma chave que lista primeiro os pacientes nascidos no mês de janeiro, e assim sucessivamente, até dezembro.

```
USE PACIENTE
INDEX ON STR( MONTH( NASCIMENTO ) ) + NOME TO TEMP
```

29.5.8 Criando vários índices

Como você já deve ter percebido, são muitas as opções de pesquisa que nós podemos criar usando índices. Dessa forma, nós podemos ter vários índices associados a um arquivo. O fragmento abaixo cria 3 índices.

```
USE PACIENTE
INDEX ON NOME TO PACIENTE01
INDEX ON STRZERO( MONTH( NASCIMENTO ) , 2 ) + NOME TO PACIENTE02
INDEX ON STR( ALTURA ) TO PACIENTE03
```

Em uma outra rotina, já com os índices criados, você deverá abrir o arquivo dessa forma.

```
USE PACIENTE
SET INDEX TO PACIENTE01, PACIENTE02, PACIENTE03
```

Agora temos uma dúvida: qual desses índices é o ativo no momento ? Via de regra, sempre o primeiro da listagem do comando SET INDEX é o índice ativo no momento. Então, caso você deseje realizar uma busca com SEEK você deve informar um nome a ser buscado.

```
SEEK "Marcia"
```

Para mudar a índice ativo use o comando SET ORDER. Por exemplo, eu quero mudar para o segundo índice para fazer uma busca pelo mês:

```
SET ORDER TO 2 // Agora o índice é o segundo
SEEK "05" // A primeira ocorrência de um nascimento no mês de maio.
```

Dica 154

Ao utilizar índices múltiplos, certifique-se de que todos eles estejam abertos antes que você escreva no banco de dados. [Spence 1991, p. 423].

Dica 155

O único problema com SET ORDER TO é que a posição numérica torna o código difícil de ler. Se você tem vários índices, procure examinar o comando USE ou SET INDEX precedente para saber qual é o quarto índice. Depois você deve criar uma constante manifesta para eliminar esse indesejável número mágico. O fragmento a seguir ilustra isso :

```
#define L_NAME_ORDER 1
#define TELEFONE_ORDER 2
PROCEDURE Main()

USE TELS NEW
SET INDEX TO NOME, TELEFONE

... Seu código

SET ORDER TO L_NAME_ORDER // Para poder buscar por nome

... Sua busca

SET ORDER TO TELEFONE_ORDER // Para buscar por telefone

// etc e etc

RETURN
```

[Spence 1991, p. 423]

Dica 156

Fique atento ao local do ponteiro do registro. Lembre-se que o início de arquivo em um índice não é o início em outro. Por isso nós alertamos no início desse capítulo que GO 1 é diferente de GO TOP. Acompanhe o exemplo a seguir, retirado de [Spence 1991, p. 424]

```
#define L_NAME_ORDER 1
#define TELEFONE_ORDER 2
PROCEDURE Main()

USE TELS NEW
SET INDEX TO NOME, TELEFONE
// etc e etc
SET ORDER TO L_NAME_ORDER
GO TOP // Vou para o primeiro registro na ordem do nome
// etc
SET ORDER TO TELEFONE_ORDER
// Aqui eu ainda estou no mesmo registro
// , mas ele não é mais o primeiro pois a ordem agora é outra
// ele pode ser o segundo, o quinto, o décimo, etc.
// pois a ordem agora é por telefone
RETURN
```

Dica 157

SET ORDER TO 0 corresponde a ordem natural do arquivo. Ou seja, os índices abertos continuam ativos, mas a ordem de exibição ignora qualquer índice. Spence nos aconselha : "se a ordem não afetar uma operação em particular, configure-a para 0". [Spence 1991, p. 424].

29.6 Conclusão

Pontos importantes que merecem atenção especial

1. todo arquivo DBF possui um ponteiro de registro;
2. qualquer operação de alteração simples irá incidir sobre o registro corrente, que é aquele no qual o ponteiro de registro está posicionado;
3. todo arquivo dbf possui um registro extra chamado de "registro fantasma". Esse registro está posicionado após o final do arquivo e possui as características de cada campo (nome, tamanho e tipo de dado);
4. quanto for excluir um registro lembre-se de que esse registro será apenas marcado para a exclusão;
5. você deve criar os seus índices caso queira encontrar determinada informação com rapidez e eficiência;

6. você pode ter vários índices abertos associados ao mesmo arquivo;
7. quando você for incluir ou alterar algum dado no seu arquivo, certifique-se de que os índices estejam abertos para que eles recebam as alterações também;

30 Arquivos DBFs : Modo programável

> Pedi, e dar-se-vos-á; buscai e encontrareis; batei, e abrir-se-vos-á. Porque aquele que pede recebe; e o que busca encontra; e, ao que bate, se abre.
>
> Mateus 6:7-8

Objetivos do capítulo

- Aprender a desenvolver um programa em Harbour que use arquivos DBFs
- Entender o conceito de área de trabalho no modo programável
- Manipular índices e entender a sua importância
- Usar as principais funções manipuladoras de banco de dados do Harbour.

30.1 O Harbour e o modo programável.

Agora, vamos desenvolver um programa simples que realiza as operações que nós estudamos. O exemplo a seguir (listagem 30.1) lhe ajudará a entender como isso funciona. Note que os mesmos comandos que nós usamos no modo interativo (via HBRUN) podem ser usados no modo programável a seguir.

Listagem 30.1: Adicionando dados a um arquivo DBF
Fonte: codigos/dbf02.prg

```
1
2  PROCEDURE Main
3  LOCAL aStruct := {  { "NOME" , "C" , 50, 0 },;
4                      { "NASCIMENTO" , "D" , 8 , 0 },;
5                      { "ALTURA" , "N" , 4 , 2 },;
6                      { "PESO" , "N" , 6 , 2 } }
7
8
9      IF .NOT. FILE( "paciente.dbf" )
10        DBCREATE( "paciente" , aStruct )
11        ? "Arquivo de pacientes criado."
12     ENDIF
13
14     USE paciente // Abrindo o arquivo paciente.dbf
15     ? "Inserindo dados no arquivo paciente.dbf"
16     APPEND BLANK
17     REPLACE NOME WITH "Miguel"
18     REPLACE NASCIMENTO WITH STOD("19900209") // 9 DE FEV DE 1990
19     REPLACE ALTURA WITH 1.70
20     REPLACE PESO WITH 65
21
22 RETURN
```

Aparentemente o programa somente exibe a mensagem reproduzida abaixo:

.:Resultado:.

```
Inserindo dados no arquivo paciente.dbf
```

Se o arquivo “paciente.dbf” não existir ele exibirá a saída a seguir:

.:Resultado:.

```
Arquivo de pacientes criado.
Inserindo dados no arquivo paciente.dbf
```

No entanto, o programa realiza outras tarefas adicionais. A seguir temos uma rápida descrição do que o programa realiza:

1. Na linha 3 o programa define um array com a estrutura do arquivo a ser criado.
2. As linhas 9 até 12 verificam se o arquivo “paciente.dbf” existe, caso ele não exista então ele é criado (linha 10).
3. A linha 14 abre o arquivo “paciente.dbf”.

4. A linha 16 cria um registro em branco para receber os dados
5. As linhas 17 até 20 gravam os dados de um paciente.

30.1.1 Área de trabalho

Nós já vimos que um arquivo DBF aberto ocupa uma área de trabalho pré-existente. Por exemplo, o arquivo notas.dbf do fragmento abaixo ocupa a área de trabalho 1.

```
PROCEDURE Main

   USE notas

RETURN
```

Um cuidado adicional que você, enquanto programador, deve ter é **não abrir mais de um arquivo na mesma área de trabalho**. O fragmento de código abaixo representa um erro comum entre os programadores iniciantes. Isso porque eu não informei que o arquivo de clientes deveria ser aberto em uma área nova (eu não selecionei a área), e a consequência disso é que o comando USE clientes usou a mesma área do comando USE notas. No final das contas, apenas o arquivo clientes ficará aberto.

```
PROCEDURE Main

   USE notas
   USE clientes
   // Apenas o arquivo clientes está aberto.

RETURN
```

Dica 158

O comando USE abre um arquivo informado, mas ele fecha o arquivo que estava aberto anteriormente na área de trabalho ativa. Para que eu abra o arquivo sempre em uma área de trabalho livre eu devo usar a cláusula NEW. Conforme o fragmento abaixo.

```
PROCEDURE Main

   USE notas
   USE clientes NEW
   // Os arquivos notas e clientes estão abertos
RETURN
```

Utilizando um ALIAS para o arquivo

ALIAS é o nome dado a um apelido que você dá para poder referenciar um arquivo de dados. Algumas vezes o nome do arquivo é muito complicado de se memorizar, e nessas horas a cláusula ALIAS pode lhe ajudar. Por exemplo, suponha que o nome do arquivo que você está abrindo é not001.dbf. Você pode querer algo mais claro, como

"Nota" ou "NotaFiscal", portanto nessas horas você pode usar a cláusula ALIAS do comando USE.

Dica 159

Por que alguém iria criar um arquivo chamado not001.dbf ? Não seria mais prático usar o nome nota.dbf ?
O que está por trás dessas perguntas é o seguinte questionamento: porque eu tenho que utilizar o ALIAS se eu já poderia criar um nome amigável para os meus arquivos ?
Para responder a esse questionamento pense antes na seguinte situação: a sua empresa desenvolveu um sistema de contabilidade que é multi-empresa, ou seja, o sistema consegue controlar a contabilidade de inúmeras empresas. Para que os arquivos não fiquem enormes com dados de inúmeras empresas, nós resolvemos criar para cada empresa seus próprios arquivos. Dessa forma nós temos as notas fiscais da empresa 001 (not001.dbf), as notas fiscais da empresa 002 (not002.dbf) e assim por diante. Contudo, os arquivos do seu programa referenciam as notas fiscais apenas pelo nome "notas". É nessas horas que a cláusula ALIAS pode lhe ajudar. Você poderia fazer conforme o código a seguir :

```
USE not001 ALIAS nota
```

Outra razão é quando você está dando manutenção em um sistema que foi criado por outro programador. Você pode, por exemplo, criar uma rotina que irá ter que realizar operações em um arquivo cujo nome não foi você que criou. Nesse caso, você pode achar conveniente se referir aos arquivos desse sistema usando um nome mais claro para você.

O comando SELECT e a função SELECT()

Quando o assunto é área de trabalho o Harbour dispõe de duas instruções bastante úteis para o controle dessas áreas: uma função chamada SELECT() e um comando também com o mesmo nome. O funcionamento dessas instruções são ligeiramente diferentes e você precisa estar atento para não cometer erros de lógica no seu código. Vamos começar pelo comando SELECT.

Descrição sintática 46

1. Nome : SELECT
2. Classificação : comando.
3. Descrição : Muda a área de trabalho corrente.
4. Sintaxe

```
SELECT <xnArea>|<idAlias>
```

Fonte : [Nantucket 1990, p. 4-98]

Os dois exemplos a seguir ilustram o uso do comando SELECT.

```
USE CLIENTE NEW
USE COMPRAS NEW
USE NOTAS NEW
// Aqui temos 3 arquivos abertos e a área de trabalho
// selecionada é a número 3, que contém o arquivo NOTAS

SELECT CLIENTE
// Agora a área corrente é a 1, ou a que tem o arquivo CLIENTES
```

Uma outra forma, bem menos usada, de se usar o comando SELECT é indicando um número no lugar do nome do ALIAS. Por exemplo, no nosso código anterior, para selecionarmos o arquivo de CLIENTE nós poderíamos fazer assim :

```
SELECT 1 // Seleciona a área 1 (CLIENTE)
```

Essa forma acima, informando o número da área de trabalho, era muito usada em códigos dBase (início da década de 1980), porque essa era a única forma de se abrir um arquivo em uma área de trabalho nova. O fragmento abaixo **não era possível nessa época** :

```
USE CLIENTE NEW
USE COMPRAS NEW
USE NOTAS NEW
```

A cláusula NEW não existia nessa época, portanto, para realizarmos a operação acima teríamos que fazer :

```
SELECT 1
USE CLIENTE
SELECT 2
USE COMPRAS
SELECT 3
USE NOTAS
SELECT 4 // Não está errado, eu apenas selecionei uma área vazia
```

A consequência prática disso é que você pode usar o comando SELECT informando um número de uma área de trabalho que não está ocupada. É bom ressaltar que: **se você informar o nome de um ALIAS inexistente irá gerar um erro de execução**, conforme o fragmento abaixo :

```
USE CLIENTE
USE COMPRAS NEW
USE NOTAS NEW

SELECT VENDAS // ERRO DE EXECUÇÃO POIS O ALIAS NÃO EXISTE
```

Agora vamos passar para a função SELECT().

Descrição sintática 47

1. Nome : SELECT()
2. Classificação : função.
3. Descrição : Determina o número da área de trabalho de um alias especificado.
4. Sintaxe

```
SELECT( [<cAlias>] ) -> O número da área de trabalho
```

Fonte : [Nantucket 1990, p. 5-206]

A função SELECT() não muda a área de trabalho, ela apenas retorna o número de uma área de trabalho. Por exemplo, no fragmento a seguir a função SELECT() deve retornar o valor 3.

```
USE CLIENTE
USE COMPRAS NEW
USE NOTAS NEW

? SELECT()  // 3
```

Se eu não passar argumento algum para a função SELECT(), ela irá retornar o número da área corrente. Mas, se eu passar um argumento para SELECT(), ela apenas irá retornar o número da área, mas não irá selecioná-la. O fragmento a seguir o valor da função SELECT("CLIENTE") é 1, mas a área ativa é a 3.

```
USE CLIENTE
USE COMPRAS NEW
USE NOTAS NEW

? SELECT("CLIENTE") // 1
? SELECT() // 3
```

Se o alias não existir, SELECT() retorna zero, conforme abaixo.

```
USE CLIENTE
USE COMPRAS NEW
USE NOTAS NEW

? SELECT(``VENDAS'') // RETORNA ZERO
```

Dica 160

Cuidado para não confundir o uso da função SELECT() com o uso do comando SELECT. Um comando pode ter parênteses envolvendo os seus argumentos, desde que tenha um espaço. Como SELECT é o nome de um comando e de uma função você deve ter cuidado com os espaços após o nome.O exemplo a seguir ilustra isso :

```
FUNCTION MAIN()
LOCAL x

   USE NOTA

   USE ITEM NEW

   SELECT NOTA // A área atual é a 1 (NOTA)

   x := SELECT( "ITEM" ) // Apenas verifiquei se esse arquivo está ocupando

   ? x // A funçao SELECT() retorna apenas o valor, mas não muda de área
   ? "A AREA ATUAL E ", ALIAS()  // A área ainda é NOTA

   SELECT ("ITEM") // O comando SELECT muda de área
   ? "A AREA ATUAL E ", ALIAS()


   // Apenas um espaço faz muita diferença SELECT( carea ) <> SELECT (carea)

RETURN NIL
```

Cuidado quando for usar o comando SELECT com parênteses. Um comando exige um espaço entre o seu nome e o parênteses.

30.1.2 Aprimorando o programa de inclusão de pacientes

O programa da listagem 30.1 pode ser melhorado em alguns pontos essenciais. De acordo com o criador da linguagem C++, “escrever um programa implica apurar gradualmente as ideias a respeito do que você quer fazer e como expressar isso.” [Stroustrup 2012, p. 219]. Nas próximas subseções nós iremos implementar algumas melhorias no respectivo programa.

Como você sabe se o arquivo paciente.dbf está aberto ?

Muitas vezes um processo de abertura de um arquivo não funciona como deveria e o arquivo não é aberto. Quando isso acontece o programa irá apresentar erros nas linhas subsequentes. Por exemplo, se o programa da listagem 30.1 não conseguir abrir o arquivo “paciente.dbf” (linha 14) um erro de execução será gerado na linha 16, pois o comando APPEND BLANK não pode realizar uma operação em um arquivo que não foi aberto. Da mesma forma os comandos REPLACEs que estão nas linhas 17 até 20. A única coisa a fazer é verificar se o arquivo foi aberto corretamente logo após a sua abertura. A listagem 30.2 modifica a listagem anterior e faz essa verificação através da função USED() (linha 15).

Listagem 30.2: Verificando se o arquivo foi aberto com sucesso.
Fonte: codigos/dbf03.prg

```
1
2  PROCEDURE Main
3  LOCAL aStruct := {  { "NOME" , "C" , 50, 0 },;
4                      { "NASCIMENTO" , "D" , 8 , 0 },;
5                      { "ALTURA" , "N" , 4 , 2 },;
6                      { "PESO" , "N" , 6 , 2 } }
7
8
9      IF .NOT. FILE( "paciente.dbf" )
10        DBCREATE( "paciente" , aStruct )
11        ? "Arquivo de pacientes criado."
12     ENDIF
13
14     USE paciente // Abrindo o arquivo paciente.dbf
15     IF .NOT. USED()
16        ? "O arquivo 'paciente.dbf' näo foi aberto com sucesso."
17        ? "A operaçäo será abortada"
18        QUIT
19     ENDIF
20
21     ? "Inserindo dados no arquivo paciente.dbf"
22     APPEND BLANK
23     REPLACE NOME WITH "Miguel"
24     REPLACE NASCIMENTO WITH STOD("19900209") // 9 DE FEV DE 1990
25     REPLACE ALTURA WITH 1.70
26     REPLACE PESO WITH 65
27
28 RETURN
```

A função USED() está descrita logo a seguir:

Descrição sintática 48

1. Nome : USED()
2. Classificação : função.
3. Descrição : USED() é uma função de tratamento de banco de dados utilizada para verificar se há um arquivo de banco de dados em uso em uma área de trabalho específica. .
4. Sintaxe

```
USED() -> lDbfAberto

lDbfAberto : .t. caso haja um arquivo de banco de dados
                 em uso; caso contrário, retorna falso (.f.).
```

Fonte : [Nantucket 1990, p. 5-240]

Como inserir no banco de dados as novas informações digitadas pelo usuário ?

Até agora o nosso exemplo tem sido pouco funcional, porque o valor que ele grava no arquivo “paciente.dbf” sempre é o mesmo. O exemplo a seguir modifica a listagem anterior para poder permitir que o usuário insira um valor.

Listagem 30.3: Gravando dados digitados pelo usuário no banco de dados.
Fonte: codigos/dbf10.prg

```
PROCEDURE Main
FIELD NOME, NASCIMENTO, ALTURA, PESO
LOCAL aStruct := {  { "NOME" , "C" , 50, 0 },;
                    { "NASCIMENTO" , "D" , 8 , 0 },;
                    { "ALTURA" , "N" , 4 , 2 },;
                    { "PESO" , "N" , 6 , 2 } }
LOCAL GetList := {}


    IF .NOT. FILE( "paciente.dbf" )
       DBCREATE( "paciente" , aStruct )
       ? "Arquivo de pacientes criado."
    ENDIF

    USE paciente // Abrindo o arquivo paciente.dbf
    ? "Inserindo dados no arquivo paciente.dbf"

    CLS
    SET DATE BRITISH
    APPEND BLANK
    @ 10,10 SAY "Nome do paciente   : " GET NOME
    @ 12,10 SAY "Data de nascimento : " GET NASCIMENTO
```

```
    @ 14,10 SAY "Altura do paciente : " GET ALTURA
    @ 16,10 SAY "Peso do paciente   : " GET PESO
    READ


RETURN
```

.:Resultado:.

```
          Nome do paciente   :

          Data de nascimento :    /  /

          Altura do paciente :  0.00

          Peso do paciente   :    0.00
```

Vamos digitar alguns dados nesse formulário (lembre-se de teclar ENTER para passar de um campo para outro).

.:Resultado:.

```
          Nome do paciente   :  Charles Haddon Spurgeon

          Data de nascimento :  12/12/34

          Altura do paciente :  1.50

          Peso do paciente   :  87.00
```

Você pode conferir, no HBRUN, se o registro está lá mesmo usando o nosso conhecido comando LIST.

Aprimorando a nossa rotina de inclusão.

A rotina que nós apresentamos cumpre o que promete, mas ela possui algumas melhorias que **devem** ser realizadas pelo programador. A listagem a seguir mostra essa mesma rotina com algumas modificações fundamentais para que o seu cadastro funcione corretamente. As modificações estão comentadas logo após a listagem.

Listagem 30.4: Rotina de inclusão de novo registro com as alterações
Fonte: codigos/dbf11.prg

```
#include "inkey.ch"
PROCEDURE Main
LOCAL cNome := SPACE(50), dNascimento := CTOD("//"), nAltura := 0, nPeso := 0
LOCAL aStruct := {  { "NOME" , "C" , 50, 0 },;
                    { "NASCIMENTO" , "D" , 8 , 0 },;
                    { "ALTURA" , "N" , 4 , 2 },;
                    { "PESO" , "N" , 6 , 2 } }
```

```
LOCAL GetList := {}                                                          8
                                                                             9
    IF .NOT. FILE( "paciente.dbf" )                                          10
       DBCREATE( "paciente" , aStruct )                                      11
       ? "Arquivo de pacientes criado."                                      12
    ENDIF                                                                    13
                                                                             14
    USE paciente // Abrindo o arquivo paciente.dbf                           15
    ? "Inserindo dados no arquivo paciente.dbf"                              16
                                                                             17
    CLS                                                                      18
    SET DATE BRITISH                                                         19
                                                                             20
    @ 10,10 SAY "Nome do paciente   : " GET cNome                            21
    @ 12,10 SAY "Data de nascimento : " GET dNascimento                      22
    @ 14,10 SAY "Altura do paciente : " GET nAltura                          23
    @ 16,10 SAY "Peso do paciente   : " GET nPeso                            24
    READ                                                                     25
    IF LASTKEY() <> K_ESC                                                    26
       APPEND BLANK                                                          27
       REPLACE FIELD->NOME WITH cNome                                        28
       REPLACE FIELD->NASCIMENTO WITH dNascimento                            29
       REPLACE FIELD->ALTURA WITH nAltura                                    30
       REPLACE FIELD->PESO WITH nPeso                                        31
    ELSE                                                                     32
       @ 18,10 SAY "Cancelando..."                                           33
    ENDIF                                                                    34
                                                                             35
                                                                             36
RETURN                                                                       37
```

No exemplo anterior (listagem 30.3) os valores são editados diretamente sobre os registros do banco de dados. Note que o comando GET da primeira listagem (linhas 22 a 25) opera diretamente sobre os campos do registro (NOME, NASCIMENTO, ALTURA e PESO).

Por que essa prática, de editar os dados diretamente sobre o campo e não sobre uma variável é uma prática desaconselhada ?

O principal motivo[1] que inviabiliza a digitação de dados diretamente no campo é a gravação imediata desses valores nos campos do arquivo. Note que no primeiro código (a listagem 30.3) eu tenho que criar um registro em branco (na linha 21), e nas linhas 22 a 25 receber os dados diretamente nos campos desse registro recém-criado. Se por acaso o usuário se arrepender e não quiser mais gravar esses valores aí já será tarde, porque eles já foram gravados e o que é pior: um registro indesejável terá sido gerado. Portanto, a causa principal que inviabiliza a edição de dados direto sobre um campo de arquivo é a impossibilidade que o usuário tem de se arrepender e cancelar a operação. Na segunda listagem as seguintes modificações foram realizadas :

1. As variáveis locais foram criadas (linha 3) para receber os dados digitados pelo

[1] Existem outros motivos, como a dificuldade gerada para se trabalhar em ambiente multiusuário. Não iremos abordar esse motivo porque ainda não vimos como trabalhar em uma rede de computadores.

usuário. Note que cada variável corresponde a um campo do registro que está sendo editado.

2. O APPEND BLANK (criação do registro em branco na linha 27) só é realizado após a digitação dos dados no GET.

3. Temos uma verificação realizada na linha 26 da segunda listagem. Essa verificação basicamente olha se o usuário abandonou a edição do GET pressionando a tecla ESC. Em caso negativo ele cria o registro em branco e realiza as gravações.

É importante verificar a forma como o usuário saiu do formulário. Como nossa programação está abordando o modo texto somente, a verificação da forma de saída é através da última tecla pressionada após o usuário abandonar a edição do formulário. A função LASTKEY() faz essa verificação da seguinte forma: cada tecla que você pressiona gera um código numérico (tudo isso sem que você note). A função LASTKEY() tem o poder de descobrir qual foi o código da última tecla pressionada. O código da tecla ESC é o número 27. Como 27 é um número mágico, o Harbour disponibilizou para você um arquivo com o código de todas as teclas. O nome desse arquivo é “inkey.ch” (incluso na linha 1) e a constante manifesta desse código é K_ESC (linha 26). O que a estrutura de decisão IF faz, na linha 26, é avaliar se a última tecla pressionada foi o ESC e, em caso negativo, realizar as operações de gravação no arquivo (linhas 27 a 31).

Dica 161

É importante não editar os dados diretamente no seu arquivo. Use variáveis para a edição e, conforme a decisão do usuário, grave os dados ou não. O uso de variáveis para a edição é semelhante ao uso de um rascunho antes de se escrever a versão final de uma redação.

Dica 162

Você aprendeu a gravar os dados ou não quando o programa é em modo texto. Quando um programa é em ambiente gráfico (estilo Microsoft Windows), sempre existem dois botões : o "OK"e o "Cancelar". Ou seja, eu não vou precisar avaliar qual foi a última tecla digitada com a função LASTKEY(). Essa forma de programação é chamada de "programação orientada por eventos".

Dica 163

Existe uma forma simples de se inicializar as variáveis locais. Note que as variáveis locais foram declaradas e inicializadas para poderem ser usadas pelo comando GET. No nosso exemplo nós fizemos assim:

```
LOCAL cNome := SPACE(50), dNascimento := CTOD("//")
LOCAL nAltura := 0, nPeso := 0
```

Existe uma forma mais prática de se inicializar essas variáveis: **simplesmente mova o ponteiro do registro para o registro fantasma e pegue os valores dos campos.**
Os valores dos campos do registro fantasma são os mesmos de acordo com a lista abaixo:

1. Campo caractere : o valor será equivalente a função SPACE(nEspaço), onde nEspaço é o tamanho do campo. Por exemplo, como o tamanho do campo NOME é 50, o valor do campo nome do registro fantasma é SPACE(50).
2. Campo numérico : sempre é zero. Se tiver casas decimais essas casas serão zeros também.
3. Campo data : equivalente a função CTOD("//")
4. Campo lógico : sempre será falso.

Como ir para o registro fantasma ?
Simplesmente execute os seguintes comandos abaixo :

```
GO BOTTOM // Vai para o último registro
SKIP  // Pula para o registro fantasma

cNome := FIELD->NOME // Atribui SPACE(50) à variável cNome
dNascimento := FIELD->NASCIMENTO // CTOD("//")
nAltura := FIELD->ALTURA // 0
nPeso := FIELD->PESO  // 0
```

30.1.3 Como alterar, no banco de dados as informações digitadas pelo usuário ?

O processo de alterar as informações, nós já vimos no modo interativo, envolve três operações :

1. Encontrar a informação.
2. Trazer a informação encontrada para o formulário.
3. Gravar a informação editada pelo usuário.

Vamos dedicar a próxima seção para revisar as formas de se encontrar uma informação em um arquivo.

30.1.4 Alteração de usuário: encontrando a informação

Vamos por partes: primeiro, para encontrar a informação nós devemos mover o ponteiro do registro até que determinada condição seja satisfeita. Nós podemos utilizar as seguintes técnicas :

Usar o comando SKIP repetidas vezes até que a condição seja satisfeita

Essa forma é desaconselhada pois ela é ineficiente. Acompanhe a listagem a seguir :

```
ACCEPT "Informe o nome que deseja buscar" TO cNome
GO TOP
DO WHILE .NOT. EOF()
     IF FIELD->NOME = cNome
          ? "Encontrei"
          // Realiza a operação desejada
          EXIT
     ENDIF
     SKIP
ENDDO
```

Usar o comando LOCATE com a cláusula FOR em conjunto com a função FOUND()

Essa é a maneira mais clara, porém tem o mesmo desempenho da rotina anterior (com SKIP). Acompanhe a listagem a seguir :

```
ACCEPT "Informe o nome que deseja buscar" TO cNome
LOCATE FOR FIELD->NOME = cNome
  IF FOUND()
          ? "Encontrei"
          // Realiza a operação desejada
  ENDIF
```

Usar o comando SEEK em um arquivo indexado pela chave correspondente a busca e verificar se achou com a função FOUND()

Essa forma é a mais eficiente de todas pois não precisamos percorrer o registro até encontrar a informação, porém é necessário que o arquivo esteja indexado pelo campo NOME. Acompanhe a listagem a seguir :

```
USE paciente
SET INDEX TO paciente01 // A chave é o NOME
ACCEPT "Informe o nome que deseja buscar" TO cNome
SEEK cNome
  IF FOUND()
          ? "Encontrei"
          // Realiza a operação desejada
  ENDIF
```

30.1.5 Alteração de usuário: "trazendo"a informação encontrada

Quando você encontrar o registro desejado você deve "trazer"a informação encontrada. Por "trazer"nós queremos dizer atribuir os campos às variáveis e montar um formulário com essas variáveis. Como nossa interface usada é modo texto nós utilizaremos o comando GET.

Listagem 30.5: Rotina básica de alteração de registros
Fonte: codigos/dbf12.prg

```
                                                                        1
PROCEDURE Main                                                          2
LOCAL cBusca := SPACE(30)                                               3
LOCAL cNome, dNascimento, nAltura, nPeso                                4
LOCAL GetList := {}                                                     5
                                                                        6
   SET DATE BRITISH                                                     7
                                                                        8
   USE paciente                                                         9
   SET INDEX TO paciente01 // A chave pelo o NOME                       10
                                                                       11
   CLS                                                                 12
   @ 05,05 SAY "Informe o nome do paciente : " GET cBusca ;            13
                                                 PICTURE "@S20"        14
   READ                                                                15
   IF LASTKEY() == K_ESC                                               16
      RETURN                                                           17
   ENDIF                                                               18
   SEEK cBusca                                                         19
   IF FOUND()                                                          20
      cNome := PACIENTE->NOME                                          21
      dNascimento := PACIENTE->NASCIMENTO                              22
      nPeso := PACIENTE->PESO                                          23
      nAltura := PACIENTE->ALTURA                                      24
      @ 07,05 SAY "Nome       : " GET cNome PICTURE "@S30"             25
      @ 09,05 SAY "Nascimento : " GET dNascimento                      26
      @ 11,05 SAY "Altura     : " GET nAltura                          27
      @ 13,05 SAY "Peso       : " GET nPeso                            28
      READ                                                             29
   ENDIF                                                               30
                                                                       31
RETURN                                                                 32
```

Note que as variáveis não precisam ser inicializadas no início do código, mas somente na hora de exibir os seus valores, a partir da linha 20.

30.1.6 Alteração de registro: gravando a informação

Para gravar a informação nós precisamos verificar se o formulário foi abandonado (pela tecla ESC) ou se foi confirmado (Geralmente a tecla ENTER). Para fazer isso nós usamos a função LASTKEY(), já vista na seção anterior. Não se esqueça de inserir o cabeçalho “inkey.ch” no início do arquivo, para que a constante manifesta seja usada.

A listagem a seguir é muito parecida com a listagem anterior, mas note que nas linhas 29 até 34 ocorre a gravação dos valores alterados pelo usuário de volta no banco de dados.

Listagem 30.6: Rotina básica de alteração de registros
Fonte: codigos/dbf13.prg

```
#include "inkey.ch"                                                    1
PROCEDURE Main                                                         2
LOCAL cBusca := SPACE(30)                                              3
LOCAL cNome, dNascimento, nAltura, nPeso                               4
LOCAL GetList := {}                                                    5
                                                                       6
   SET DATE BRITISH                                                    7
                                                                       8
   USE paciente                                                        9
   SET INDEX TO paciente01 // A chave pelo o NOME                      10
                                                                      11
   CLS                                                                12
   @ 05,05 SAY "Informe o nome do paciente : " GET cBusca;            13
                                                PICTURE "@S20"        14
   READ                                                               15
   IF LASTKEY() == K_ESC                                              16
      RETURN                                                          17
   ENDIF                                                              18
   SEEK cBusca                                                        19
   IF FOUND()                                                         20
      cNome := PACIENTE->NOME                                         21
      dNascimento := PACIENTE->NASCIMENTO                             22
      nPeso := PACIENTE->PESO                                         23
      nAltura := PACIENTE->ALTURA                                     24
      @ 07,05 SAY "Nome       : " GET cNome PICTURE "@S30"            25
      @ 09,05 SAY "Nascimento : " GET dNascimento                     26
      @ 11,05 SAY "Altura     : " GET nAltura                         27
      @ 13,05 SAY "Peso       : " GET nPeso                           28
      READ                                                            29
      IF LASTKEY() <> K_ESC                                           30
         REPLACE PACIENTE->NOME WITH cNome                            31
         REPLACE PACIENTE->NASCIMENTO WITH dNascimento                32
         REPLACE PACIENTE->ALTURA WITH nAltura                        33
         REPLACE PACIENTE->PESO WITH nPeso                            34
      ENDIF                                                           35
   ENDIF                                                              36
                                                                      37
RETURN                                                                38
```

30.1.7 Exclusão de registro

A rotina modelo para exclusão de registros é muito parecida com a rotina de alteração vista na subseção anterior.

Listagem 30.7: Rotina básica de exclusão de registros
Fonte: codigos/dbf14.prg

```
#include "inkey.ch"
PROCEDURE Main
LOCAL cBusca := SPACE(30)
LOCAL cResposta := SPACE(1)
LOCAL GetList := {}

   SET DATE BRITISH
   hb_cdpselect("UTF8")
   USE paciente
   SET INDEX TO paciente01 // A chave pelo o NOME

   CLS
   @ 05,05 SAY "Informe o nome do paciente : " GET cBusca;
                                               PICTURE "@S20"
   READ
   IF LASTKEY() == K_ESC
      RETURN
   ENDIF
   SEEK cBusca
   IF FOUND()
      @ 07,05 SAY "Nome       : " + PACIENTE->NOME PICTURE "@S30"
      @ 09,05 SAY "Nascimento : " + DTOC( PACIENTE->NASCIMENTO )
      @ 11,05 SAY "Altura     : " + STR( nAltura )
      @ 13,05 SAY "Peso       : " + STR( nPeso )

      @ 15,05 SAY "Confirma a exclusão desse registro ? (S/N)";
              GET cResposta
      IF LASTKEY() $ "Ss"
         DELETE // Após a confirmação o registro é excluído
      ENDIF
   ENDIF

RETURN
```

Note que eu não preciso do comando GET para alterar o conteúdo de registro algum (eles foram trocados pelo comando SAY e uma operação de concatenação). Outro detalhe importante é que o GET pode ser usado para obter uma resposta do usuário (tipo SIM ou NÃO) para a exclusão do registro.

30.1.8 Funções especiais que informam onde o ponteiro de registro está.

O Harbour também tem algumas funções especiais que retornam informações sobre um determinado arquivo. Iremos abordar agora as funções BOF(), EOF() e RECNO(). Ao final veremos um exemplo abordando SKIP e essas funções.

EOF : End of file

EOF() é uma função de tratamento de banco de dados utilizada para verificar se o processamento chegou no final do arquivo.

Descrição sintática 49

1. Nome : EOF()[a]
2. Classificação : função.
3. Descrição : Determina quando é encontrado o final do arquivo.
4. Sintaxe

```
EOf() -> lLimite
```

Fonte : [Nantucket 1990, p. 5-78]

[a]EOF significa End of file, ou Fim do arquivo.

O exemplo a seguir (listagem 30.8) lhe ajudará a entender como isso funciona.

Listagem 30.8: EOF
Fonte: codigos/dbf15.prg

```
#include "inkey.ch"
PROCEDURE Main

   SET DATE BRITISH

   USE paciente
   SET INDEX TO paciente01

   DO WHILE .NOT. EOF()
      ? PACIENTE->NOME, PACIENTE->NASCIMENTO, ;
        PACIENTE->ALTURA, PACIENTE->PESO
      SKIP
   ENDDO

RETURN
```

.:Resultado:.

BOF : Begin of file

BOF() é uma função de tratamento de banco de dados utilizada para verificar se o processamento chegou no início do arquivo.

Descrição sintática 50

1. Nome : BOF()[a]
2. Classificação : função.
3. Descrição : Determina quando é encontrado o início do arquivo.
4. Sintaxe

```
BOf() -> lLimite
```

Fonte : [Nantucket 1990, p. 5-34]

[a]BOF significa Begin of file, ou Inicio do arquivo.

O exemplo a seguir (listagem 30.9) lhe ajudará a entender como isso funciona.

Listagem 30.9: BOF
Fonte: codigos/dbf16.prg

```
#include "inkey.ch"
PROCEDURE Main

   SET DATE BRITISH

   USE paciente
   SET INDEX TO paciente01
   GO BOTTOM // VAI PARA O FINAL
   DO WHILE .NOT. BOF()
      ? PACIENTE->NOME, PACIENTE->NASCIMENTO,;
        PACIENTE->ALTURA, PACIENTE->PESO
      SKIP -1 // RETROCEDE UM
   ENDDO

RETURN
```

30.1.9 Fechando um arquivo DBF

Para fechar um arquivo previamente aberto você pode usar os seguintes comandos :

1. CLOSE <idAlias> : onde <idAlias> é o nome da área de trabalho.
2. USE : sem argumentos.

O comando USE sem argumentos serve para fechar o arquivo da área de trabalho corrente. Já o comando CLOSE pode fechar o arquivo de uma outra área de trabalho, mas o comando não altera a área de trabalho atual.

Alguns exemplos :

```
USE PACIENTE
USE EXAMES NEW
CLOSE PACIENTE // Fecha o arquivo PACIENTE, mas continua na área 2.
```

Esse fragmento a seguir funciona como se fosse o comando USE sem parâmetros.

```
USE PACIENTE
USE EXAMES NEW
CLOSE  // Fecha o arquivo EXAMES, mas continua na área 2.
```

O comando CLOSE ALL, a seguir, fecha todos os arquivos DBFs e também outros formatos, como os arquivos ALTERNATE e os formatos ativos.

```
USE PACIENTE
USE EXAMES NEW
CLOSE  ALL// Fecha TODOS os arquivos abertos
          // e vai para a área de trabalho 1.
```

O comando CLOSE DATABASES, a seguir, fecha todos os arquivos DBFs abertos.

```
USE PACIENTE
USE EXAMES NEW
CLOSE  DATABASES// Fecha TODOS os arquivos DBFs abertos e vai
                  //para a área de trabalho 1.
```

O comando CLOSE INDEXES fecha todos os arquivos de índices **da área de trabalho corrente**.

Dica 164

O comando CLEAR ALL é uma das formas de se fechar todos os arquivos, mas ele também libera os GETs ainda não lidos de um formulário. Portanto não use esse comando para fechar os seus arquivos.

30.1.10 Mais sobre os índices

Cuidado quando estiver processando índices dentro de laços

Você precisa tomar muito cuidado quando for realizar múltiplas alterações em um arquivo. Por exemplo, vamos supor que você possui um arquivo com os preços de custo de uma empresa. Você foi chamado para criar uma rotina que reajusta os preços de custo de todos esses produtos. A rotina é simples:

1. O usuário informa de quanto será o percentual de aumento e
2. o programa percorre o arquivo reajustando os registros de um por um.

A rotina criada abre o arquivo mais os índices conforme a listagem a seguir. Note que, entre as linhas 8 e 18, nós criamos uma base de testes com 10 registros. Contudo a listagem possui um erro de lógica. Vamos discutir sobre esse erro após a listagem.

Listagem 30.10: Erro de lógica envolvendo índices
Fonte: codigos/indices.prg

```
/*
Exemplo de uma rotina com erros
*/
PROCEDURE Main
LOCAL x , nAumento


   // CRIANDO OS DADOS PARA O NOSSO TESTE

   DBCREATE( "produtos" , {{"CODIGO","N",3,0},{"CUSTO","N",10 , 2 }} )
   USE produtos
   INDEX ON STR( FIELD->CUSTO ) TO produtos
   FOR x := 1 TO 10
      APPEND BLANK
      REPLACE FIELD->CODIGO WITH x
      REPLACE FIELD->CUSTO WITH 100
   NEXT
   USE
   // AQUI INICIO O TESTE
   ? "Essa rotina reajusta o preço de custo dos seus produtos"
   USE produtos
   SET INDEX TO produtos

   INPUT "Informe o percentual de aumento (p.ex: 10 se for 10%) : " TO nAument
   DO WHILE .NOT. EOF()
      REPLACE FIELD->CUSTO WITH FIELD->CUSTO * ( 1 + (nAumento/100) )
      SKIP
   ENDDO

RETURN
```

Vamos supor que você digitou um percentual de reajuste de 10% para todos os produtos.

.:Resultado:.

```
Essa rotina reajusta o preço de custo dos seus produtos
Informe o percentual de aumento (p.ex: 10 se for 10%) : 10
```

O programa executa o que lhe foi ordenado. Você pensa consigo mesmo: “se todos os produtos da minha base de testes possuem o preço de custo igual a 100, então um reajuste de 10% irá alterar os preços todos para 110.” Para verificar se deu tudo certo, você executa o HBRUN e lista os registros:

```
USE produtos
LIST codigo, custo
```

Para a sua surpresa apenas um produto, o primeiro, sofreu o reajuste:

.:Resultado:.

```
  1     110.00
  2     100.00
  3     100.00
  4     100.00
  5     100.00
  6     100.00
  7     100.00
  8     100.00
  9     100.00
 10     100.00
```

Por que isso aconteceu ?

A resposta para esse problema está nos índices. O índice ativo possui o campo CUSTO na sua chave (linha 11), e quando você reajusta em 10% o primeiro preço de custo ele passa a ser o último (por causa do índice) e o laço termina. Ou seja, eu não cheguei a percorrer o arquivo todo. A solução para esse problema é desativar temporariamente o índice ou mudar para outro índice que não receba alterações. Como eu só tenho um índice eu devo desativar temporariamente esse índice, mas sem fechá-lo. Isso é feito com o comando SET ORDER TO 0 (que nós já vimos anteriormente).

O exemplo da listagem 30.11 mostra como se faz isso. Note que nós apenas incluímos o comando SET ORDER TO 0 na linha 22.

Listagem 30.11: Solução do erro de lógica envolvendo índices
Fonte: codigos/indice2.prg

```
/*                                                                                1
Exemplo de uma rotina com erros                                                   2
*/                                                                                3
PROCEDURE Main                                                                    4
LOCAL x , nAumento                                                                5
                                                                                  6
                                                                                  7
   // CRIANDO OS DADOS PARA O NOSSO TESTE                                         8
                                                                                  9
   DBCREATE( "produtos" , {{"CODIGO","N",3,0},{"CUSTO","N",10 , 2 }} )
   USE produtos                                                                   10
   INDEX ON STR( FIELD->CUSTO ) TO produtos                                       11
   FOR x := 1 TO 10                                                               12
      APPEND BLANK                                                                13
      REPLACE FIELD->CODIGO WITH x                                                14
      REPLACE FIELD->CUSTO WITH 100                                               15
   NEXT                                                                           16
   USE                                                                            17
   // AQUI INICIO O TESTE                                                         18
   ? "Essa rotina reajusta o preço de custo dos seus produtos"                    19
   USE produtos                                                                   20
   SET INDEX TO produtos                                                          21
   SET ORDER TO 0                                                                 22
                                                                                  23
   INPUT "Informe o percentual de aumento (p.ex: 10 se for 10%) : " TO nAument
```

```
   DO WHILE .NOT. EOF()
      REPLACE FIELD->CUSTO WITH FIELD->CUSTO * ( 1 + (nAumento/100) )
      SKIP
   ENDDO

RETURN
```

Dica 165

Alterar um campo que faz parte de um índice ativo é um erro comum entre os iniciantes (e entre os veteranos também), e as suas consequências são imprevisíveis. Nos casos abordados o ponteiro foi para o último registro, mas existem casos onde o índice alterado é para um valor menor e o ponteiro vai para o primeiro registro, fazendo com que o seu programa fique preso dentro de um loop infinito. **Quando for alterar campos dentro de um laço, certifique-se de que o arquivo não usa esse campo como uma chave de índice ativa.** Se esse for o caso então use SET ORDER TO 0.

30.1.11 Campo auto-numerado

No ato da criação de um arquivo DBF, você pode criar um campo auto-numerado. Esse campo é automaticamente preenchido sempre que um registro é incluído. Vejamos alguns exemplos. A listagem 30.12 nos mostra um exemplo bem simples.

Listagem 30.12: Criando um campo auto-numerado
Fonte: codigos/autonum01.prg

```
PROCEDURE MAIN

    LOCAL aStruct := {}
    LOCAL x, nTot := 10

    AADD( aStruct , { "CODIGO" ,  "I:+" , 10 , 0 })
    AADD( aStruct , { "NOME" , "C" , 30 , 0 })

    dbCreate( "contatos" , aStruct )
    USE contatos
    FOR x := 1 TO nTot
        APPEND BLANK
        // O código foi gravado automaticamente
        // só é necessário gravar o NOME
        REPLACE NOME WITH "CONTATO " + STRZERO( x , 3 )
    NEXT

    GO TOP
    DO WHILE .NOT. EOF()
        ? FIELD->CODIGO , FIELD->NOME
        SKIP
    ENDDO

RETURN
```

.:Resultado:.

```
 1 CONTATO 001
 2 CONTATO 002
 3 CONTATO 003
 4 CONTATO 004
 5 CONTATO 005
 6 CONTATO 006
 7 CONTATO 007
 8 CONTATO 008
 9 CONTATO 009
10 CONTATO 010
```

Note que a rotina já grava o valor do código automaticamente. Outro fato digno de nota é que um campo auto-numerado é do tipo numérico mas, para que ele seja auto-numerado, o seu tipo deve ser definido por "I:+"na função DbCreate().

30.1.12 Gravando dados em um campo auto-numerado

Você tem liberdade para gravar seus próprios valores em um campo auto-numerado. Isso é simples e pode ser feito através de um comando REPLACE normal. O exemplo 30.13 mostra como isso é feito. Esse exemplo é uma alteração do exemplo anterior, por isso só mostraremos as linhas incluídas. Qualquer dúvida, consulte o arquivo na pasta codigo.

Listagem 30.13: Trecho do código onde nós gravamos nosso próprio valor em um campo auto-numerado.
Fonte: codigos/autonum02.prg

```
    APPEND BLANK                                                    1
    REPLACE CODIGO WITH 1000  // Incluído "manualmente"             2
    REPLACE NOME WITH "CONTATO 1000"                                3
                                                                    4
    APPEND BLANK                                                    5
    REPLACE NOME WITH "CONTATO NOVO"                                6
```

O resultado é

.:Resultado:.

```
    1 CONTATO 001
    2 CONTATO 002
    3 CONTATO 003
    4 CONTATO 004
    5 CONTATO 005
    6 CONTATO 006
    7 CONTATO 007
    8 CONTATO 008
    9 CONTATO 009
   10 CONTATO 010
 1000 CONTATO 1000
   12 CONTATO 12
```

Note que o próximo valor auto-numerado não é o valor 11, mas sim o 12. Isso porque o valor 11 foi gerado mas não foi gravado, já que nós gravamos o valor 1000 no seu lugar.

30.2 Conclusão

Assumir o controle total sobre a forma como os dados são recuperados e tratados é uma tarefa de grande responsabilidade, mas não é nenhum bicho-de-sete-cabeças. Caso você deseje se aprofundar nesse assunto, o próximo capítulo irá abordar algumas técnicas para o uso profissional de arquivos DBFs, caso você não queira se aprofundar, uma sugestão interessante é estudar o acesso aos banco de dados SQL.

31 Arquivos DBFs : Programação multi-usuário

> Pedi, e dar-se-vos-á; buscai e encontrareis; batei, e abrir-se-vos-á. Porque aquele que pede recebe; e o que busca encontra; e, ao que bate, se abre.
>
> Mateus 6:7-8

Objetivos do capítulo

- Programação multi-usuário
- Entendendo o processo de leitura e gravação em um ambiente multi-usuário
- Conhecer as funções de travamento/travamento de registro e arquivo
- Criar suas funções de abertura e gravação de arquivos.

31.1 Introdução

Já estudamos o arquivo DBF através do modo interativo e realizamos um pequeno programa que manipula arquivos DBFs. Agora iremos trabalhar algumas técnicas que permitem um desenvolvimento profissional de aplicações que fazem uso dos arquivos DBFs.

31.2 Programação multiusuário

31.2.1 O que é uma rede de computadores ?

Uma rede de computadores é um dispositivo eletrônico que serve para conectar dois pontos. Essa definição, dada por Gabriel Torres em `https://www.udemy.com/course/redes-modulo-1/` parece simples demais, mas ela é verdadeira. O assunto, contudo, pode se tornar árido e muito complexo, a depender da profundidade do estudo realizado. Felizmente o Harbour trás recursos que permitem trabalhar de forma eficiente os arquivos DBFs em um ambiente multiusuário. Basicamente o problema que um programador enfrenta em um ambiente multi-usuário está relacionado com a abertura dos arquivos DBFs da aplicação e com a gravação dos dados.

Figura 31.1: Rede de computadores: interligando e compartilhando recursos

31.2.2 A principal preocupação

Em um sistema operacional multiusuário, qualquer alteração feita em um arquivo (um documento do Word, uma planilha do Excel, um DBF, etc.) requer exclusividade. Ou seja, na hora da gravação o arquivo deve estar reservado somente para o programa que faz essa gravação. Por exemplo, já pensou, que bagunça seria, se o mesmo documento físico fosse editado por duas pessoas ao mesmo tempo ? Essa é uma das principais tarefas de um sistema operacional multi-usuário: impedir que dois programas editem o mesmo arquivo (ou parte dele) ao mesmo tempo. A figura 31.2 ilustra essa situação através de uma pequena metáfora: a moça representa o sistema operacional, e os dois homens representam dois programas que querem usar o mesmo arquivo.

Figura 31.2: O sistema operacional não permite que dois programas editem o mesmo recurso simultaneamente

Esclarecendo um pouco mais a nossa analogia: o arquivo (ou parte dele), pode ser visto por duas ou mais pessoas, mas só pode ser editado por uma pessoa.

Dica 166

Por que foi usado a expressão "o arquivo (ou parte dele)"?
Existem programas, como uma planilha eletrônica, que requerem que o documento inteiro seja exclusivo do usuário que vai editar. Nesse caso, se você quiser gravar na planilha, você deve reservar a planilha toda. Mas existem outros programas, como um gerenciador de banco de dados, que não requer que a tabela toda esteja reservada. Nesse caso, o programa pode reservar apenas o registro que vai ser editado. No caso do Harbour, veremos que podemos tanto reservar o arquivo DBF inteiro (a tabela) ou apenas o registro a ser editado.

Vamos tentar exemplificar, na próxima seção, usando dois softwares de planilha eletrônica: o Calc e o GNumeric.

31.2.3 Simulando um erro

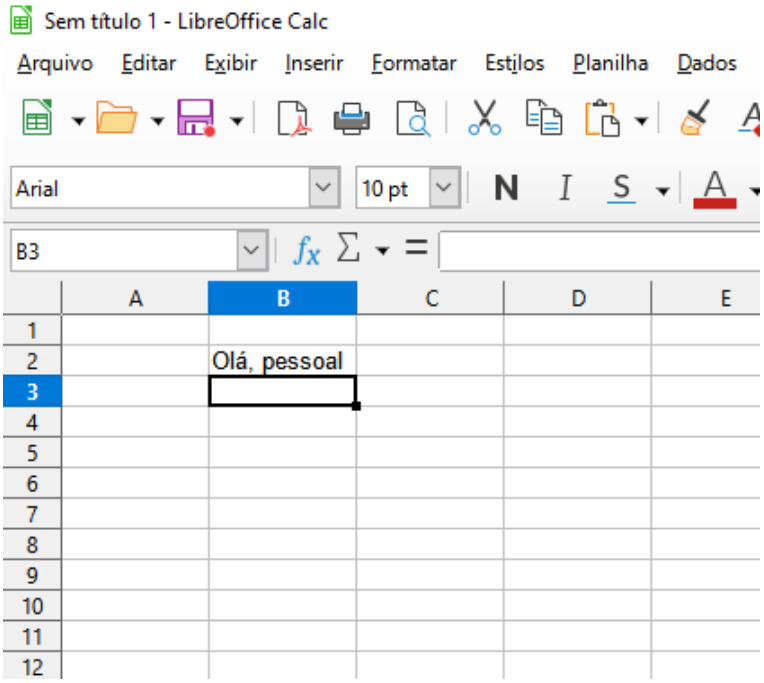
Iremos realizar uma simulação de erro. Primeiro iremos abrir uma planilha usando o LibreOffice Calc, em seguida abriremos a mesma planilha usando o Gnumeric (outro software para edição de planilha). Mantendo o mesmo arquivo aberto simultaneamente pelos dois softwares, tentaremos salvar uma alteração. Note que nós não podemos efetivar qualquer alteração com um arquivo aberto simultaneamente por dois softwares. É importante que você acompanhe o nosso teste, e leia com atenção as observações posteriores.

Parte I: Criando um arquivo no Calc

Primeiramente iremos usar o software LibreOffice Calc para criar uma planilha simples, conforme a figura 31.3.

Nessa planilha iremos editar uma célula qualquer, escrevendo alguma coisa nela.

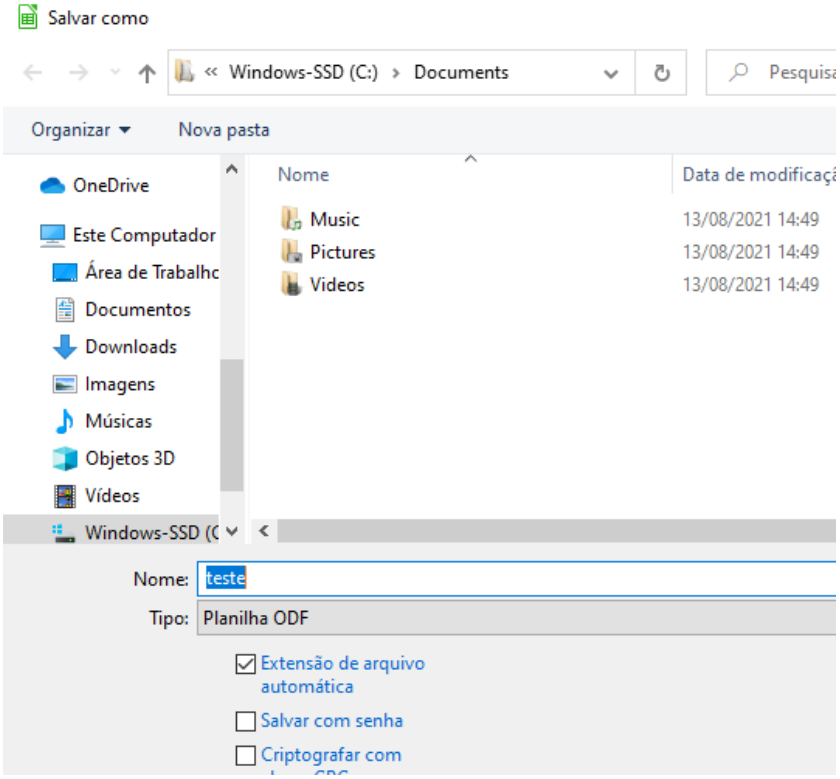
Figura 31.3: Abrindo a planilha através do LibreOffice Calc



Parte II

Agora iremos salvar esse arquivo com o nome de "teste.odf".
O arquivo foi salvo e a planilha continua aberta com o arquivo "teste.odf".

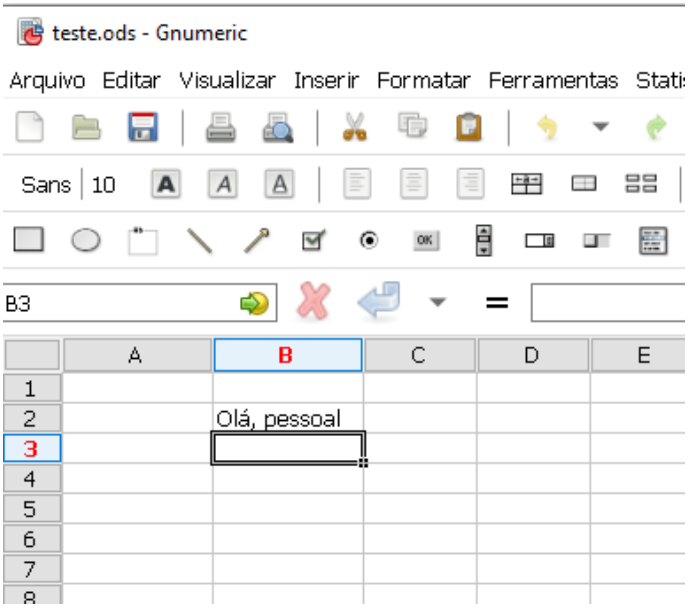
Figura 31.4: Abertura do mesmo arquivo simultaneamente por dois programas



Parte III

Vamos agora abrir esse mesmo arquivo usando uma outra planilha, a Gnumeric. Note que o mesmo arquivo foi aberto duas vezes. Um por cada planilha.

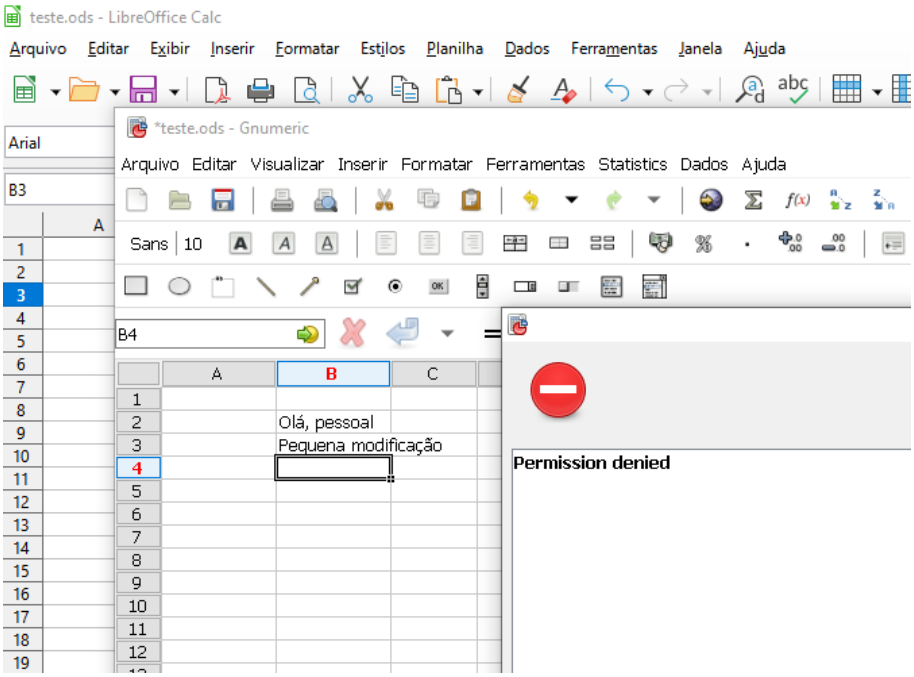
Figura 31.5: Abertura do mesmo arquivo simultaneamente por dois programas



Parte IV

Agora tentaremos realizar uma alteração no Gnumeric e salvaremos a mesma. Note que um erro ocorreu.

Figura 31.6: Abertura do mesmo arquivo simultaneamente por dois programas



Conclusão

Realizar operações em um mesmo arquivo ao mesmo tempo é típico de aplicações desenvolvidas para o ambiente multi-usuário. Não importa se o arquivo é uma planilha, um documento texto ou um arquivo DBF. A aplicação deve estar pronta para poder responder a determinadas questões, que são, basicamente duas:

1. Duas ou mais aplicações podem abrir o mesmo arquivo simultaneamente ?
2. Se duas ou mais aplicações podem abrir o mesmo arquivo simultaneamente, elas vão poder realizar alterações ?

No caso do nosso exemplo, um erro ocorreu, mas esse erro foi previsto pelo programador e uma mensagem de erro foi exibida: "Permission denied"(Permissão negada).

Que permissão foi negada ?

A permissão para gravar o arquivo em disco.

Quem negou a permissão ?

O sistema operacional

Qual a função do programador ?

Perguntar ao sistema operacional se tem permissão para gravar o arquivo. Se tiver permissão, então grava o arquivo, se não tiver permissão, uma mensagem de erro deve ser exibida para o usuário e o processo não deve ser levado adiante. Se o programador não fizer essa verificação prévia, o seu software tentará gravar o arquivo, o sistema operacional não irá deixar, e um erro de execução irá ocorrer. Nesse último caso, o programa é abortado e os dados serão perdidos.

No caso do nosso exemplo, o erro foi previsto e o processo foi interrompido. Contudo, os dados não foram perdidos. O usuário pode salvar a planilha com outro nome, ou se tiver em um ambiente com várias máquinas em uma sala, pode perguntar quem está usando a planilha dele e solicitar ao usuário que feche o arquivo para que ele possa trabalhar em paz. Em todo caso, os dados não serão perdidos.

Essa, basicamente, é a função do programador Harbour quando ele está trabalhando com arquivos DBFs. Quando ele for realizar qualquer operação no arquivo, ele deve perguntar ao sistema operacional (Windows, Linux, MacOS, etc.) se tal operação pode ser realizada. Se o sistema operacional responder que pode, então ele segue com a operação, caso contrário, ele deve realizar alguma ação para que o usuário não perca os dados.

Figura 31.7: A torre de controle é o S.O. e o avião é o seu software.

É mais ou menos como se o software desenvolvido fosse um avião que vai pousar, e a torre de controle fosse o sistema operacional. O avião precisa pedir permissão para pousar. Se ele não pedir permissão e simplesmente tentar pousar, pode ser que dê certo ou pode ser que não. A torre de controle não tem como impedir a loucura do piloto. E mais, na nossa analogia, a torre de controle nem comunica nada ao piloto sem ser perguntada. Ou seja: o piloto precisa perguntar se pode pousar, para garantir que o pouso seja seguro. Prosseguindo com o nosso exemplo, se a torre de controle negar, o piloto vai ficar sobrevoando a pista e depois de N segundos/minutos irá perguntar de novo: "eu posso pousar ?", e assim sucessivamente (notou que isso é uma estrutura de repetição ?). Finalmente, se essa situação demorar muito, o piloto (o usuário) pode decidir esperar ou não pousar mais naquele aeroporto.

Moral da história: antes de fazer operações em arquivos, pergunte ao sistema operacional se pode.

31.3 Cuidados que um programador deve ter

Veremos os cuidados mas não faremos exemplos completos. Por isso não se preocupe se não entender completamente os tópicos a seguir. Na próxima seção, faremos vários testes contemplando cada caso. Basicamente, o programador deve realizar três operações :

1. abrir o arquivo no modo compartilhado;
2. travar o registro a ser alterado;
3. liberar o bloqueio após finalizar.

31.3.1 Abrir o arquivo no modo compartilhado

Se você quer que o seu programa trabalhe em ambiente multi-usuário, você deve abrir os seus arquivos em modo compartilhado. Existem duas formas de se abrir um arquivo DBF: exclusivo ou compartilhado. Um arquivo aberto no modo exclusivo não permite que duas ou mais pessoas trabalhem nele simultaneamente. Já um arquivo aberto no modo compartilhado, irá permitir que N pessoas abram esse arquivo. Para garantir que o arquivo seja aberto em modo compartilhado, o programador precisa seguir apenas dois passos:

Abra o arquivo com a cláusula SHARED

A cláusula SHARED informa que o arquivo deve ser aberto em modo compartilhado.

```
USE clientes SHARED
```

Isso quer dizer que duas ou mais pessoas poderão usar esse arquivo ao mesmo tempo. Lembra do nosso exemplo com as planilhas ? Pois bem, nele os dois programas abriram o mesmo arquivo em modo compartilhado. O LibreOffice foi escrito em C++, Java e Python, e o Gnumeric foi escrito em C e C++, e mesmo assim, eles conseguem compartilhar o mesmo documento, pois ambos, cada qual a seu modo, conseguem se comunicar com o sistema operacional e realizar as verificações necessárias.

Verificar se o arquivo foi aberto

Uma boa prática é verificar se o sistema operacional deixou que o arquivo fosse aberto. Nós fazemos isso através da função NETERR(). O nosso código ficaria assim :

```
USE clientes SHARED
IF NETERR()
    // O sistema operacional informou que houve uma falha qualquer
    // e o arquivo não foi aberto
    // Devo realizar alguma operação de correção aqui.
ENDIF
// Tudo bem, posso prosseguir com o processamento.
```

A linguagem Harbour possui um SET só para dizer se os seus arquivos devem ser abertos em modo exclusivo ou compartilhado.

```
SET EXCLUSIVE ON
USE arquivo // Abre em modo exclusivo
SET EXCLUSIVE OFF
USE notas // Abre em modo compartilhado
```

O programador não é obrigado a definir esse SET. Na verdade, é até aconselhável que ele não mecha nessa configuração. O ideal é sempre informar no ato da abertura, se o arquivo será aberto em modo exclusivo ou compartilhado. Assim :

```
USE cliente EXCLUSIVE
```

ou

```
USE cliente SHARED
```

Assim, evitamos o trabalho adicional de ter que saber qual o estado de SET EXCLUSIVE.

Dica 167

Evite o comando USE sem as cláusulas EXCLUSIVE ou SHARED. **Não faça assim**:

```
USE cliente
```

O comando USE isolado das cláusulas depende do estado de SET EXCLUSIVE, e isso pode deixar seus programas difíceis de serem lidos. Abra seus arquivos sempre com as cláusulas SHARED ou EXCLUSIVE explicitamente declaradas.

31.3.2 Bloquear/Travar o registro antes de gravar

Na hora de gravar qualquer informação no seu arquivo DBF, você deve obter um travamento do registro. Essa é outra regra da programação em ambiente multi-usuário. Bloquear/Travar significa obter o recurso só para si, sem compartilhar o mesmo com ninguém. Para que o travamento seja realizado com sucesso, duas condições são necessárias :

1. O arquivo deve estar aberto em modo compartilhado (já fizemos isso)
2. Ninguém deve estar usando o recurso na hora em que você fizer o travamento. Ou seja, o registro deve estar desbloqueado.

O Harbour possui duas formas de travamento : a nível de registro e a nível de arquivo. O travamento mais usado é a nível de registro, porque se você quer alterar um dado em um registro, então você **não** precisa travar o arquivo inteiro, basta travar o registro.

O travamento (lock) de registro é feito pela função RLOCK() - "R"de Record (Registro).

Existem casos, mais raros, onde o travamento do arquivo inteiro é necessário. Nesse caso você deve usar a função FLOCK() - "F"de File (Arquivo).

No caso de travamento de registro, o nosso código ficaria assim :

```
IF RLOCK()
    REPLACE NOME WITH cNome
ENDIF
```

Só para complementar. Voltando ao exemplo do começo do capítulo, aquele das planilhas, onde o Gnumeric não conseguiu gravar as alterações. Isso aconteceu porque ele não conseguiu bloquear o recurso para gravação, daí aquela mensagem "Permission denied". Ou seja, o arquivo foi aberto compartilhado, mas o mesmo não pode ser travado para gravação.

31.3.3 Liberar o registro após a gravação

Depois que você terminou de gravar os dados, você deve emitir uma ordem de desbloqueio, para que o registro fique disponível para outros usuários gravarem caso isso seja necessário. Existem duas formas de desbloqueio: o desbloqueio explícito e o desbloqueio implícito.

Desbloqueio explícito

Um desbloqueio é feito através do comando UNLOCK.

Desbloqueio implícito

Existem com casos onde nenhum comando UNLOCK foi emitido, mas o seu registro foi desbloqueado. Isso ocorre quando o ponteiro do arquivo é movido para outro registro.

31.3.4 Conclusão

Nos próximos tópicos, nós veremos formas melhores de se realizar a abertura, o travamento e o desbloqueio. Essas formas mostradas foram uma simplificação para facilitar o entendimento. Por exemplo: se a função RLOCK() não conseguir travar o registro, ela vai retornar falso e o programa segue adiante sem gravar os dados. Isso evita um erro de execução, mas a gravação não foi efetuada. O ideal é ficarmos tentando até conseguir o travamento (laço de repetição). Se o travamento não for conseguido em N segundos, um aviso deve ser emitido para o usuário e ele decide se vai esperar mais ou vai abandonar a operação. É mais ou menos como o nosso exemplo do avião e da torre de controle. Faremos isso adiante.

31.4 Entendendo o processo passo-à-passo

31.4.1 Teste 1 : tentando abrir um arquivo que já foi aberto em modo exclusivo

Criando o arquivo para exemplo

Usaremos o seguinte arquivo da listagem 31.1. É bem simples, ele apenas abre o arquivo conforme estávamos fazendo nos capítulos anteriores.

Listagem 31.1: Versão inicial do nosso teste de travamento.
Fonte: codigos/travamento01.prg

```
/*
Simulando travamento
*/
PROCEDURE Main
LOCAL aStruct := { { "CODIGO" , "C" , 10 , 0 } }

    CLS
    IF .NOT. FILE( "travamento.dbf" )
        DBCREATE( "travamento" , aStruct )
    ENDIF

    USE travamento
    ? "Tecle algo para fechar e encerrar o teste"
    INKEY(0)

RETURN
```

Algumas notas importantes :

1. O arquivo não é fechado, ele somente é fechado se o usuário teclar algo.
2. O SET EXCLUSIVE não foi definido, portanto o seu estado é ON.
3. O arquivo foi aberto sem definirmos o modo de compartilhamento (EXCLUSIVE ou SHARED)
4. Portanto, o arquivo foi aberto em modo exclusivo.

Gere o executável :

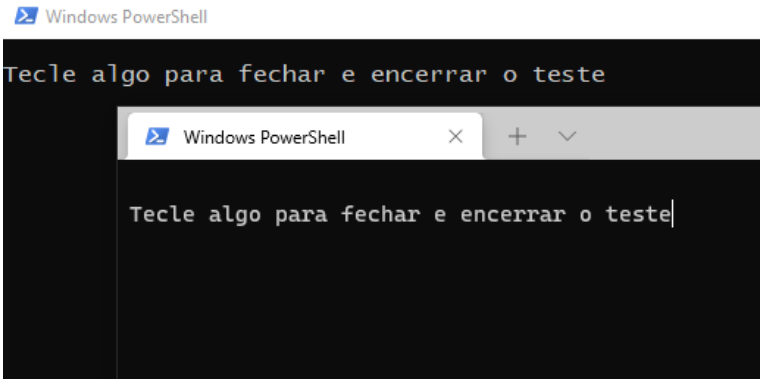
.:Resultado:.

```
hbmk2 travamento01
```

Abrindo dois prompts de comando

Vamos abrir dois prompts de comando e, em cada um deles, abrir o programa que foi gerado. Faça conforme a figura 31.8

Figura 31.8: Abrindo dois prompts para teste.



Aparentemente os dois arquivos foram abertos sem maiores problemas, mas isso não é verdade. O que aconteceu foi o seguinte:

1. O programa foi executado corretamente da primeira vez (lembre-se que ele não foi finalizado)
2. A segunda instância do programa (o segundo prompt) foi executado, tentou abrir o arquivo mas não conseguiu. Uma mensagem de erro foi gerada, mas o programador não verificou a existência do erro. Consequentemente nenhum erro foi reportado.

Um comando USE mal sucedido não reporta mensagem de erro. Tal característica é comum também em outras linguagens. Precisamos ter uma forma de verificar se a operação foi realizada com sucesso.

Colocando o teste de verificação

Vamos agora inserir um teste de verificação.

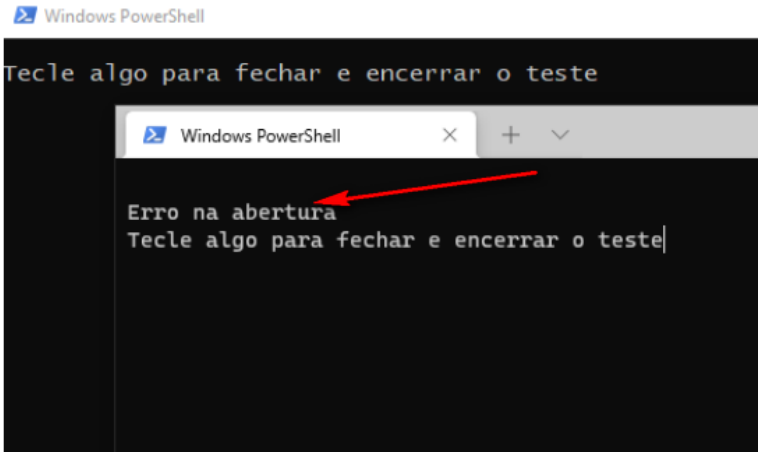
Listagem 31.2: Alterando o programa para que ele verifique se o arquivo foi aberto. Fonte: codigos/travamento02.prg

```
/*
Simulando travamento
*/
PROCEDURE Main
LOCAL aStruct := { { "CODIGO" , "C" , 10 , 0 } }

    CLS
```

```
    IF .NOT. FILE( "travamento.dbf" )
        DBCREATE( "travamento" , aStruct )
    ENDIF

    USE travamento
    IF NETERR()
        ? "Erro na abertura"
    ENDIF
    ? "Tecle algo para fechar e encerrar o teste"
    INKEY(0)

RETURN
```

Voltando a executar, conforme a figura 31.9

Figura 31.9: A segunda instância já reportou um erro.



Conclusão do teste 1

Ao abrir um arquivo, sempre verifique se ele foi aberto corretamente. A função NETERR() realiza essa verificação em ambientes multi-usuários.

Descrição sintática 51

1. Nome : NETERR()
2. Classificação : função.
3. Descrição : NETERR() é uma função de tratamento de ambientes de rede. O seu valor de retorno é configurado por USE e APPEND BLANK em ambientes de rede. É utilizado para testar se algum desses comandos falhou.
4. Sintaxe

```
NETERR() -> lErro

lErro : .t. caso haja um erro;
caso contrário, retorna falso (.f.).
```

Fonte : [Nantucket 1990, p. 5-167]

Operações em modo compartilhado são mais lentas

Spence nos adverte que usar banco de dados no modo compartilhado é uma operação muito mais lenta do que usá-los no modo exclusivo. [...] No modo compartilhado [o Harbour] pode armazenar temporariamente registros e índices localmente. No modo compartilhado, ele precisará verificar continuamente os arquivos do servidor, para assegurar que ninguém os tenha alterado [Spence 1994, p. 775].

Dica 168

Se sua aplicação só precisa ler o banco de dados, você pode abrir em modo compartilhado, mas em modo somente leitura (READONLY), conforme abaixo:

```
USE CLIENTE SHARED READONLY
```

Dessa forma o Habour irá abrir em ambiente compartilhado, mas o desempenho será superior, porque as verificações no servidor serão desativadas.

31.4.2 Teste 2 : tentando gravar dados em um registro já travado por outro usuário

Criando o arquivo para exemplo

Usaremos o seguinte arquivo da listagem 31.3. Vamos aproveitar a listagem anterior e acrescentar um registro.

Listagem 31.3: Versão inicial do nosso teste de travamento.
Fonte: codigos/travamento03.prg

```
/*                                                                    1
```

```
Simulando travamento                                                    2
*/                                                                      3
PROCEDURE Main                                                          4
LOCAL aStruct := { { "CODIGO" , "C" , 20 , 0 } ,;                       5
                   { "NOME" , "C" , 20 , 0 }  }                         6
                                                                        7
    CLS                                                                 8
    // Cria o arquivo para teste, caso não exista                       9
    IF .NOT. FILE( "trava.dbf" )                                        10
        DBCREATE( "trava" , aStruct )                                   11
        USE trava EXCLUSIVE                                             12
        APPEND BLANK                                                    13
        REPLACE CODIGO WITH "001"                                       14
        REPLACE NOME WITH "APPEND BLANK"                                15
        ? "Inserindo o valor de teste"                                  16
        ? "SAINDO DO SISTEMA"                                           17
        ? "Abra dois Terminais de Comando (Prompt/Shell)"               18
        ? "e em cada um deles execute esse programa"                    19
        QUIT                                                            20
    ENDIF                                                               21
    // Aqui começa o teste realmente                                    22
    USE trava SHARED                                                    23
    IF NETERR()                                                         24
        ? "Erro na abertura"                                            25
        QUIT                                                            26
    ENDIF                                                               27
    IF RLOCK()                                                          28
        ? "Registro travado"                                            29
    ELSE                                                                30
        ? "Registro foi travado por outro usuario"                      31
    ENDIF                                                               32
    ? "Tecle algo para fechar e encerrar o teste"                       33
    ? hb_Valtoexp( dbrlocklist() )                                      34
    INKEY(0)                                                            35
                                                                        36
RETURN                                                                  37
```

Observações importantes: esse nosso teste exige três passos descritos abaixo:

1. Execute o programa pela primeira vez para poder criar o arquivo. O nosso programa irá criar o arquivo para testes e irá sair.

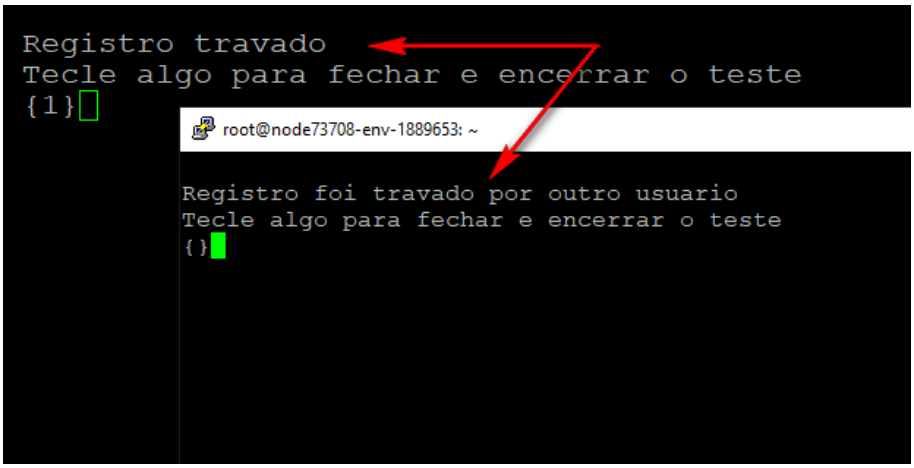
2. Execute novamente e não feche. Aqui o arquivo trava.dbf teve o primeiro registro travado.

3. Agora abra outro prompt de comando e execute novamente o arquivo.

Na figura 31.10 vemos as duas janelas abertas, referente aos passos 2 e 3.

Realizando o teste

Figura 31.10: Abrindo dois prompts para teste.



Dica 169

Os sistemas operacionais multiusuários podem reservar algumas surpresas desagradáveis. É extremamente aconselhável a você testar o ambiente antes de implementar a sua aplicação. Use o teste da listagem 31.3 para verificar se o travamento está funcionando. Geralmente uma aplicação usando o Subsistema Windows para Linux apresenta problemas no bloqueio de registros [a], de modo que você deve realizar testes no ambiente através de programas simples (esse é um deles).
É preciso que o desenvolvedor interaja com o pessoal administrador de sistemas, pois muitos problemas decorrem de configurações do ambiente multiusuário. As vezes um servidor Linux, usando o Samba, apresenta problemas na gravação de arquivos. Muitas vezes tais problemas não são do Harbour e nem do Linux, mas de algum problema no compartilhamento. Por isso é importante ter programas pequenos que realizem testes de travamento e gravação de dados em bases de teste. Muita calma nessa hora, as vezes o administrador de sistemas não quer reconhecer o problema ou não quer parar a sua rotina para verificar com calma o que aconteceu, por isso é importante você ter os seus testes preparados. Quanto mais informação você puder fornecer para o administrador de sistemas melhor para você.

[a] Testes realizados em 10/Ago/2021

Conclusão do teste 2

Antes de gravar dados no arquivo DBF, você precisa usar a função RLOCK() para travar o registro antes de qualquer operação de gravação.

Descrição sintática 52

1. Nome : RLOCK()
2. Classificação : função.
3. Descrição : RLOCK() é uma função de tratamento de rede utilizada para travar o registro corrente, evitando que outros usuários atualizem o registro até que o travamento seja liberado.
4. Sintaxe

```
RLOCK() -> lSucesso

lSucesso  : .t. se for obtido o travamento;
caso contrário, retorna falso (.f.).
```

Fonte : [Nantucket 1990, p. 5-197]

O travamento com RLOCK() é utilizado para operações no registro corrente da área de trabalho selecionada. Isso inclui os comandos:

- @ ... GET (quando usado diretamente sobre o registro. Já vimos que devemos evitar essa prática.)
- DELETE (um só registro)
- RECALL (um só registro)
- REPLACE (um só registro)

Caso vá usar RLOCK() em uma outra área de trabalho, você deve "ir para essa área"ou então referenciar essa área com um alias, conforme o exemplo abaixo:

```
IF CLIENTE->( RLOCK() )
    REPLACE NOME WITH "CLAUDIO"
ENDIF
```

31.4.3 Teste 3 : liberando um registro travado

Criando o arquivo para exemplo

Sempre que você travar um registro, você deve liberar o bloqueio após as operações serem realizadas.

Usaremos o seguinte arquivo da listagem anterior e faremos uma pequena modificação (confira a alteração feita na listagem 31.4). Não iremos listar o arquivo completo porque ele é praticamente igual ao anterior, vamos apenas chamar a atenção para o comando UNLOCK que destrava o registro.

Listagem 31.4: Versão inicial do nosso teste de travamento.
Fonte: codigos/travamento04.prg

```
IF RLOCK()
    ? "Registro travado"
    ? "Aqui eu posso usar replaces"
    REPLACE NOME WITH "NOVO VALOR"
    UNLOCK // Destravou o registro para outros usarem
ELSE
    ? "Registro foi travado por outro usuario"
ENDIF
```

Toda operação de alteração em registros, em ambiente multiusuário, obedece a essa sequência:

1. Abertura do arquivo compartilhado (USE arquivo SHARED)
2. Bloqueio (Travamento) do registro a ser alterado (RLOCK()).
3. Alteração do registro (REPLACE).
4. Desbloqueio do registro (UNLOCK).

Na próxima seção iremos aprimorar essa sequência acima com técnicas que tornarão a nossa programação mais segura. Contudo, os passos para se trabalhar com arquivos em ambiente multiusuário são os listados acima, nada mais será acrescentado.

31.5 Criando uma função para bloquear o registro

Lembra da nossa metáfora do avião e da torre de controle ? Quando o avião recebe uma negativa da torre ele fica sobrevoando até perguntar de novo se pode pousar. Se não puder ele fica sobrevoando, até que ele desiste, vai pousar em outro lugar ou ele espera até que uma hora a torre vai responder que ele pode pousar.

É mais ou menos isso que iremos fazer agora. Iremos entrar em um Loop, e dentro desse loop, nós realizaremos uma chamada a função RLock(). Só sairemos do LOOP quando os dados tiverem o travamento ou quando o operador desistir de gravar.

31.6 Conclusão

Entender os requisitos para um bom funcionamento do seu software em rede é importante por dois motivos: o mais óbvio deles é o bom funcionamento da sua aplicação em rede durante o acesso aos seus arquivos DBFs. Mas também tem um segundo motivo que torna esse conhecimento importante. Muitos programadores utilizam bancos de dados clientes servidores, padrão SQL. Esses bancos são ótimos e representam um avanço em relação aos DBFs, conforme já foi dito em capítulos anteriores. O problema é que muitas pessoas acabam ficando sem saber como as coisas funcionam "por debaixo do capô". O conhecimento que adquirimos nesse capítulo é bem raso em comparação a complexidade que é a gerência de um banco de dados PostgreSQL ou Oracle, mas a ausência desse conhecimento nos deixa em desvantagem, quando precisarmos realizar algum ajuste fino no banco de dados

(tunning). Quando vamos estudar como esse processo funciona nos bancos SQL, nos deparamos com termos como índices, travamento ou "tamanho da página"(pagesize), e não nos damos conta que tais conceitos estão baseados nessa simples teoria vista durante esses capítulos.

32 RDD

> Pedi, e dar-se-vos-á; buscai e encontrareis; batei, e abrir-se-vos-á. Porque aquele que pede recebe; e o que busca encontra; e, ao que bate, se abre.
>
> Mateus 6:7-8

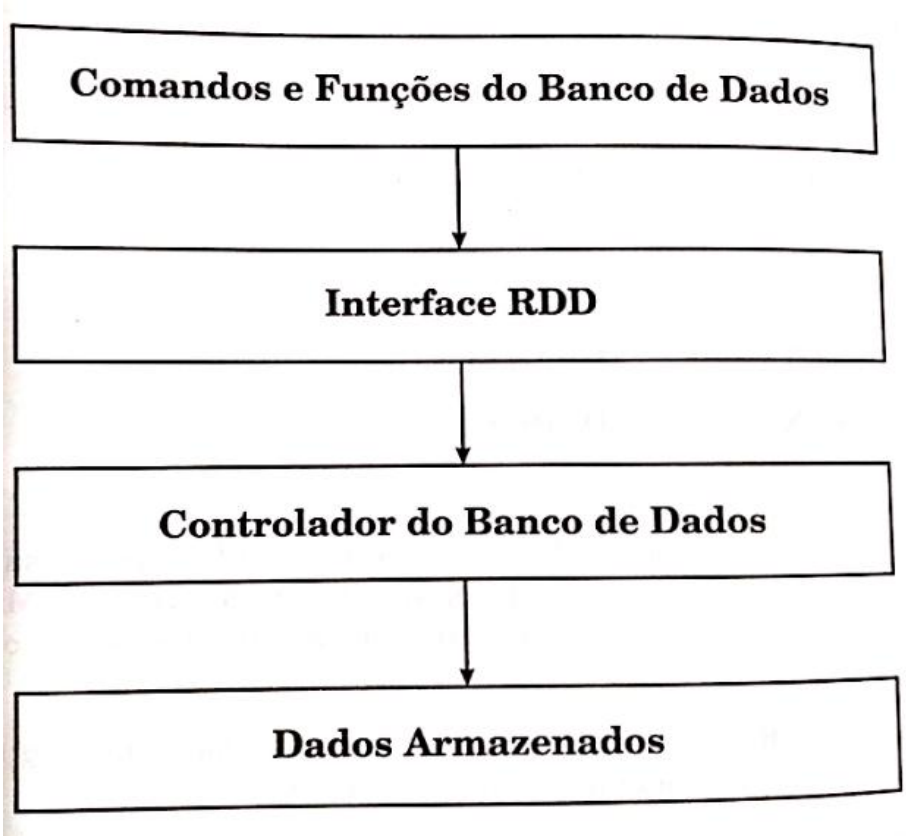
Objetivos do capítulo

- Obter conhecimentos básicos de Windows

32.1 Introdução

O Harbour permite que seus dados sejam acessados independentemente do formato. Isso é possível graças ao sistema de controladores intercambiáveis de banco de dados (RDD[1]). Nos últimos capítulos, vimos como se trabalhar com arquivos DBFs. Vimos também que esse sistema possui um conjunto de funções e comandos que facilitam a leitura e gravação dos dados. A proposta inicial do RDD é permitir que você codifique apenas uma vez, e depois passe a usar esse mesmo código, com pouca ou nenhuma alteração, para acessar outras fontes de dados ou para ordenar o DBF com outro índice diferente do padrão NTX. Um RDD significa uma camada extra de dados que permite a seu programa acessar as tabelas de dados físicos (figura 32.1).

Figura 32.1: O sistema RDD do Harbour - [Spence 1994, p. 225]



Basicamente temos os seguintes tipos de RDDs:

1. base de dados DBF locais, ou em rede local.
2. base de dados DBF remota, simulando a arquitetura cliente x servidor, mas os arquivos continuam sendo DBFs
3. base de dados SQL

O Harbour fornece os seguintes RDDs :

- .DBF/.NTX - O padrão nativo do Clipper e do Harbour

[1] Replaceable Data Driver

- .DBF/.CDX - Nativos do FoxPro da Microsoft
- SQLMIX - acesso a dados armazenados em banco de dados SQL e criação de DBFs virtuais em memória.

Além dos citados, temos também os RDDs disponibilizados por terceiros, dentre eles citamos :

- Mediator - RDD comercial para bancos Oracle, MS SQL Server, PostgreSQL e MySQL[2].
- ADS - Cliente para o Advantage Database Server[3]
- LetoDB - RDD open source para acessar banco de dados no formato DBF, mas utilizando todo o poder da arquitetura cliente x servidor[4].
- SQLRDD - Produto comercial com suporte a uma variedade de banco de dados. [5]

32.2 Procedimento geral para se trabalhar com um RDD

32.2.1 Inclua a biblioteca

Cada RDD está armazenado em uma lib a parte. Essa lib precisa ser incluída durante o processo de linkedição. Além de incluir a lib durante esse processo você precisa referenciar o RDD dentro do seu código
Segundo Spence,

> Os RDDs ficam armazenados em bibliotecas e, como você sabe, os linkeditores só linkeditam o que eles precisam das bibliotecas. Você precisa solicitar explicitamente os RDDs que você usará para que o linkeditor possa linkeditá-los. Faça isso usando a instrução REQUEST. [Spence 1994, p. 225]

Resumindo:

1. inclua a lib do RDD, através do seu arquivo hbc correspondente. Isso é feito durante o processo de compilação;
2. no seu código, antes da rotina MAIN, faça uma referência a esse RDD através do comando REQUEST.

Veremos isso com detalhes nos próximos tópicos

32.2.2 Ao abrir o arquivo, informe qual RDD ele deve usar

Se você não fizer isso o Harbour usará o DBFNTX, que é o padrão.
Nos tópicos seguintes nós abordaremos dois RDDs : o SQLMIX e o DBFCDX.

[2] Maiores informações em `http://www.otc.pl/index.asp?s=35`

[3] O ADS é um produto comercial da multinacional SAP, maiores informações em `https://devzone.advantagedatabase.com/`

[4] `https://www.kresin.ru/en/letodb.html`

[5] `https://www.xharbour.com/sqlrdd1.html`

32.3 SQLMIX

O SQLMIX é um RDD que possibilita ao Harbour acessar bases de dados em outros formatos, e apresentar o resultado como se fosse um DBF. Explicando de outra forma: você vai acessar uma base de dados SQL, mas verá o resultado da pesquisa em formato DBF. Com isso, você vai poder usar uma variedade grande de comandos e funções do Harbour nesse DBF virtual.

Importante: a apresentação do SQLMIX será feita em duas etapas. A primeira delas será vista nesse capítulo, onde iremos usar o SQLMIX para acessar um array, e a segunda parte usaremos o SQLMIX para acessar uma base de dados SQL. Isso porque o pré-requisito para o entendimento completo do SQLMIX é conhecer SQL e também configurar o Harbour para acessar essas bases de dados. Como ainda não temos esse conhecimento, iremos deixar o acesso a base de dados SQL para um capítulo posterior. Mas não se preocupe, pois tudo o que você aprender aqui será aproveitado no futuro.

32.3.1 Iniciando com o SQLMIX

Listagem 32.1: Programa simples para uso do SQLMIX.
Fonte: codigos/sqlmix01.prg

```
REQUEST SQLMIX                                                          1
PROCEDURE Main                                                          2
                                                                        3
    LOCAL aStruct := { { "NOME", "C", 20, 0 },;                         4
                       { "TELEFONE", "C", 10, 0 } }                     5
                                                                        6
    dbCreate( "agenda", aStruct , "SQLMIX", .T.)                        7
    APPEND BLANK                                                        8
    REPLACE FIELD->NOME WITH "GERALDO"                                  9
    REPLACE FIELD->TELEFONE WITH "980768-9087"                          10
    ? FIELD->NOME, FIELD->TELEFONE                                      11
    ? "O arquivo tem " , RECCOUNT() , " registro"                       12
                                                                        13
    WAIT                                                                14
                                                                        15
RETURN                                                                  16
```

Para compilar o programa faça assim

.:Resultado:.

```
hbmk2 sqlmix01 rddsql.hbc
```

Observações iniciais

1. Quando for compilar, inclua o arquivo rddsql.hbc. Esse arquivo adicional incluirá a biblioteca que dá suporte ao SQLMIX, sem ele o seu programa não irá compilar.
2. Antes da função MAIN, informe REQUEST SQLMIX. Isso irá incluir a biblioteca que você disponibilizou no passo anterior.

3. Note que a função DBCREATE() já abriu o "arquivo"usando o RDD SQLMIX.

Normalmente nós usamos a função DBCREATE() apenas com os dois parâmetros obrigatórios: o nome do arquivo a ser criado e a matriz com a estrutura do arquivo. Contudo, essa função tem o poder de abrir o arquivo após ele ser criado, e foi assim que usamos essa função no exemplo da listagem 32.1.

Qualquer dúvida consulte a descrição sintática de DBCREATE() na página 533.

O "arquivo"que foi criado na realidade não existe no disco, ele é um DBF que existe só na memória. Use o gerenciador de arquivos ou o comando dir (Prompt de Comando) e veja que nenhum arquivo foi criado. Agora, se você trocar o driver para DBFNTX, conforme o trecho abaixo, verá que o arquivo será criado em disco :

```
dbCreate( "agenda", aStruct , "DBFNTX", .T.)
```

Execute o programa após as alterações e note que o arquivo "agenda.dbf"foi criado em disco e um registro foi adicionado. Isso prova que o driver "DBFNTX", o padrão do Harbour, trabalha com arquivos reais. O "SQLMIX"só manipula arquivos em memória.

Na prática, isso significa que os arquivos :

1. não podem ser armazenados para uso posterior;
2. não podem ser compartilhados, ou seja, as funções de rede não surtem efeito.
3. não podem ser usados por funções que manipulam arquivos em disco, por exemplo: FILE().

Ou seja, os "arquivos"não são arquivos de verdade. Se você se lembrar disso não terá problemas com o SQLMIX.

32.3.2 O tamanho do campo caractere é variável

Você precisa ficar atento ao tamanho dos campos em um DBF Virtual, isso porque eles são variáveis, diferentes dos campos de um arquivo DBF real. O exemplo da listagem 32.2 irá criar o mesmo arquivo usando dois drivers distintos: o SQLMIX e o DBFNTX. Veja a diferença no tamanho dos campos.

Listagem 32.2: Programa simples para uso do SQLMIX.
Fonte: codigos/sqlmix02.prg

```
REQUEST SQLMIX                                                      1
PROCEDURE Main                                                      2
                                                                    3
    LOCAL aDriver := { "SQLMIX" , "DBFNTX" }, x                     4
    LOCAL aStruct := { { "NOME", "C", 20, 0 },;                     5
                       { "TELEFONE", "C", 10, 0 } }                 6
                                                                    7
    FOR x := 1 TO LEN( aDriver )                                    8
        dbCreate( "agenda", aStruct , aDriver[x], .T.)              9
        APPEND BLANK                                                10
        REPLACE FIELD->NOME WITH "GERALDO"                          11
        REPLACE FIELD->TELEFONE WITH "980768-9087"                  12
        ? "Usando o driver " + aDriver[x]                           13
```

```
        ? FIELD->NOME , LEN( FIELD->NOME )
        ? FIELD->TELEFONE, LEN( FIELD->TELEFONE)
    NEXT

RETURN
```

Execute o programa e comprove a diferença:

.:Resultado:.

```
Usando o driver SQLMIX
GERALDO          7
980768-9087         11
Usando o driver DBFNTX
GERALDO                     20
980768-908         10
```

Na prática, isso significa que **o tamanho do campo se altera de acordo com o valor gravado nele.**

32.3.3 O conteúdo do campo pode extrapolar o seu tamanho

Em condições normais (driver DBFNTX) o tamanho máximo do campo sempre será o limite máximo do valor a ser gravado, se o campo tem tamanho 10, então eu só posso gravar, no máximo, dez bytes nesse campo. O SQLMIX trabalha diferente, o valor pode extrapolar o tamanho do campo. O exemplo da listagem 32.3 irá ilustrar essa situação.

Listagem 32.3: SQLMIX: extrapolando o tamanho do campo.
Fonte: codigos/sqlmix03.prg

```
REQUEST SQLMIX
PROCEDURE Main

    LOCAL aDriver := { "SQLMIX" , "DBFNTX" }, x
    LOCAL aStruct := { { "NOME", "C", 10, 0 } }

    FOR x := 1 TO LEN( aDriver )
        dbCreate( "agenda", aStruct , aDriver[x], .T.)
        APPEND BLANK
        REPLACE FIELD->NOME WITH "SUPERIOR AO TAMANHO DO CAMPO"
        ? "Usando o driver " + aDriver[x]
        ? FIELD->NOME , LEN( FIELD->NOME )
    NEXT

RETURN
```

Execute o programa e comprove a diferença:

.:Resultado:.

```
Usando o driver SQLMIX
UM NOME GRANDE, SUPERIOR AO TAMANHO DO CAMPO         44
Usando o driver DBFNTX
UM NOME GR         10
```

O próximo exemplo tenta gravar um dado numérico superior ao tamanho do campo. Veja que o SQLMIX aceita também:

Listagem 32.4: SQLMIX: extrapolando o tamanho do campo numérico.
Fonte: codigos/sqlmix07.prg

```
REQUEST SQLMIX
PROCEDURE Main

    LOCAL aDriver := { "SQLMIX" , "DBFNTX" }, x
    LOCAL aStruct := { { "IDADE", "N", 2, 0 } }

    FOR x := 1 TO LEN( aDriver )
        ? "Usando o driver " + aDriver[x]
        dbCreate( "agenda", aStruct , aDriver[x], .T.)
        APPEND BLANK
        REPLACE IDADE WITH 1000
        ? "Valor do campo : " , hb_ValToExp( FIELD->IDADE )
        ? "Campo tipo : " , VALTYPE( FIELD->IDADE )
    NEXT

RETURN
```

Execute o programa e comprove a diferença:

.:Resultado:.

```
Usando o driver SQLMIX
Valor do campo :  1000
Campo tipo :  N
Usando o driver DBFNTX
Error DBFNTX/1021  Data width error: IDADE
Called from MAIN(11)
```

32.3.4 Um campo não gravado possui o tamanho definido em sua estrutura

Nesse caso, o driver SQLMIX se comporta igual ao driver padrão do Harbour, confira a listagem 32.5.

Listagem 32.5: SQLMIX: o tamanho default do campo.
Fonte: codigos/sqlmix04.prg

```
REQUEST SQLMIX
PROCEDURE Main

    LOCAL aDriver := { "SQLMIX" , "DBFNTX" }, x
    LOCAL aStruct := { { "NOME", "C", 10, 0 } }

    ? "NÃO IREI GRAVAR, APENAS CRIAR O REGISTRO"
    FOR x := 1 TO LEN( aDriver )
        dbCreate( "agenda", aStruct , aDriver[x], .T.)
        APPEND BLANK
        ? "Usando o driver " + aDriver[x]
```

```
        ? "Campo tipo : " , VALTYPE( FIELD->NOME ) ,;
          " Tamanho : " , LEN( FIELD->NOME )
    NEXT

RETURN
```

Execute o programa e comprove a diferença:

.:Resultado:.

```
NÃO IREI GRAVAR, APENAS CRIAR O REGISTRO
Usando o driver SQLMIX
Campo tipo :  C  Tamanho :          10
Usando o driver DBFNTX
Campo tipo :  C  Tamanho :          10
```

32.3.5 O valor NIL é aceito pelo SQLMIX

O driver SQLMIX aceita valores NIL gravado nos campos do DBF virtual, diferente do DBFNTX. Confira a listagem 32.6.

Listagem 32.6: SQLMIX: o valor NIL.
Fonte: codigos/sqlmix05.prg

```
REQUEST SQLMIX
PROCEDURE Main

    LOCAL aDriver := { "SQLMIX" , "DBFNTX" }, x
    LOCAL aStruct := { { "NOME", "C", 10, 0 } }

    FOR x := 1 TO LEN( aDriver )
        ? "Usando o driver " + aDriver[x]
        dbCreate( "agenda", aStruct , aDriver[x], .T.)
        APPEND BLANK
        REPLACE NOME WITH NIL
        ? "Valor do campo : " , hb_ValToExp( FIELD->NOME )
        ? "Campo tipo : " , VALTYPE( FIELD->NOME )
    NEXT

RETURN
```

Execute o programa e comprove a diferença:

.:Resultado:.

```
Usando o driver SQLMIX
Valor do campo :  NIL
Campo tipo :  U
Usando o driver DBFNTX
Error DBFNTX/1020  Data type error: NOME
Called from MAIN(11)
```

Note que o SQLMIX aceita que um dado NIL seja gravado, mas o DBFNTX não.

32.3.6 Tipos de dados diferentes

O SQLMIX não aceita que um tipo de dado diferente do que foi definido no campo seja gravado. Confira a listagem 32.7.

Listagem 32.7: SQLMIX: tipo de dado diferente.
Fonte: codigos/sqlmix06.prg

```
REQUEST SQLMIX
PROCEDURE Main

    LOCAL aDriver := { "SQLMIX" , "DBFNTX" }, x
    LOCAL aStruct := { { "NOME", "C", 10, 0 } }

    FOR x := 1 TO LEN( aDriver )
        ? "Usando o driver " + aDriver[x]
        dbCreate( "agenda", aStruct , aDriver[x], .T.)
        APPEND BLANK
        REPLACE NOME WITH 1000
        ? "Valor do campo : " , hb_ValToExp( FIELD->NOME )
        ? "Campo tipo : " , VALTYPE( FIELD->NOME )
    NEXT

RETURN
```

.:Resultado:.

```
Error SQLMIX/0  Data type error: NOME
```

Nesse caso, o SQLMIX se comporta da mesma forma que o DBFNTX.

32.3.7 Conclusão

O SQLMIX é o RDD que foi desenvolvido para simular o resultado de consultas SQL utilizando DBFs virtuais (em memória). Você pode usar o SQLMIX para trabalhar com arrays como se fossem DBFs (foi o que nós fizemos). Ele difere do DBFNTX em alguns pontos, mas já podemos adiantar que essas diferenças são aceitáveis, já que um banco de dados SQL funciona diferente de um banco DBF. Basicamente uma tabela SQL :

1. Tem seus campos caracteres de tamanho variável (VARCHAR)
2. Aceita valores nulos em seus campos (para otimizar o espaço utilizado)

Veremos essas e outras características quando formos estudar os bancos SQL. Depois do estudo do SQL, voltaremos a estudar o acesso a esses dados através do SQLMIX.

32.4 DBF/CDX

32.4.1 Introdução

Já vimos que o sistema de indexação do Harbour é o DBF/NTX. O DBF/CDX é um sistema de indexação utilizado pelo Microsoft FoxPro e tem uma aceitação muito grande entre os desenvolvedores Harbour.

32.4.2 Iniciando com DBFCDX

Informando a biblioteca

Quando você for trabalhar com o índice CDX, é necessário que a biblioteca seja informada através do arquivo dbfcdx.hbc. Por exemplo:

.:Resultado:.

```
hbmk2 seuprograma dbfcdx.hbc
```

Antes da função MAIN, faça

```
REQUEST DBFCDX
```

Ao abrir o arquivo faça assim

Quando for abrir o arquivo informe o driver :

```
USE funcionario VIA "DBFCDX"
```

Se você não informar o driver, o padrão DBFNTX será usado.

32.4.3 O índice CDX

A diferença entre CDX e NTX

Basicamente, a diferença principal entre o índice CDX e o NTX é a forma com que os índices são referenciados. No caso dos índices NTX, cada índice equivale a um arquivo. Por exemplo, no trecho abaixo temos um arquivo DBF usando três índices. Cada índice equivale a um arquivo separado :

```
USE funcionario
SET INDEX TO fnome, fdata, fsalario
```

Já um arquivo usando o driver DBFCDX, o mesmo exemplo seria assim:

```
USE funcionario
SET INDEX TO indice
```

Ou seja, um mesmo arquivo CDX pode armazenar vários índices.

Criando um arquivo CDX

Você deve estar lembrado que para criarmos um arquivo NTX, nós fazemos assim :

```
USE funcionario
INDEX ON nome TO fnome
INDEX ON STR(salario) TO fsalario
INDEX ON DTOS(niver) TO fdata
```

Para criarmos um índice CDX, nós fazemos assim :

```
USE funcionario VIA "DBFCDX"
INDEX ON nome TAG ORDEM_NOME TO indice
INDEX ON STR(salario) TAG ORDEM_SAL  TO indice
INDEX ON DTOS(niver) TAG ORDEM_DATA TO indice
```

Como o nome do arquivo de índices é o mesmo, nós repetimos o nome do arquivo de índices. O que diferencia um índice de outro é a cláusula "TAG". Dessa forma, todos os índices ficarão agrupados dentro de um mesmo arquivo. Só tenha cuidado para não repetir a mesma TAG duas ou mais vezes, já que é ela que diferencia um índice de outro dentro do arquivo.

Importante: Se você criar um arquivo de índices com o mesmo nome do arquivo DBF, então esse arquivo sempre será aberto automaticamente.

Exemplo de uso

Um exemplo completo de uso está descrito na listagem 32.8.

Listagem 32.8: DBFCDX: exemplo de uso.
Fonte: codigos/dbfcdx01.prg

```
REQUEST DBFCDX                                                          1
#define ESTRUTURA_DBF { { "NOME" , "C" , 30 , 0 },;                     2
                        { "SALARIO" , "N" , 10 , 0 },;                  3
                        { "NIVER" , "D" , 8 , 0 } }                     4
                                                                        5
FIELD NOME, SALARIO, NIVER                                              6
                                                                        7
PROCEDURE MAIN                                                          8
                                                                        9
    CriaDadosParaTeste() // Cria base para testes                       10
    USE funcionario VIA "DBFCDX"                                        11
    ? "ORDEM NOME : "                                                   12
    GO TOP                                                              13
    DO WHILE .NOT. EOF()                                                14
        ?? FIELD->NOME , " "                                            15
        SKIP                                                            16
    ENDDO                                                               17
    ? "ORDEM SALARIO : "                                                18
    SET ORDER TO "ORDEM_SAL"                                            19
    GO TOP                                                              20
    DO WHILE .NOT. EOF()                                                21
```

```
        ?? FIELD->NOME , " "
        SKIP
    ENDDO
    ? "ORDEM DATA : "
    SET ORDER TO "ORDEM_DATA"
    GO TOP
    DO WHILE .NOT. EOF()
        ?? FIELD->NOME , " "
        SKIP
    ENDDO

RETURN
/* Função que cria a base de dados de testes */
STATIC PROCEDURE CriaDadosParaTeste

    IF .NOT. FILE( "funcionario.dbf" )
        FERASE("funcionario.cdx")
        // 1. Cria o arquivo
        DBCREATE( "funcionario" , ESTRUTURA_DBF )
        // 2. Abre e indexa
        USE funcionario VIA "DBFCDX" EXCLUSIVE
        INDEX ON NOME TAG ORDEM_NOME TO funcionario
        INDEX ON STR(SALARIO) TAG ORDEM_SAL TO funcionario
        INDEX ON DTOS(NIVER) TAG ORDEM_DATA TO funcionario
        // 3. Insere dados para teste
        APPEND BLANK
        * A PRIMEIRA EM ORDEM ALFABÉTICA (LETRA A)
        REPLACE NOME WITH "ANA MARIA"
        REPLACE SALARIO WITH 8000
        REPLACE NIVER WITH STOD( "19990608" )
        APPEND BLANK
        REPLACE NOME WITH "CHICO CARLOS"
        * GANHA MENOS (SALÁRIO MENOR)
        REPLACE SALARIO WITH 800
        REPLACE NIVER WITH STOD( "20000707" )
        APPEND BLANK
        REPLACE NOME WITH "VLADIMIR"
        REPLACE SALARIO WITH 5000
        * O MAIS VELHO (DATA MENOR)
        REPLACE NIVER WITH STOD( "19700707" )
        // 4. Fecha o arquivo
        CLOSE
    ENDIF
```

O resultado do programa é

.:Resultado:.

```
ORDEM NOME : ANA MARIA    CHICO CARLOS    VLADIMIR
ORDEM SALARIO : CHICO CARLOS    VLADIMIR    ANA MARIA
ORDEM DATA : VLADIMIR    ANA MARIA    CHICO CARLOS
```

Aspectos importantes :

1. Informe a referencia a lib : REQUEST DBFCDX
2. Informe o driver ao abrir : USE funcionario VIA "DBFCDX"
3. Ao indexar use a cláusula TAG para diferenciar um índice de outro
4. A ordem do índice ativo pode ser feita através do nome da TAG : SET ORDER TO "NOME_DA_TAG"

32.4.4 Funções que auxiliam no uso de RDDs

RDDSetDefault

Define um RDD default ou retorna o valor do RDD atual. Até agora nós vimos o comando USE associado com a cláusula VIA, para poder determinar o RDD utilizado. Se por acaso o programador não colocar a cláusula VIA, determinando o RDD, o Harbour irá usar o RDD default que é o DBFNTX. Ocorre que você pode mudar o RDD default. Assim, o código abaixo irá abrir três arquivos usando o RDD DBF/CDX.

```
RDDSetDefault("DBFCDX")
USE cliente SHARED
USE fornecedor SHARED NEW
USE pdv EXCLUSIVE NEW
```

Caso seja necessário recuperar o RDD anterior, é só armazenar no retorno da função RDDSetDefault(), por exemplo:

```
cRDDAnterior := RDDSetDefault("DBFCDX")
? cRDDAnterior // Imprime DBFNTX
// Realiza operações usando DBFCDX
RDDSetDefault( cRDDAnterior ) // Retorna o default para o anterior
```

Se você usar essa função sem parâmetros, então ela apenas irá retornar o valor do RDD atual, por exemplo:

```
? RDDSetDefault() // Imprime o RDD atual
```

Descrição sintática 53

1. Nome : RDDSetDefault
2. Classificação : função.
3. Descrição : Define ou retorna o RDD padrão para o aplicativo
4. Sintaxe

```
rddSetDefault([<cNovoRDD>])
→ cRDDAnterior

cRDDAnterior : string contendo o RDD anterior.
```

5. Parâmetros
   - [<cNovoRDD>] : Nome do novo RDD padrão

Fonte : https://harbour.github.io/doc/clc53.html#rddsetdefault

RDDName

RDDName retorna o nome do RDD da área de trabalho ativa. Essa função depende do contexto da área de trabalho. A listagem abaixo, retirado de https://harbour.github.io/doc/clc53.html#rddname, ilustra essa função.

```
    USE Customer VIA "DBFNTX" NEW                                   1
    USE Sales    VIA "DBFCDX" NEW                                   2
                                                                    3
    ? rddName()                             //  Retorna DBFCDX      4
    ? Customer->( rddName() )               //  Retorna DBFNTX      5
    ? Sales->( rddName() )                  //  Retorna DBFCDX      6
```

RDDList

RDDList() retorna um array com a lista de RDDs disponíveis. A listagem abaixo (listagem 32.9) mostra os RDDs disponíveis pelo Harbour. Desses disponíveis, o default é o DBF/NTX.

Listagem 32.9: Lista de RDDs disponíveis.
Fonte: codigos/rddlist01.prg

```
                                                                    1
PROCEDURE Main                                                      2
    LOCAL aRdd := rddList(),x                                       3
                                                                    4
    FOR x := 1 TO LEN( aRdd )                                       5
        ? aRdd[x]                                                   6
    NEXT                                                            7
```

```
RETURN
```

Compile assim :

.:Resultado:.

```
hbmk2 rddlist01
```

O resultado é

.:Resultado:.

```
DBF
DBFFPT
DBFNTX
DBFBLOB
```

Vamos incluir mais dois RDDs, o DBF/CDX e o SQLMIX, e executar o nosso programa novamente.

.:Resultado:.

```
hbmk2 rddlist01 dbfcdx.hbc rddsql.hbc
```

Ao executar o programa o resultado é **exatamente o mesmo**.

.:Resultado:.

```
DBF
DBFFPT
DBFNTX
DBFBLOB
```

Isso acontece porque o REQUEST precisa ser incluído no código. Vamos fazer uma pequena modificação no código, conforme a listagem 32.10.

Listagem 32.10: Lista de RDDs disponíveis II.
Fonte: codigos/rddlist02.prg

```
REQUEST DBFCDX // <--- Suporte ao DBFCDX
REQUEST SQLMIX // <--- Suporte ao SQLMIX
PROCEDURE Main
    LOCAL aRdd := rddList(),x

    FOR x := 1 TO LEN( aRdd )
        ? aRdd[x]
    NEXT

RETURN
```

Compile assim:

.:Resultado:.

```
hbmk2 rddlist02 dbfcdx.hbc rddsql.hbc
```

Agora sim, os drivers foram incluídos

.:Resultado:.

```
    SQLBASE
    SQLMIX
    DBF
    DBFFPT
    DBFNTX
    DBFCDX
    DBFBLOB
```

DBSetDriver

A função DBSetDriver foi criada no tempo do Clipper 5.2. Evidentemente ela ainda está disponível no Harbour, mas os desenvolvedores aconselham que ela seja trocada pela função rddSetDefault().

Um exemplo, retirado de `https://harbour.github.io/doc/clc53.html#dbsetdriver`, exemplifica o uso de dbSetDriver().

```
dbSetDriver("DBFNDX")                                                  1
IF ( dbSetDriver() <> "DBFNDX" )                                       2
     ? "DBFNDX driver not available"                                   3
ENDIF                                                                  4
```

32.5 Conclusão

Parte IV

Programação TUI

33 Achoice

> Uma árvore boa não pode dar maus frutos, nem uma árvore má pode dar frutos bons.
>
> Jesus Cristo

Objetivos do capítulo

- Entender o que é um computador.
- Compreender o que é um programa de computador.
- Saber a diferença básica entre um montador, um compilador e um interpretador.
- Categorizar a linguagem Harbour entre as linguagens existentes.
- Instalar o material de aprendizado e apoio no seu computador.
- Compilar um exemplo que veio com o material de apoio.

33.1 O que é a função Achoice() ?

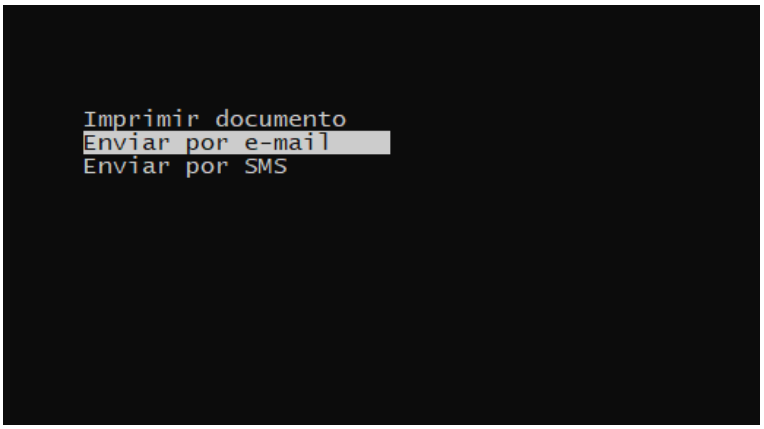
Quando queremos criar um menu do tipo "popup"a partir de um array, usamos a função Achoice(). Quando a função Achoice() é invocada, ela exibe um uma lista de ítens de menu é exibida dentro das coordenadas de uma área retangular.

Um exemplo básico da função Achoice() :

Listagem 33.1: Um exemplo básico de uso da função Achoice().
Fonte: codigos/achoice01.prg

```
PROCEDURE main

    LOCAL nOpt
    LOCAL aOpcoes := { "Imprimir documento" ,;
                       "Enviar por e-mail",;
                       "Enviar por SMS" }

    CLS
    nOpt := Achoice( 10 , 10 , 12 , 30 , aOpcoes )

    @ 15,0 SAY "Retorno de achoice() : " + STR(nOpt)
    ?
    WAIT "Tecle algo para continuar"

RETURN
```

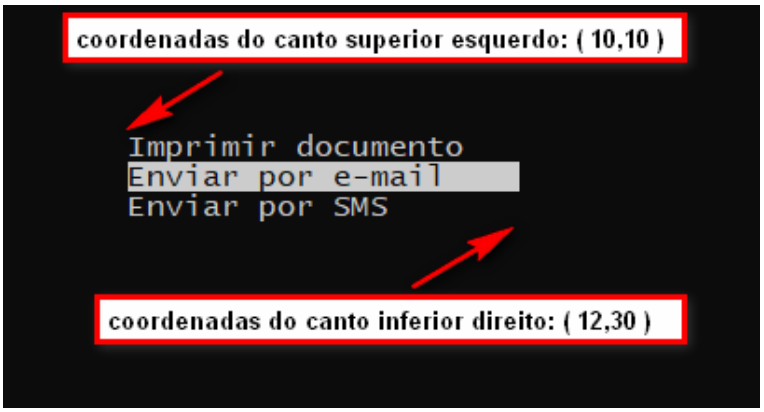
Figura 33.1: Um exemplo básico de uso da função Achoice.



1. Os quatro primeiros parâmetros da função são as coordenadas.
2. Os dois primeiros (no exemplo: 10 e 10) são do canto superior esquerdo da área retangular.
3. Os dois últimos (no exemplo: 12 e 30) são do canto superior esquerdo da área retangular.
4. O quinto parâmetro é o array com as opções.

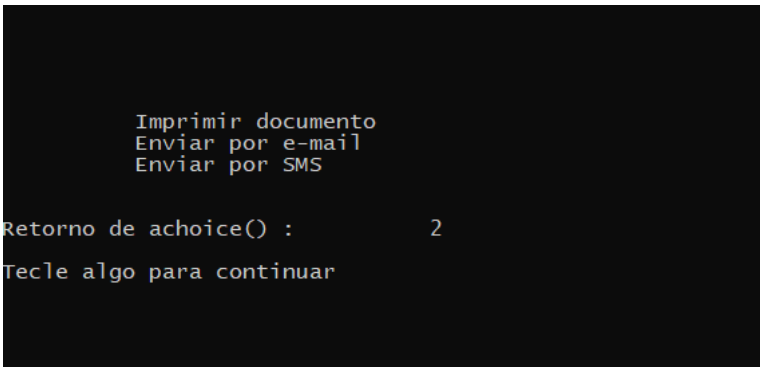
Veja abaixo a mesma figura com as coordenadas ilustradas :

Figura 33.2: Ilustração com as coordenadas da função Achoice().



O usuário pode usar as setas (para cima e para baixo) para navegar entre as opções. No nosso exemplo, o ítem selecionado está realçado com um fundo branco. Para selecionar o ítem tecle ENTER. Ao teclar ENTER "você sairá"da função e ela retornará um valor numérico referente a posição do ítem que foi selecionado. Confira esse comportamento na figura 33.3, onde o usuário selecionou o segundo elemento.

Figura 33.3: Saída da função Achoice() após teclar ENTER.



Se o usuário teclar ESC, então o retorno da função Achoice() será 0.
Dois detalhes :

1. A função achoice() não providencia uma moldura, você terá que fazer isso. Veremos adiante a forma ideal de se fazer isso.
2. É responsabilidade sua calcular a largura e o comprimento necessários para a exibição dos elementos do array.

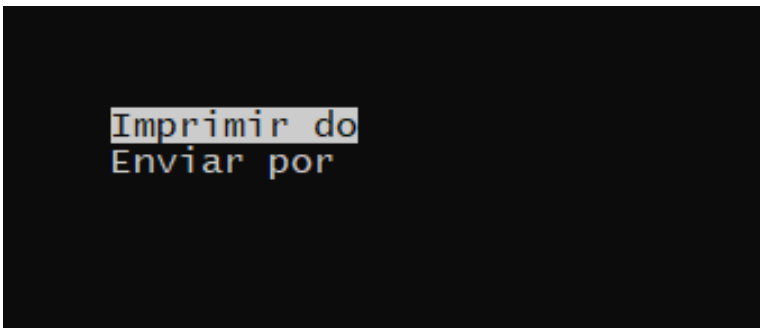
Por exemplo, a seguir temos o mesmo exemplo com uma área retangular menor.

Listagem 33.2: Uma área retangular menor.
Fonte: codigos/achoice0103.prg

```
nOpt := Achoice( 10 , 10 , 11 , 20 , aOpcoes )
```

O resultado :

Figura 33.4: O mesmo exemplo com uma área retangular menor.



O usuário poderá selecionar qualquer um dos elementos, mesmo que algum não apareça, a função fará um rolamento de tela automaticamente.

Bem, a ideia é essa. Nas próximas seções nós expandiremos esse exemplo.

34 DbEdit

As coisas têm vida própria.
Tudo é questão de despertar a sua alma.

Gabriel García Márquez

Objetivos do capítulo

- Aprender função Browser.
- Aprender a função dbEdit no formato mais simples.
- Aprender a usar a dbEdit com todos os parâmetros
- Escrever as próprias funções de usuário.

34.1 Introdução

A função DbEdit() é usada para exibir dados armazenados em um arquivo DBF na forma tabular. Essa função possui uma variedade de formas de uso, de modo que esse capítulo foi dedicado ao seu uso.

34.2 Browse()

Antes de iniciarmos o estudo da DbEdit(), iremos estudar a função Browse(). De acordo com a documentação do desenvolvedor, o Browse() é uma função de interface com o usuário que invoca um editor e um browser[1] (orientado por tabela) de utilização geral para os registros na área de trabalho corrente [Nantucket 1990, p. 5-36]. A listagem 34.1 nos mostra um exemplo bem simples da função Browse().

Listagem 34.1: Versão inicial do Browse() em tela cheia.
Fonte: codigos/browse.prg

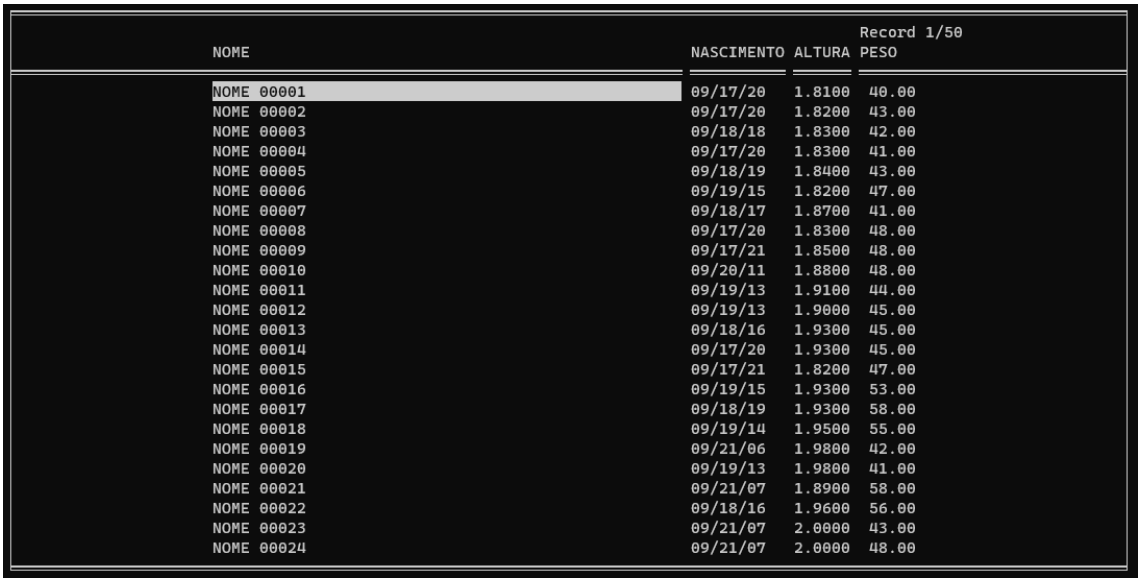
```
/**********************/
PROCEDURE Main

    CriaArquivo()
    USE ficha
    Browse()

RETURN
/**********************/
PROCEDURE CriaArquivo()

    LOCAL x, aStruct := {  { "NOME" , "C" , 50, 0 },;
                    { "NASCIMENTO" , "D" , 8 , 0 },;
                    { "ALTURA" , "N" , 6 , 4 },;
                    { "PESO" , "N" , 6 , 2 } }


    IF .NOT. FILE( "ficha.dbf" )
       DBCREATE( "ficha" , aStruct )
       USE ficha NEW
       FOR x := 1 TO 50
          APPEND BLANK
          REPLACE NOME WITH "NOME " + STRZERO( x , 5 )
          REPLACE NASCIMENTO WITH ;
                    DATE() - ( 365 * hb_RandomInt( 0 , x ) )
          REPLACE ALTURA WITH 1.6 + hb_RandomInt( 1 , x ) / 100
          REPLACE PESO WITH 30+hb_RandomInt( 10 , x + 11 )
          SKIP
       NEXT
       USE
    ENDIF
```

[1] O termo "browser"(navegador) aqui não se refere a web browser, mas a um grid com o conteúdo de um arquivo, ou vários. Quando nos referirmos a um navegador de internet, usaremos o termo “Web browser”

```
                                                                    32
RETURN                                                              33
/**********************/                                            34
```

O resultado está a seguir, na figura 34.1.

Figura 34.1: Função Browse()



Essa função é bem simples, ela possui apenas quatro parâmetros opcionais, correspondente as coordenadas. Caso você deseje informar as coordenadas da função Browse(), você precisa definir dois pontos[2] correspondentes à "quina"superior esquerda e a "quina"inferior direita do "grid".

O exemplo da listagem 34.2 mostra a função Browse() com as coordenadas definidas.

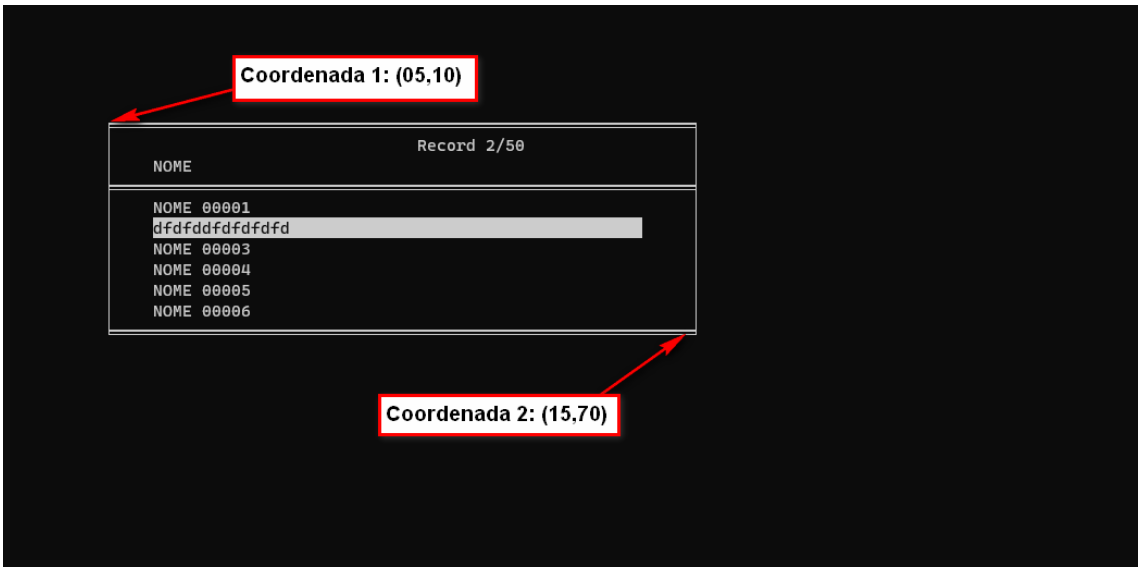
Listagem 34.2: Versão do Browse() com coordenadas definidas.
Fonte: codigos/browse02.prg

```
/**********************/                                            1
PROCEDURE Main                                                      2
                                                                    3
    CriaArquivo()                                                   4
    USE ficha                                                       5
    CLS                                                             6
    Browse( 05 , 10 , 15 , 70 )                                     7
                                                                    8
                                                                    9
RETURN                                                              10
/**********************/                                            11
```

O resultado é o da figura 34.2.

[2]Caso ache necessário, você pode revisar o sistema de coordenadas na página 259

Figura 34.2: Função Browse() com coordenadas



Descrição sintática 54

1. Nome : Browse()
2. Classificação : função.
3. Descrição : cria um "grid"com registros editáveis.
4. Sintaxe

```
Browse( [<nTopo>] , [<nEsquerda>] ,;
        [<nBase>] , [<nDireita>]) -> NIL
```

5. Parâmetros
   - [<nTopo>] e [<nEsquerda>] : coordenadas da "quina"superior esquerda do "grid".
   - [<nBase>] e [<nDireita>] : coordenadas da "quina"inferior direita do "grid".
6. Fonte : [Nantucket 1990, p. 5-36]

Importante: A função Browse() tem as seguintes características :

1. no canto superior direito temos uma linha de status informando :
   a) o registro / o total de registros;
   b) se o registro tiver sido excluído exibe a palavra «Deleted>";
   c) se chegou no início do arquivo exibe «bof>";
   d) se está no final do arquivo exibe «eof>".

2. ao se pressionar qualquer uma das teclas de navegação, a barra luminosa move-se para uma nova linha ou coluna;
3. quando você pressiona ENTER entra-se na edição do campo corrente;
4. a tecla DEL exclui o registro da linha corrente, caso o registro já esteja excluído ele recupera o registro (para recuperar, o SET DELETED deve estar OFF, senão o registro excluído não aparece);
5. se deseja ir para a última linha (final do arquivo) tecle Control + Page Down ou mantenha a seta para baixo pressionada até chegar no último registro.
6. para inserir um registro, vá para o final do arquivo (veja ítem anterior) e em seguida tecle a seta para baixo. Se você não digitar nada os dados não serão gravados.

Dica 170

A função Browse() não é indicada para usuários finais, porque ela sempre vem com o seu modo de edição e inclusão ativos. Outra coisa: você não vai poder suprimir alguma coluna ou personalizar o nome das mesmas, de modo que o nome da coluna sempre será o nome do respectivo campo no banco de dados. **Use essa função para checagens rápidas ou para testes**. Em resumo: evite a função Browse() no cliente final porque ela não pode ser personalizada, daí alguém pode querer uma pequena mudança que não poderá ser feita. Para poder personalizar o seu "grid", use a função DbEdit() ou o objeto TBrowse.

34.3 DbEdit : Introdução

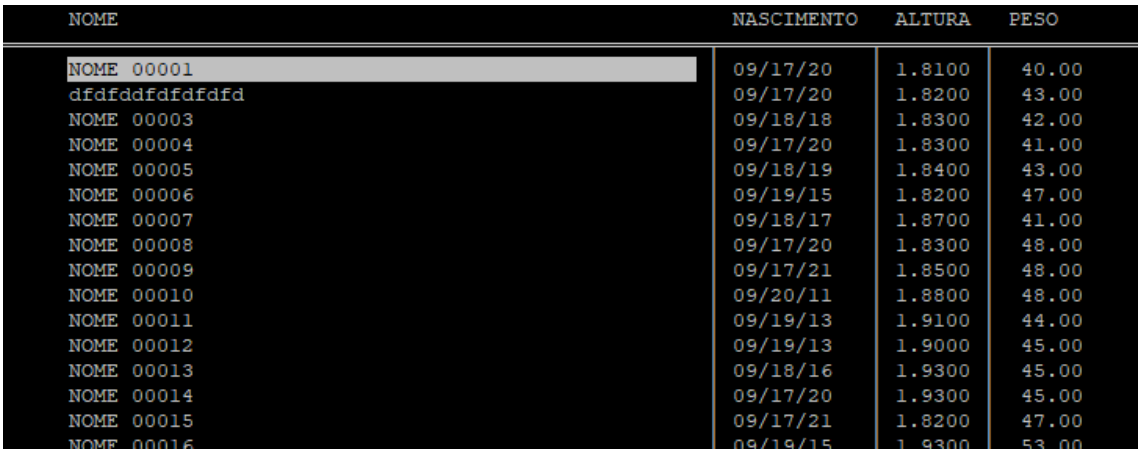
34.3.1 DbEdit() sem argumentos

A função DbEdit() em sua forma mais simples ésta descrita na listagem 34.3.

Listagem 34.3: DbEdit sem argumentos
Fonte: codigos/dbedit01.prg

```
PROCEDURE Main

    CriaArquivo()
    USE ficha
    CLS
    DbEdit()

RETURN
/*********************/
```

Figura 34.3: DbEdit() sem argumentos.



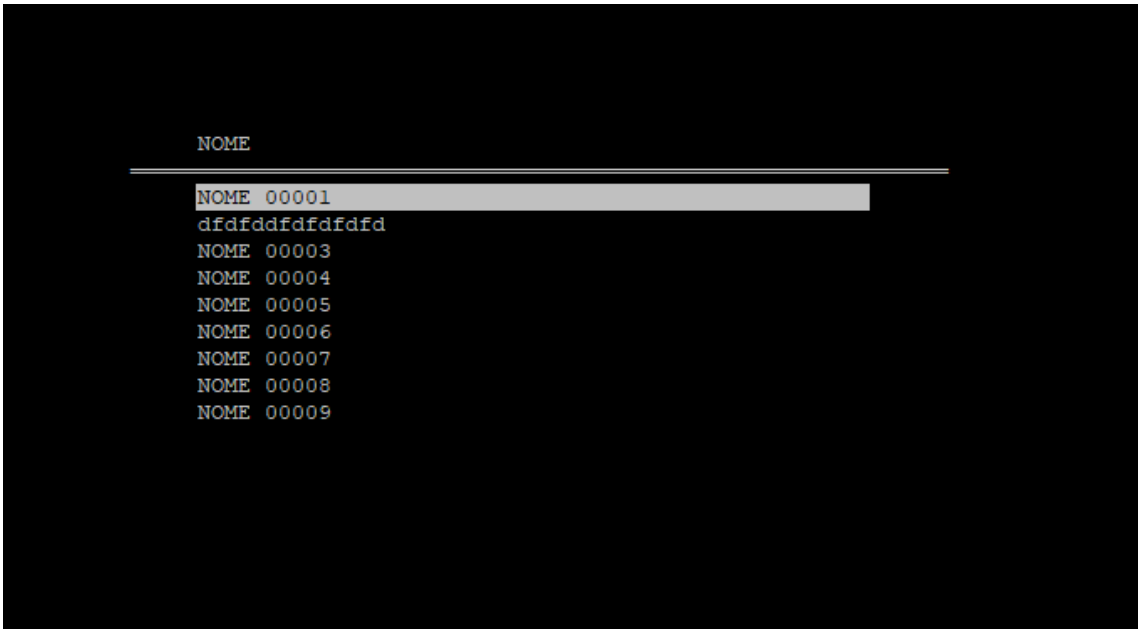
34.3.2 DbEdit() : adicionando as coordenadas

Vamos agora adicionar as coordenadas, conforme a listagem 34.4.

Listagem 34.4: DbEdit apenas com as coordenadas
Fonte: codigos/dbedit02.prg

```
PROCEDURE Main

    CriaArquivo()
    USE ficha
    CLS
    DbEdit( 5 , 10 , 15 , 70 )

RETURN
/*********************/
```

Figura 34.4: DbEdit() sem argumentos.



34.3.3 DbEdit() : exibindo apenas algumas colunas

Vamos agora exibir apenas algumas colunas, conforme a listagem 34.5.

Listagem 34.5: DbEdit com apenas algumas colunas
Fonte: codigos/dbedit03.prg

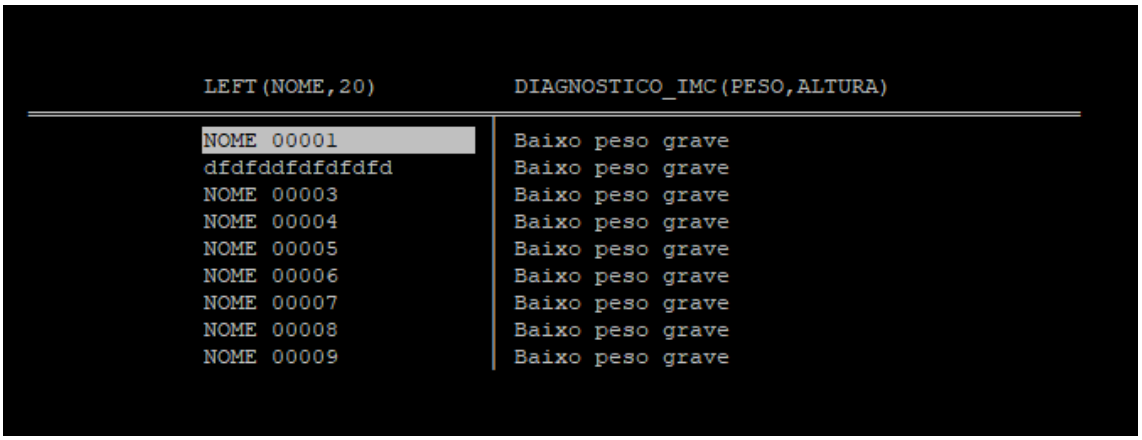
```
PROCEDURE Main

    LOCAL aCab := { "NOME" , "PESO" }

    CriaArquivo()
    USE ficha
    CLS
    DbEdit( 5 , 3 , 15 , 80 , aCab )

RETURN
```

Figura 34.5: DbEdit() com apenas algumas colunas.

Listagem 34.6: DbEdit() com funções nas colunas
Fonte: codigos/dbedit0301.prg

```
PROCEDURE Main

    LOCAL aCab := { "LEFT(NOME,20)" , "DIAGNOSTICO_IMC(PESO,ALTURA)"}
```

Figura 34.6: DbEdit() com funções nas colunas.



34.3.4 DbEdit() : formatando a saída através de máscaras

Vamos agora exibir apenas algumas colunas, conforme a listagem 34.7.

Listagem 34.7: DbEdit com apenas algumas colunas
Fonte: codigos/dbedit04.prg

```
PROCEDURE Main

    LOCAL aCab := { "NOME" , "PESO"  }
    LOCAL aPict := { "@S10" , "@RE 999.99"  }

    CriaArquivo()
    USE ficha
    CLS
    DbEdit( 5 , 3 , 15 , 80 , aCab , , aPict )
```

Figura 34.7: DbEdit() : uso de máscaras.

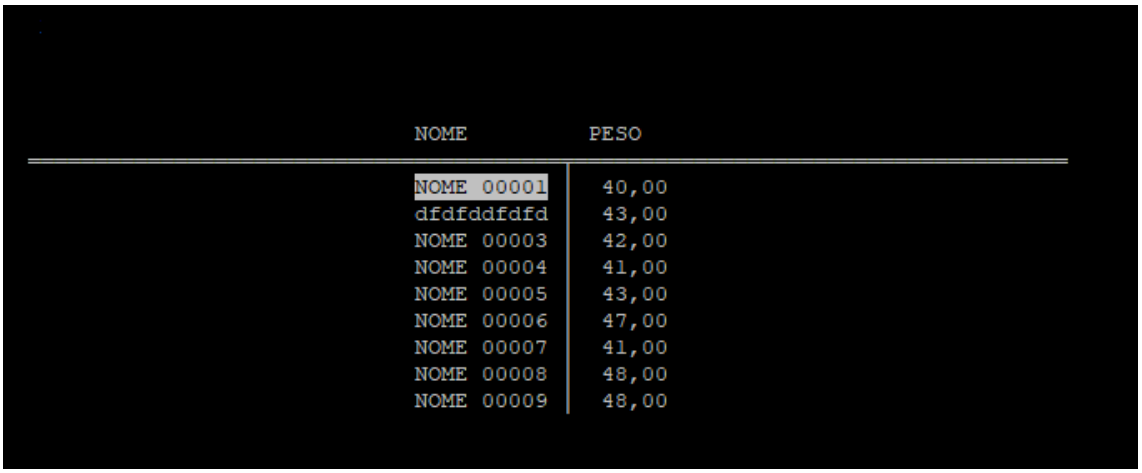


Figura 34.8: DbEdit() : uso de máscaras.

34.4 DbEdit : Sintaxe

Descrição sintática 55

- Nome : DbEdit()
- Classificação : função.
- Descrição : faz um browse em registros no formato de tabela (grid).
- Sintaxe

```
DbEdit( [<nTopo>] , [<nEsquerda>] ,;
[<nBase>] , [<nDireita>],;
[<acColunas>],;
[<cFuncaoUsuario>],;
[<acMascara>|<cMascara>],;
[<acCabecalhoColunas>|<cCabecalhoColunas>],;
[<aSeparadoresCabecalho>|<cSeparadoresCabecalho>],;
[<aSeparadoresColunas>|<cSeparadoresColunas>],;
[<aSeparadoresRodapes>|<cSeparadoresRodapes>],;
[<acRodapeColunas>|<cRodapeColunas>]
     ) -> NIL
```

- Parâmetros
  - [<nTopo>] e [<nEsquerda>] : coordenadas da "quina"superior esquerda do "grid".
  - [<nBase>] e [<nDireita>] : coordenadas da "quina"inferior direita do "grid".
  - [<acColunas>] : Um vetor contendo o nome dos campos.
  - [<cFuncaoUsuario>] : É o nome de uma função definida pelo usuário que é executada automaticamente quando uma tecla não reconhecível é pressionada ou quando não há nenhuma tecla pendente no buffer do teclado. Não coloque parênteses ou argumentos.
  - [<acMascara>|<cMascara>] : Esse parâmetro pode ser um vetor ou uma string. Se especificar o vetor, o seu conteúdo deve ser as máscaras de formatação de cada coluna. Se especificar uma string, a mesma formatação será aplicada a todas as colunas.
  - [<acCabecalhoColunas>|<cCabecalhoColunas>] : Esse parâmetro pode ser um vetor ou uma string. Se especificar um vetor, o seu conteúdo deve ser os cabeçalhos de cada coluna. Se especificar uma string, todos os cabeçalhos das colunas terão o mesmo caracter.

Descrição sintática 56

- Nome : DbEdit() - Continuação
- Parâmetros
  - [<aSeparadoresCabecalho>|<cSeparadoresCabecalho>] : Semelhante aos parâmetros anteriores, esse também pode ser um vetor ou uma string. Se for um vetor, o seu conteúdo deve ser os separadores dos cabeçalhos de cada coluna. Se for uma string, o mesmo separador será usado para todas as colunas.
  - [<aSeparadoresRodapes>|<cSeparadoresRodapes>] : Da mesma forma, pode ser um vetor ou uma string. Se for um vetor, o seu conteúdo deve ser os separadores de rodapé para cada coluna. Se for uma string, o mesmo separador será usado para todas as colunas.
  - [<acRodapeColunas>|<cRodapeColunas>] : Ainda seguindo o mesmo formato, pode ser um vetor ou uma string. Se for um vetor, o seu conteúdo deve ser os rodapés para cada coluna. Se for uma string, o mesmo rodapé será usado para todas as colunas.
- Fonte : [Nantucket 1990, p. 5-49]

34.5 DbEdit : Avançado

34.6 Criando nossas próprias funções

Parte V

Programação Orientada a Objetos

35 Programação orientada a objetos

Nada é mais poderoso do que
uma ideia cujo tempo chegou.

Victor Hugo

35.1 O que é orientação a objetos ?

Consultando a wikipédia, nós vemos que “programação orientada a objetos (também conhecida pela sua sigla POO) é um modelo de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.” [1] Spence, no seu clássico sobre Clipper 5.2 inicia a explanação assim : “a programação orientada a objetos [...] é um método de implementar sistemas de software que promete enormes ganhos na produtividade do programador, na confiabilidade e no suporte dos programas” [Spence 1994, p. 251].

Particularmente, eu tive dificuldades para entender o que é orientação a objetos. Na época em que eu tive contato com o tema, eu já programava profissionalmente, e os conceitos que eu lia me pareciam óbvios demais. Tão simples que eu não conseguia visualizar o real poder da programação orientada a objetos. De tanto ouvir os outros falarem que a POO [2] veio para ficar e que ela iria facilitar o desenvolvimento de software, a minha curiosidade ficava aguçada, mas a minha mente não conseguia entender direito o que era programar orientado a objeto.

O criador de uma das primeiras linguagens orientadas a objetos, o cientista da computação Bjarne Stroustrup lançou uma luz sobre o meu problema. Segundo ele, você pode usar uma linguagem orientada a objetos (como C++ , C# ou Java) e mesmo assim, não programar orientado a objetos. Isso quer dizer que, mais do que uma tecnologia, orientação a objetos é uma forma de se resolver um problema: um paradigma.

Nós já falamos um pouco sobre paradigmas, mas não custa nada repetir: paradigma é um conjunto de princípios que norteiam uma forma de pensar. Não é um conceito exclusivo da programação de computadores, mas da filosofia. Sem saber, estamos cercado por paradigmas diversos, em todas as áreas. Por exemplo, para um economista de formação marxista, o mundo é um palco onde classes lutam entre si. Já um economista liberal desconhece a luta de classes, mas reconhece apenas o homem e as suas liberdades individuais. Desconhecer os paradigmas é desconhecer a estrutura de uma forma de pensamento. Nos dias de hoje, onde a informação nos chega fragmentada (facebook, twitter, blogs, etc.), é cada vez mais comum as pessoas (principalmente os jovens) defenderem ideias que pertencem a paradigmas contraditórios entre si.

Bem, mas voltando para a programação de computadores: o que é o paradigma orientado a objetos ? A única forma que eu encontrei para explicar o paradigma orientado a objetos foi recorrendo a um evento que aconteceu a mais ou menos dois mil e quatrocentos anos.

35.1.1 Sócrates e o nascimento de uma nova forma de pensamento

Sócrates foi o primeiro homem no ocidente que estudou a forma que nós usamos para nos comunicarmos uns com os outros. Ele não deixou nada escrito, mas o seu discípulo Platão escreveu vários textos, chamados diálogos, onde diversos temas são abordados. O problema que Sócrates tentava resolver era a da busca pela verdade através da linguagem falada. Para Sócrates, se você estruturasse o seu raciocínio de uma forma errada (paradigma errado), os seus resultados seriam ineficazes. Ao

[1] Fonte: `https://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos` - Acessado em 17-Dez-2016

[2] Em alguns textos você poderá encontrar a sigla OOP (Object Oriented Programming).

analisar um discurso qualquer, Sócrates observou que os homens estruturavam um raciocínio sobre o verbo. Por exemplo: quando alguém era indagado sobre o que era a virtude, esse alguém poderia dizer: “virtude é **gerir** as coisas da cidade, é **fazer** bem aos amigos e [fazer] mal aos inimigos, é **administrar** a casa com sabedoria, é **cuidar** de seu interior, é **falar** a verdade, etc.”[3] Note que sempre o verbo tinha a proeminência dentro da argumentação. Sócrates entendeu que isso era um erro que nos levava a conclusões equivocadas. Segundo ele, o substantivo (o conceito) era quem deveria receber o destaque dentro de um diálogo. Por isso os diálogos socráticos abordam conceitos, como o amor, a justiça, a virtude, a paixão, a religião, etc. Dentro desses diálogos o substantivo deve ser definido, para depois o verbo entrar em ação. Os filósofos que vieram depois de Sócrates, até discordavam dele em alguns aspectos, mas não deixavam de reconhecer a importância do substantivo (conceito) em uma explicação qualquer.

Mas afinal, o que isso tem a ver com orientação a objetos ? Se você notou, o paradigma que temos estudado até agora tem como elemento primordial a rotina (função ou procedimento). O paradigma orientado a objetos inverte isso e coloca o objeto (a variável) em primeiro lugar[4]. A função (que agora recebe o nome de método) ainda existe, mas vem subordinada ao objeto (que é uma variável).

Platão tentou justificar a importância do substantivo da seguinte forma: todos os objetos que nós vemos, existem em um outro mundo anterior ao nosso. Antes de nós nascermos, as nossas almas contemplaram esses objetos nesse mundo anterior (conhecido como “mundo das ideias”). Quando chegamos a esse nosso mundo (o mundo real), nós trazemos as lembranças desse mundo anterior, que faz com que a gente saiba (embora confusamente) o que é o amor, a justiça, a religião, a paz, etc. A única forma de fazer com que a gente reconheça claramente tais conceitos é a linguagem, através do diálogo focado nos substantivos. Temos, então dois mundos: o mundo das ideias e o mundo real.

Isso também foi usado pela computação na orientação a objetos ? Sim. Quando você programa orientado a objetos, você primeiro tem que definir o que é o objeto e o que ele pode fazer. Isso, na programação de computadores, chama-se "criar uma classe".

Figura 35.1: A planta da casa está no "mundo das ideias", e a casa no "mundo real".

Uma classe equivale ao mundo das ideias de Platão. Uma classe não é usada diretamente pelo seu programa, ela é usada para criar o objeto que será usado pelo seu programa. Grosso modo, uma classe não ocupa espaço significativo na memória RAM do computador, mas o objeto sim. A figura 35.1 ilustra essas afirmações através

[3] Trecho do diálogo Menon.

[4] Isso quer dizer que o conceito/substantivo deve ser ter a primazia dentro de qualquer raciocínio.

de uma simples analogia: a classe equivale a planta de uma casa, enquanto que o objeto é a própria casa.

35.2 Na prática, o que é uma classe ?

Uma classe é um tipo definido pelo usuário [Stroustrup 2012, p 301]. Esse tipo é composto por tipos de dados primitivos[5] e por outros tipos definidos pelo usuário (outros objetos).

35.2.1 A primeira coisa a se fazer

Definir uma classe em Harbour é muito fácil, você deve incluir o arquivo **hbclass.ch** no seu programa. Portanto, a primeira linha do seu programa orientado a objetos deve começar com :

```
#include "hbclass.ch"
```

35.2.2 Em seguida definimos a interface

Agora vamos criar a interface da nossa classe, veja um exemplo a seguir :

```
CLASS MinhaClasse

         DATA m
         DATA n

         METHOD meuMetodo()
         METHOD meuMetodo2( cValor )

END CLASS
```

A interface é a parte da declaração da classe que pode ser acessada diretamente pelos seus usuários. Toda a nossa interface deve ficar entre CLASS e END CLASS.

35.2.3 Se necessário, crie a implementação

Após definir a interface, vamos verificar se algum método foi definido. Caso haja, você deve implementar esses métodos. A implementação é aquela parte da declaração da classe que seus usuários acessam apenas indiretamente, por meio da interface[Stroustrup 2012, p 301]. Na nossa classe de exemplo temos dois métodos definidos na interface: meuMetodo() e meuMetodo2().

```
METHOD meuMetodo() CLASS MinhaClasse

// Código do método
```

[5]Esses tipos de dados já foram abordados no começo do nosso estudo. Eles são os tipos que já vem definidos pela própria linguagem. No caso do Harbour são os tipos númericos, data, lógico, etc.

```
RETURN NIL

METHOD meuMetodo2( cValor ) CLASS MinhaClasse

// Código do método

RETURN cRetorno
```

Os métodos se parecem com funções, note que eles tem parâmetros e valores de retorno. A diferença é que essas "funções"estão subordinadas a classe. Lembra dos verbos e substantivos ?

Sintaticamente um método difere de uma função em apenas dois pontos:

1. no lugar de FUNCTION nós colocamos METHOD e
2. logo após nós acrescentamos o termo CLASS, seguido do nome da classe a qual o método pertence.

35.2.4 Agora você já pode criar seus objetos

Nesse ponto a classe foi criada e definida. Mas ela não é usada diretamente, lembra do nosso exemplo da planta da casa e da casa ? Pois bem, nós criamos a planta da casa. Tecnicamente, o que foi feito até agora foi apenas definir um tipo de dado. Para usar esse tipo de dado (construir a casa) faça no seu programa :

```
PROCEDURE MAIN

LOCAL oMeuObjeto := MinhaClasse():New() // Declaro a variável

? oMeuObjeto:MeuMetodo()


RETURN
```

Essa sintaxe é meio esquisita, mas ela não é uma invenção do Harbour. Todas as linguagens usam algo semelhante. Primeiro você deve criar uma variável para receber o objeto instanciado: no nosso caso essa variável chama-se oMeuObjeto. Para instanciar o objeto coloque o nome da classe seguido de parênteses, logo em seguida coloque a instrução New(). Essa instrução será vista mais adiante, por enquanto vamos nos conformar com ela sem saber maiores detalhes.

```
oMeuObjeto := MinhaClasse():New()
```

Dica 171

Para ajudar a fixar essa sintaxe, basta você se lembrar que uma classe é diferente de uma função, e se eu fizer somente assim :

```
oMeuObjeto := MinhaClasse()
```

O Harbour vai "pensar"que se tratar de uma função. Eu preciso de algo para diferenciar uma classe de uma função na hora de instanciar o objeto e esse "algo"é a palavra "New".

```
oMeuObjeto := MinhaClasse():New()
```

Nenhum membro da classe pode ser usado desvinculado do nome do objeto. Para usar, coloque o nome do objeto, seguido de dois pontos (":") e finalmente o nome do membro.

```
oMeuObjeto:MeuMetodo()
```

35.3 Características de um programa orientado a objetos

Vamos analisar detalhadamente as características de um programa orientado a objetos:

1. abstração;
2. encapsulamento;
3. polimorfismo;
4. herança;

35.3.1 Abstração

Essa característica pode causar confusão, afinal de contas a programação trata de coisas abstratas. Uma variável, uma rotina, um dado, enfim, tudo é abstrato. O que significa o termo abstração no âmbito da orientação a objetos ? De acordo com Goodrich e Tamassia:

> A noção de abstração significa decompor um sistema complicado em suas partes fundamentais e descrevê-las em uma linguagem simples e precisa. [Goodrich e Tamassia 2002, p. 65]

A tarefa de abstração é puramente intelectual e não depende de linguagem alguma. Contudo, apenas linguagens orientadas a objetos tem o poder de traduzir, de forma satisfatória, uma abstração em código.Isso é feito através de classes, onde os dados são definidos, e os métodos, onde as operações sobre os dados são implementadas.

Até agora, dentro do paradigma estruturado, já estudamos códigos como :

```
LOCAL nTotalDaNotaFiscal := 10
LOCAL dData := DATE()
```

No paradigma estruturado, nós podemos até criar uma variável numérica e chamá-la de nTotalDaNotaFiscal, mas somente a orientação a objetos permite abstrações que aproximem mais o código do mundo real. Uma abstração é a criação de tipos de dados que simulem o comportamento de objetos reais. Assim, podemos realizar operações sobre esses tipos de dados.

```
LOCAL nTemp := -20 // Temperatura mínima
LOCAL oGeladeira := Geladeira():New()

oGeladeira:Ligar()
DO WHILE oGeladeira:Temperatura() < nTemp
   oGeladeira:StandBy()
ENDDO
```

Mais adiante veremos que uma abstração é feita na interface da própria classe.

35.3.2 Encapsulamento

Nós já estudamos o que é encapsulamento mesmo sem saber. Encapsulamento "se refere à combinação de dados e rotinas para processar dados em uma única entidade" [Hyman 1995, p. 130].

Em um programa estruturado o encapsulamento é conseguido através de variáveis locais e estáticas. Nós também podemos usar essas variáveis para reduzir o famigerado acoplamento, que foi visto no capítulo sobre escopo de variáveis. O objetivo do encapsulamento é isolar (encapsular) os dados dentro da classe, de forma que eles fiquem protegidos de mudanças externas.

35.3.3 Polimorfismo

Em um programa estruturado eu preciso criar nomes de funções distintos uns dos outros. Nós vimos como contornar esse problema com funções estáticas, mas isso não resolve a maioria dos problemas. Eu sempre vou ter que disponibilizar a rotina para que ela possa ser usada em outros locais do programa, e isso me leva a nomes longos para evitar o choque de nomes. Um exemplo abaixo mostra três funções que processam o valor final de um pedido, de uma nota fiscal e de um orçamento.

```
ValorFinalDoPedido()
ValorFinalDaNotaFiscal()
ValorFinalDoOrcamento()
```

Já em um programa orientado a objetos, o método é subordinado a classe, logo eu posso ter métodos com o mesmo nome, mas em classes diferentes. Veja, a seguir, o equivalente orientado a objetos do fragmento anterior :

```
oPedido:ValorFinal()
oNotaFiscal:ValorFinal()
oOrcamento:ValorFinal()
```

Ou seja, eu posso ter classes com métodos que tem o mesmo nome mas se comportando de forma diferente. O nome dessa característica chama-se polimorfismo (POLI = Muitas, MORFOS = formas). Com isso eu posso limitar a quantidade de nomes criados e ter nomes repetidos. Essa característica torna os meus programas mais fáceis de serem lidos.

Dica 172

Para ajudar a entender o que é polimorfismo, pense no método acelerar() de um objeto avião e o método acelerar() de um objeto carro. Esses métodos tem o mesmo nome, mas estão subordinados a classes diferentes. Se você acompanhou a introdução desse capítulo verá que o verbo (método) sempre deve se subordinar ao substantivo/conceito (classe).

Dica 173

Como nós temos métodos com o mesmo nome em classes diferentes, é necessário que eu informe a classe a qual o método pertence[a]. Por isso, quando nós definimos o método, nós podemos informar a classe logo após.

```
METHOD ValorFinal()  CLASS CalculaPizza
```

[a]Em algumas linguagens isso não é necessário, pois os métodos ficam totalmente inseridos dentro do código da classe.

35.3.4 Herança

O conceito de herança, tão comum hoje nas linguagens de programação, não surgiu com a programação de computadores. Os pensadores antigos já costumavam agrupar os seres em categorias. Por volta de 300 a.C. o filósofo Aristóteles criou uma classificação primitiva dos seres que foi aperfeiçoada ao longo dos anos. O que tem isso a ver com a herança em programação orientada a objetos ?

Tem tudo a ver. Observe como os seres são classificados. Eles recebem inicialmente uma classificação inicial chamada “reino”. Temos o reino animal, vegetal e mineral. A partir do reino os seres recebem subclassificações que herdam as características das classes superiores. Por exemplo: tomemos um cachorro e um gato. Apesar de pertencerem a classes diferentes : canino e felino, eles partilham características comuns. Por exemplo, eles mamam quando pequenos, tem sangue quente (com temperatura constante) e tem quatro membros que usam para locomoção. Ou seja, eles “herdam” características de uma classe superior. Se uma nova raça de cachorro surgir hoje eu não vou precisar definir todas as suas características desde o início, pois eu sei que essa raça nova irá herdar as características dos caninos, que por sua vez, herdam características de outras classes superiores. Essa característica de transmitir comportamentos de classes superiores faz com a herança seja um dos aspectos mais interessantes da programação orientada a objeto. Se você criou uma classe chamada documento, você poderá criar a classe nota fiscal e herdar as características da classe documento. Desse forma, eu não precisarei mais recriar características que já existem em classes mais elevadas. Se no futuro surgir um tipo especial de documento eu não preciso definir esse tipo do zero, basta herdar as características da classe documento ou até mesmo de uma outra classe derivada de documento.

Para herdar uma classe de outra você precisa definir a herança na definição da classe. Para fazer isso, simplesmente coloque a palavra-chave INHERIT logo após o nome da classe, seguido da classe superior que você quer herdar.

```
CLASS NotaFiscal INHERIT Documento

... Definições da nossa classe.

END CLASS
```

35.4 Nossa primeira classe em Harbour

Vamos agora criar uma classe através de uma série de exemplos. Iremos evoluindo do paradigma estruturado até o paradigma orientado a objetos.

O objetivo aqui é criar um retângulo.

35.4.1 O retângulo inicial

O primeiro exemplo (listagem 35.1) não usará classes, ele irá simplesmente definir o retângulo usando o comando @ ... TO do Harbour.

Listagem 35.1: Versão inicial.
Fonte: codigos/oop01.prg

```
1
PROCEDURE Main                                                  2
                                                                3
    CLS                                                         4
    @ 10, 10 TO 20,20                                           5
    @ 30,10  SAY "Tecle algo para sair"                         6
    INKEY(0)                                                    7
                                                                8
RETURN                                                          9
```

Figura 35.2: Resultado do programa.

35.4.2 Criando uma função para desenhar o retângulo

Vamos agora encapsular o código que gera o retângulo em uma função. Ainda estamos no paradigma estruturado.

Listagem 35.2: Versão inicial usando o paradigma estruturado.
Fonte: codigos/oop02.prg

```
PROCEDURE Main

    CLS
    Retangulo( 10 , 10 , 20 , 20 )
    @ 30,10  SAY "Tecle algo para sair"
    INKEY(0)

RETURN

FUNCTION Retangulo( nLin01, nCol01 , nLin02 , nCol02 )

    @ nLin01, nCol01 TO nLin02, nCol02

RETURN NIL
```

Visivelmente nada mudou.

35.4.3 O retângulo usando o paradigma orientado a objetos

Primeiro cria uma classe que irá definir o retângulo.

Listagem 35.3: Versão inicial orientado a objetos.
Fonte: codigos/oop03.prg

```
/* Cabeçalho obrigatório */
#include "hbclass.ch"
/* Interface da classe */
CLASS TRetangulo

    DATA nLin01
    DATA nCol01
    DATA nLin02
    DATA nCol02

    METHOD Desenha()

END CLASS
/* Implementação da classe */
METHOD Desenha()

    @ ::nLin01, ::nCol01 TO ::nLin02, ::nCol02

RETURN Nil
/*
   Esse trecho a seguir não faz parte da classe.
   Ele é apenas um exemplo de uso.
   É aconselhável que ele fique em outro arquivo.
   Ele só está aqui para facilitar o entendimento.
*/
PROCEDURE MAIN

    LOCAL oRetangulo := TRetangulo():New()

    oRetangulo:nLin01 := 10
    oRetangulo:nCol01 := 10
    oRetangulo:nLin02 := 20
    oRetangulo:nCol02 := 20

    CLS
    oRetangulo:Desenha()
    @ 30,10  SAY "Tecle algo para sair"
    INKEY(0)

RETURN
```

Esse é o nosso retângulo inicial. Basicamente os parâmetros das funções se transformaram em atributos.

35.4.4 Valores default para os atributos

Da mesma forma que o parâmetro de uma função possui pode possuir um valor padrão, os atributos também podem ser definidos com seus respectivos valores já inicializados. Isso é feito através da cláusula INIT após a definição do atributo. A listagem 35.4 já nos trás essa mudança.

Listagem 35.4: Atribuindo valor inicial aos parâmetros.
Fonte: codigos/oop04.prg

```
    DATA nLin01 INIT 10
    DATA nCol01 INIT 10
    DATA nLin02 INIT 20
    DATA nCol02 INIT 20
```

Dessa forma, podemos omitir, caso seja possível, a etapa de definição de valores (Listagem 35.5).

Listagem 35.5: Nesse exemplo os parâmetros não foram definidos.
Fonte: codigos/oop04.prg

```
PROCEDURE MAIN

    LOCAL oRetangulo := TRetangulo():New()

    //oRetangulo:nLin01 := 10
    //oRetangulo:nCol01 := 10
    //oRetangulo:nLin02 := 20
    //oRetangulo:nCol02 := 20

    CLS
    oRetangulo:Desenha()
    @ 30,10  SAY "Tecle algo para sair"
    INKEY(0)

RETURN
```

35.5 Evoluindo a nossa classe TRectangulo para proteger os atributos

Agora veremos como proteger os atributos de uma classe. A nossa classe funciona perfeitamente, mas precisamos proteger os atributos de valores indesejáveis. No nosso código atual, se alguém informar o seguinte valor para o atributo:

```
oRetangulo:nLin01 := "Uma string em vez de número"
```

irá gerar um erro de execução, afinal de contas, a classe TRetangulo espera que o atributo nLin tenha um valor numérico.

35.5.1 Protegendo um atributo

Para se proteger um atributo de uma classe, a primeira coisa a se fazer é "dizer"para a classe que um determinado atributo não pode sofrer alterações externas. Isso é feito através dos especificadores de acesso.

Especificadores de acesso

Por padrão, todo atributo de uma classe pode ser alterado livremente através da sintaxe abaixo.

```
objeto:atributo := valor
```

Contudo, como já vimos, essa forma de atribuição pode levar a erros de execução. Pensando em proteger o atributo de uma alteração indesejável, a POO criou os seguintes especificadores de acesso:

1. EXPORTED : o atributo pode ser alterado livremente. Esse é o padrão.
2. HIDDEN : o atributo só pode ser alterado internamente e somente pela classe criadora do respectivo atributo.
3. PROTECTED : muito semelhante ao HIDDEN, o atributo só pode ser alterado internamente. Contudo as classes "filhas"dessa classe **também podem alterar o atributo**.

Vamos estudar caso a caso, primeiramente iremos estudar o especificador de acesso HIDDEN, e só depois de estudarmos a herança, nós veremos como se comporta o especificador PROTECTED.

Modificando as coordenadas do retângulo para HIDDEN

Vamos fazer um pequeno teste. No código abaixo nós incluímos os especificadores de acesso EXPORTED e HIDDEN nos atributos. Repare como isso é feito na listagem 35.6:

Listagem 35.6: Nesse exemplo os parâmetros não foram definidos.
Fonte: codigos/oop05erro.prg

```
CLASS TRetangulo                                                    1
                                                                    2
    HIDDEN: // Especificador HIDDEN                                 3
    DATA nLin01 INIT 10                                             4
    DATA nCol01 INIT 10                                             5
    DATA nLin02 INIT 20                                             6
    DATA nCol02 INIT 20                                             7
                                                                    8
    EXPORTED: // Especificador EXPORTED                             9
    METHOD Desenha()                                                10
                                                                    11
END CLASS                                                           12
```

Note que após cada especificador de acesso, o modo que foi invocado por esse especificador de acesso a membro se aplica até o próximo especificador de acesso a membro ou até [...] o término [Deitel e Deitel 2001, p. 411]. No caso do Harbour, sempre coloque ":"após o nome do especificador, conforme a listagem 35.6 acima.

O programa irá compilar perfeitamente, mas quando tentarmos atribuir valores diretamente nos atributos um erro de execução será gerado:

.:Resultado:.

```
Error BASE/41  Scope violation (hidden): TRETANGULO:_NLIN01
Called from TRETANGULO:_NLIN01(0)
Called from MAIN(32)
```

Esse erro aconteceu porque os atributos agora são acessíveis somente através da classe. Para resolver esse problema, teremos que criar os métodos que são os responsáveis pela atribuição de valores aos atributos.

35.5.2 Getters e Setters : uma forma de proteção contra alterações indesejáveis

Getters e Setters são métodos especializados em controlar o acesso aos atributos HIDDEN e PROTECTED de uma classe. Essa ideia não é específica do Harbour, mas um termo bastante conhecido da POO. Como veremos nas linhas a seguir, trata-se de uma ideia bastante simples.

Vamos partir do problema da seção anterior. Como faremos para gravar valores nos atributos nLin01, nLin02, nCol01 e nCol02 ? Simples, crie um método para cada uma delas. No trecho abaixo (listagem 35.7) criamos um método Set para o atributo nLin01. **Você precisa fazer isso para todos os atributos que terão seus valores modificados externamente**.

Listagem 35.7: Método Set para o atributo nLin01.
Fonte: codigos/oop06.prg

```
METHOD SetLin01( nLin01 )                                    1
                                                             2
    ::nLin01 := hb_DefaultValue( nLin01 , 10 )               3
                                                             4
RETURN Nil                                                   5
```

Note que, dentro do método, nós atribuímos um valor padrão caso o tipo de dado seja incorreto. Lembre-se que a função hb_DefaultValue() atribui um valor padrão também se o tipo de dado for diferente do tipo de dado passado como padrão. Quando você for passar o valor para o atributo faça conforme a listagem 35.8 :

Listagem 35.8: Atribuindo valor para os atributos.
Fonte: codigos/oop06.prg

```
    oRetangulo:setLin01( 30 )                                1
    oRetangulo:setCol01( 25 )                                2
    oRetangulo:setLin02( 70 )                                3
    oRetangulo:setCol02( 85 )                                4
```

Quando for criar um Setter, obedeça a nomenclatura preconizada pela POO :

```
Nome do Setter = Set + Nome do Atributo
```

Por exemplo, se o nome do atributo for nLin01, o nome do setter pode ser SetLin01() (omitindo o prefixo "n").

Getters

O mesmo raciocínio também se aplica quando queremos ler um valor de um atributo HIDDEN ou PRIVATE de fora da classe. Só que, em vez de criarmos um Setter, devemos criar um Getter. No nosso exemplo da classe TRetangulo() poderíamos fazer assim:

Listagem 35.9: Getters em TRetangulo()
Fonte: codigos/oop07.prg

```
    METHOD GetLin01()                                                    1
    METHOD GetCol01()                                                    2
    METHOD GetLin02()                                                    3
    METHOD GetCol02()                                                    4
```

Um exemplo, pode ser visto na listagem 35.10, onde definimos um Getter para o atributo nLin01.

Listagem 35.10: Um modelo de um Getter
Fonte: codigos/oop07.prg

```
METHOD GetLin01()                                                        1
                                                                         2
RETURN ::nLin01                                                          3
```

Um exemplo de uso:

Listagem 35.11: Usando o Getter para acessar o valor
Fonte: codigos/oop07.prg

```
    ? "Coordenadas : " ,;                                                1
            oRetangulo:GetLin01(),;                                      2
            oRetangulo:GetCol01(),;                                      3
            oRetangulo:GetLin02(),;                                      4
            oRetangulo:GetLin02()                                        5
```

35.5.3 Getters e Setters : simplificando a sintaxe

De acordo com Kresin

> SETGET é usado para os dados a serem calculados, ele permite que você use o nome do método como o nome da variável a ser lida / escrita nos dados. Por exemplo, se sua classe é uma variável, cujo valor não é desejável instalar diretamente, uma vez que deve ter um determinado formato ou deve estar apenas em um determinado intervalo. Você define esta variável como HIDDEN para excluir sua mudança direta, e define SETGET o método que será usado para definir o valor de uma variável, ele verifica o valor passado, formata-o como deveria ser e assim por diante. Agora, para definir o valor de uma variável, você escreve: obj: <methodName>: = <value>. [Kresin 2016] (Acessado em 28-Out-2021)

Listagem 35.12: Getters e Setters: sintaxe simplificada
Fonte: codigos/oop08.prg

```
METHOD GetLin01()
METHOD GetCol01()
METHOD GetLin02()
METHOD GetCol02()
```

35.6 Conclusão

Nas próximas seções nós iremos abordar a orientação a objetos[6] através de um exemplo mais prático.

[6] Alguém já me disse que o termo deveria ser “orientação a classes”, já que a classe deve ser definida antes do objeto.

36 Programação orientada a objetos

Lembre-se, as coisas
consomem tempo.

Piet Hein

36.1 Pensando orientado a objetos

Criar uma classe no Harbour não é difícil. O mais difícil é “pensar orientado a objetos”. Para facilitar as coisa, vamos desenvolver um programa que calcula o custo de uma pizza [1]. Primeiramente vamos entender o problema:

“Um comerciante deseja que você crie um programa simples para calcular o preço de uma pizza. Uma pizza é composta por uma base comum que custa $ 10,00 acrescido da cobertura (cujo preço deve ser informado pelo vendedor). Cada cliente pode exigir uma pizza com mais de uma cobertura do mesmo tipo.”

Temos as seguintes observações preliminares :

1. uma pizza é composta por base e cobertura;
2. uma pizza pode ter mais de uma cobertura;
3. as coberturas são do mesmo tipo;
4. o custo da base é $ 10,00;
5. o custo da cobertura deve ser informado pelo vendedor ao sistema.

Vamos evoluir o nosso problema, mas primeiro iremos mostrar uma solução baseada no paradigma estruturado.

36.1.1 Versão inicial do programa

A versão a seguir, adaptada de [Hyman 1995, p. 125], adota o paradigma estruturado para resolvermos esse problema.

Listagem 36.1: Versão 1 do programa que calcula o preço da pizza
Fonte: codigos/pizza01.prg

```
#define BASE 10.00 // Custo da base da pizza
PROCEDURE Main
LOCAL nNumCob // Número de cobertura
LOCAL nValCob // Preço por cobertura
LOCAL nTotal // Preço final

    // Descobre o número de coberturas
    INPUT "Informe a quantidade de coberturas : " TO nNumCob
    INPUT "Informe o preço da cobertura : " TO nValCob

    nValFinal := BASE + ( nNumCob * nValCob )

    ? "A pizza custa $ ", nValFinal

RETURN
```

[1] Esse projeto foi adaptado do livro “Borland C++ para leigos”, de Michael Hyman.

.:Resultado:.

```
Informe a quantidade de coberturas : 2
Informe o preço da cobertura : 10
A pizza custa $          30.00
```

36.1.2 Versão 2 do programa que calcula o preço da pizza

Analisando o nosso problema, podemos criar 3 rotinas, de acordo com o que aprendemos com a programação top down. As rotinas abordam os seguintes problemas :

1. **Obter** o número de coberturas;
2. **Calcular** o preço;
3. **Imprimir** o preço;

O programa da listagem a seguir é uma evolução do programa anterior.

Listagem 36.2: Versão 2 do programa que calcula o preço da pizza
Fonte: codigos/pizza02.prg

```
#define BASE 10.00 // Custo da base da pizza
PROCEDURE Main
LOCAL nNumCob // Número de cobertura
LOCAL nValCob // Preço por cobertura
LOCAL nTotal // Preço final

    // Descobre o número de coberturas
    nNumCob := ObterQtdCoberturas()
    nValCob := ObterValCoberturas()

    nValFinal := ValorFinal( nNumCob , nValCob )

    ? "A pizza custa $ ", nValFinal

RETURN
/**
Obtem a quantidade de coberturas
*/
FUNCTION ObterQtdCoberturas()
LOCAL nQtd

INPUT "Informe a quantidade de coberturas : " TO nQtd

RETURN nQtd
/**
Obtem o valor das coberturas
*/
FUNCTION ObterValCoberturas()
LOCAL nVal

```

```
INPUT "Informe o preço da cobertura : " TO nVal

RETURN nVal
/**
Imprime o valor final
*/
FUNCTION ValorFinal( nNumCob , nValCob )

RETURN BASE + ( nNumCob * nValCob )
```

.:Resultado:.

```
Informe a quantidade de coberturas : 2
Informe o preço da cobertura : 10
A pizza custa $          30.00
```

O programa apresenta a mesma saída da sua versão anterior, e talvez tal refino não seja necessário, afinal de contas, parece uma complicação desnecessária. As técnicas que ensinaremos podem parecer um exagero, e talvez até seja quando o código é simples demais. A questão é que um código simples sofre mudanças com o passar dos anos e acaba se tornando complexo e de difícil manutenção. O paradigma orientado a objetos nos ajuda a criar programas cuja manutenção menos traumática.

O criador da linguagem C++ descreve a maior parte do conteúdo sobre programação como "uma discussão sobre como expressamos um programa como um conjunto de partes cooperativas e como elas podem compartilhar e trocar dados"[Stroustrup 2012]. Esperamos, portanto, que você entenda as melhorias no nosso pequeno projeto por esse prisma. No final, esperamos lhe convencer que a versão final estará mais fácil de se manter, pois os problemas foram isolados dentro das suas respectivas rotinas.

36.1.3 Versão 3 do programa que calcula o preço da pizza

O nosso programa que calcula o preço final da pizza funciona perfeitamente. Porém, passados algumas semanas o seu cliente volta a lhe ligar. Ele quer incrementar o seu programa mais um pouco, de modo que ele passe a **gravar** esses dados em um banco de dados. O seu cliente não quer relatórios, pois ele usará a planilha Microsoft Excel para visualizar o arquivo DBF que será criado[2]. Você topa o desafio, implementa um arquivo chamado pizza.dbf e aproveita para criar outras funções extras. O resultado está listado a seguir:

Listagem 36.3: Versão 3 do programa que calcula o preço da pizza
Fonte: codigos/pizza03.prg

```
#define BASE 10.00 // Custo da base da pizza
#define ESTRUTURA_DBF { { "NUMCOB" , "N" , 3 , 0 },;
                        { "VALCOB" , "N" , 6 , 2 },;
                        { "TOTAL" , "N" , 6 , 2 } }
PROCEDURE Main
LOCAL nNumCob // Número de cobertura
LOCAL nValCob // Preço por cobertura
```

[2] O Microsoft Excel abre diretamente os arquivos DBFs e os transforma em uma planilha. O LibreOffice Calc também faz isso.

```
LOCAL nTotal // Preço final
    // Abertura de arquivo
    IF .NOT. AbreArquivo()
       ? "Problemas na abertura do arquivo"
       ? "Saindo do sistema"
       QUIT
    ENDIF
    // Descobre o número de coberturas
    nNumCob := ObterQtdCoberturas()
    nValCob := ObterValCoberturas()
    // Cálculo do valor final
    nValFinal := ValorFinal( nNumCob , nValCob )
    // Exibir total
    MostraTotal( nValFinal )
    // Gravo os dados
    Grava( nNumCob , nValCob , nValFinal )


RETURN
/**
Obtem a quantidade de coberturas
*/
FUNCTION ObterQtdCoberturas()
LOCAL nQtd

INPUT "Informe a quantidade de coberturas : " TO nQtd

RETURN nQtd
/**
Obtem o valor das coberturas
*/
FUNCTION ObterValCoberturas()
LOCAL nVal

INPUT "Informe o preço da cobertura : " TO nVal

RETURN nVal
/**
Imprime o valor final
*/
FUNCTION ValorFinal( nNumCob , nValCob )

RETURN BASE + ( nNumCob * nValCob )

/**
Abre o arquivo
*/
FUNCTION AbreArquivo()
LOCAL lSucesso

      IF .NOT. FILE( "pizza.dbf" )
```

```
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )
      ENDIF
      USE pizza
      IF USED()
         lSucesso := .t.
      ELSE
         lSucesso := .f.
      ENDIF

RETURN lSucesso

/**
Exibe o total
*/
PROCEDURE MostraTotal()

   ? "A pizza custa $ ", nValFinal

RETURN

/**
Grava o resultado
*/
PROCEDURE Grava( nNumCob , nValCob , nValFinal )

   APPEND BLANK
   REPLACE NUMCOB WITH nNumCob
   REPLACE VALCOB WITH nValCob
   REPLACE TOTAL WITH nValFinal

RETURN
```

Vamos tecer alguns comentários sobre a nova versão do nosso programa.

1. A estrutura do arquivo pizza.dbf (que é uma matriz) está definido como uma constante manifesta na linha 2 (com quebras de linha até a linha 4).
2. A abertura do arquivo foi colocada em uma rotina chamada AbreArquivo(). Essa rotina está definida entre as linhas 55 e 68 e a chamada dela é feita na linha 10. Caso o arquivo não possa ser aberto, a rotina retorna falso e o programa é finalizado com uma mensagem de erro (linha 13).
3. A impressão do preço, que antes era feito na rotina principal, foi transferido para a rotina MostraTotal().
4. Finalmente temos a rotina Grava(), que tem a função de gravar as variáveis no arquivo pizza.dbf.

36.1.4 Versão 4 do programa que calcula o preço da pizza

Não demora muito e o seu cliente lhe liga novamente e pede novas alterações. Ele quer agora gravar no arquivo pizza.dbf o nome da pessoa que pediu a pizza, o

endereço dessa pessoa e o telefone dela. Você concorda e o programa recebe novas alterações.

Você continua criando novas rotinas, as novas são:

1. **Obter** o nome do cliente;
2. **Obter** o endereço do cliente;
3. **Obter** o telefone do cliente;
4. **Gravar** o nome do cliente;
5. **Gravar** o endereço do cliente;
6. **Gravar** o telefone do cliente;

Note que nós estamos, desde as primeiras versões do programa, colocando os verbos em negrito. Essa prática irá nos ajudar quando nós formos contrastar com o paradigma orientado a objetos (cuja ênfase é no substantivo).

A versão do nosso novo programa está listado logo a seguir. Um detalhe importante: não esqueça de excluir o arquivo pizza.dbf para que o sistema recrie-o novamente com os campos adicionais.

No final nós comentaremos o que foi feito.

Listagem 36.4: Versão 4 do programa que calcula o preço da pizza
Fonte: codigos/pizza04.prg

```
#define BASE 10.00 // Custo da base da pizza
#define ESTRUTURA_DBF { { "NUMCOB" , "N" , 3 , 0 },;
                        { "VALCOB" , "N" , 6 , 2 },;
                        { "TOTAL" , "N" , 6 , 2 },;
                        { "NOME" , "C" , 30 , 0 },;
                        { "TELEFONE" , "C" , 30 , 0 },;
                        { "ENDERECO" , "C" , 30 , 0 } }
PROCEDURE Main
LOCAL nNumCob // Número de coberturas
LOCAL nValCob // Preço por coberturas
LOCAL nTotal // Preço final
LOCAL cNome // Nome do cliente
LOCAL cTelefone // Telefone do cliente
LOCAL cEndereco // Endereço do cliente

    // Abertura de arquivo
    IF .NOT. AbreArquivo()
       ? "Problemas na abertura do arquivo"
       ? "Saindo do sistema"
       QUIT
    ENDIF
    // Dados do cliente
    cNome := ObterCliente()
    cTelefone := ObterTelefone()
    cEndereco := ObterEndereco()
    // Descobre o número de coberturas
```

```
    nNumCob := ObterQtdCoberturas()
    nValCob := ObterValCoberturas()
    // Cálculo do valor final
    nValFinal := ValorFinal( nNumCob , nValCob )
    // Exibir total
    MostraTotal( nValFinal )
    // Gravo os dados
    Grava( nNumCob , nValCob , nValFinal ,;
           cNome, cTelefone, cEndereco )


RETURN
/**
Obtem a quantidade de coberturas
*/
FUNCTION ObterQtdCoberturas()
LOCAL nQtd

INPUT "Informe a quantidade de coberturas : " TO nQtd

RETURN nQtd
/**
Obtem o valor das coberturas
*/
FUNCTION ObterValCoberturas()
LOCAL nVal

INPUT "Informe o preço da cobertura : " TO nVal

RETURN nVal
/**
Imprime o valor final
*/
FUNCTION ValorFinal( nNumCob , nValCob )

RETURN BASE + ( nNumCob * nValCob )

/**
Obtem o nome do cliente
*/
FUNCTION ObterCliente()
LOCAL cVal

ACCEPT "Informe o nome do cliente : " TO cVal

RETURN cVal
/**
Obtem o telefone do cliente
*/
FUNCTION ObterTelefone()
LOCAL cVal
```

```
                                                                                78
ACCEPT "Informe o telefone do cliente : " TO cVal                               79
                                                                                80
RETURN cVal                                                                     81
/**                                                                             82
Obtem o endereço do cliente                                                     83
*/                                                                              84
FUNCTION ObterEndereco()                                                        85
LOCAL cVal                                                                      86
                                                                                87
ACCEPT "Informe o endereço do cliente : " TO cVal                               88
                                                                                89
RETURN cVal                                                                     90
                                                                                91
                                                                                92
/**                                                                             93
Abre o arquivo                                                                  94
*/                                                                              95
FUNCTION AbreArquivo()                                                          96
LOCAL lSucesso                                                                  97
                                                                                98
      IF .NOT. FILE( "pizza.dbf" )                                              99
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )                                100
      ENDIF                                                                     101
      USE pizza                                                                 102
      IF USED()                                                                 103
         lSucesso := .t.                                                        104
      ELSE                                                                      105
         lSucesso := .f.                                                        106
      ENDIF                                                                     107
                                                                                108
RETURN lSucesso                                                                 109
                                                                                110
/**                                                                             111
Exibe o total                                                                   112
*/                                                                              113
PROCEDURE MostraTotal()                                                         114
                                                                                115
   ? "A pizza custa $ ", nValFinal                                              116
                                                                                117
RETURN                                                                          118
                                                                                119
/**                                                                             120
Grava o resultado                                                               121
*/                                                                              122
PROCEDURE Grava( nNumCob , nValCob , nValFinal ,;                               123
                 cNome, cTelefone, cEndereco )                                  124
                                                                                125
   APPEND BLANK                                                                 126
   REPLACE NUMCOB WITH nNumCob                                                  127
   REPLACE VALCOB WITH nValCob                                                  128
```

```
   REPLACE TOTAL WITH nValFinal                                    129
   REPLACE NOME WITH cNome                                         130
   REPLACE TELEFONE WITH cTelefone                                 131
   REPLACE ENDERECO WITH cEndereco                                 132
                                                                   133
RETURN                                                             134
```

As alterações efetuadas

1. nós alteramos o array que define o pizza.dbf (linha 2 a 7)
2. as rotinas extras foram criadas: Obter o nome, o endereço e o telefone do cliente. Note que nessas novas rotinas foi utilizada o comando ACCEPT, porque os dados são caracteres.
3. a rotina Grava() foi alterada para poder gravar também o nome, o telefone e o endereço.

Até agora não temos novidade alguma. O paradigma que nós usamos para desenvolver esse programa foi amplamente discutido durante todo o livro. Na verdade a grande maioria dos curso de introdução a programação abordam somente o paradigma estruturado [3]. Na próxima seção iremos iniciar a abordagem do paradigma orientado a objetos.

36.2 A ideia por trás do paradigma orientado a objetos

Hyman esclarece que

> a ideia básica da programação orientada a objetos é simples. Os dados e as rotinas para processar os dados são combinados em uma única entidade chamada de classe. Se você deseja acessar os dados da classe, use as rotinas da classe [Hyman 1995, p. 127].

No nosso caso os dados são :

1. número de coberturas (nNumCob);
2. valor por cobertura (nValCob);
3. valor final (nTotal);
4. nome do cliente (cCliente);
5. telefone do cliente (cTelefone);
6. endereço do cliente (cEndereco).

As rotinas são :

[3] Mesmo cursos introdutórios que usam linguagens totalmente orientadas a objeto, como Java, abordam somente o paradigma estruturado. Lembre-se que você pode usar uma linguagem orientada a objeto mas não usar o paradigma orientado a objetos, por isso existem tantos cursos de introdução usando linguagens orientadas a objetos.

1. ObterQtdCoberturas();
2. ObterValCoberturas();
3. ValorFinal(nNumCob, nValCob);
4. MostraTotal(nValFinal);
5. ObterCliente();
6. ObterEndereco();
7. ObterTelefone();
8. Grava(nNumCob, nValCob, nValFinal, cNome, cTelefone, cEndereco);

Se nós fossemos aplicar o raciocínio de uma aplicação orientada a objetos, como você acha que seriam essas rotinas ?

Na verdade, essas rotinas não iriam mudar muito. Para que o problema abordado se transforme em um programa orientado a objetos nós, primeiramente, temos que detectar as classes (os substantivos principais do nosso problema) e agrupar as rotinas dentro dessas classes[4]. Vamos rever o problema novamente :

A versão inicial do problema :

"Um comerciante deseja que você crie um programa simples para calcular o preço de uma pizza. Uma pizza é composta por uma base comum que custa $ 10,00 acrescido da cobertura (cujo preço deve ser informado pelo vendedor). Cada cliente pode exigir uma pizza com mais de uma cobertura do mesmo tipo."

As alterações solicitadas :

"Posteriormente, o comerciante solicitou que o programa armazenasse o pedido através do nome, do endereço e do telefone do cliente. Esses dados devem ficar armazenados em um arquivo para posterior consulta através de planilhas eletrônicas."

Note que nós não temos apenas um problema complexo, mas dois problemas simples. O primeiro deles requer que você calcule o preço da pizza (versão inicial) e o segundo problema envolve um pedido de venda. Temos duas classes : a classe **Calculadora de Pizza** e a classe **Pedido**[5]. Vamos prosseguir com o nosso raciocínio: agora que nós temos duas classes definidas, vamos dividir os dados e as rotinas de cada classe. Note que as rotinas são exatamente as mesmas do nosso programa anterior, mas elas agora foram agrupadas em classes.

Temos a classe **CalculadoraDePizza**

1. Dados
    a) número de coberturas (nNumCob);
    b) valor por cobertura (nValCob);
    c) valor final (nTotal).

[4]Essa nossa abordagem é puramente didática, para se especializar no raciocínio orientado a objetos você deve prosseguir estudando assuntos relacionados a Engenharia de software, como padrões de projetos e UML.

[5]Essa não é a única forma de resolver o nosso problema, alguns programadores podem optar por criar uma classe única chamada **Pizza**

2. Rotinas
    a) ObterQtdCoberturas();
    b) ObterValCoberturas();
    c) ValorFinal(nNumCob, nValCob);
    d) MostraTotal(nValFinal).

E agora a classe **Pedido**

1. Dados
    a) nome do cliente (cCliente);
    b) telefone do cliente (cTelefone);
    c) endereço do cliente (cEndereco).
2. Rotinas
    a) ObterCliente();
    b) ObterEndereco();
    c) ObterTelefone();
    d) Grava(nNumCob, nValCob, nValFinal, cNome, cTelefone, cEndereco);

Vamos desenvolver o programa novamente, mas agora usando a orientação a objetos. Primeiramente iremos criar a versão que calcula o preço da pizza, e em seguida iremos criar o pedido de venda.

36.2.1 Versão inicial do programa orientado a objetos

Inclua o cabeçalho

```
#include "hbclass.ch"
```

Vou manter a constante que define o valor do custo da base da pizza.

```
#define BASE 10.00 // Custo da base da pizza
```

Agora vamos criar o “esqueleto” da nossa classe :

```
CLASS CalculaPizza

END CLASS
```

Toda a nossa classe deverá ficar entre CLASS e END CLASS.

Vamos agora definir as variáveis da nossa classe. Para definir as variáveis use a palavra chave DATA ou VAR (elas são equivalentes). Iremos usar a palavra DATA.

```
CLASS CalculaPizza

   DATA nNumCob // Número de cobertura
   DATA nValCob // Preço por cobertura
   DATA nTotal // Preço final

END CLASS
```

O seu programa deve se parecer com a listagem abaixo:

```
#include "hbclass.ch"                                                    1
#define BASE 10.00 // Custo da base da pizza                             2
CLASS CalculaPizza                                                       3
                                                                         4
   DATA nNumCob // Número de cobertura                                   5
   DATA nValCob // Preço por cobertura                                   6
   DATA nTotal // Preço final                                            7
                                                                         8
END CLASS                                                                9
```

Vamos prosseguir agora trazendo as funções para dentro da nossa classe. Nesse caso nós colocamos dentro da classe apenas o cabeçalho da função. Quando você programa orientado a objetos, as “funções” da classe chamam-se “métodos”. Por isso passaremos a usar esse termo.

Primeiramente vamos incluir o cabeçalho dos nossos métodos. O nosso programa ficará assim :

```
#include "hbclass.ch"                                                    1
#define BASE 10.00 // Custo da base da pizza                             2
CLASS CalculaPizza                                                       3
                                                                         4
   DATA nNumCob // Número de cobertura                                   5
   DATA nValCob // Preço por cobertura                                   6
   DATA nTotal // Preço final                                            7
                                                                         8
   METHOD ObterQtdCoberturas()                                           9
   METHOD ObterValCoberturas()                                           10
   METHOD ValorFinal()                                                   11
                                                                         12
END CLASS                                                                13
```

Agora vamos definir os nossos métodos. Vamos ilustrar a diferença entre um método e uma função a seguir com a rotina ObterQtdCoberturas().

Primeiramente veja a função:

```
FUNCTION ObterQtdCoberturas()
LOCAL nQtd

   INPUT "Informe a quantidade de coberturas : " TO nQtd

RETURN nQtd
```

Agora o método :

```
METHOD ObterQtdCoberturas() CLASS CalculaPizza

   INPUT "Informe a quantidade de coberturas : " TO ::nNumCob

RETURN NIL
```

Note que as mudanças foram poucas,

1. no lugar de FUNCTION nós colocamos METHOD e logo após nós acrescentamos o termo CLASS CalculaPizza.
2. nós não usamos mais a variável LOCAL nQtd, mas a variável definida na seção DATA nNumCob
3. a função retorna o valor nQtd, mas o método não, pois a quantidade de cobertura (::nNumCob) é compartilhada entre os demais métodos da classe.

É bom reforçar que não precisamos mais retornar valor algum desse método, por isso o return do método é NIL. Esse conceito é importante, pois essas variáveis são visíveis e compartilhadas por todos os métodos do nosso objeto.

Dica 174

Para diferenciar uma variável do objeto de uma variável comum nós colocamos o prefixo :: antes da variável do objeto.
Uma variável de objeto:

```
DATA nNumCob
```

Dentro do método ela é referenciada assim :

```
::nNumCob
```

Os demais métodos devem ser criados usando esse mesmo procedimento.

36.2.2 Comparando as duas versões da função MAIN

Vamor agora comparar a versão da função MAIN desenvolvida usando o paradigma estruturado com a função MAIN que usa as classes criadas. A versão abaixo usa funções.

```
#define BASE 10.00 // Custo da base da pizza
PROCEDURE Main
LOCAL nNumCob // Número de cobertura
LOCAL nValCob // Preço por cobertura
LOCAL nTotal // Preço final

    // Descobre o número de coberturas
```

```
    nNumCob := ObterQtdCoberturas()
    nValCob := ObterValCoberturas()

    nValFinal := ValorFinal( nNumCob , nValCob )

    ? "A pizza custa $ ", nValFinal

RETURN
```

Agora a versão orientada a objetos. Por questões didáticas, nós vamos colocar a nossa rotina no mesmo arquivo que definiu a classe. Veja a nossa rotina :

```
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()

    ? "A pizza custa $ ", oPizza:ValorFinal()

RETURN
```

O fato mais importante é a ausência de variáveis para intercambiar dados entre as funções. Agora as variáveis são internas a classe. Outro fato digno de nota é que nós temos que criar o objeto (o termo técnico é instanciar). Para instanciar o objeto nós fazemos assim :

```
oPizza := CalculaPizza():New()
```

Depois de instanciada, a variável oPizza pode ser usada com os métodos definidos. Sempre obedecendo a sintaxe : Objeto:Método() ou Objeto:VariavelDoObjeto.

Apesar dos termos novos, o programa final ficou mais simples porque as variáveis foram escondidas (o termo técnico é “encapsuladas”) dentro dos métodos. Logo abaixo, veja a listagem completa do nosso programa.

Listagem 36.5: Versão 1 do programa que calcula o preço da pizza (orientado a objetos)
Fonte: codigos/pizza05.prg

```
#include "hbclass.ch"
#define BASE 10.00 // Custo da base da pizza
CLASS CalculaPizza

   DATA nNumCob // Número de cobertura
   DATA nValCob // Preço por cobertura
   DATA nTotal // Preço final

   METHOD ObterQtdCoberturas()
   METHOD ObterValCoberturas()
   METHOD ValorFinal()

```

```
END CLASS

/**
Obtem a quantidade de coberturas
*/
METHOD ObterQtdCoberturas() CLASS CalculaPizza


INPUT "Informe a quantidade de coberturas : " TO ::nNumCob

RETURN NIL
/**
Obtem o valor das coberturas
*/
METHOD ObterValCoberturas()  CLASS CalculaPizza


INPUT "Informe o preço da cobertura : " TO ::nValCob

RETURN NIL
/**
Imprime o valor final
*/
METHOD ValorFinal()  CLASS CalculaPizza

  ::nTotal := BASE + ( ::nNumCob * ::nValCob )

RETURN ::nTotal
/*
Função Main
*/
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    // Descobre o número de coberturas
    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()

    ? "A pizza custa $ ", oPizza:ValorFinal()

RETURN
```

A execução é exatamente igual a versão estruturada, a diferença está somente no código.

.:Resultado:.

```
Informe a quantidade de coberturas : 2
Informe o preço da cobertura : 10
A pizza custa $          30.00
```

Será que esse código é suficiente para lhe convencer a abandonar o paradigma estruturado e usar o paradigma orientado a objetos ?

Não, porque mesmo usando o paradigma estruturado eu posso ter as minhas variáveis isoladas, basta usar as variáveis estáticas externas. Veja a seguir um código estruturado com variáveis estáticas externas.

Listagem 36.6: Versão estruturada usando variáveis estáticas externas
Fonte: codigos/pizza06.prg

```
#define BASE 10.00 // Custo da base da pizza
STATIC nNumCob // Número de cobertura
STATIC nValCob // Preço por cobertura
STATIC nTotal // Preço final
PROCEDURE Main

    // Descobre o número de coberturas
    ObterQtdCoberturas()
    ObterValCoberturas()
    ? "A pizza custa $ ", ValorFinal()

RETURN
/**
Obtem a quantidade de coberturas
*/
FUNCTION ObterQtdCoberturas()

INPUT "Informe a quantidade de coberturas : " TO nNumCob

RETURN NIL
/**
Obtem o valor das coberturas
*/
FUNCTION ObterValCoberturas()

INPUT "Informe o preço da cobertura : " TO nValCob

RETURN NIL
/**
Imprime o valor final
*/
FUNCTION ValorFinal()
RETURN BASE + ( nNumCob * nValCob )
```

Portanto, até agora, a orientação a objetos não se justifica. Mas antes de tomarmos essa conclusão precipitadamente

36.2.3 Implementando a classe de pedidos

Vamos voltar aos requisitos do nosso sistema. Lembra quando o cliente pediu que você gravasse o pedido em um arquivo ? Vamos rever o que ele solicitou.

"O comerciante solicitou que o programa armazenasse o pedido através do nome, do endereço e do telefone do cliente. Esses dados devem ficar armazenados em um arquivo para posterior consulta através de planilhas eletrônicas."

Nós fizemos esse programa usando o paradigma estruturado, agora vamos criar uma classe de pedido para fazermos uma implementação orientada a objetos. Primeiramente vamos raciocinar juntos: o pedido possui os seguintes atributos :

1. Dados
    a) nome do cliente (cCliente);
    b) telefone do cliente (cTelefone);
    c) endereço do cliente (cEndereco).
2. Rotinas
    a) ObterCliente();
    b) ObterEndereco();
    c) ObterTelefone();
    d) Grava(nNumCob, nValCob, nValFinal, cNome, cTelefone, cEndereco);

Se você observar bem, verá que faz todo sentido criar a classe Pedido em um nível superior e depois alterar a classe CalculaPizza para herdar os métodos da classe Pedido. Assim, quando eu for usar a classe CalculaPizza, eu não preciso modificar nada nela, pois o pedido está definido em uma classe superior.

Vamos agora criar a classe pedido. Veja na listagem a seguir que ela não é muito diferente da classe CalculaPizza. Note que nós não implementamos a gravação em DBFs.

Listagem 36.7: Classe Pedido
Fonte: codigos/pedido.prg

```
#include "hbclass.ch"                                                    1
CLASS Pedido                                                             2
                                                                         3
   DATA cNome // Nome do cliente                                         4
   DATA cTelefone // Endereço do cliente                                 5
   DATA cEndereco // Endereço do cliente                                 6
                                                                         7
   METHOD ObterCliente()                                                 8
   METHOD ObterTelefone()                                                9
   METHOD ObterEndereco()                                                10
                                                                         11
END CLASS                                                                12
                                                                         13
METHOD ObterCliente() CLASS Pedido                                       14
                                                                         15
   ACCEPT "Informe o nome do cliente : " TO ::cNome                      16
                                                                         17
RETURN NIL                                                               18
                                                                         19
```

```
METHOD ObterTelefone() CLASS Pedido

   ACCEPT "Informe o telefone do cliente : " TO ::cTelefone

RETURN NIL

METHOD ObterEndereco() CLASS Pedido

   ACCEPT "Informe o endereço do cliente : " TO ::cEndereco

RETURN NIL
```

Agora vamos voltar a classe CalculaPizza e fazer com que ela herde as características da classe Pedido. Não tem muito o que fazer, basta alterar o seu cabeçalho e incluir a cláusula “INHERIT”.

```
CLASS CalculaPizza INHERIT Pedido
```

Pronto, a nossa classe CalculaPizza agora tem os métodos e as variáveis definidas na classe Pedido. O nosso programa principal agora pode ser alterado para que fique assim :

```
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()
    oPizza:ObterCliente() // Novos metodos
    oPizza:ObterTelefone() // Novos metodos
    oPizza:ObterEndereco() // Novos metodos

    ? "A pizza custa $ ", oPizza:ValorFinal()

RETURN
```

O arquivo com a classe CalculaPizza e a procedure Main após as alterações está listada abaixo:

Listagem 36.8: Classe CalculaPizza após as alterações
Fonte: codigos/pizza.prg

```
#include "hbclass.ch"
#define BASE 10.00 // Custo da base da pizza
CLASS CalculaPizza INHERIT Pedido

   DATA nNumCob // Número de cobertura
   DATA nValCob // Preço por cobertura
   DATA nTotal // Preço final

   METHOD ObterQtdCoberturas()
```

```
   METHOD ObterValCoberturas()
   METHOD ValorFinal()

END CLASS

/**
Obtem a quantidade de coberturas
*/
METHOD ObterQtdCoberturas() CLASS CalculaPizza


INPUT "Informe a quantidade de coberturas : " TO ::nNumCob

RETURN NIL
/**
Obtem o valor das coberturas
*/
METHOD ObterValCoberturas()  CLASS CalculaPizza


INPUT "Informe o preço da cobertura : " TO ::nValCob

RETURN NIL
/**
Imprime o valor final
*/
METHOD ValorFinal()  CLASS CalculaPizza

  ::nTotal := BASE + ( ::nNumCob * ::nValCob )

RETURN ::nTotal
/*
Função Main
*/
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()
    oPizza:ObterCliente() // Novos metodos
    oPizza:ObterTelefone() // Novos metodos
    oPizza:ObterEndereco() // Novos metodos

    ? "A pizza custa $ ", oPizza:ValorFinal()

RETURN
```

Note que temos dois arquivos : pedido.prg e pizza.prg. Para compilar faça :

.:Resultado:.

```
hbmk2 pizza pedido
```

Uma simulação de execução :

.:Resultado:.

```
Informe a quantidade de coberturas : 2
Informe o preço da cobertura : 10
Informe o nome do cliente : Rubens
Informe o telefone do cliente : 75648-3232
Informe o endereço do cliente : Rua Clipper, No 7
A pizza custa $          30.00
```

Ainda não implementamos as rotinas de gravação nos arquivos. Iremos realizar isso após uma breve conversa sobre banco de dados relacionais e banco de dados orientado a objetos.

36.3 Banco de dados relacionais e banco de dados orientado a objetos

Até agora nós vimos apenas o banco de dados do Harbour, que são os arquivos DBFs. Vimos também que esse tipo de arquivo perdeu espaço para um padrão chamado SQL. Não vamos nos deter no padrão SQL nesse livro, basta você entender que, tanto os arquivos DBFs quanto o padrão SQL pertencem a um tipo de modelo chamado de "relacional". Os dados no modelo relacional estão agrupados em tabelas (semelhante a uma planilha eletrônica). A grande maioria das organizações obedecem a esse modelo bidimensional: bancos, supermercados, universidades e empresas vários portes. As vantagens de se usar o padrão relacional são diversos, dentre eles citamos:

1. a existência de uma teoria relacional criada por pesquisadores renomados;
2. a padronização de práticas;
3. a popularização dessa teoria através de universidades, centros de pesquisa, escolas técnicas, etc.;
4. um número muito grande de organizações que usam esse padrão;
5. a existência de comitês internacionais que criam normas para esse modelo.

Diante de tantos motivos, o modelo relacional tornou-se, praticamente, sinônimo de banco de dados.

Contudo, uma mudança ocorreu durante os anos de 1980. Com o advento das linguagens orientadas a objetos, surgiu também o desejo de se replicar esse modelo também para os bancos de dados. Dessa forma, os primeiros bancos de dados orientados a objetos foram criados. "Um banco de dados orientado a objetos é um banco de dados em que cada informação é armazenada na forma de objetos" [6]. **Um**

[6] https://pt.wikipedia.org/wiki/Banco_de_dados_orientado_a_objetos - OnLine : 11-Fev-2017

banco de dados relacional se torna difícil de manipular com dados complexos, tornando-se necessário uma etapa de conversão do modelo orientado a objetos para o modelo relacional para que a gravação possa ser feita. É justamente nesse ponto que nós iremos nos deter. Como o nosso curso é introdutório e atualmente o mundo relacional domina o universo dos banco de dados, não faz sentido estudar um banco de dados orientados a objetos, que carece até de padronização. O que nós queremos dizer é que você deve criar nas suas aplicações uma interface de tradução que faça com que o seu software orientado a objetos consiga gravar em um banco de dados relacional. Essa etapa de "tradução" é conhecida como "mapeamento objeto-relacional".

Dito isso, na nossa próxima seção nós iremos realizar um mapeamento objeto relacional **bem simples** para que os dados da nossa aplicação Pizza consiga gravar os pedidos. Dessa forma, esperamos que você consiga ter uma experiência prática com esse tipo de mapeamento.

36.3.1 Gravando dados do pedido

Para realizar a gravação dos dados vamos atentar primeiramente para os seguintes fatos :

1. temos duas classes: Pedido e CalculaPizza;
2. a classe CalculaPizza herda todas as características da classe Pedido;
3. o programador usa diretamente a classe CalculaPizza;
4. tanto a classe Pedido quanto a classe CalculaPizza possuem dados que devem ser gravados.

A conclusão inicial a que chegamos é que : "devemos criar o método Grava() na classe CalculaPizza().

Os passos para a gravação do pedido estão definidos nas próximas subseções.

Definir a estrutura do arquivo no cabeçalho

Nas definições faça :

```
#define ESTRUTURA_DBF { { "NUMCOB" , "N" , 3 , 0 },;                    1
                        { "VALCOB" , "N" , 6 , 2 },;                    2
                        { "TOTAL" , "N" , 6 , 2 },;                     3
                        { "NOME" , "C" , 30 , 0 },;                     4
                        { "TELEFONE" , "C" , 30 , 0 },;                 5
                        { "ENDERECO" , "C" , 30 , 0 } }                 6
```

Até agora tudo bem, nós já fizemos isso anteriormente na versão estruturada do nosso programa.

Criar o cabeçalho do método Grava

```
CLASS CalculaPizza INHERIT Pedido

   DATA nNumCob
   DATA nValCob
   DATA nTotal

   METHOD ObterQtdCoberturas()
   METHOD ObterValCoberturas()
   METHOD ValorFinal()
   METHOD Grava()  // <======== MÉTODO GRAVA

END CLASS
```

Criar o método Grava

```
METHOD Grava()  CLASS CalculaPizza
LOCAL lSucesso

      IF .NOT. FILE( "pizza.dbf" )
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )
      ENDIF
      USE pizza
      IF USED()
         lSucesso := .t.
      ELSE
         lSucesso := .f.
      ENDIF

      IF lSucesso
         APPEND BLANK
         REPLACE NUMCOB WITH ::nNumCob
         REPLACE VALCOB WITH ::nValCob
         REPLACE TOTAL WITH ::nValFinal
         REPLACE NOME WITH ::cNome
         REPLACE TELEFONE WITH ::cTelefone
         REPLACE ENDERECO WITH ::cEndereco
      ENDIF


RETURN lSucesso
```

Alterar a procedure MAIN para incluir o método Grava

Altere a procedure MAIN e coloque logo no seu final o seguinte código:

```

```

```
    IF oPizza:Grava()
       ? "Pedido gravado com sucesso"
    ELSE
       ? "Problemas na gravação"
    ENDIF
```

36.3.2 A listagem final

O arquivo pedido.prg não sofreu alterações, por isso vamos listar somente o arquivo pizza com as alterações. Note que nós renomeamos ele para pizzafinal.prg para preservar o arquivo anterior.

Listagem 36.9: Versão quase final
Fonte: codigos/pizzafinal.prg

```
#include "hbclass.ch"
#define BASE 10.00 // Custo da base da pizza
#define ESTRUTURA_DBF { { "NUMCOB" , "N" , 3 , 0 },;
                        { "VALCOB" , "N" , 6 , 2 },;
                        { "TOTAL" , "N" , 6 , 2 },;
                        { "NOME" , "C" , 30 , 0 },;
                        { "TELEFONE" , "C" , 30 , 0 },;
                        { "ENDERECO" , "C" , 30 , 0 } }

CLASS CalculaPizza INHERIT Pedido

   DATA nNumCob // Número de cobertura
   DATA nValCob // Preço por cobertura
   DATA nTotal // Preço final

   METHOD ObterQtdCoberturas()
   METHOD ObterValCoberturas()
   METHOD ValorFinal()
   METHOD Grava()  // <======== MÉTODO GRAVA

END CLASS

/**
Obtem a quantidade de coberturas
*/
METHOD ObterQtdCoberturas() CLASS CalculaPizza


INPUT "Informe a quantidade de coberturas : " TO ::nNumCob

RETURN NIL
/**
Obtem o valor das coberturas
*/
METHOD ObterValCoberturas()  CLASS CalculaPizza

```

```
                                                                                
INPUT "Informe o preço da cobertura : " TO ::nValCob                            

RETURN NIL
/**
Imprime o valor final
*/
METHOD ValorFinal()  CLASS CalculaPizza

  ::nTotal := BASE + ( ::nNumCob * ::nValCob )

RETURN ::nTotal
/**
Grava
*/
METHOD Grava()  CLASS CalculaPizza
LOCAL lSucesso

      IF .NOT. FILE( "pizza.dbf" )
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )
      ENDIF
      USE pizza
      IF USED()
         lSucesso := .t.
      ELSE
         lSucesso := .f.
      ENDIF

      IF lSucesso
         APPEND BLANK
         REPLACE NUMCOB WITH ::nNumCob
         REPLACE VALCOB WITH ::nValCob
         REPLACE TOTAL WITH ::nTotal
         REPLACE NOME WITH ::cNome
         REPLACE TELEFONE WITH ::cTelefone
         REPLACE ENDERECO WITH ::cEndereco
      ENDIF


RETURN lSucesso

/*
Função Main
*/
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()
```

```
    oPizza:ObterCliente() // Novos metodos
    oPizza:ObterTelefone() // Novos metodos
    oPizza:ObterEndereco() // Novos metodos

    ? "A pizza custa $ ", oPizza:ValorFinal()
    IF oPizza:Grava()
       ? "Pedido gravado com sucesso"
    ELSE
       ? "Problemas na gravação"
    ENDIF

RETURN
```

Para compilar faça :

.:Resultado:.

```
hbmk2 pizzafinal pedido
```

.:Resultado:.

```
Informe a quantidade de coberturas : 3
Informe o preço da cobertura : 7
Informe o nome do cliente : Vlademiro
Informe o telefone do cliente : 3712-1212
Informe o endereço do cliente : Rua dos Aprendizes
A pizza custa $          31.00
Pedido gravado com sucesso
```

36.3.3 O construtor da classe

Quando uma classe é criada ela recebe um método especial chamado construtor. Um construtor é um método que é executado automaticamente sempre que a sua classe é criada. Toda classe tem um construtor e se você não criar um construtor, a própria classe se encarregará de criar um automaticamente e lhe dará o nome de NEW().

```
oPizza := CalculaPizza():New()
```

Você mesmo pode criar o seu método construtor. Para fazer isso apenas crie um método e, preferencialmente, dê-lhe o nome de NEW(). Quando for declarar esse método no interior da classe coloque a palavra “CONSTRUCTOR” logo após a declaração.

```
CLASS CalculaPizza INHERIT Pedido

   DATA nNumCob
   DATA nValCob
   DATA nTotal
```

```
   METHOD ObterQtdCoberturas()
   METHOD ObterValCoberturas()
   METHOD ValorFinal()
   METHOD Grava()
   METHOD New() CONSTRUCTOR // O construtor

END CLASS
```

Os parâmetros do construtor e valores de retorno

Todo método (assim como qualquer rotina) pode receber parâmetros e retornar valores. Um erro comum entre os iniciantes é passar o parâmetro certo, mas no local errado. Por exemplo, o código a seguir está errado, pois o parâmetro está no lugar errado.

```
oArquivo := ArquivoTexto( "arquivo.txt" ):New()
```

O correto seria :

```
oArquivo := ArquivoTexto():New(  "arquivo.txt"  )
```

O valor que um construtor deve retornar é o próprio objeto e para que isso ocorra você sempre deve retornar o valor SELF. SELF é uma palavra-chave que significa mais ou menos “EU MESMO”.

```
METHOD New() CLASS ArquivoTexto

    ... Código da classe

RETURN Self
```

36.4 Métodos com o mesmo nome em classes herdadas

Sabemos que duas classes podem ter métodos com o mesmo nome, mas ainda não estudamos o que acontece quando uma classe, que é pai de outra classe, tem um método com o mesmo nome da classe filha. Nesse caso, se você estiver usando a classe filha, o método que será chamado será a da classe filha (como era de se esperar), e se você estiver chamando a classe pai, o método será a da classe pai (nenhuma novidade aqui). Vamos realizar uma simulação com a nossa classe CalculaPizza e a nossa classe Pedido na próxima seção, fique atento.

36.5 Tornando a classe Pedido mais genérica possível

Vamos partir do seguinte caso: você concluiu o programa CalculaPizza() e o seu cliente esta satisfeito com o seu trabalho, tão satisfeito que lhe indicou para um amigo dele. O problema agora é que o amigo dele não trabalha no ramo de pizzarias, mas no

ramo de venda de implementos agrícolas. Você ainda não visitou o seu provável novo cliente, mas já pensa em usar parte do código do seu programa quando for emitir o pedido para esse novo cliente. Na verdade você não vai usar a classe CalculaPizza(), mas vai usar a classe Pedido() e criar uma nova classe do tipo CalculaImplementos(). Porém, você detectou um pequeno inconveniente no seu código. Vamos descrever esse inconveniente logo a seguir:

- a classe Pedido() não grava o pedido, quem faz isso é a classe CalculaPizza().
- quando você for criar a classe para o seu novo sistema, a que vai ficar no lugar de CalculaPizza(), você também irá criar um método para gravar o pedido.
- quando você for criar esse novo método, alguns campos que são específicos do pedido de venda serão repetidos nesse novo método. Os campos são : Nome do cliente, endereço do cliente e telefone do cliente.
- o ideal é que os campos genéricos do pedido, como Nome do cliente, endereço do cliente e telefone do cliente, fiquem na classe Pedido.

O inconveniente é o seguinte: toda vez que você for criar uma classe filha da classe Pedido, você terá que criar um método Grava() e gravar os dados que pertencem somente ao Pedido. Por exemplo, uma classe filha nova irá requerer os seguintes comandos no seu método Grava():

```
REPLACE NOME WITH ::cNome
REPLACE TELEFONE WITH ::cTelefone
REPLACE ENDERECO WITH ::cEndereco
```

Em um futuro não tão distante você poderá conseguir um novo cliente de um outro ramo (hotelaria, por exemplo) e terá que criar um método Grava() na futura classe filha também com os comandos acima. Isso é uma repetição desnecessária que possui um outro inconveniente: e se a sua classe Pedido mudar para poder ter o campo “data do pedido” ? Nesse caso, você teria o trabalho dobrado de incluir esse código nas suas classes filhas, pois é lá que o método Grava() está.

Pensando nisso você (inteligentemente) resolve fazer uma pequena alteração na suas classes já criadas: CalculaPizza() e Pedido(). Você resolve criar um método Grava() também na classe Pedido() apenas com os campos do pedido.

Você pode estar achando tudo um pouco confuso, mas vamos demonstrar através do código.

Primeiro, vamos criar o método Grava() da classe pedido. Nós fazemos isso transferindo parte do código do método Grava() da classe CalculaPizza(). Primeiramente, copie a classe pedido.prg para pedido2.prg. Essa cópia é só para preservar o modelo anterior, caso queira consultar mais tarde. Agora inclua o método Grava. Observe a seguir o método Grava() da classe Pedido() :

```
#include "hbclass.ch"                                                    1
CLASS Pedido                                                             2
                                                                         3
   DATA cNome // Nome do cliente                                         4
   DATA cTelefone // Telefone do cliente                                 5
```

```
   DATA cEndereco // Endereço do cliente

   METHOD ObterCliente()
   METHOD ObterTelefone()
   METHOD ObterEndereco()

   METHOD Grava()

END CLASS
```

Feito isso, vá para o final do arquivo e acrescente ao método Grava() da classe Pedido apenas os campos do arquivo pizza.dbf que correspondem a classe Pedido (você está fazendo um mapeamento objeto-relacional de verdade agora).

```
METHOD Grava() CLASS Pedido

  REPLACE NOME WITH ::cNome
  REPLACE TELEFONE WITH ::cTelefone
  REPLACE ENDERECO WITH ::cEndereco

RETURN NIL
```

Pronto, salve o seu arquivo pedido2.prg. Agora vamos copiar o arquivo pizzafinal.prg para pizzafinal2.prg. Essa cópia é apenas para preservar o arquivo anterior, para que você possa comparar depois. Abra o arquivo pizzafinal2.prg e faça as alterações no método Grava().

```
/**
Grava
*/
METHOD Grava()  CLASS CalculaPizza
LOCAL lSucesso

      IF .NOT. FILE( "pizza.dbf" )
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )
      ENDIF
      USE pizza
      IF USED()
         lSucesso := .t.
      ELSE
         lSucesso := .f.
      ENDIF

      IF lSucesso
         APPEND BLANK
         ::Super:Grava()
         REPLACE NUMCOB WITH ::nNumCob
         REPLACE VALCOB WITH ::nValCob
         REPLACE TOTAL WITH ::nTotal
      ENDIF

```

```
                                                                    25
RETURN lSucesso                                                     26
```

Note que a única alteração feita foi trocar as instruções REPLACE que fazia a gravação dos campos NOME, TELEFONE e ENDERECO pela linha ::Super:Grava().

Pronto. Agora é só compilar:

.:Resultado:.

```
hbmk2 pizzafinal2 pedido2
```

Pronto. Você agora fez um mapeamento objeto relacional simples. Não queremos nos estender muito nesse assunto, mas se você notar, verá que agora as duas classes partilham o mesmo arquivo (isso porque o banco de dados não é orientado a objeto, mas relacional), mas cada classe só grava no arquivo os seus campos.

Você deve estar se perguntando como isso foi feito. Primeiramente nós criamos dois métodos Grava(), um para cada classe (isso você já sabe). O método Grava() chamado pela sua aplicação continua sendo o da classe mais inferior, ou seja, CalculaPizza(). Só que, logo após o APPEND BLANK, nós chamamos o método Grava() da classe Pedido. Como nós fazemos isso, já que os métodos tem o mesmo nome ? É simples, é só colocar o prefixo Super:: antes do nome do método. Esse prefixo indica que o método a ser usado não é o da classe corrente, mas o da classe pai. Se no futuro você quiser incluir algo na classe Pedido (como por exemplo, a data do pedido, o usuário responsável, a transportadora, etc.), você deverá alterar apenas a classe Pedido. Isso torna as coisas mais claras para o programador.

O código final está listado a seguir :

Aqui temos o arquivo que define a classe Pedido.

Listagem 36.10: Versão final da classe Pedido
Fonte: codigos/pizzafinal2.prg

```
#include "hbclass.ch"                                                   1
#define BASE 10.00 // Custo da base da pizza                            2
#define ESTRUTURA_DBF { { "NUMCOB" , "N" , 3 , 0 },;                    3
                        { "VALCOB" , "N" , 6 , 2 },;                    4
                        { "TOTAL" , "N" , 6 , 2 },;                     5
                        { "NOME" , "C" , 30 , 0 },;                     6
                        { "TELEFONE" , "C" , 30 , 0 },;                 7
                        { "ENDERECO" , "C" , 30 , 0 } }                 8
                                                                        9
CLASS CalculaPizza INHERIT Pedido                                       10
                                                                        11
   DATA nNumCob // Número de cobertura                                  12
   DATA nValCob // Preço por cobertura                                  13
   DATA nTotal // Preço final                                           14
                                                                        15
   METHOD ObterQtdCoberturas()                                          16
   METHOD ObterValCoberturas()                                          17
   METHOD ValorFinal()                                                  18
   METHOD Grava()  // <======== MÉTODO GRAVA                            19
                                                                        20
END CLASS                                                               21
                                                                        22
```

```
/**
Obtem a quantidade de coberturas
*/
METHOD ObterQtdCoberturas() CLASS CalculaPizza


INPUT "Informe a quantidade de coberturas : " TO ::nNumCob

RETURN NIL
/**
Obtem o valor das coberturas
*/
METHOD ObterValCoberturas()  CLASS CalculaPizza


INPUT "Informe o preço da cobertura : " TO ::nValCob

RETURN NIL
/**
Imprime o valor final
*/
METHOD ValorFinal()  CLASS CalculaPizza

  ::nTotal := BASE + ( ::nNumCob * ::nValCob )

RETURN ::nTotal
/**
Grava
*/
METHOD Grava()  CLASS CalculaPizza
LOCAL lSucesso

      IF .NOT. FILE( "pizza.dbf" )
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )
      ENDIF
      USE pizza
      IF USED()
         lSucesso := .t.
      ELSE
         lSucesso := .f.
      ENDIF

      IF lSucesso
         APPEND BLANK
         ::Super:Grava()
         REPLACE NUMCOB WITH ::nNumCob
         REPLACE VALCOB WITH ::nValCob
         REPLACE TOTAL WITH ::nTotal
      ENDIF
```

```
RETURN lSucesso

/*
Função Main
*/
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()
    oPizza:ObterCliente() // Novos metodos
    oPizza:ObterTelefone() // Novos metodos
    oPizza:ObterEndereco() // Novos metodos

    ? "A pizza custa $ ", oPizza:ValorFinal()
    IF oPizza:Grava()
       ? "Pedido gravado com sucesso"
    ELSE
       ? "Problemas na gravação"
    ENDIF

RETURN
```

Agora a classe CalculaPizza, filha da classe Pedido.

Listagem 36.11: Versão final
Fonte: codigos/pizzafinal2.prg

```
#include "hbclass.ch"
#define BASE 10.00 // Custo da base da pizza
#define ESTRUTURA_DBF { { "NUMCOB" , "N" , 3 , 0 },;
                        { "VALCOB" , "N" , 6 , 2 },;
                        { "TOTAL" , "N" , 6 , 2 },;
                        { "NOME" , "C" , 30 , 0 },;
                        { "TELEFONE" , "C" , 30 , 0 },;
                        { "ENDERECO" , "C" , 30 , 0 } }

CLASS CalculaPizza INHERIT Pedido

   DATA nNumCob // Número de cobertura
   DATA nValCob // Preço por cobertura
   DATA nTotal // Preço final

   METHOD ObterQtdCoberturas()
   METHOD ObterValCoberturas()
   METHOD ValorFinal()
   METHOD Grava()  // <======== MÉTODO GRAVA

END CLASS

/**
```

```
*Obtem a quantidade de coberturas
*/
METHOD ObterQtdCoberturas() CLASS CalculaPizza


INPUT "Informe a quantidade de coberturas : " TO ::nNumCob

RETURN NIL
/**
*Obtem o valor das coberturas
*/
METHOD ObterValCoberturas()  CLASS CalculaPizza


INPUT "Informe o preço da cobertura : " TO ::nValCob

RETURN NIL
/**
*Imprime o valor final
*/
METHOD ValorFinal()  CLASS CalculaPizza

  ::nTotal := BASE + ( ::nNumCob * ::nValCob )

RETURN ::nTotal
/**
*Grava
*/
METHOD Grava()  CLASS CalculaPizza
LOCAL lSucesso

      IF .NOT. FILE( "pizza.dbf" )
         DBCREATE( "pizza.dbf" , ESTRUTURA_DBF )
      ENDIF
      USE pizza
      IF USED()
         lSucesso := .t.
      ELSE
         lSucesso := .f.
      ENDIF

      IF lSucesso
         APPEND BLANK
         ::Super:Grava()
         REPLACE NUMCOB WITH ::nNumCob
         REPLACE VALCOB WITH ::nValCob
         REPLACE TOTAL WITH ::nTotal
      ENDIF


RETURN lSucesso
```

```
/*
Função Main
*/
PROCEDURE Main
LOCAL oPizza

    oPizza := CalculaPizza():New()

    oPizza:ObterQtdCoberturas()
    oPizza:ObterValCoberturas()
    oPizza:ObterCliente() // Novos metodos
    oPizza:ObterTelefone() // Novos metodos
    oPizza:ObterEndereco() // Novos metodos

    ? "A pizza custa $ ", oPizza:ValorFinal()
    IF oPizza:Grava()
       ? "Pedido gravado com sucesso"
    ELSE
       ? "Problemas na gravação"
    ENDIF

RETURN
```

Para o usuário final, quase nada mudou. A vantagem (esperamos ter conseguido lhe convencer) é que as mudanças futuras serão mais fáceis. Você terá menos trabalho não só para alterar o programa da pizzaria, mas também para a criar outros programas futuros que tenham pedidos de venda. A classe Pedido deverá ficar apenas com os atributos comuns a todas as classes, e o banco de dados deverá ser criado na classe filha mesmo, pois cada empresa terá a sua própria estrutura de arquivo.

.:Resultado:.

```
Informe a quantidade de coberturas : 3
Informe o preço da cobertura : 7
Informe o nome do cliente : Vlademiro
Informe o telefone do cliente : 3712-1212
Informe o endereço do cliente : Rua dos Aprendizes
A pizza custa $          31.00
Pedido gravado com sucesso
```

37 UML

Assim como é inútil estender a rede se as aves o observam, também esses homens não percebem que fazem tocaia contra a própria vida; armam emboscadas contra eles mesmos! Tal é o caminho de todos os gananciosos; quem assim procede se destrói

Provérbios 1.17-19

37.1 Uma conversa inicial

Imagine que você já desenvolve sistemas há alguns anos. Você já tem os seus próprios clientes e também já criou as suas próprias técnicas para agilizar a criação dos seus sistemas. Você também já deve ter passado por alguns problemas comuns a todos os programadores, e também é possível que já tenha enfrentado situações inusitadas para a maioria das outras profissões, como ser despertado de madrugada com algum cliente desesperado com o seu sistema de informações inoperante.

Mesmo com todos as dificuldades e prazos apertados, o seu negócio vai bem, a sua renda mensal está garantida e talvez você esteja se perguntando qual a vantagem de se aprender sobre Casos de uso e UML.

Um Diagrama de Caso de uso é, tão somente uma ferramenta que ajuda você a visualizar e a transmitir os requisitos do seu sistema. Algumas pessoas querem aprender HTML para criar páginas belíssimas, mas depois que aprendem HTML, CSS e JS descobrem que não sabem criar a tão sonhada página, e ficam copiando modelos pré-prontos para fazerem pequenas customizações. Saber HTML não garante a construção de sites arrasadores. Da mesma forma, algumas pessoas passam a programar usando uma linguagem orientada a objetos, mas depois descobrem que a tal linguagem não agregou em nada no resultado final dos programas criados. Usar uma linguagem orientada a objetos não garante a você um bom desempenho. Já vi diversos programadores (diversos mesmo) programarem usando C++ mas usando a velha técnica estruturada da Linguagem C.

UML, HTML, C++, e tantas outras linguagens são "balas de prata". Elas não irão funcionar se você não aprender determinadas teorias, e dependem também da sua experiência pessoal. Não é algo que se aprende somente nos livros. Com certeza essas linguagens fazem a diferença, mas a pessoa que as utiliza precisa dominar determinadas técnicas e ter alguma vivência. Por isso não fique desapontado com esse capítulo. Ele está aqui porque a UML faz parte do arsenal de todo desenvolvedor. Vamos usar exemplos simples, e não iremos abordar todos os diagramas usados na UML. Vamos iniciar com o diagrama de casos de uso.

37.1.1 Para que serve um Caso de uso ?

Vamos fazer novamente o nosso exercício de imaginação: durante todos esses anos que você vem desenvolvendo sistemas profissionalmente, você já deve ter entrado em contato com vários clientes para fazer o levantamento das suas necessidades, para daí, desenvolver uma solução. Essa etapa, tão importante, recebe o nome de **levantamento de requisitos de sistema**. Pois bem, um Diagrama de Caso de uso é uma representação gráfica dos requisitos do seu sistema. Saber "desenhar"um Diagrama de Caso de uso é fácil, a parte que demanda trabalho é o levantamento dos requisitos do sistema a ser desenvolvido. É uma etapa que é crucial, pois evita problemas futuros do tipo : "não foi isso o que eu lhe pedi". Usar a UML para desenvolver seus Casos de uso diminui as falhas na comunicação, pois tornam as coisas mais claras para você, para sua equipe e para seu cliente. O levantamento de requisitos de sistema não é um documento desenvolvido pelo Programador, mas sim pelo Analista de Sistemas em conjunto com a organização onde ele presta consultoria. Contudo, na vida real, existem muitos programadores que possuem as suas próprias empresas e atuam também como analistas de sistemas. Tendo em mente esse cenário, tão comum

atualmente, nós decidimos incluir o "Caso de uso"dentro do nosso estudo sobre UML.

Antes de prosseguirmos dois pequenos esclarecimentos:

1. UML é uma linguagem usada para documentar e visualizar o seu projeto de software. Ela não vai lhe dizer o que fazer, nem quais passos seguir[1].

2. Um Diagrama de Caso de uso é um dos diagramas que a UML possui. Além do caso de uso, a UML possui diversos outros diagramas, como o Diagrama de Classes, Diagrama de sequência, Diagrama de pacotes, etc. Cada um desses diagramas documenta uma determinada faceta do sistema de informação. O Caso de uso documenta os requisitos do seu sistema.

37.2 Nosso primeiro programa

O objetivo dessa seção é mostrar, através de uma estória, a criação de um Diagrama de Caso de uso através da linguagem UML.[2]

37.2.1 Uma palavrinha sobre os requisitos

Antes de mais nada, para evitar qualquer falha na nossa comunicação posterior, vamos definir o que queremos dizer com "requisitos de sistema".

Não vamos aqui nos demorar nos diversos tipos de requisitos [3], os requisitos de que trataremos aqui são as tarefas e serviços que o seu sistema deve executar. Na teoria eles são chamados de "Requisitos Funcionais".

O que é um requisito ?

É uma função específica que o seu sistema deve realizar para funcionar corretamente. É uma necessidade que detalha o que o sistema deve fazer.[McLaughlin, Pollice e West 200 p. 44]

Na próxima seção iremos acompanhar a estória fictícia do desenvolvimento de um programa de computador.

37.2.2 Seu novo projeto de programação

Você foi chamado para desenvolver um software para controlar uma porta para cachorros.

A porta para cachorro não é uma porta qualquer. Ela é acionada por controle remoto. Assim, o seu dono não vai precisar se levantar para abrir a porta. Como a porta tem um controle remoto, basta o dono apertar o botão do controle para abrir a porta, e depois apertar o botão do controle para fechar a porta.

A primeira coisa que devemos fazer é obter a lista de requisitos funcionais do sistema. Os requisitos não precisam ser apenas de software, algumas características adicionais podem não depender do programador, mas serem cruciais para o sucesso da solução apresentada:

1. a abertura da porta para cachorros deve ter pelo menos 30 cm de altura;

[1] Maiores detalhe em https://pt.wikipedia.org/wiki/UML

[2] Essa estória foi baseada em [McLaughlin, Pollice e West 2007, p. 40].

[3] Existem os requisitos funcionais, requisitos de qualidade, requisitos comportamentais, etc.

Figura 37.1: Porta para cachorros

2. um botão de controle remoto abre a porta se ela estiver fechada, e fecha a porta se ela estiver aberta.

A lista de requisitos pode ser complementada com uma lista do processo completo, como se fosse uma estória:

1. O cachorro late para poder sair;
2. o dono ouve o latido;
3. o dono aperta o botão do controle remoto;
4. a porta abre;
5. o cachorro sai;
6. o cachorro faz o que queria fazer;
7. o cachorro entra;
8. o dono verifica se o cachorro entrou;
9. o dono aperta o botão do controle remoto;
10. a porta fecha.

Note que existem ítens que não são de responsabilidade do programador, como por exemplo o ítem 8. Esse ítem, mais na frente, irá causar um pequeno problema entre o programador e o seu cliente.

Agora vamos codificar a solução.

37.2.3 Codificação

Teremos que desenvolver dois programas: o primeiro irá ficar na porta, em algum dispositivo de hardware, e o segundo irá ficar no controle remoto.

Vamos começar pela porta

Listagem 37.1: Controlador da Porta

```
#include "hbclass.ch"

/****c* DogDoor/DogDoor
* NAME
*
*  DogDoor
*
* DESCRIPTION
*
* Porta para cachorros
*
* SOURCE
*/

CREATE CLASS DOGDOOR
    * Attributes and methods
    HIDDEN:
    DATA lIsOpen INIT .f.

    EXPORTED:
    METHOD open()
    METHOD close()
    METHOD isOpen()

ENDCLASS


/****m* DogDoor/open
*
* Abre a porta
*/
METHOD open() CLASS DogDoor

    ::lIsOpen := .t.

    RETURN NIL
/************/

/****m* DogDoor/close
*
*  Fecha a porta
*/
METHOD close() CLASS DogDoor

    ::lIsOpen := .f.

    RETURN NIL
/************/

/****m* DogDoor/isOpen
```

```
*
*  Mostra o estado da porta
*
*  .t. - aberta
*  .f. - fechada
*/
METHOD isOpen() CLASS DogDoor

    RETURN ::lIsOpen
/************/
```

Para testarmos a classe que controla a porta, faça o seguinte código de teste

Listagem 37.2: Teste do Controlador da Porta

```

PROCEDURE MAIN

    LOCAL oDogDoor := DogDoor():New()

    ? "A porta esta aberta ? " , oDogDoor:IsOpen()
    ? "Abra a porta"
    oDogDoor:Open()
    ? "A porta esta aberta ? " , oDogDoor:IsOpen()
    ? "Feche a porta"
    oDogDoor:Close()
    ? "A porta esta aberta ? " , oDogDoor:IsOpen()

    RETURN
```

O resultado do teste é o seguinte :

```
A porta está aberta ?  .F.
Abra a porta
A porta está aberta ?  .T.
Feche a porta
A porta está aberta ?  .F.
```

Agora vamos para o controle remoto

Listagem 37.3: Controle remoto

```
#include "hbclass.ch"

/****c* Remote/Remote
* NAME
*
*  Remote
*
* DESCRIPTION
*
* Controle remoto
*
```

```
* SOURCE
*/

CREATE CLASS REMOTE
    * Attributes and methods
    HIDDEN:
    DATA oDogDoor INIT DogDoor():New()

    EXPORTED:
    METHOD pressButton()

ENDCLASS
/******/


/****m* Remote/pressButton
* NAME
*
*  pressButton
*
*
*/
METHOD pressButton() CLASS Remote

    IF ::oDogDoor:IsOpen()
        ::oDogDoor:Close()
        cReturn := "A PORTA ESTA ABERTA. VOU FECHAR A PORTA"
    ELSE
        ::oDogDoor:Open()
        cReturn := "A PORTA ESTA FECHADA. VOU ABRIR A PORTA"
    ENDIF

    RETURN cReturn
/************/
```

O teste a seguir verifica se o controle remoto está funcionando.

Listagem 37.4: Teste do Controle remoto

```

PROCEDURE MAIN

    LOCAL oRemote := Remote():New()

    ? "O cachorro quer sair"
    ? oRemote:pressButton()
    ? "O cachorro quer entrar"
    ? oRemote:pressButton()
    ? "O cachorro quer sair"
    ? oRemote:pressButton()

    RETURN
```

O resultado do teste é o seguinte :

```
Pressionando o botao do controle remoto
A PORTA ESTA FECHADA. VOU ABRIR A PORTA
Pressionando o botao do controle remoto
A PORTA ESTA ABERTA. VOU FECHAR A PORTA
Pressionando o botao do controle remoto
A PORTA ESTA FECHADA. VOU ABRIR A PORTA
```

37.2.4 Os problemas começam a aparecer

Se você entregou a primeira versão do seu programa para o cliente e ele não entrou em contato para reclamar de algum problema, uma das duas opções abaixo está correta:

1. Você é um ótimo programador, conseguiu mapear todos os requisitos de primeira e a primeira versão do seu programa foi livre de erros.
2. O seu cliente ainda não começou a usar o seu programa.

Geralmente, a primeira execução de um programa recém-criado contém alguma falha nos requisitos. A culpa não é necessariamente sua. Muitas vezes o próprio cliente não conseguiu antecipar todos os problemas que podem ocorrer. Nesse caso, não é uma boa ideia ficar discutindo com o cliente, principalmente se você foi contratado para desenvolver um programa do zero para ele.[4]

Foi o que aconteceu com a nossa porta para cachorros. Depois de um tempo com a porta instalada, as reclamações começaram a surgir. Basicamente eram ratos, gatos e outros animais indesejáveis que usavam a porta para cachorros. Alguém estava esquecendo a porta aberta.

Figura 37.2: Visitantes indesejáveis

[4] É claro que essa regra não deve ser levada ao pé da letra. Existem casos em que a culpa não pode ser atribuída a você. Um exemplo clássico disso é quando o seu programa já foi criado, testado e já funciona em muitos clientes. Você pode até ceder uma cópia de demonstração para um potencial cliente testar. Nesse caso, quando os requisitos não atendem à necessidade do cliente, a culpa não é sua. Simplesmente o programa não satisfaz a demanda do seu cliente, cabe a você decidir se vale a pena alterar o programa ou não.

38 Padrões de Projeto

> Eu lhe prometo decepções em todos os filmes, pois eles estão muito distantes da perfeição da imaginação, assim como todas as coisas que fazemos.
>
> Dudley Nichols

Objetivos do capítulo

- O que são padrões de projetos
- Reconhecer os padrões de projetos
- Implementar o padrão Singleton

38.1 Introdução

A palavra "padrão"pode causar confusão no entendimento porque ela pode significar três coisas diferentes em inglês. **(1)** Em primeiro lugar temos a palavra "default", que já foi vista durante o capítulo sobre funções. Nesse contexto, "valor padrão"(default value) significa aquele valor que um parâmetro assume, caso ele não tenha sido declarado explicitamente pelo programador. **(2)** Em seguida, "padrão"pode assumir o significado de modelo ("standard"). Nesse caso, ele assume o significado de "modelo", ou aquilo que se pauta para construir outra coisa idêntica [1]. **(3)** Finalmente, temos a palavra padrão em sua forma mais geral, significando "modo como algo costuma acontecer"(pattern). É nesse terceiro sentido que traduzimos a palavra "padrão"dentro do termo "padrão de projeto". Em programação, um problema qualquer pode ser resolvido de várias formas diferentes. Algumas soluções são adequadas, mas outras nem tanto. Algumas soluções consomem muito tempo, já outras são mais eficientes. Umas soluções são obscuras e de difícil manutenção, mas outras permitem uma manutenção futura menos traumática. **Em outras palavras:** um "padrão de projeto"é uma solução geral para um problema que ocorre com frequência. O contexto que adotaremos é o contexto do projeto de software. Já vimos que o software passou por duas crises. A primeira delas nós já abordamos (página 311), trata-se da crise na criação do software. Contudo a indústria do software passou por outra crise, chamada de "crise na manutenção". Vimos também que essa "crise na manutenção"só viria a ser resolvida com técnicas de programação ligadas a Orientação o Objetos. Esse capítulo trata dessa segunda crise e as formas de resolução da mesma.

38.2 A gangue dos quatro

A programação orientada à objetos começou a popularizar-se na segunda metade da década de 1980. Uma das promessas desse novo paradigma era reduzir os custos na manutenção do software criado.

38.3 O padrão Singleton

O padrão Singleton é classificado como padrão criacional

38.3.1 Que tipo de problema o Singleton resolve ?

Você desenvolveu um sistema de vendas para uma empresa chamada "Nova Escola Ltda". O sistema possui cadastros e relatórios, assim como qualquer outro sistema. Contudo, o nome da empresa "Nova Escola Ltda"está presente em diversas partes do seu sistema. Algo como a listagem abaixo:

```
@ 10,10 SAY "Nova Escola Ltda."
```
1

[1] Antigamente, no ano de 1955, o jornal "O Globo"lançou um prêmio chamado "Operário padrão". Essa campanha tinha como objetivo enaltecer o operário, e também criar "modelos"a serem seguidos, em termos de competência, esforço e disciplina.

Um belo dia você consegue outro cliente, uma empresa chamada "Elvac, Geradores Eólicos S.A.". Você, então resolve fazer uma alteração no seu sistema. Primeiramente você cria um arquivo chamado "CONFIG.DBF", com a estrutura abaixo :

```
    aStruct := { {"EMPRESA","C",50,0} }
    DBCREATE( "config.dbf" , aStruct )
```

Dentro desse arquivo você grava o nome da empresa. Algo como:

```
    APPEND BLANK
    REPLACE EMPRESA WITH "Elvac, Geradores Eólicos S.A."
```

ou algo como

```
    APPEND BLANK
    REPLACE EMPRESA WITH "Nova Escola Ltda."
```

Pronto, agora sempre que surgir um cliente novo eu altero o arquivo "config.dbf"com o nome da empresa. Agora eu tenho que ler esse arquivo dentro do meu programa.

Dentro do seu sistema, nas primeiras linhas você faz algo como :

```
PROCEDURE MAIN

    PUBLIC M->EMPRESA

    USE config SHARED
    M->EMPRESA := FIELD->EMPRESA
    CLOSE DATABASE
```

Assim, nos seus relatórios e telas, você pode fazer assim:

```
    @ 10,10 SAY M->EMPRESA
```

Tais códigos são bastante comuns ainda, e muitos sistemas funcionam perfeitamente bem com eles. No entanto existe um problema grave nessa forma de trabalho: uma variável do tipo PUBLIC pode ter o seu valor alterado em qualquer parte do sistema, e isso pode levar a problemas futuros. Alguém poderia discordar dizendo: "eu nunca iria alterar o valor dessa variável". Contudo, um Engenheiro aeroespacial chamado Edward Murphy ficou famoso com a seguinte citação: "Qualquer coisa que possa ocorrer mal, ocorrerá mal, no pior momento possível". Essa citação ficou tão famosa que ganhou até um nome: "Lei de Murphy". Essa lei não pode ser interpretada de maneira uniforme para todas as coisas. Murphy trabalhava com experimentos científicos de alto risco, de modo que um pequeno sistema de informação possui uma tolerância a falhas bem menor do que um experimento envolvendo foguetes em trilhos. Contudo a Lei de Murphy se aplica ao nosso exemplo, embora com uma gravidade bem menor.

Imagine que em algum trecho do seu programa, coincidentemente, existe um arquivo chamado "fornecedor.dbf". Dentro desse arquivo existe um campo chamado EMPRESA. Esse campo é alterado da seguinte forma :

```
```

```
    M->EMPRESA := SPACE(60)

    ... Códigos diversos

    @ 10,10 SAY "DIGITE O NOME DO FORNECEDOR " GET M->EMPRESA
    READ
```

Pronto. A variável de configuração pública M->EMPRESA teve o seu conteúdo acidentalmente alterado.

Você poderia insistir dizendo: "Eu jamais faria isso, é só declarar EMPRESA como uma variável LOCAL, conforme nós já estudamos."

Contudo, essa não é a questão. A questão é que existe uma variável PUBLIC totalmente exposta em todas as suas rotinas. Como o acidente vai acontecer, nós não sabemos. Contudo, sabemos que **é possível que um acidente ocorra**. Essa é a questão.

Outra coisa torna esse problema ainda pior. Geralmente um arquivo de configuração não tem somente o campo EMPRESA. Existem outros campos que um arquivo de configuração contém. Arquivos de configuração tem sempre algo referente ao Endereço da empresa, telefone, e-mail, CNPJ, esquema de cores do programa, senhas de banco de dados, etc. Gerenciar uma miríade de valores com variáveis públicas é algo extremamente arriscado.

Conforme nós sabemos, existem várias soluções para esse problema. Mas como o Padrão Singleton resolveria esse problema ?

38.3.2 O Harbour e o padrão Singleton

Formalmente uma classe Singleton é uma classe que não é instanciada como um objeto normal. Na realidade, é como se a classe mesmo fosse um objeto. O Harbour não tem suporte a esse tipo de classe. Como então simular o Padrão Singleton dentro do Harbour ?

Listagem 38.1: Criando uma classe de configuração.
Fonte: codigos/singleton01.prg

```
PROCEDURE MAIN

    LOCAL

RETURN

FUNCTION ConfigSingleton()

    STATIC oObj

    IF EMPTY( oObj )
        USE config SHARED

        CLOSE DATABASE
    ENDIF
```

```
                                                                    16
RETURN oObj                                                         17
                                                                    18
#include "hbclass"                                                 19
                                                                    20
CREATE CLASS MyConfig                                               21
                                                                    22
    EXPORTED:                                                       23
    METHOD GetEmpresa()                                             24
    METHOD GetEndereco()                                            25
    METHOD GetEmail()                                               26
    METHOD GetTelefone()                                            27
    HIDDEN:                                                         28
    DATA cEmpresa INIT ""                                           29
    DATA cEndereco INIT ""                                          30
    DATA cEmail INIT ""                                             31
    DATA cTelefone INIT ""                                          32
                                                                    33
END CLASS                                                           34
                                                                    35
METHOD GetEmpresa() CLASS MyConfig                                  36
RETURN ::cEmpresa                                                   37
```

Crie uma função e instancie o objeto dentro dela através de uma variável estática.

Chame a função no início do programa

Instancie o objeto através da função quando precisar dos dados

38.3.3 Ponto negativo do padrão Singleton

38.4 Conclusão

O Harbour possui funções simples e poderosas que permitem a criação do nosso próprio servidor Web.

Parte VI

Programação Web

39 Programação CGI

> Um projeto honesto deve fluir de dentro para fora, nunca de fora para dentro.
>
> Henry Dreyfuss

Objetivos do capítulo

- Entender os conceitos básicos da programação CGI
- Configurar um servidor WEB para responder à requisições CGI
- Compreender os pontos positivos e negativos da programação CGI
- Diferenciar CGI de fast CGI

39.1 Introdução

Esse capítulo é diferente da maioria dos outros capítulos porque ele **não** necessita apenas do Harbour. Além do Harbour você vai precisar conhecer um pouco sobre servidores Web, porque o Harbour vai trabalhar em conjunto com um deles: o Apache 2[1]. Mas, antes de mais nada você precisa entender como funciona um servidor Web.

39.2 O que é um servidor Web ?

Primeiramente, um esclarecimento necessário nas palavras de Reichard:

> muitas pessoas confundem a Web com a Internet. Entretanto, as duas não são intercambiáveis; a Web refere-se a World Wide Web que é um serviço específico da grande Internet. [Reichard 1998, p. 170]

A Internet compreende uma série de serviços, tais como E-mail, SSH, Telnet, FTP, etc. Um servidor de E-mail é o responsável por prover o nosso acesso à nossa correspondência eletrônica, o servidor de SSH é o responsável pelo terminal seguro que nos conecta a uma máquina qualquer, e assim por diante. O presente capítulo tem por base principal o entendimento do que é um servidor Web, que é o software que responde as requisições dos navegadores Web. Durante o presente capítulo usaremos como servidor de testes o Apache 2 para Windows fornecido pelo pacote XAMPP. Segundo o site do desenvolvedor do pacote, "XAMPP é uma distribuição Apache completamente livre e fácil de instalar contendo MariaDB, PHP e Perl. O pacote de código aberto XAMPP foi configurado para ser incrivelmente fácil de instalar e usar"[2]. O servidor Web Apache é um software que sempre foi líder em sua categoria, desde os primórdios da Web não-acadêmica até os dias atuais. O sucesso foi tanto que a marca Apache deixou de estar associada apenas a um servidor, e passou a ser uma fundação[3] com poder de decisão no destino de inúmeros softwares[4]. Apache também virou o nome de uma licença de software.

Figura 39.1: Logomarca da Apache Software Foudation é baseada no símbolo do Apache.

[1] O Harbour depende de um servidor Web para implementar a programação CGI, mas não necessariamente esse servidor deve ser o Apache 2. Existem diversos servidores no mercado, desde servidores pagos, como o IIS da Microsoft, até servidores open source, como o Ngix e o LightHttpd. Toda a teoria que você aprender com o Apache 2 vai servir para os demais servidores, apenas a implementação irá mudar um pouco.

[2] Fonte : `http://apachefriends.org` acessado em 24-Set-2021.

[3] A Apache Software Foundation (ASF)

[4] Além do Servidor Apache, a fundação mantém o Openoffice, o SpamAssassin , o Jakarta, dentre outros.

39.3 Instalando o XAMPP

39.3.1 Baixe o pacote XAMPP

Acesse o site `http://apachefriends.org/` e baixe o pacote XAMPP, conforme a figura 39.2.

Figura 39.2: Baixando o XAMPP.

Clique em "XAMPP for Windows"e aguarde o download terminar. Para instalar o XAMPP, basta executar o arquivo e confirmar qualquer janela até aparecer o quadro da figura 39.3.

Figura 39.3: Tela inicial de instalação do XAMPP.

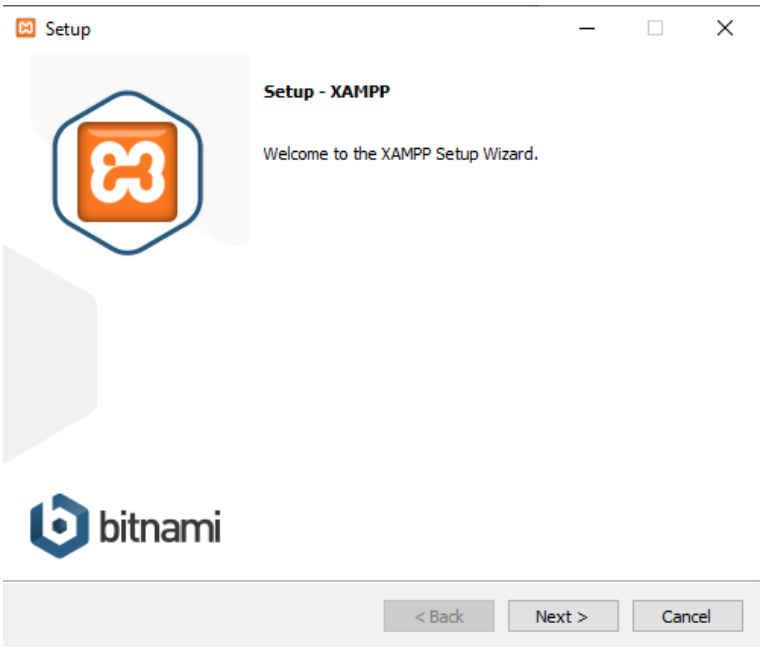


Figura 39.4: O mínimo necessário.

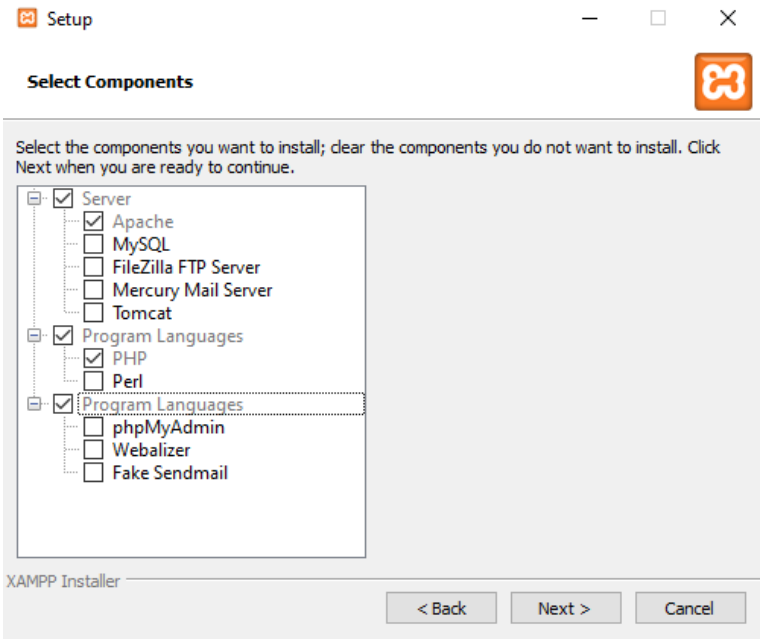


Figura 39.5: Diretório padrão.

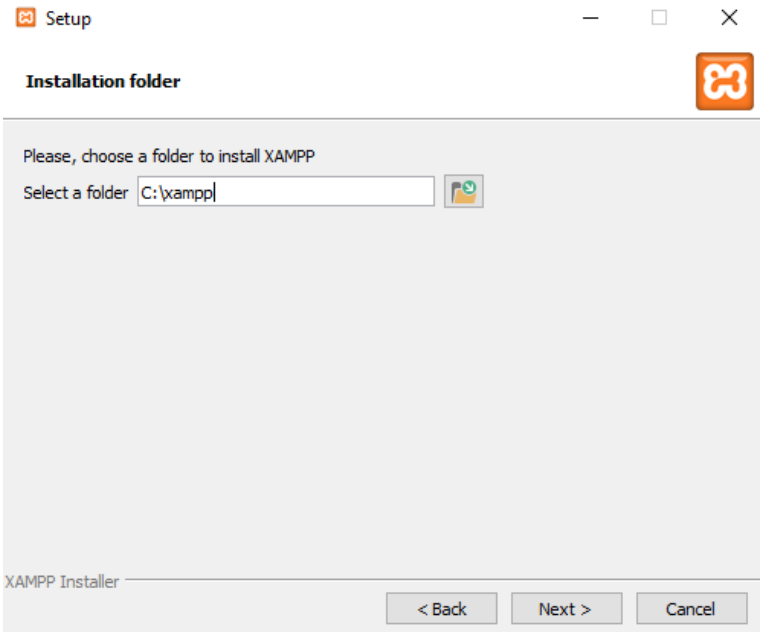


Figura 39.6: Painel de controle.

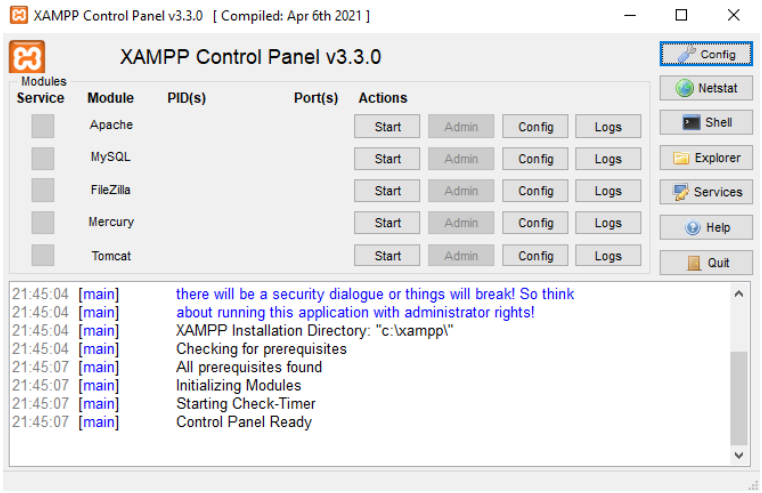
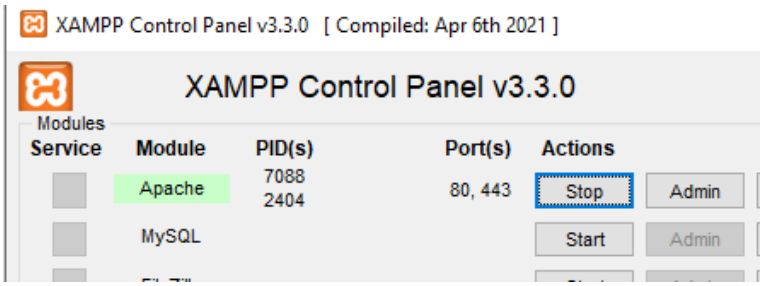


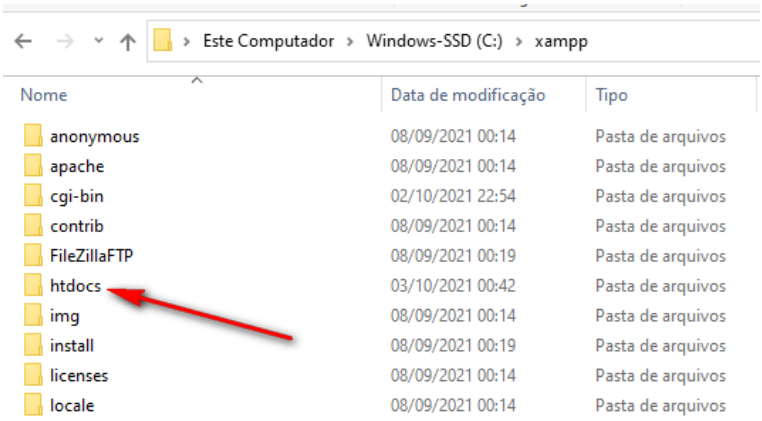
Figura 39.7: Após iniciar o Apache 2.



39.3.2 Pastas virtuais

Agora que já temos o servidor funcionando, vamos entender o seu funcionamento. O servidor Web trabalha com o conceito de pastas virtuais. Não iremos aqui ver esse conceito em detalhes, **inclusive faremos algumas simplificações para facilitar o seu entendimento**. Preste atenção na figura 39.8, ele é o conteúdo da pasta onde o xampp está instalado.

Figura 39.8: Conteúdo de c:\xampp.



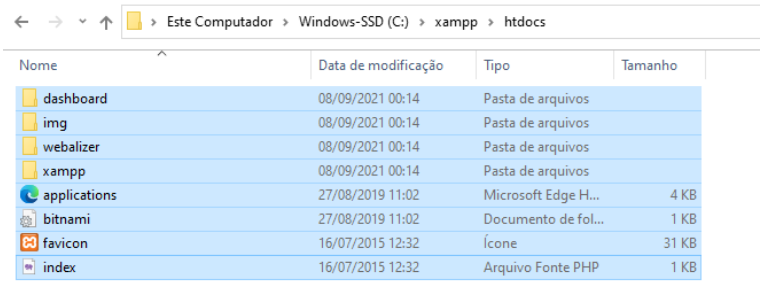
A pasta `htdocs` é o local onde seus arquivos ficarão. Essa pasta (c:\xampp\htdocs.) é conhecida por ser uma "pasta real". Nada demais até agora, todas as pastas de

um sistema operacional qualquer são "pastas reais". Por exemplo: c:\Windows e c:\Xampp são exemplos de pastas reais. Contudo, quando você acessa um servidor Web através do navegador as pastas reais não estão acessíveis, você só terá acesso as "pastas virtuais". O sistema de pastas virtuais do servidor Web tem a sua raiz em c:\xampp\htdocs. Quando você digita o endereço IP do seu servidor Web no navegador, ele vai diretamente para essa pasta. Vamos fazer uns testes agora.

I. Apague o conteúdo da pasta c:\xampp\htdocs

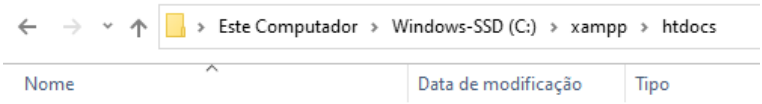
A pasta `htdocs` já vem com alguns arquivos de exemplo, vamos apagar todos eles. Use o gerenciador de arquivos do Windows se desejar. Faça conforme a figura 39.9.

Figura 39.9: Excluindo os arquivos de c:\xampp\htdocs.



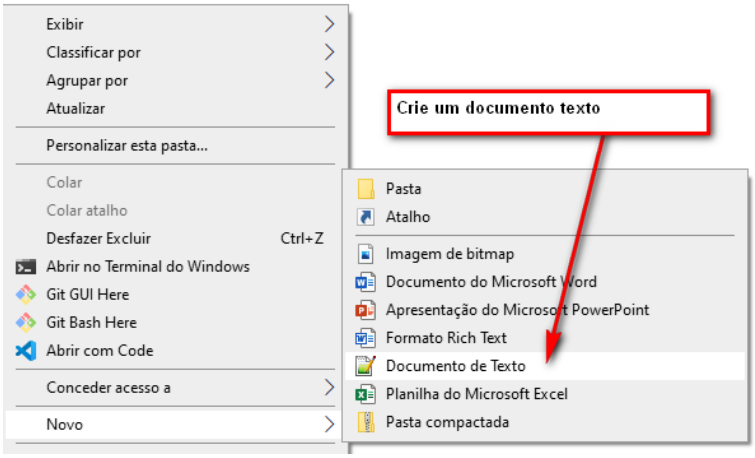
Sua pasta deve ficar conforme a figura 39.10 :

Figura 39.10: Sua pasta c:\xampp\htdocs após a exclusão



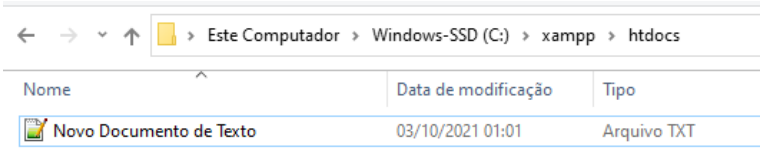
Agora crie um arquivo texto conforme as figuras a seguir. Primeiro clique com o botão direito sobre algum local da pasta e selecione o menu "Novo"e em seguida "Documento de Texto". Isso está ilustrado na figura 39.11.

Figura 39.11: Criando um arquivo (Parte I)



Sua pasta deve ficar assim (figura 39.12.) :

Figura 39.12: Criando um arquivo (Parte II)



II. Acesse o servidor Web através do seu navegador

Primeiramente inicialize o servidor Web no painel do xampp. Faça conforme a figura 39.13.

Figura 39.13: Inicializando o servidor Web

Figura 39.14: Servidor inicializado

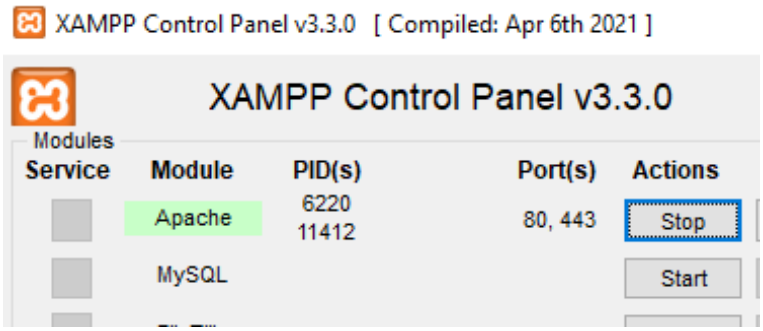
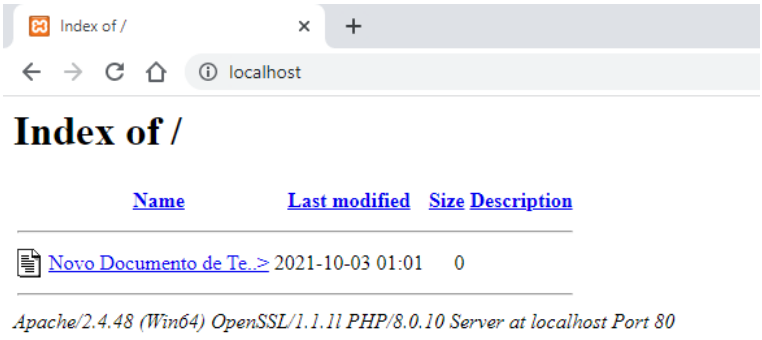


Figura 39.15: Coloque o endereço localhost no seu navegador



O seu servidor Web utiliza o endereço IP da sua máquina para escutar as requisições. Como o servidor está na mesma máquina de testes, vamos utilizar o endereço "localhost". Esse endereço é um padrão internacional, e equivale ao endereço IP 127.0.0.1. Esse endereço também é um padrão, e **sempre** quer dizer "a própria máquina". Caso o servidor Web esteja em outra máquina, o endereço IP deve ser o da máquina onde está o servidor. No nosso estudo sempre usaremos "localhost", pois estamos supondo que você está usando apenas um computador para todos os testes.

39.4 O que é CGI ?

Como você deve ter observado nos exemplos da seção anterior, sempre que alguém clica sobre um arquivo, o servidor Web exibe o conteúdo do mesmo. Esse é o comportamento padrão do servidor Web. Se o arquivo for do tipo executável, o servidor Web enviará o arquivo para o usuário (download). Com o passar do tempo, surgiu a necessidade de prover algum tipo de comportamento dinâmico antes do servidor Web exibir o conteúdo para o usuário. Por exemplo, se o usuário buscar um determinado produto, de alguma forma o servidor Web deve contactar o banco de dados e retornar o resultado da pesquisa. Esse recurso é implementado através da CGI (Common Gateway Interface). Um programa CGI pode ser um script ou um executável. Quando o usuário clica sobre esse programa, ou quando ele é "chamado"através de uma página HTML, o programa não é "baixado"(download), mas sim executado. Essa é, basicamente, a característica básica da CGI. Reichard [Reichard 1998, p. 501] resume o processo no seguinte:

1. Uma requisição é feita ao servidor Web (por exemplo, o Apache) por um navegador Web (por exemplo, o Edge).
2. A requisição é enviada a um script CGI.
3. O script CGI processa a requisição.
4. As informações geradas pelo script CGI são formatadas (geralmente HTML) e enviadas para o navegador requisitante.

39.5 Criando um programa CGI usando Harbour

Criar um programa CGI não é difícil, o principal obstáculo é entender as regrinhas que esse tipo de programação impõe. Vamos listar as regras a seguir, elas valem para qualquer linguagem.

1. O script ou executável deve estar em uma pasta habilitada para execução de CGI.
2. O script ou executável deve iniciar informando o tipo de documento que ele está gerando, seguido de duas linhas em branco.
3. Os comandos de impressão devem ser enviados diretamente para a saída padrão do sistema operacional, sem nada intermediando.

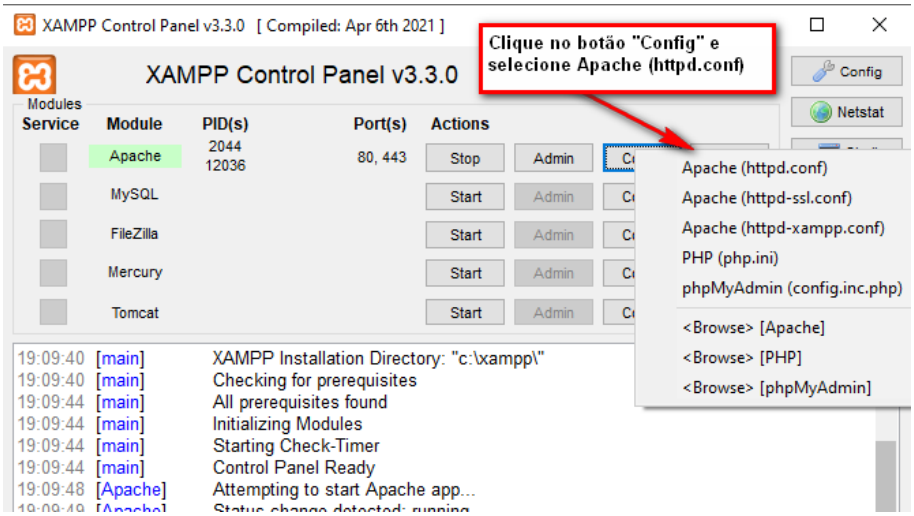
Vamos analisar caso a caso.

39.5.1 O script ou executável deve estar em uma pasta habilitada para execução de CGI

Essa configuração é feita no servidor Web, se o seu programa executável não estiver na pasta utilizada para CGI, o servidor tentará "baixar"o arquivo em vez de executar. No nosso exemplo, estamos usando o Apache 2 que é distribuído no pacote Xampp. Agora, atenção: **o Apache 2 já vem com uma pasta habilitada para a execução de CGI. Você pode alterar essa pasta.** . Ou seja, os passos a seguir devem ser seguidos apenas se você quiser mudar a pasta padrão. Se você não for mudar a pasta, então copie o seu executável para a pasta C:\xampp\cgi-bin e passe para a próxima seção 39.5.2.

A. Abra o arquivo httpd.conf

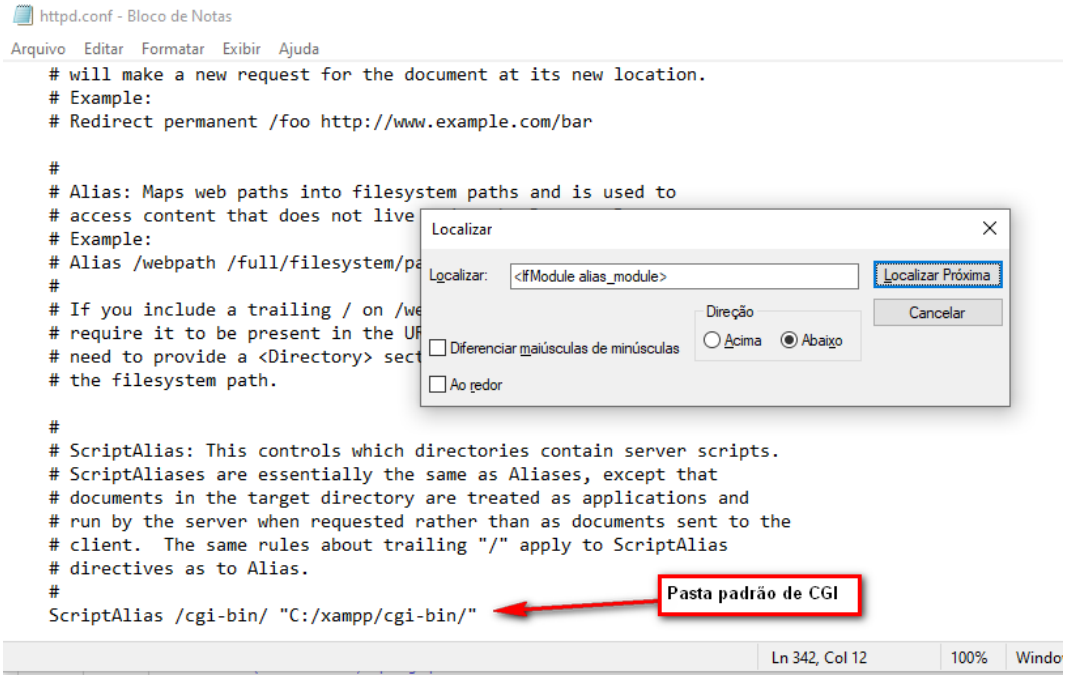
Figura 39.16: Abrindo o httpd.conf.



B. Altere a pasta padrão se for o caso

Faça uma busca no arquivo pelo termo `<IfModule alias_module>` conforme a figura 39.17.

Figura 39.17: Abrindo o httpd.conf.



A pasta padrão de CGI é a `c:\xampp\cgi-bin`, você pode mudar para outra pasta se desejar. Caso queira manter essa pasta, você deve copiar o executável para ela.

Importante: caso você mude o nome da pasta, você vai precisar inserir essas linhas :

```
<Directory "C:/caminho/para/nova/pasta/">
```

```
    AllowOverride All
    Options Indexes FollowSymLinks ExecCGI
    AllowOverride All
    Require all granted
</Directory>
```

Supondo que o seu novo caminho é `C:/caminho/para/nova/pasta/` o aspecto final do seu arquivo será :

```
<IfModule alias_module>
    #
    # Redirect: Allows you to tell clients about documents that used to
    # exist in your server's namespace, but do not anymore. The client
    # will make a new request for the document at its new location.
    # Example:
    # Redirect permanent /foo http://www.example.com/bar

    ScriptAlias /cgi-bin/ "C:/caminho/para/nova/pasta/"
</IfModule>


<Directory "C:/caminho/para/nova/pasta/">
    AllowOverride All
    Options Indexes FollowSymLinks ExecCGI
    AllowOverride All
    Require all granted
</Directory>
```

Dica 175

Gostaríamos de oferecer uma configuração que sirva para todos os servidores, mas isso não é possível. A seguinte configuração foi testada com sucesso no Apache versão 2.4.49. Como o objetivo desse capítulo é aprender a programação CGI, vamos evitar complicações desnecessárias. O ideal é você copiar os seus executáveis para a pasta c:\xampp\cgi-bin caso tenha algum problema. Caso você seja usuário do Linux, verifique as permissões de execução para o seu executável. O Linux geralmente utiliza um usuário chamado nobody ou www-data para executar os seus scripts. Verifique a documentação do Apache e os logs de erro do servidor. O log de erro do Apache chama-se error.log e pode ser acessado no painel do xampp através do botão Logs.

39.5.2 Criando o primeiro programa CGI

Um programa CGI feito em Harbour pode ser visto na listagem 39.1 a seguir :

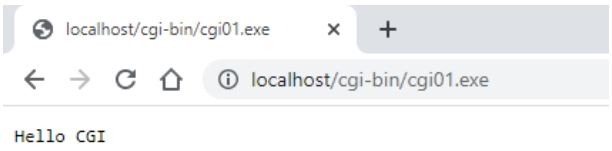
Listagem 39.1: Meu primeiro CGI.
Fonte: codigos/cgi01.prg

```
REQUEST HB_GT_CGI_DEFAULT                                                    1
```

```
PROCEDURE Main

    OutStd( "Content-type: text/plain" + hb_eol() + hb_eol() )
    OutStd( "Hello CGI" )

RETURN
```

Para executar o programa abra o seu navegador e digite o endereço `http://localhost/cgi-bin/cgi01.exe`, conforme a figura 39.18.

Figura 39.18: Executando o programa.



O script ou executável deve iniciar informando o tipo de documento que ele está gerando, seguido de duas linhas em branco

39.6 Conclusão

O Harbour possui funções simples e poderosas que permitem a criação do nosso próprio servidor Web.

40 Criando o nosso próprio servidor

> A web evoluiu para um motor de injustiça e divisão; balanceada por poderosas forças que usam-na para seus próprios interesses(agendas).
>
> Tim Berners-Lee

Objetivos do capítulo

- Criar um servidor de web simples
- Aprender a mudar os cabeçalhos Mime

40.1 Introdução

Esse capítulo é dedicado a criação de um servidor Web.
Para compilar os exemplos desse capítulo faça assim :

.:Resultado:.

```
hbmk2 arquivo hbhttpd.hbc
```

40.2 Um servidor Web bem simples

Vamos começar com um servidor web bem simples, com o mínimo necessário para que ele funcione. Antes de analisarmos o código, vamos ver o seu funcionamento básico.

40.2.1 Exemplo de uso do servidor

Help do servidor

O nosso servidor funciona na linha de comando. No nosso exemplo inicial chamaremos ele de "httpd01.exe"(no Linux, apenas "httpd01"). Quando digitamos o nome do executável e teclamos enter, o nosso servidor exibe um pequeno Help e sai.

.:Resultado:.

```
//start              Start
//stop               Stop
```

Iniciando e parando o servidor

Para iniciar o nosso servidor use o parâmetro //start e para parar use //stop. Você vai precisar de duas janelas de prompt de comando abertas na mesma pasta, porque o servidor ao ser iniciado bloqueia o terminal. Mais na frente veremos como evitar esse bloqueio de terminal, mas como queremos abstrair somente o código essencial, vamos abrir duas janelas.

Ao iniciar o servidor

Quando formos iniciar o nosso servidor ele irá exibir uma mensagem simples de "Ok"na tela.

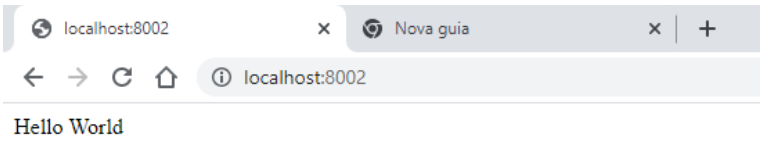
.:Resultado:.

```
httpd01 //start
Ok
```

Caso você esteja usando o Windows pode ser necessário liberar a porta 8002 no Firewall. Quando você executa pela primeira vez o próprio Windows pergunta se você deseja liberar essa porta.

Agora vá no navegador e digite "localhost:8002"na barra de endereços.

Figura 40.1: Acessando o servidor pelo navegador.



Parando o servidor

Na outra janela do Prompt de comando digite

.:Resultado:.

```
httpd01 //stop
```

Quando você voltar para o primeiro prompt verá que ele já está liberado para uso. O seu servidor agora está "off-line".

40.2.2 Análise do código fonte

Antes de iniciarmos a análise propriamente dita do nosso código, é bom ressaltar que o nosso servidor deve ser entendido como sendo um programa desenvolvido para funcionar na Web, sobre o protocolo HTTP. Ele é um programa que não depende de um servidor instalado, porque ele já contém o próprio servidor no seu código.

O código completo da versão inicial

Listagem 40.1: Versão inicial do nosso servidor.
Fonte: codigos/httpd01.prg

```
#require "hbhttpd"
PROCEDURE Main

     LOCAL oServer, hConfig, hMount

     hb_CdpSelect("UTF8")

     IF hb_argCheck( "stop" )
          MemoWrit( ".uhttpd.stop", "" )
          RETURN
     ELSEIF hb_argCheck( "start" )
          ? "Ok"
     ELSE
          ? "  //start             Start "
          ? "  //stop              Stop"
```

```
          RETURN
     ENDIF

     hMount := { => }
     hMount[ "/" ] := {|| Start() }

     hConfig := { ;
     "FirewallFilter"  => "", ;
     "Port"  => 8002, ;
     "Idle"  => {| o | iif( hb_vfExists( ".uhttpd.stop" ),;
     ( hb_vfErase( ".uhttpd.stop" ), o:Stop() ), NIL ) }, ;
     "Mount" => hMount }

     oServer := UHttpdNew()
     IF ! oServer:Run( hConfig )
          ? "Server error:", oServer:cError
          RETURN
     ENDIF

RETURN
/***************/
FUNCTION Start()

RETURN "Hello World"
/***************/
```

Recebendo os parâmetros de linha de comando

Listagem 40.2: Versão inicial do nosso servidor.
Fonte: codigos/httpd01.prg

```
     IF hb_argCheck( "stop" )
          MemoWrit( ".uhttpd.stop", "" )
          RETURN
     ELSEIF hb_argCheck( "start" )
          ? "Ok"
     ELSE
          ? "  //start             Start "
          ? "  //stop              Stop"
          RETURN
     ENDIF
```

Configurando as pastas virtuais

Listagem 40.3: Configurando as pastas virtuais.
Fonte: codigos/httpd01.prg

```
     hMount := { => }
     hMount[ "/" ] := {|| Start() }
```

Configurando o comportamento do servidor

Listagem 40.4: Configurando o comportamento do servidor.
Fonte: codigos/httpd01.prg

```
hConfig := { ;
"FirewallFilter"  => "", ;
"Port"  => 8002, ;
"Idle"  => {| o | iif( hb_vfExists( ".uhttpd.stop" ),;
( hb_vfErase( ".uhttpd.stop" ), o:Stop() ), NIL ) }, ;
"Mount" => hMount }
```

Executando o servidor

Listagem 40.5: Executando o servidor.
Fonte: codigos/httpd01.prg

```
oServer := UHttpdNew()
IF ! oServer:Run( hConfig )
     ? "Server error:", oServer:cError
     RETURN
ENDIF
```

40.3 Conclusão

O Harbour possui funções simples e poderosas que permitem a criação do nosso próprio servidor Web.

Parte VII

Programação GUI

41 Programação GUI

> Não é que o desconhecido seja complicado; o que já conhecemos é que nos parece simples.
>
> Will Rogers

Objetivos do capítulo

- Entender o que é uma GUI.

41.1 Introdução

Lembro-me da primeira vez em que vi uma máquina com o Windows. Até então eu nunca tinha visto uma interface gráfica, a única coisa que eu conhecia era o MS-DOS e uma rede chamada Novell Netware 3. Me senti pouco a vontade com aquele ambiente de janelas, e realmente não conseguia entender nada do que via. Eu vivia em um ambiente onde o raciocínio era linear. Cada programa era executado um de cada vez, e aquela visão de uma calculadora, uma planilha e um editor de textos me deixou realmente desconfortável. Hoje em dia talvez seja difícil transmitir o que era viver em um mundo onde só existia o MS-DOS. Alguns colegas meus lembram daquela época com um certo saudosismo. Realmente as coisas eram mais simples, mas eu não trocaria o ambiente de hoje pela simplicidade de ontem. Ter que sair do editor de textos para poder compilar o programa, fazer malabarismos com programas residentes na memória, como o editor SideKick e calcular cada byte a ser usado, senão a memória RAM era insuficiente, são coisas que não me deixaram muita saudade. Quando o Windows chegou no mercado, foi mais ou menos como um tsunami invadindo uma praia tranquila. No início ele recebia o nome de "ambiente operacional", porque o sistema operacional na realidade era o MS-DOS. Depois, com as primeiras versões do Windows 95, finalmente tinhamos um sistema operacional e a velha interface "tela preta"ficou restrito a uma dentre tantas janelas.

Nesse capítulo iremos estudar as interfaces gráficas. A sigla GUI (Graphical User Interface) é mais do que simplesmente o Windows. Existem outras interfaces gráficas, como o GNome e o KDE do Linux, entretanto o presente capítulo usará o Windows da Microsoft para explicar, através de exemplos, o que é uma GUI. De acordo com Pappas e Murray

> a interface gráfica é a mais notável e importante capacidade oferecida pelo Windows. A interface consistente com o usuário utiliza figuras para representar unidades de disco, arquivos, subdiretórios e diversos comandos e ações do sistema operacional. [...] A maioria dos programas Windows oferece uma interface de teclado e uma interface com o mouse. Embora muitas funções dos programas Windows possam ser controladas via teclado, o uso do mouse é frequentemente mais simples para muitas tarefas[Papas e Murray 1994, p. 661].

Não iremos aqui nos demorar mais na introdução ao Windows, porque ela já deve ser conhecida de todos aqui, nas próximas seções iremos usar o Harbour para criar programas que se utilizam dessas interfaces, contudo alguns conceitos se fazem necessários para que você entenda o que vai mudar daqui para frente.

41.2 Como o seu programa irá se transformar em um programa Windows

Se você estudou a interface modo texto (TUI) fornecida pelo Harbour, verá que todos os controles são criados pela própria linguagem. Cada comando @ ... GET ou cada DBEDIT() são elementos que o próprio Harbour cria "do zero"para que você possa utilizar esses controles. Quando você trabalha com GUI você vai solicitar ao Windows, através de uma mensagem, um elemento gráfico. Isso muda tudo. Antigamente, as

interações do seu programa com o sistema operacional eram feito de forma mais ou menos direta, mas agora as coisas mudaram. O Windows vai intermediar cada uma das solicitações. Isso se dá através de mensagens.

Em uma interface modo-texto (TUI), o próprio Harbour cria os controles, já em uma interface gráfica (GUI), o Harbour vai pedir que o Sistema Operacional crie tais controles. A consequência prática disso são controles padronizados, mas altamente customizáveis. Uma outra consequência disso é que os seus controles modo-texto não servirão para uma aplicação Windows. Como você vai trabalhar com uma interface gráfica, e os nossos controles são modo texto e são executados linearmente, pouca coisa se aproveita[1].

41.3 O que é HMG ?

O Harbour não vem nativamente configurado para trabalhar com interfaces gráficas e para conseguir essa interação, ele se utiliza de uma biblioteca externa. A bibloteca que usaremos chama-se "Harbour MiniGui"(HMG)[2], ela foi desenvolvida pelo programador e professor Roberto Lopez, e agora conta com uma comunidade viva e atuante que pode ser contactada em `www.hmgforum.com`. A HMG é uma das bibliotecas que o Harbour se utiliza para gerar tais interfaces[3]. Escolhemos o HMG pela sua facilidade e simplicidade. Nas palavras do próprio autor :

> Quando o Windows se tornou o sistema operacional padrão para PC, fiquei desapontado com as novas linguagens de programação desenvolvidas para ele. Na minha opinião, na maioria dos casos, essas novas linguagens constituíram um retrocesso na evolução da informática, pois, o código se tornou maior, menos intuitivo e mais complexo do que na geração anterior. O xBase original era poderoso, simples e intuitivo. São as encarnações do Windows e as novas ferramentas principais do Windows, que perderam esses recursos, tornando as coisas cada vez mais difíceis. Em 2000, descobri o projeto Harbour, um compilador baseado em Clipper multiplataforma totalmente novo e gratuito. Algum tempo depois, comecei a experimentar a interface Harbour-C e o compilador MinGW. Finalmente percebi que meu antigo "sonho"poderia ser possível. Crie uma ferramenta gratuita do Windows com o espírito xBase original (poderoso, intuitivo e fácil de usar). O HMG pretende ser "natural". A ideia básica por trás disso, é deixar o programador expressar suas ideias de forma direta, usando um código simples, fácil de escrever e fácil de manter, tornando seu trabalho mais produtivo. Fonte: `https://sites.google.com/site/hmgweb/`. Acessado em 28-Set-2021.

[1] Existem bibliotecas que permitem que um código fonte escrito em uma interface TUI seja executado em uma interface GUI. Particularmente, não acho uma boa ideia isso. Somente se você tiver um sistema grande é que vale a pena usar esse artifício, mas se você for criar algo novo, ou migrar um sistema não tão grande, eu aconselho você a programar tudo em modo gráfico (GUI) e não aproveitar os controles TUI. Há quem pense diferente de mim, como disse, essa é uma opinião pessoal.

[2] Antigamente o nome dessa lib era somente Minigui. Contudo, já existia um software com esse nome. Tratava-se de uma lib para dispositivos embarcados sem relação alguma com o Harbour. Alguns anos depois, a lib foi rebatizada de Harbour MiniGui e nós a chamamos simplemente de HMG.

[3] O Harbour possui várias outras libs independentes, como a HWGui, QtContribs, HMG Extend e outras.

Aconselhamos você a descobrir a HMG também através dos inúmeros exemplos que estão na pasta SAMPLES. Dentro dessa pasta tem inúmeras subpastas. Após ler a nossa introdução, vá para a subpasta "Basics"e compile os exemplos usando a IDE.

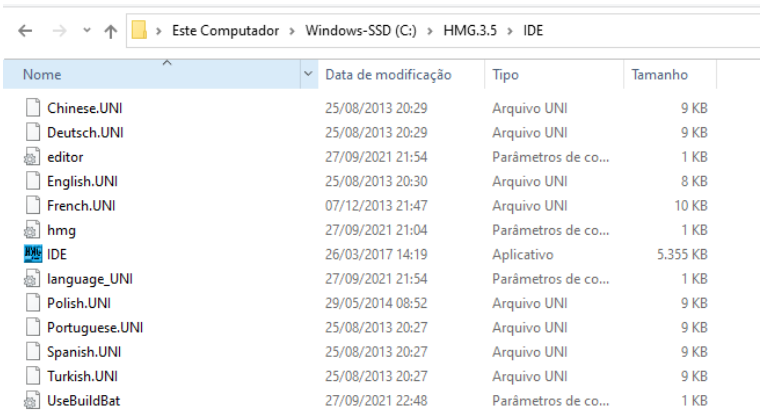
41.4 Hello Windows

Se você instalou o Harbour seguindo as instruções do capítulo 2, você já tem a HMG instalada na sua máquina. Você pode gerar um programa para windows usando a linha de comando, mas a HMG disponibiliza um Ambiente de Desenvolvimento Integrado[4] (IDE) para que você possa montar as suas telas clicando e arrastando componentes. Usaremos a IDE padrão do HMG porque ela facilita a confecção das telas e ajuda muito durante o processo de desenvolvimento.

41.4.1 Localizando a IDE e criando um atalho

A IDE da HMG fica na pasta IDE, dentro da pasta principal onde você instalou a HMG. A figura 41.1 mostra onde a IDE se localiza em uma instalação padrão da HMG.

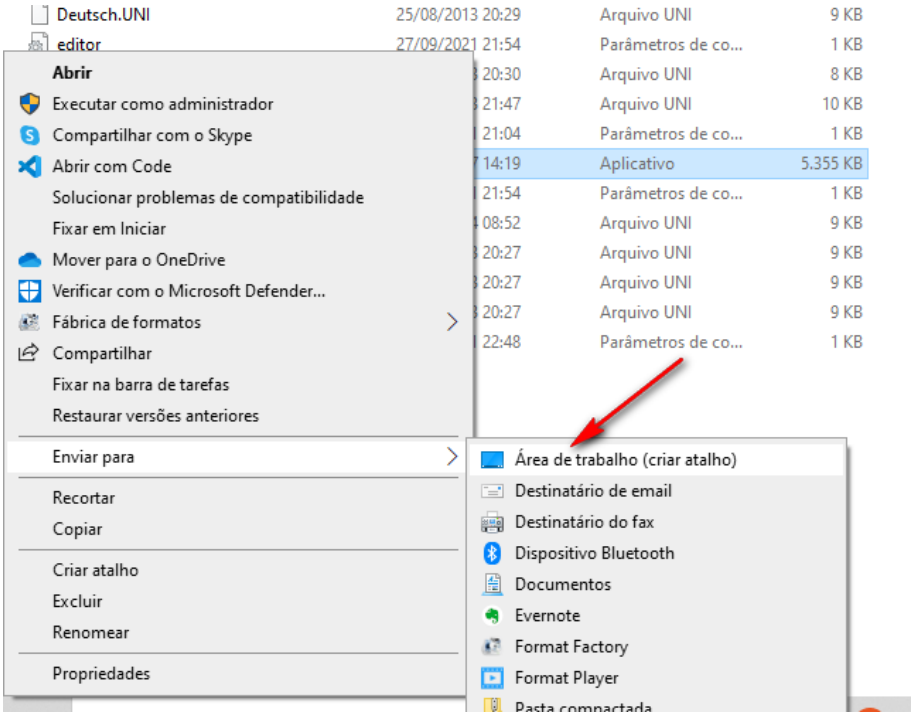
Figura 41.1: Localizando a IDE da HMG.



Para facilitar daqui para frente, crie um atalho para a IDE.exe na sua área de trabalho. Para criar um atalho, basta clicar com o botão direito sobre o arquivo IDE.exe, e selecionar a opção "Enviar para -> Área de Trabalho (Criar atalho)"conforme a figura 41.2.

[4]Integrated Development Environment

Figura 41.2: Criando um atalho para a HMG.



Pronto, agora procure salvar os seus exemplos na nossa pasta de trabalho :

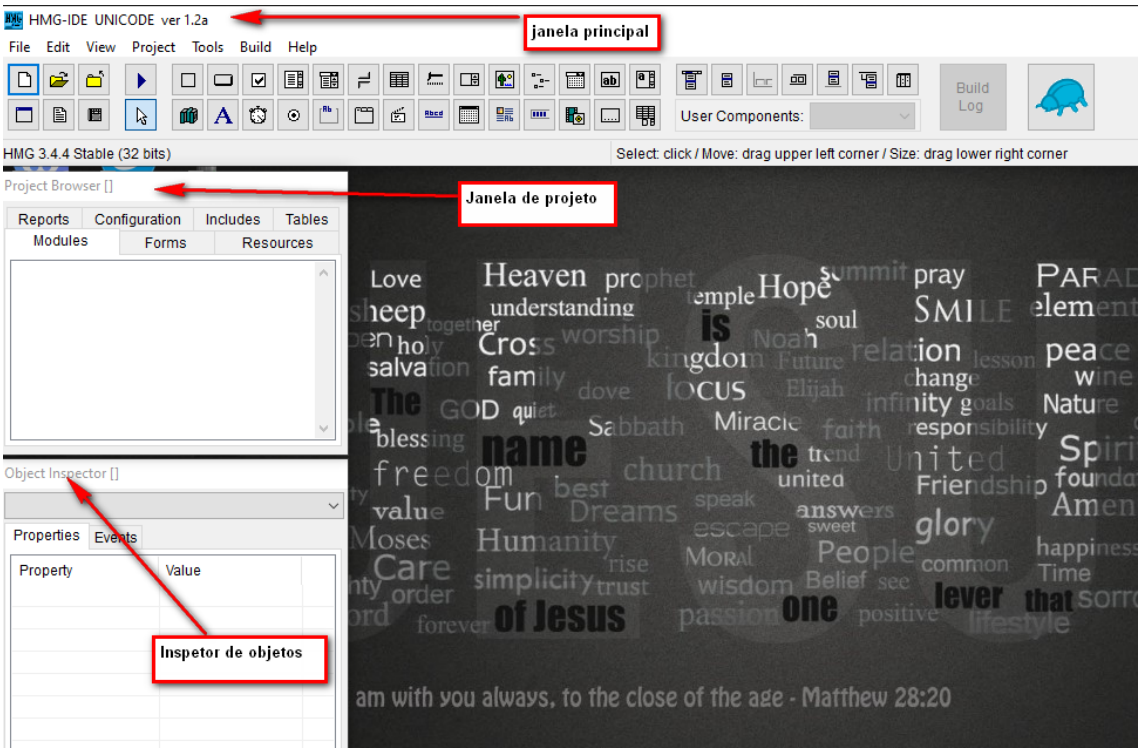
"C:\Curso_Harbour\pratica"

Contudo, tem um detalhe: você vai ter que criar uma subpasta para cada exemplo. Isso porque um programa feito usando a IDE irá criar alguns arquivos extras, todos eles são velhos conhecidos nossos, mas não convém que eles fiquem em uma mesma pasta, porque eles iriam se sobrepor uns aos outros.

41.4.2 Abrindo a IDE e conhecendo o ambiente

Agora execute o programa IDE.exe clicando duas vezes sobre o atalho criado. A tela inicial irá se parecer com a figura 41.3.

Figura 41.3: A IDE e suas 3 janelas.



Note que são 3 janelas flutuantes. Vamos descrever rapidamente cada uma delas:

1. Janela principal: essa janela contém o menu principal e a paleta de componentes do windows. A grande maioria desses componentes são fornecidos pelo próprio Windows. A HMG faz a intermediação e torna cada um desses componentes disponíveis para o Harbour.

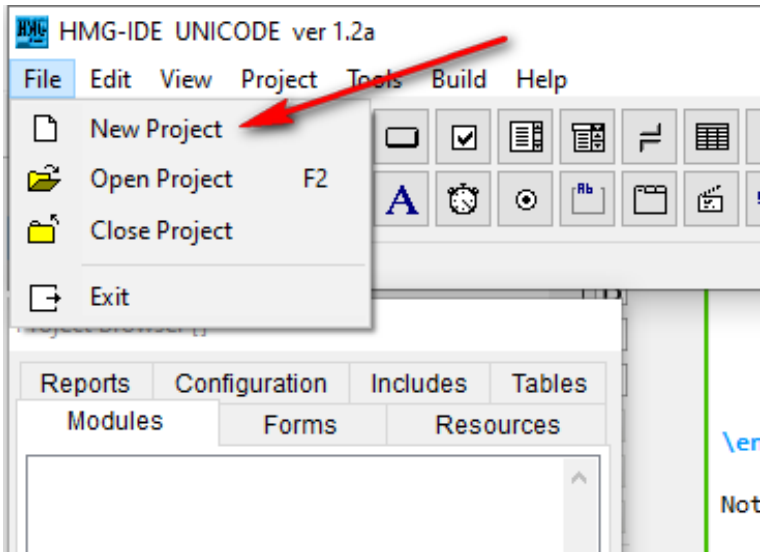
2. Janela de projeto: quando um projeto for aberto, você poderá navegar pelos arquivos do seu projeto usando essa janela.

3. Inspetor de Objeto: você vai poder alterar as propriedades de cada objeto através dessa janela.

41.4.3 Criando o primeiro projeto

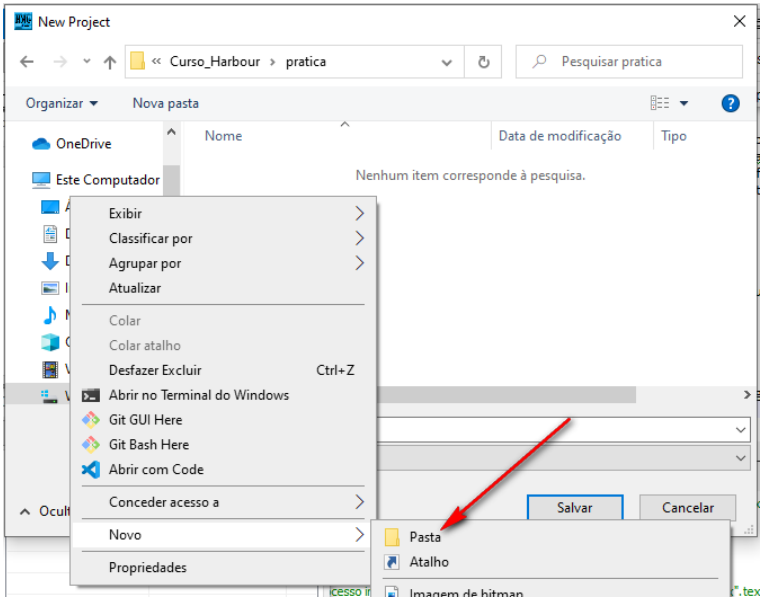
Para criar o seu primeiro projeto clique no menu "File"e selecione "New Project", conforme a figura 41.4.

Figura 41.4: Novo projeto.



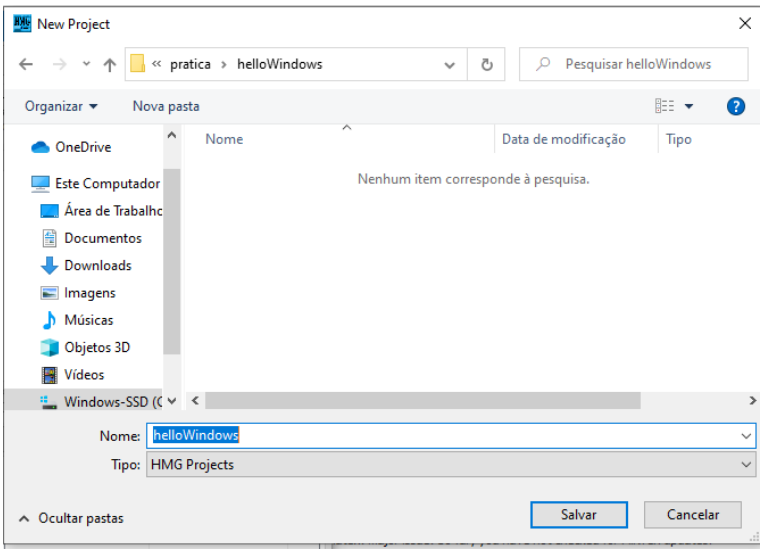
A IDE irá agora pedir a pasta onde o seu projeto será gravado. É necessário que a pasta já exista, por isso faça como a figura 41.5, crie uma pasta chamada "helloWindows"dentro da nossa pasta "pratica".

Figura 41.5: Crie a pasta destino.



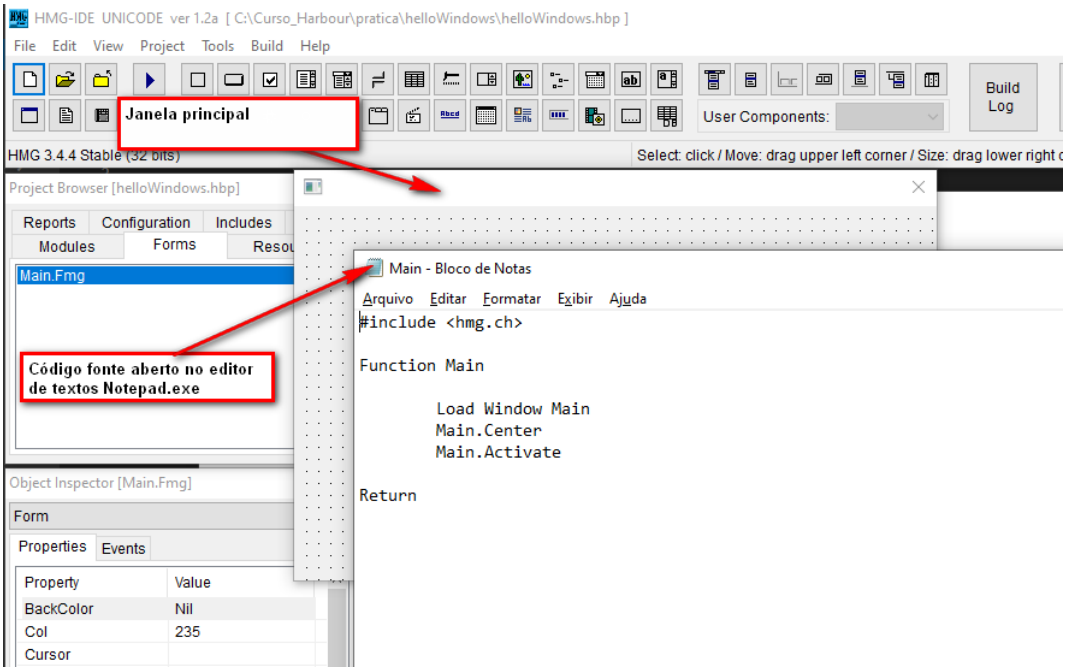
"Entre"na pasta recém-criada e digite "helloWindows"no nome do arquivo a ser criado, conforme a figura 41.6.

Figura 41.6: Atribua um nome ao projeto.



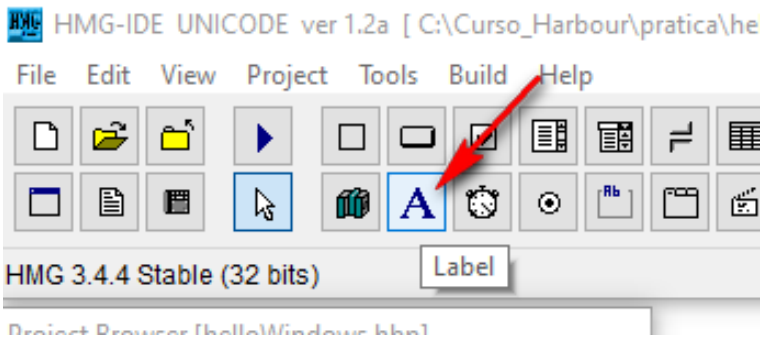
Ao fazer isso, uma janela surgirá e também o código fonte do seu projeto. O código fonte do seu projeto será aberto no Bloco de Notas do Windows, mais na frente nós veremos como mudar o editor padrão.

Figura 41.7: Projeto salvo.



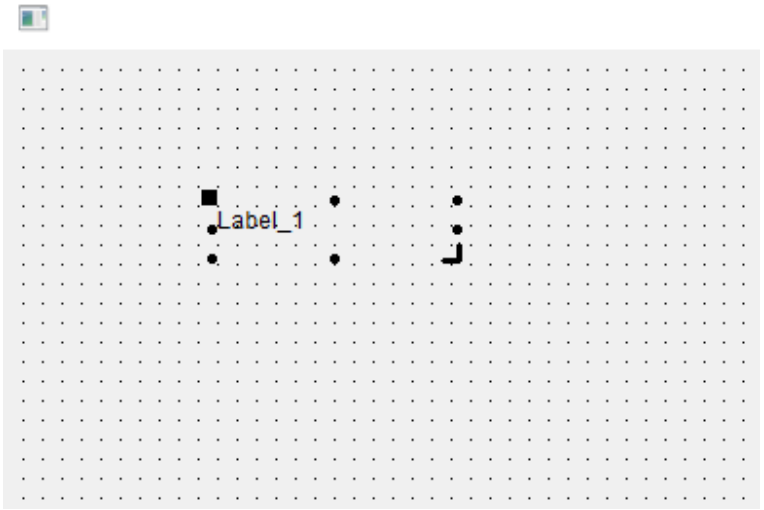
Vamos fazer um pequeno teste. Clique em um controle label, na Janela principal, conforme a figura 41.8. **Importante:** não arraste o controle para a janela. Simplesmente clique e solte.

Figura 41.8: Inserindo um label.



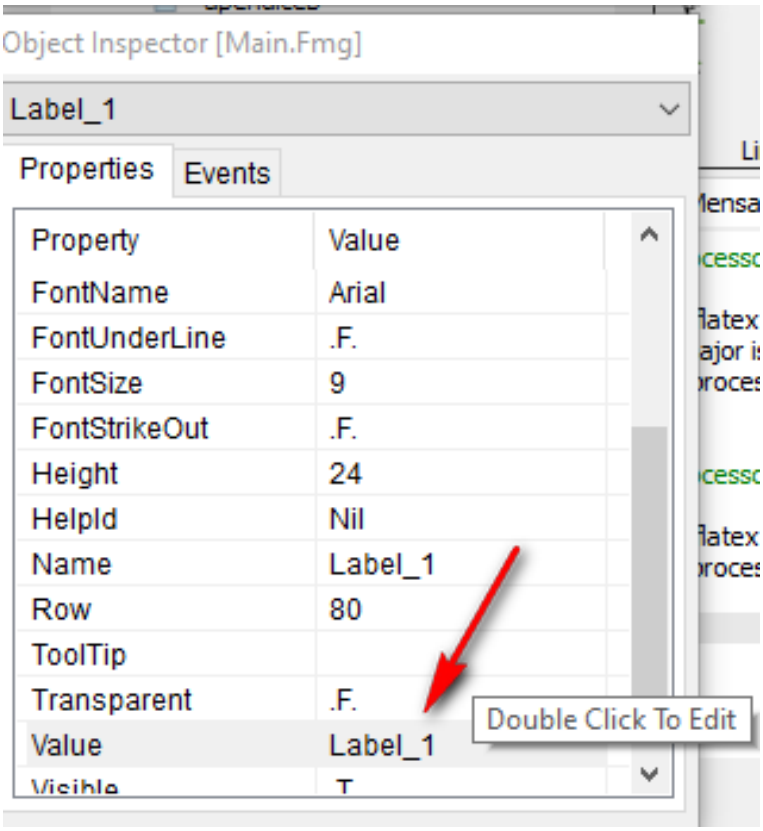
Agora clique sobre a janela do seu projeto, em qualquer lugar na sua área interna (não na barra de títulos), conforme a figura 41.9

Figura 41.9: Inserindo um label (Parte II).



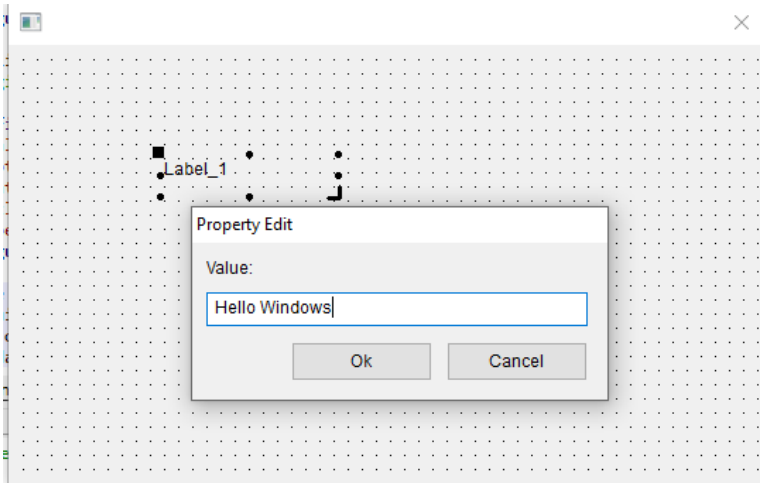
Note que o Harbour acabou de inserir um controle Label na sua janela. Um controle do tipo Label é usado para inserir textos na sua aplicação. Ele é muito usado para criar nomes de campos (se você estudou TUI, é equivalente ao comando @ ... SAY). Você vai poder alterar o conteúdo do que está escrito. Para fazer isso, deixe o Label selecionado, vá no Inspetor de Objetos e clique duas vezes sobre a propriedade Value, conforme a figura 41.10.

Figura 41.10: Clique duas vezes para editar uma propriedade.



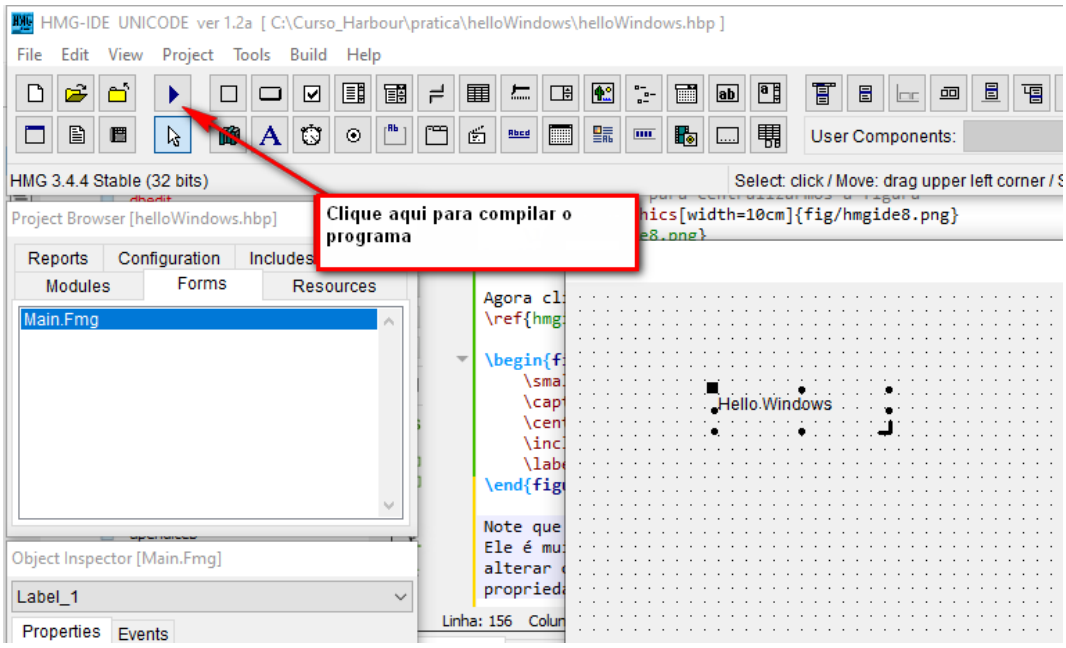
Após o duplo clique, uma janela se abre e você pode digitar o conteúdo de Value.

Figura 41.11: Digitando o conteúdo da propriedade Value.



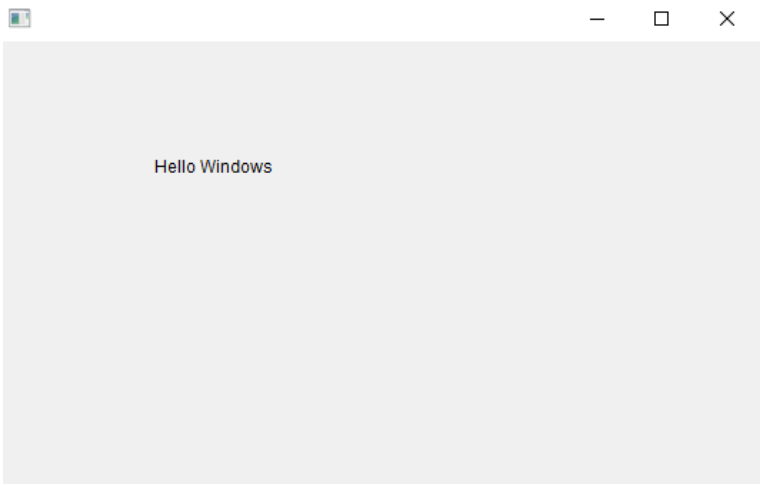
Pronto, agora clique na seta, conforme a figura 41.12, para compilar o programa. Após o clique, a IDE irá perguntar se você deseja salvar as alterações em disco. Confirme o salvamento.

Figura 41.12: Compilando o programa



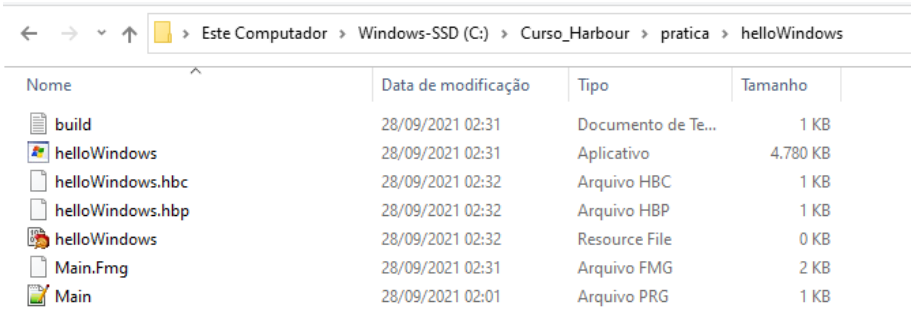
Pronto, após o processo de compilação, o programa recém-gerado será aberto automaticamente (figura 41.13).

Figura 41.13: .



Faça agora o seguinte teste, abra o Gerenciador de Arquivos do Windows e navegue até a pasta onde o programa foi gerado. Veja que alguns arquivos foram criados.

Figura 41.14: .



Os arquivos criados :

1. build.log : Arquivo log com o resultado do hbmk2. Lembre-se, ainda estamos usando o hbmk2 e o processo de compilação ainda é o mesmo. Apenas estamos fazendo tudo graficamente através da IDE.
2. helloWindows.exe : É o programa gerado.
3. helloWindows.hbc : O nosso arquivo de configuração auxiliar do hbmk2.
4. helloWindows.hbb : O arquivo que contém a lista de arquivos fonte (.prg) do projeto.
5. helloWindows.rc : Arquivo de recursos. Veremos mais sobre ele mais tarde. É nesse arquivo que ficam os icones da aplicação, por exemplo.
6. Main.fmg : A janela.
7. Main.prg : O código fonte que usará a janela de mesmo nome.

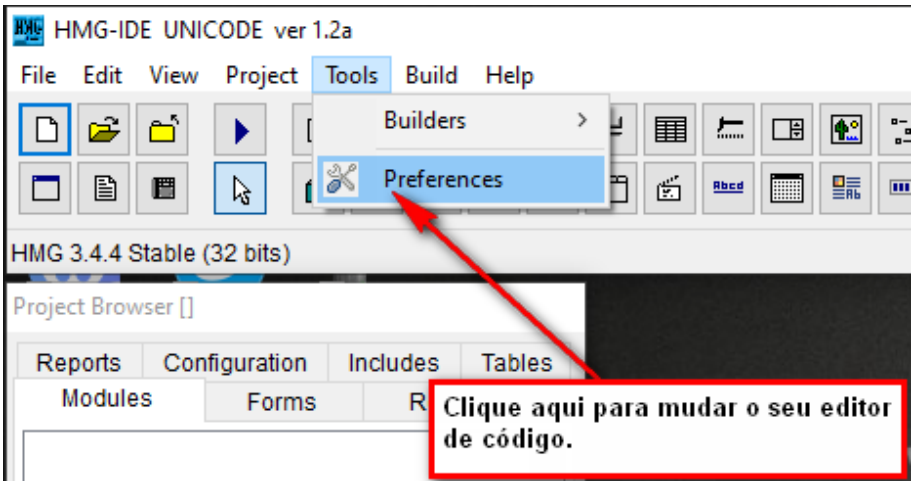
Dica 176

Se você já programou em Delphi, o arquivo .fmg equivale ao arquivo .dfm do Delphi. Mas se você não tem essa experiência,não se preocupe porque explicaremos tudo com detalhes nas próximas seções.

41.5 Analisando o programa helloWindows

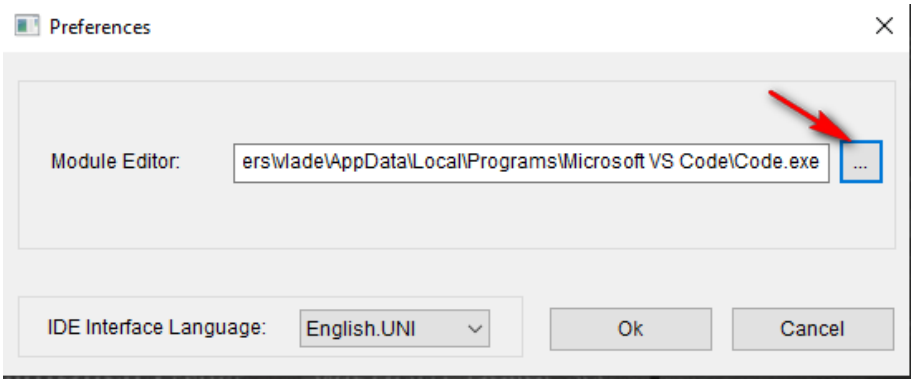
Antes de mais nada, vamos configurar um editor de textos para ser usado durante o nosso desenvolvimento. Nós usaremos o editor Visual Studio Code, mas você é livre para escolher o seu editor. Para mudar o editor, vá no menu Tools e clique em Preferences, conforme a figura 41.15.

Figura 41.15: Mudando o editor de código



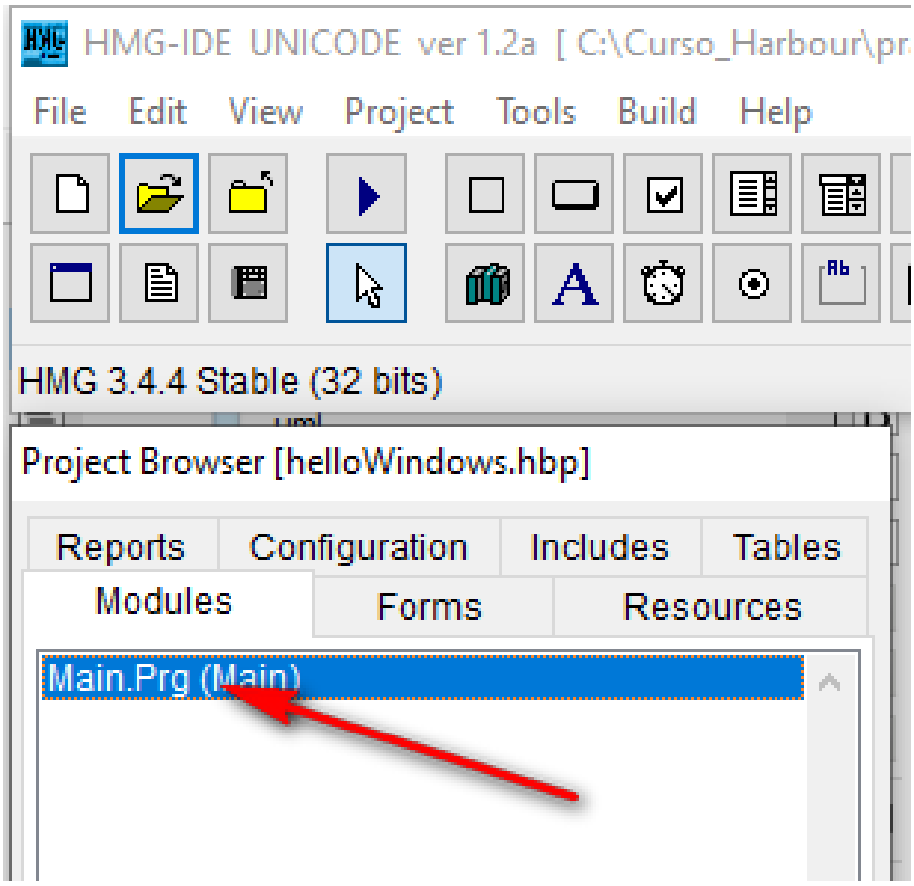
Em seguida clique no botão de busca (Figura 41.16) e selecione o executável. No caso do Visual Studio Code ele chama-se "code.exe".

Figura 41.16: Mudando o editor de código II



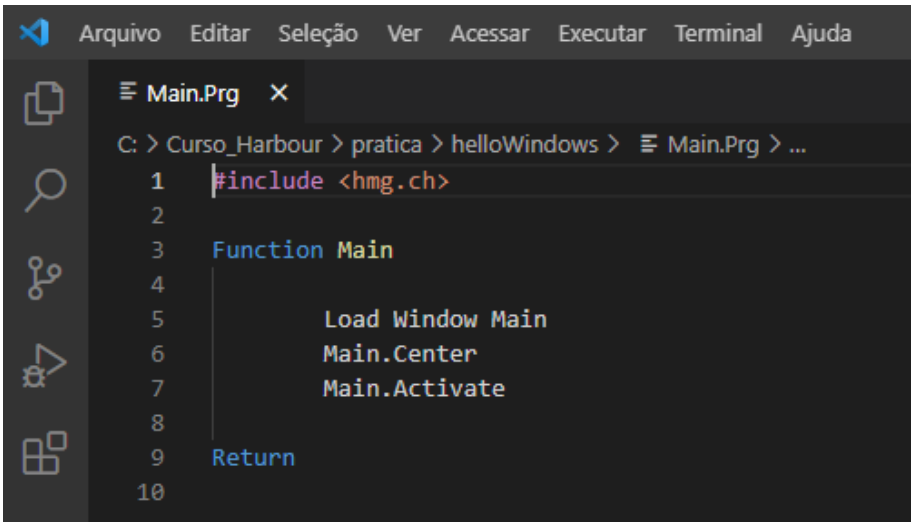
Vamos agora abrir o arquivo Main.prg usando a IDE. Na janela de Projetos, na aba Modules, clique duas vezes sobre o nome do arquivo. No nosso caso o arquivo chama-se "Main.prg".

Figura 41.17: Abrindo o arquivo pela IDE



A IDE irá executar o editor de código com o conteúdo do arquivo selecionado.

Figura 41.18: Abrindo o arquivo pela IDE (II)



A figura 41.18 nos mostra o código gerado. Até agora não digitamos uma linha de código. Vamos agora analisar essas poucas linhas:

1. Na linha 1 temos o include do arquivo "hmg.ch". Esse arquivo é necessário para que você possa usar a grande maioria dos recursos da HMG.

2. Da linha 3 até a 9 temos o procedimento principal, nós costumamos usar PROCEDURE, contudo FUNCTION também está correto.

3. Na linha 5 nós "carregamos"o arquivo "Main.fmg"com as definições da janela para a memória. Basta informar o nome do arquivo sem a extensão, conforme está no exemplo.

4. Na linha 6 centralizamos a janela, mas ela ainda não "apareceu"para a gente.

5. Finalmente na linha 7 a janela aparece através de Activate.

Um aspecto importante de todo programa HMG é a relação direta que existe entre uma janela (arquivo FMG) e um código fonte (arquivo PRG). Essa relação sempre existe e na imensa maioria das vezes o nome dos arquivos só diferem um do outro pela extensão. Assim, temos Main.prg e Main.fmg. Quando você for construindo outras janelas, procure manter esse padrão porque isso irá lhe facilitar muito.

Todo programa Windows obedece ao conceito já visto de "mensagens", por isso ele parece estranho aos olhos de um programador acostumado com o modo-texto. Sempre seguiremos a seguinte sequência :

1. Carregar a janela (LOAD)

2. Realizar alguma configuração, se necessário

3. Exibir a janela (Activate)

Esse método Activate, na verdade, é um LOOP onde a janela ficará aguardando as mensagens que você enviará para ela. Não adianta você digitar nada após esse loop (Activate), porque tudo o que você digitar só será executado após a janela ser fechada. Todo programa GUI trabalha com esse conceito de loop de mensagens, existem diferenças entre as linguagens, mas o loop está lá, aguardando as mensagens serem enviadas para execução. Faça o teste: a HMG possui uma função que exibe uma janela do tipo "alert". Vamos colocar uma janela após o Activate. ERRADO

Parte VIII

Metaprogramação

42 Geradores de programas

> Entre todas as criações do homem, a linguagem talvez seja a mais surpreendente.
>
> Giles Strachey

Objetivos do capítulo

- Gerar seus próprios programas através de templates

42.1 Introdução

A medida em que vamos avançando no aprendizado de uma linguagem, nós passamos a identificar padrões dentro dos nossos códigos. Esses padrões nos faz pensar se o próprio processo de desenvolvimento não é passível de automação. Basicamente, essa ideia nos leva a dois caminhos dentro do processo de desenvolvimento :

1. Programação baseada em gabarito (Template programming)
2. Programação baseada em dados (Data-driven programming)

42.2 Data-driven programming

42.3 Template programming

42.4 O dicionário de dados

42.5 Conclusão

REFERÊNCIAS BIBLIOGRÁFICAS

[Aguilar 2008]AGUILAR, L. J. **Fundamentos de programação: algoritmos, estruturas de dados e objetos**. Editora McGraw-Hill, São Paulo, 2008.

[Ascencio e Campos 2014]ASCENCIO, A. F. G.; CAMPOS, E. A. V. de. **Fundamentos da programação de computadores: algoritmos, PASCAL, C/C++ (padrão ANSI) e JAVA**. Pearson, São Paulo, 2014.

[Damas 2013]DAMAS, L. **Linguagem C**. LTC, Rio de Janeiro, 2013.

[Deitel e Deitel 2001]DEITEL, H. M.; DEITEL, P. J. **C++ Como programar**. Bookman, Porto Alegre, 2001.

[Forbellone e Eberspacher 2005]FORBELLONE, A. L.; EBERSPACHER, H. F. **Lógica de programação: a construção de algoritmos e estrutura de dados**. Prentice Hall, São Paulo, 2005.

[Goodrich e Tamassia 2002]GOODRICH, M. T.; TAMASSIA, R. **Estrutura de Dados e Algoritmos em Java** . 2002.

[Guimarães e Lages 1994]GUIMARãES, A. d. M.; LAGES, N. A. d. C. **Algoritmos e estruturas de dados**. LTC - Livros Técnicos e Científicos, Rio de Janeiro, 1994.

[Heimendinger 1994]HEIMENDINGER, L. **Clipper 5.0 - Avançado**. Campus, São paulo, 1994.

[Horstman 2005]HORSTMAN, C. **Conceitos de computação com o essencial de C++ .** Bookman, Porto Alegre, 2005.

[Hyman 1995]HYMAN, M. I. **Borland C++ para leigos**. Berkeley Brasil Editora, São Paulo, 1995.

[Kernighan e Pike 2000]KERNIGHAN, B. W.; PIKE, R. **A prática da programação**. Campus, Rio de Janeiro, 2000.

[Kernighan e Ritchie 1986]KERNIGHAN, B. W.; RITCHIE, D. M. **C: a linguagem de programação**. Campus, Rio de Janeiro, 1986.

[Kresin 2016]KRESIN, A. *Harbour for beginners*. 2016. `http://www.kresin.ru/en/hrbfaq.html/`.

[Manzano e Oliveira 2008]MANZANO, J. A.; OLIVEIRA, J. F. **Estudo dirigido Algoritmos**. Editora Érica, São Paulo, 2008.

[McLaughlin, Pollice e West 2007]MCLAUGHLIN, B.; POLLICE, G.; WEST, D. **Use a cabeça - Análise e projeto orientado ao objeto**. O'Reilly, São paulo, 2007.

[Mizrahi 2004]MIZRAHI, V. V. **Treinamento em linguagem C++.** Pearson, São Paulo, 2004.

[Nantucket 1990]NANTUCKET. **Clipper 5.0 - manual de referência**. Nantucket Corporation, EUA, 1990.

[Norton e Hahn 1992]NORTON, P.; HAHN, H. **O guia do UNIX** . 1992.

[OnLine 2016]ONLINE, F. C. **Fórum Clipper OnLine** . 2016.

[Papas e Murray 1994]PAPAS, C. H.; MURRAY, W. H. **Borland C++ 4**. Makron, Rio de Janeiro, 1994.

[Ramalho 1991]RAMALHO, J. A. A. **Clipper 5.0 avançado**. McGraw-Hill, São paulo, 1991.

[Reichard 1998]REICHARD, K. **Servidor Internet com Linux** . 1998.

[Salus 1994]SALUS, P. H. **A Quarter-Century of Unix** . 1994.

[Schach 2009]SCHACH, S. R. **Engenharia de Software: os paradigmas clássico e orientado a objetos** . 2009.

[Spence 1991]SPENCE, R. **Clipper 5.0 - release 5.01**. Makron, Rio de Janeiro, 1991.

[Spence 1994]SPENCE, R. **Clipper 5.2**. Makron, Rio de Janeiro, 1994.

[Stroustrup 2012]STROUSTRUP, B. **Princípios e práticas de programação com C++ .** Bookman, Porto Alegre, 2012.

[Swan 1994]SWAN, T. **Tecle e aprenda C**. Berkeley Brasil editora, São Paulo, 1994.

[Tenenbaum, Langsam e Augenstein 1995]TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de dados usando C.** Makron Books, São Paulo, 1995.

[Vidal 1989]VIDAL, A. G. d. R. **Clipper**. LTC - Livros Técnicos e Científicos, Rio de Janeiro, 1989.

[Vidal 1991]VIDAL, A. G. d. R. **Clipper 5**. LTC - Livros Técnicos e Científicos, Rio de Janeiro, 1991.

[Yourdon 1992]YOURDON, E. **Análise estruturada moderna**. Campus, Rio de Janeiro, 1992.

Parte IX

Apêndice

A Fluxogramas

A.1 Estruturas de controle

Veja a seguir uma representação de um loop. Note que não existe uma notação específica para representar um loop. Essa falta de notação para loops é um ponto negativo dos fluxogramas.

Figura A.1: Estrutura de seleção IF (Fluxograma da listagem 7.5)

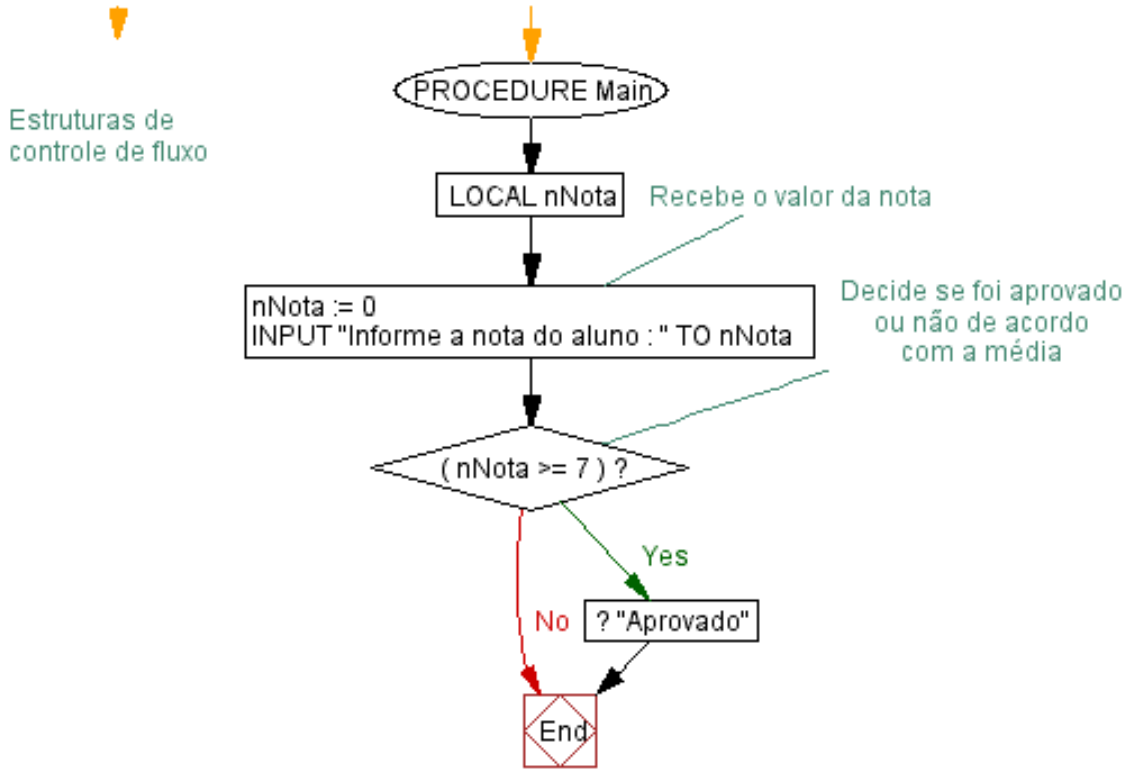


Figura A.2: Fluxograma de uma estrutura de seleção DO CASE ... ENDCASE da listagem 8.5)

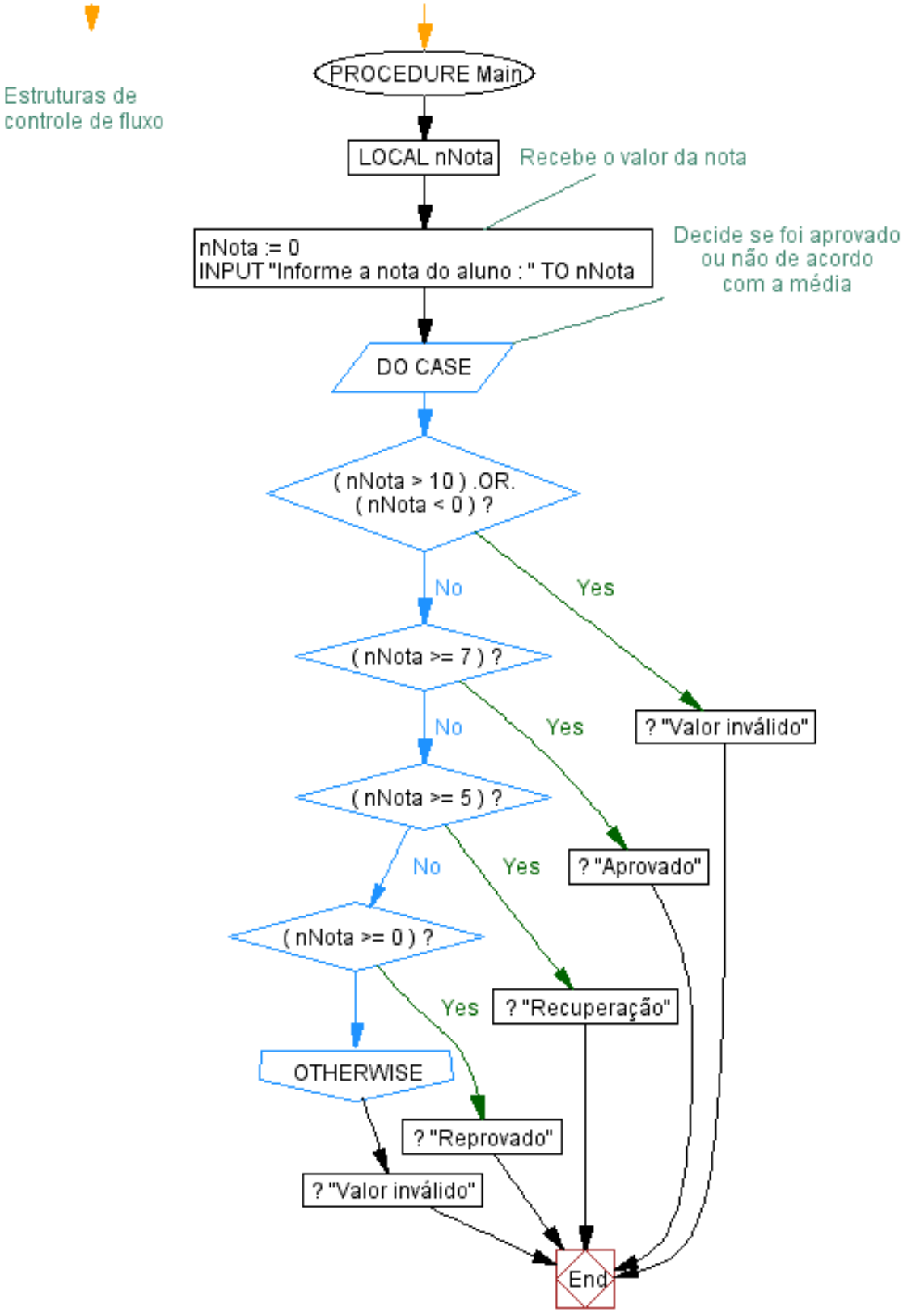
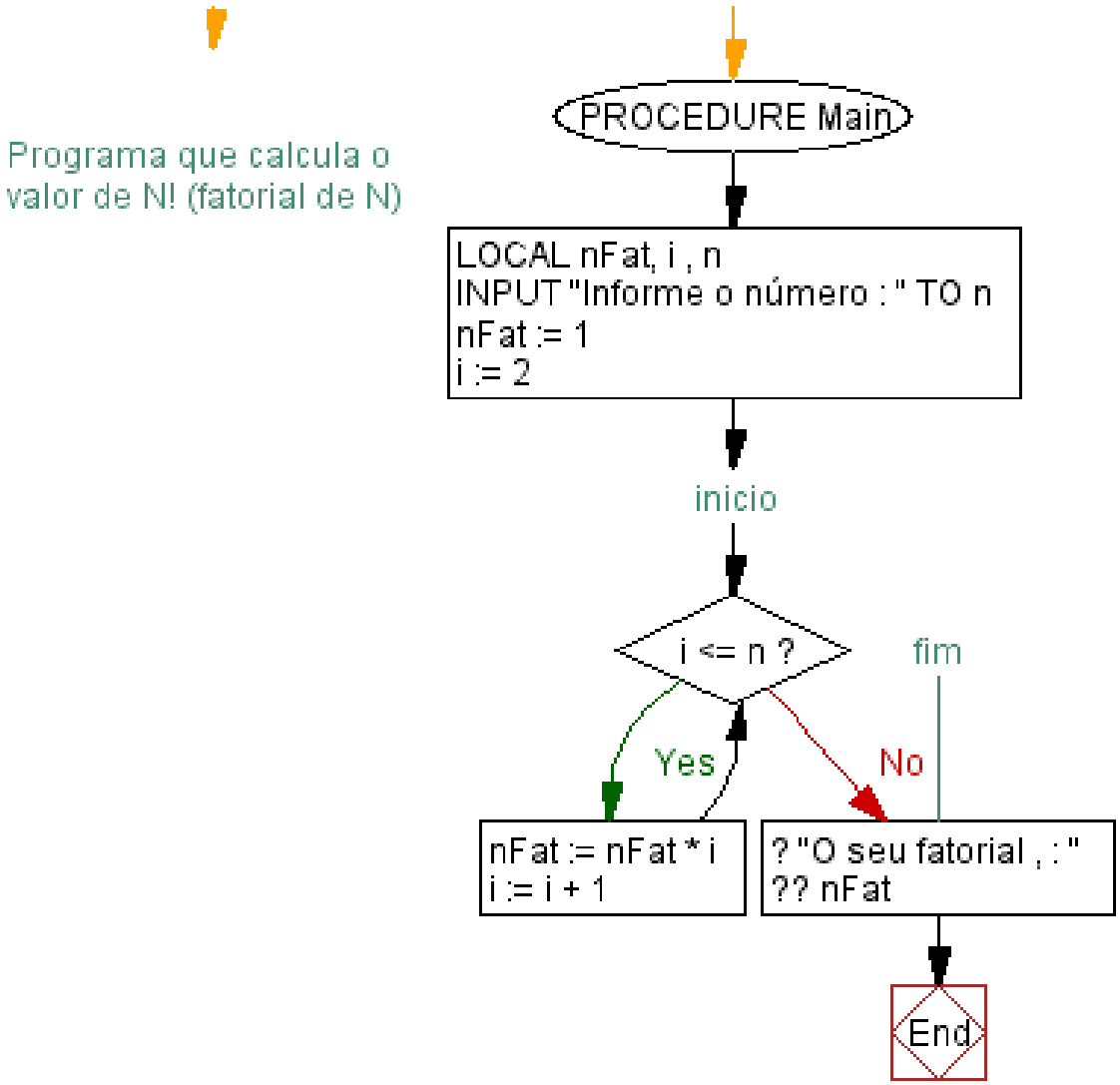


Figura A.3: Calculando o fatorial através de um laço 9.4)



B Usando o hbmk2

B.1 Introdução

O hbmk2 é o utilitário de linha de comando que nós usamos durante todo esse livro para poder compilar e linkeditar os nossos programas. Caso você não esteja familiarizado com esse processo, sugerimos uma leitura no capítulo 2 (“O processo de criação de um programa de computador”). Durante o livro esse utilitário foi utilizado apenas de duas formas possíveis:

1.criando um executável de apenas um arquivo fonte;

2.criando um executável de dois arquivos fonte.

A forma mais usual de criação de programas está exemplificado a seguir:

.:Resultado:.

```
hbmk2 hello
```

Essas formas de uso do hbmk2, apesar de corretas, não são as mais usadas na prática, pois geralmente os programas possuem vários arquivos (as vezes centenas deles). Nesse caso, a forma usual de se criar um programa através do hbmk2 é gerando um arquivo com a lista de arquivos que fazem parte do programa a ser gerado. Esse arquivo possui a extensão hbp e o seu conteúdo consiste simplesmente de uma lista de arquivos, conforme o exemplo a seguir:

```
principal.prg
menu.prg
cad01.prg
cad02.prg
cad03.prg
```

Supondo que o arquivo acima possua o nome exemplo.hbp, o processo de criação do programa envolve apenas :

.:Resultado:.

```
hbmk2 exemplo
```

Note que não precisamos digitar a extensão (.hbp), pois o hbmk2 irá procurar um arquivo chamado exemplo.hbp.

Na seção a seguir, nós iremos ver alguns parâmetros usados dentro do arquivo hbp e que podem facilitar, e muito, a geração do programa.

B.2 Alguns parâmetros usados pelo hbmk2

B.2.1 A compilação incremental através do parâmetro -inc

Em primeiro lugar: o que é compilação incremental ? Bem, vamos supor que você esteja trabalhando em um sistema que possui dezenas de arquivos. Vamos supor também que você deseja alterar um pequeno detalhe em um arquivo específico. A alteração, por menor que seja, irá requerer a recompilação de todos os arquivos do sistema, mesmo que eles não tenham sofrido alteração. Foi pensando nisso que os desenvolvedores da linguagem Harbour criaram a compilação incremental, que nada mais é que a compilação apenas do módulo que foi alterado, poupando tempo e recursos.

No exemplo a seguir nós acrescentamos a compilação incremental, no nosso arquivo exemplo.hbp, através do parâmetro -inc.

```
-inc
principal.prg
menu.prg
cad01.prg
cad02.prg
cad03.prg
```

Pronto, se você alterar o arquivo menu.prg, por exemplo, apenas ele será compilado quando você digitar hbmk2 exemplo.

B.2.2 Informações de depuração através do parâmetro -b

Se você quiser usar o depurador para encontrar erros basta incluir a opção -b no seu arquivo hbp. Nós já vimos como usar o depurador no capítulo **??**.

```
-b
principal.prg
menu.prg
cad01.prg
cad02.prg
cad03.prg
```

Você pode, obviamente, usar dois ou mais parâmetros ao mesmo tempo, conforme a listagem a seguir, que incluiu -b e -inc na geração do executável:

```
-inc
-b
principal.prg
menu.prg
cad01.prg
cad02.prg
cad03.prg
```

C Os *SETs*

C.1 Estruturas de controle

```
\#include "set.ch"
FUNCTION Main()

xAnt := Set(_SET_CONSOLE,.T.)  //SET CONSOLE ON

 ... Operações ...
Set(_SET_CONSOLE,xAnt )

... Restante do programa
```

[OnLine 2016]

D Resumo de Programação Estruturada

D.1 Fluxogramas

Figura D.1: Estrutura de seleção IF (Seleção única)

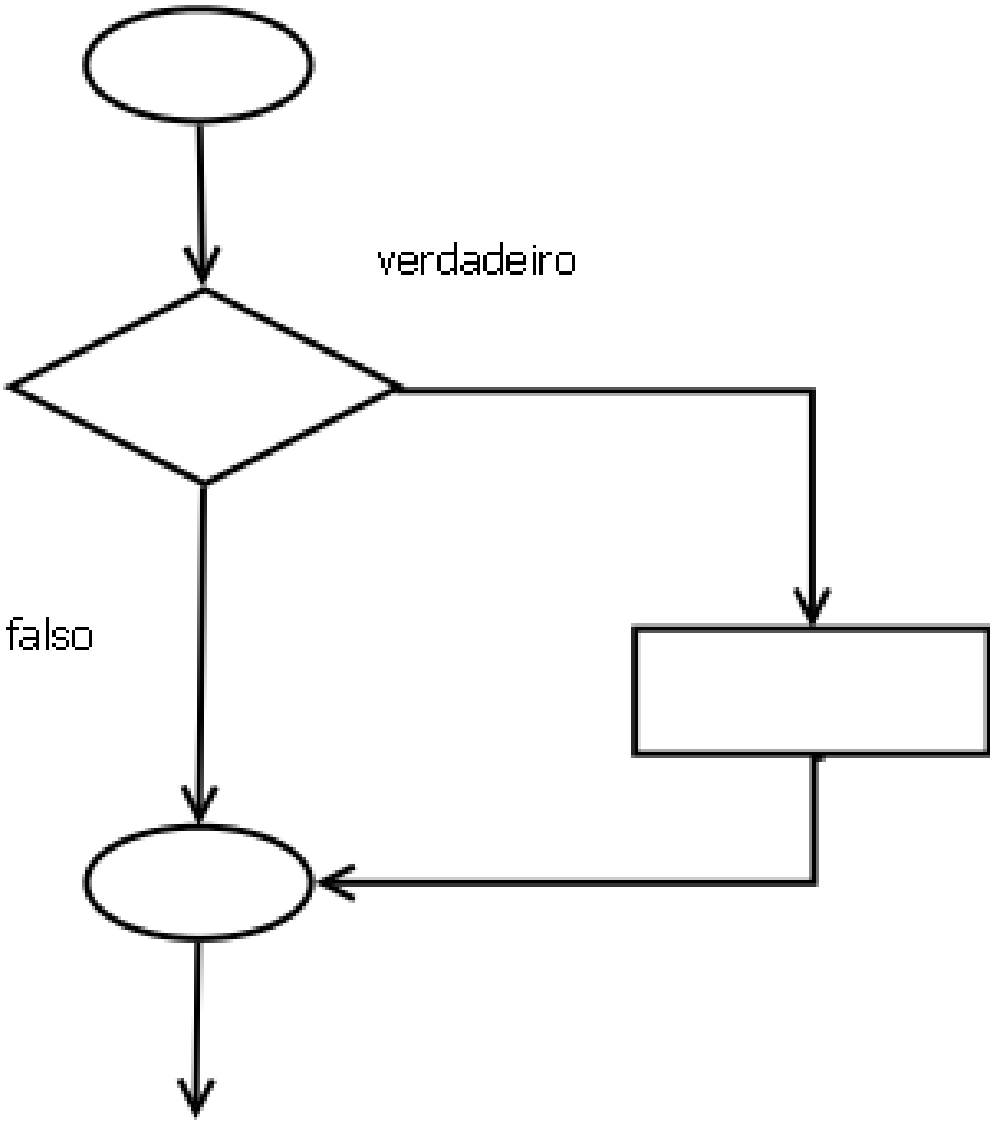


Figura D.2: Estrutura de seleção IF (Seleção dupla)

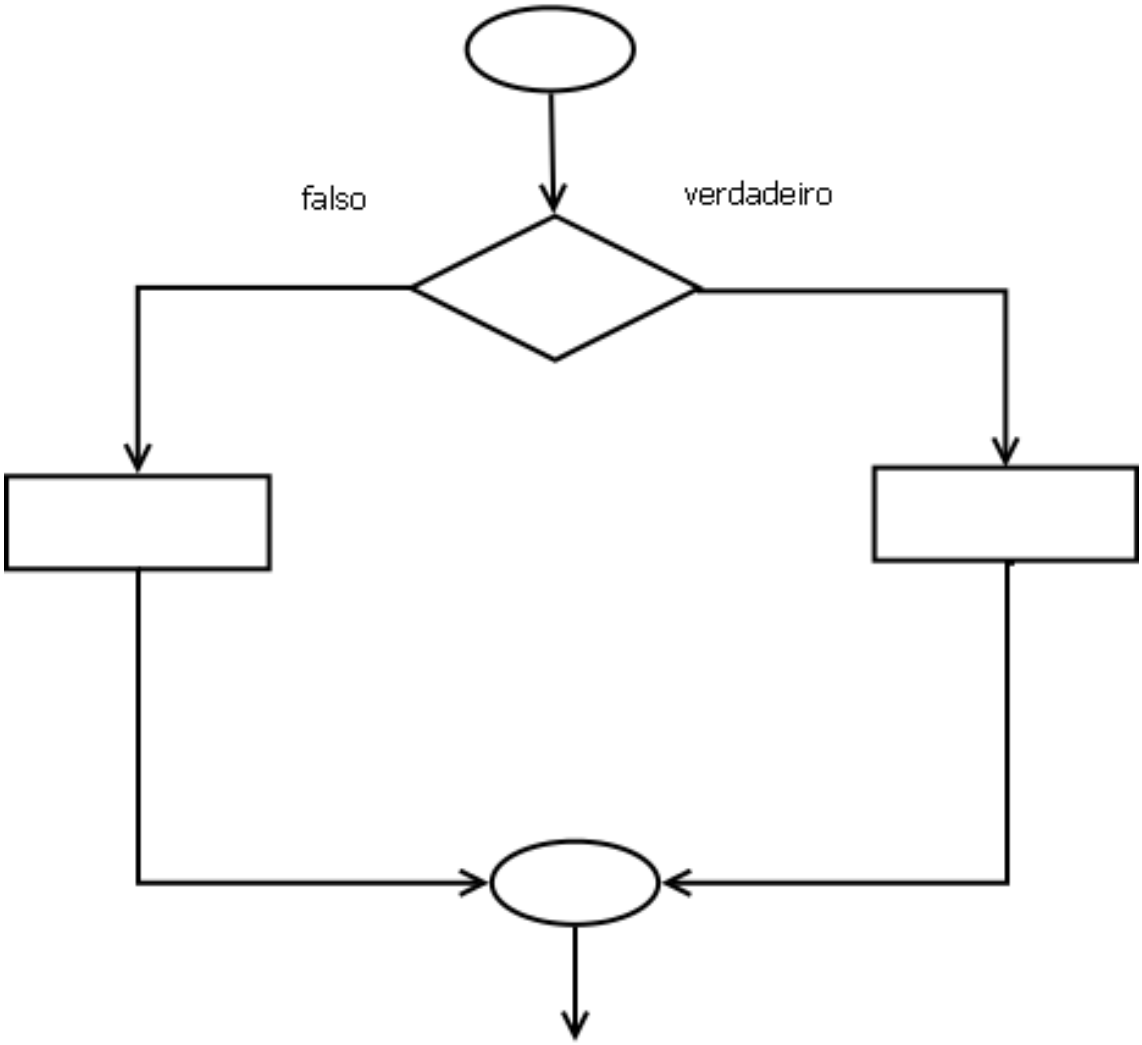


Figura D.3: Estrutura de seleção CASE (Seleção múltipla)

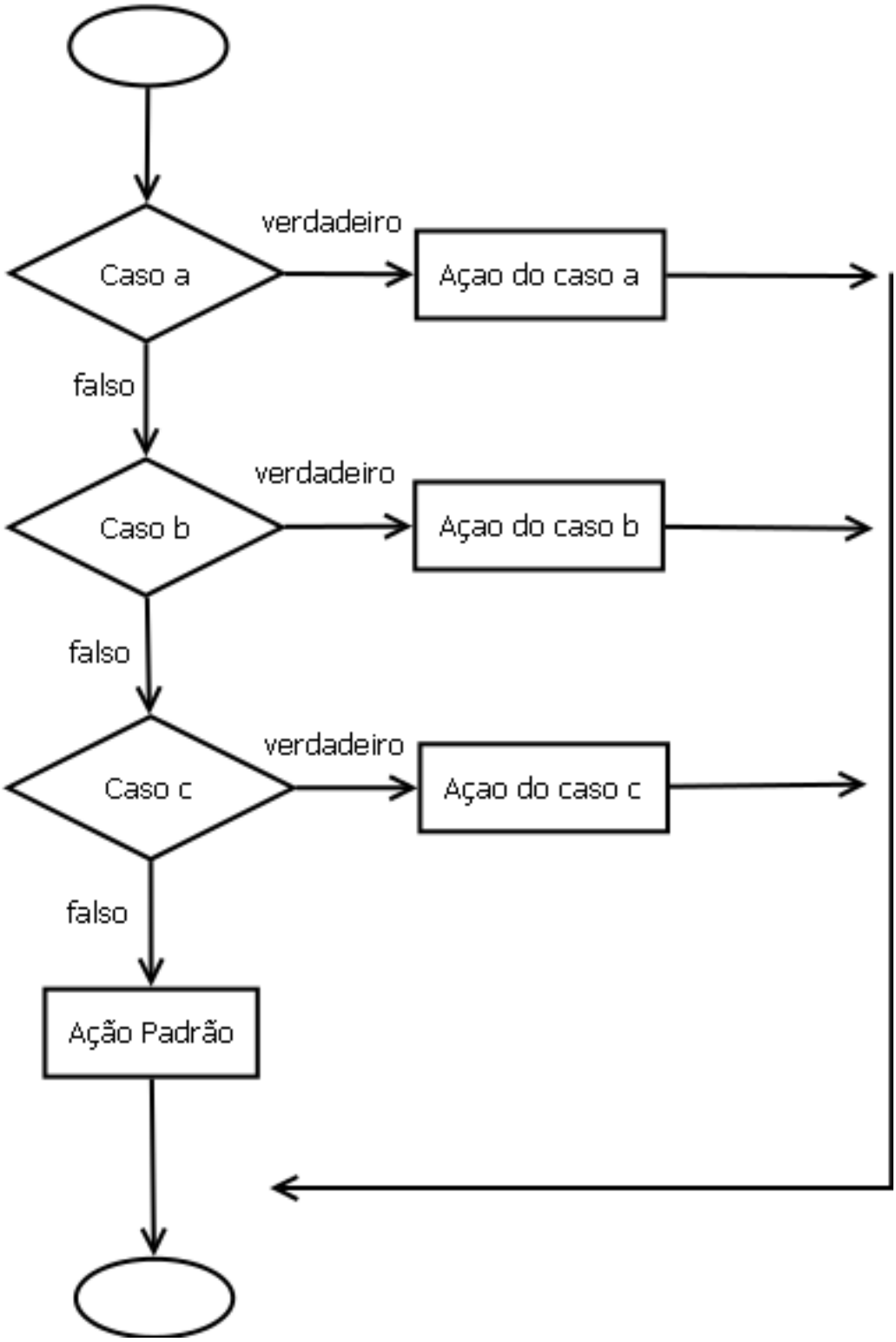


Figura D.4: Estrutura de repetição WHILE

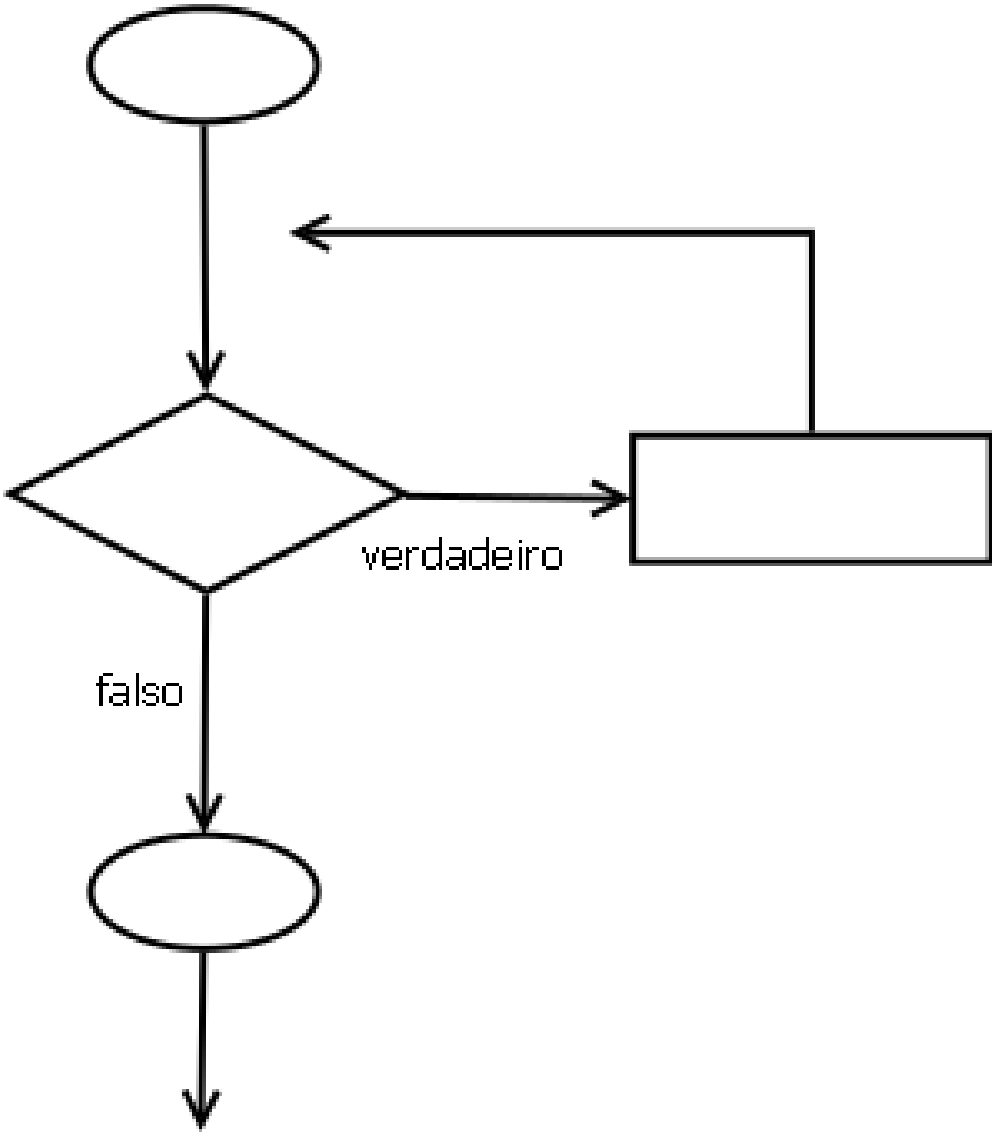


Figura D.5: Estrutura de repetição FOR

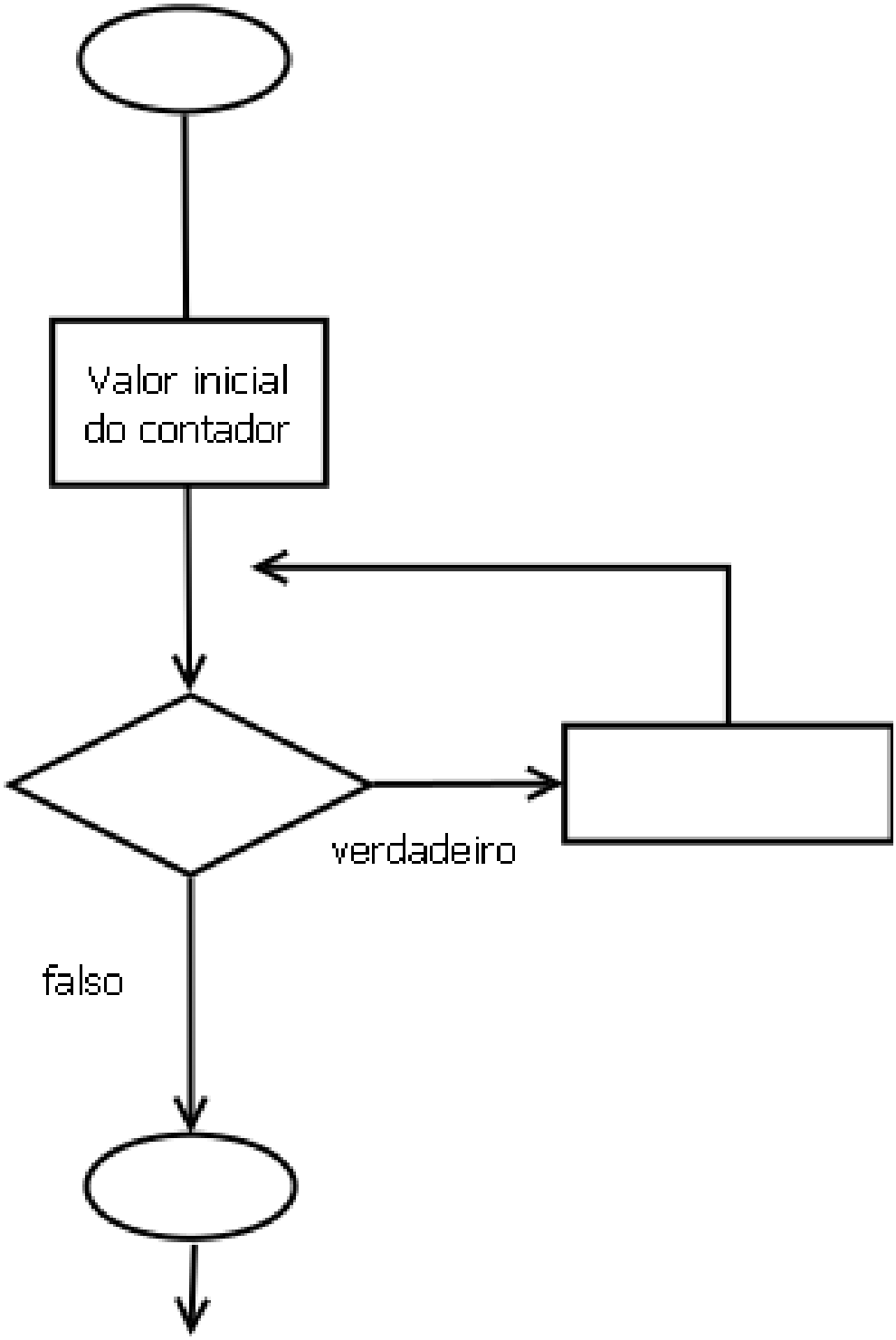
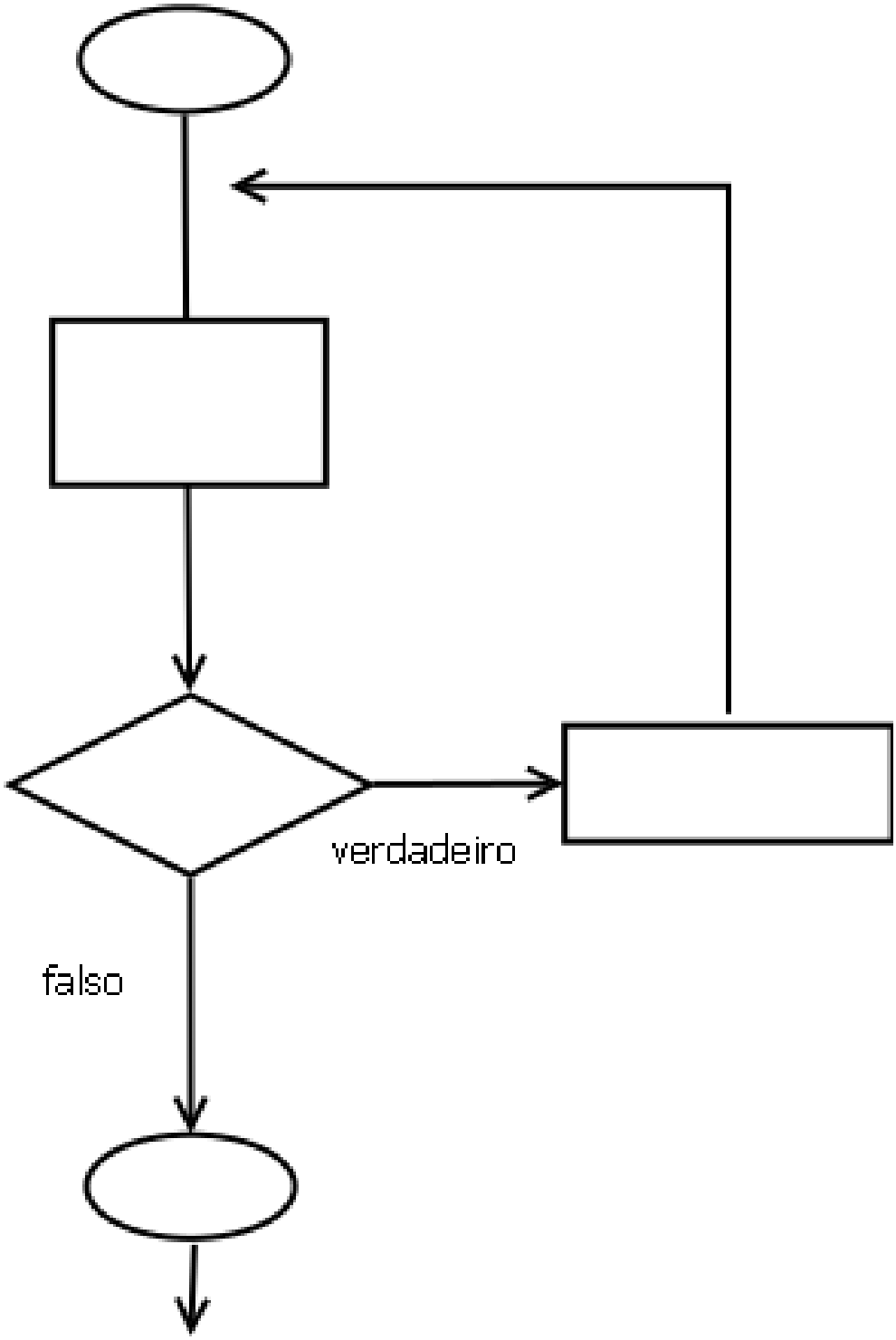


Figura D.6: Estrutura de repetição REPITA ATÉ



E Codificação e acentuação

E.1 Por que meus acentos não aparecem corretamente ?

Esse livro foi escrito para pessoas que estão iniciando no mundo da programação, por isso a explicação a seguir será bem simples e sem muitos detalhes técnicos. Se você já é um programador experiente, talvez você discorde de alguns pontos a seguir. É importante que você (iniciante) leia a sequência de perguntas e respostas a seguir. Elas constituem uma "trilha de aprendizado", portanto procure não pular perguntas.

E.1.1 O que acontece quando eu pressiono uma tecla ?

Um sinal elétrico é gerado. Na verdade são oito sinais (oito bits), que constituem uma unidade chamada Byte. Cada Byte possui uma representação numérica.

E.1.2 Como o computador faz para exibir letras ?

Quando o computador recebe o Byte gerado pelo teclado, ele vai converter esse Byte em um número. Após essa conversão, ele vai fazer uma busca em uma tabela contendo letras, números e outros símbolos.

No começo da era dos computadores modernos, cada fabricante tinha a sua própria tabela incorporada no hardware. Isso era um problema, pois a letra "A", por exemplo, era representada por códigos distintos em diferentes fabricantes. Por exemplo, em um computador da marca XYZ, a letra "A"era representada pelo número 100, e em outro computador da marca ABC, a mesma letra "A"era representada pelo número 200.

Essa variedade de tabelas causava dores de cabeça para os desenvolvedores de software.

E.1.3 Se cada fabricante tinha a sua própria tabela de símbolos, como esse problema foi resolvido ?

O problema foi resolvido com a criação de uma tabela universal chamada de Tabela ASCII. Essa tabela possui 256 números, ela começa no 0 (zero) e vai até o número 255. Esse tamanho não é por acaso, ele resulta de todas as combinações possíveis dos 8 bits que compõem um Byte.

2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 = 256

A tabela ASCII funciona assim: os números de 0 (zero) até 31 servem para representar comandos e ações. O número 27 representa a tecla ESC, o número 7 emite um beep, o número 10 é uma quebra de linha, o número 11 é a tabulação, e por aí vai. A partir do número 32 até o 255, temos os caracteres imprimíveis, ou seja, as letras, os números, o espaço em branco, o asterisco, e outros símbolos.

Essa tabela resolveu o problema para os Estados Unidos, mas outro problema acabou surgindo: à medida que os computadores foram se popularizando, a quantidade de símbolos exigidos se mostrava superior à quantidade de elementos da tabela. Ou seja, a quantidade requerida era superior aos 256 espaços disponíveis da tabela ASCII.

E.1.4 Como fazer para representar a infinidade de símbolos existentes ?

Os criadores da Tabela ASCII resolveram dividir a tabela em duas partes. A primeira era uma tabela comum a todos os computadores do mundo. Essa "base comum"vai do 0 (zero) até o 127. A partir do número 128 até o número 255 os caracteres variavam de idioma para idioma. Essa "segunda parte"chama-se "Tabela ASCII estendida".

Essa solução ainda hoje é usada. Nas linguagens de programação, por exemplo, a tabela original é a ASCII padrão norte-americano. Por isso que se você for tentar exibir um caracter acentuado e não "mudar"para a tabela ASCII do seu país, o que vai aparecer é um outro símbolo estranho. Isso acontece porque o tal "caracter estranho"pertence a tabela ASCII original. Para você imprimir um cedilha, por exemplo, você primeiro terá que selecionar a tabela ASCII com a codificação latina na sua parte estendida.

Com o advento da Internet, os programas não eram mais restritos a uma determinada cultura. A solução definitiva foi a criação de uma tabela única, compartilhada por todos os computadores do mundo.

E.1.5 A criação de um padrão mundial

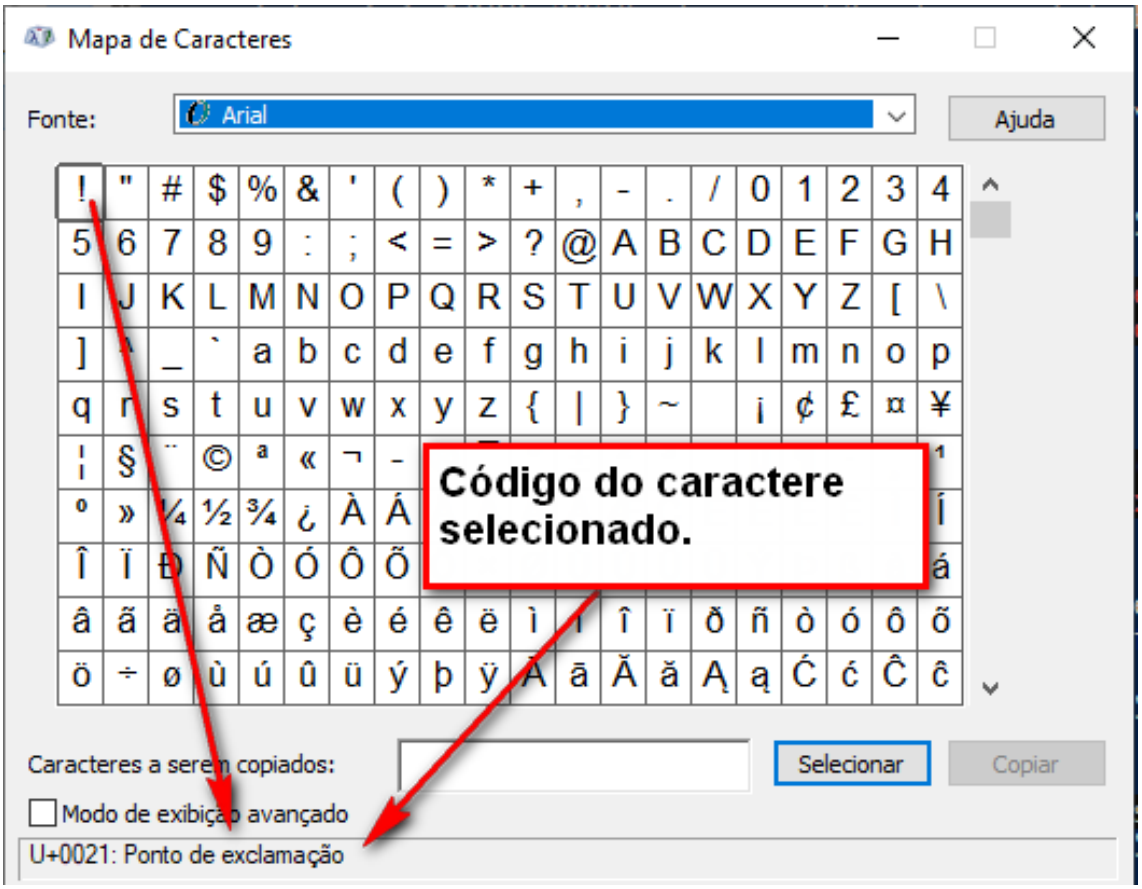
O padrão UNICODE foi criado para resolver o problema da "falta de espaço"da tabela ASCII. Trata-se de uma tabela gigantesca, que contém todos os caracteres de todas os alfabetos do mundo, e ainda sobra espaço para outros símbolos.

Esse padrão, contudo, é algo abstrato. Ou seja, ele só existe "no papel". Para que esse padrão seja representado pelo computador ele precisa ser armazenado na memória RAM, e esse armazenamento é feito através de uma técnica chamada de "encoding". O enconding mais usado é o UTF-8, pois ele armazena todos os caracteres definidos no padrão UNICODE.

Os primeiros 127 caracteres são exatamente iguais aos caracteres usados pela primeira tabela ASCII, dessa forma mantêm-se a compatibilidade. Da posição 128 até a 255 cada caractere ocupa apenas 1 Byte. A partir da posição 256, cada caractere representado passa a ter o tamanho de 2 Bytes, já que o número de combinações possíveis acabou. E não fica por aí. Como a quantidade de caracteres dessa tabela é gigantesca o número de combinações pode requerer até 6 Bytes para representar um único caractere.

O Windows possui um programa nativo chamado de "Mapa de caracteres", nele você pode pesquisar os caracteres da tabela.

Figura E.1: Mapa de caracteres



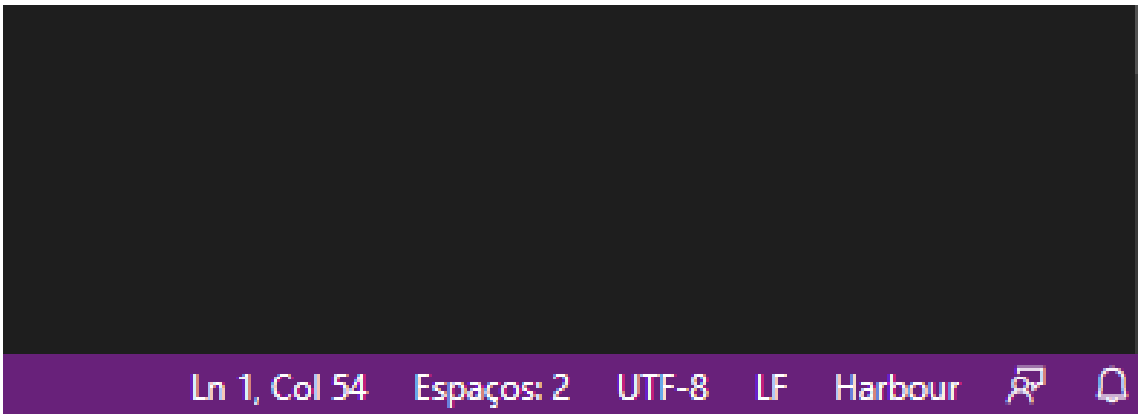
E.2 Como selecionar o UTF-8 no Harbour ?

E.2.1 Selecione um editor com suporte a UTF-8

Primeiramente, você deve usar um editor de programas com suporte a UTF-8. Todos os modernos editores de código possuem esse suporte, e na grande maioria deles o UTF-8 já vem definido por padrão. Geralmente no rodapé do seu editor existe uma referência a codificação usada.

A figura E.2 abaixo vemos o rodapé do Visual Studio Code com o UTF-8.

Figura E.2: Rodapé do Visual Studio Code



E.2.2 Ative o suporte a UTF-8 no Harbour

A seguir temos um exemplo de código que imprime acentos através do encoding UTF-8.

Listagem E.1: Exibindo acentos

```
REQUEST HB_CODEPAGE_UTF8  // <--- Insira essa linha
PROCEDURE Main

   hb_cdpSelect( "UTF8" ) // <---- Insira essa linha
   ? "João chegou cedo hoje ", "mas Tainá chegou atrasada."

RETURN
```

Ou seja, basta inserir o código abaixo antes da chamada da função MAIN

```
REQUEST HB_CODEPAGE_UTF8
```

e dentro de MAIN chame a função a seguir:

```
hb_cdpSelect("UTF8")
```

A primeira instrução ativa o suporte do Harbour a UTF-8. Uma instrução REQUEST é lida pelo Harbour uma única vez em tempo de compilação [1]. Já a função hb_cdpSelect("UTF8") serve para selecionar o suporte a UTF-8.

Por que isso é necessário ? Não bastava somente colocar REQUEST ?

A resposta é **não**. Isso porque o seu programa pode usar duas ou mais codificações. Por exemplo, vamos supor que você foi contratado para escrever uma aplicação que usa dois banco de dados: um deles foi implementado há trinta anos e usa uma codificação ISO-8859-1, bastante comum ainda. O outro banco de dados está na WEB e usa o UTF-8 como padrão. Em casos assim, o seu programa precisa selecionar a página de código correspondente a cada banco que irá acessar. Isso é feito através da função hb_cdpSelect().

Dica 177

Habitue-se a digitar os seus códigos utilizando um editor com suporte a UTF-8. Atualmente todas as aplicações estão utilizando esse padrão, inclusive os modernos banco de dados. Se a sua aplicação não tiver suporte a UTF-8 ela poderá ter problemas com acentuação quando for se conectar com aplicativos de terceiros, como banco de dados por exemplo. Você precisará de um editor de texto com suporte a UTF-8 e deve selecionar esse modo no seu editor (na grande maioria o UTF-8 já é o padrão quando você cria um novo arquivo).

E.2.3 Fique atento

Geralmente codificar em UTF-8 requer apenas os cuidados já mencionados. Contudo, existem casos onde você faz tudo certo e continua a ter problemas com acentuação. Por isso, siga a lista de verificação abaixo caso você tenha problemas.

[1] A expressão “em tempo de compilação” quer dizer que foi durante a fase da geração do código objeto. O programa continua iniciando a partir de *Main*

1. Certifique-se de que o seu editor de código tem suporte a UTF-8;
2. Veja se o UTF-8 está selecionado no seu editor de código (lembra da figura E.2, do VS Code ?);
3. Selecione o UTF-8 no Harbour, conforme já vimos.

As vezes o código que você está trabalhando foi originalmente criado em uma outra codificação. Se for esse o caso, talvez seja necessário você converter o seu código para UTF-8. Existem programas que fazem isso facilmente. No mundo *NIX temos o utilitário de linha de comando "iconv", ele é útil para realizar conversões de vários arquivos (em lote). O Visual Studio Code também realiza a conversão do arquivo aberto. Basta clicar na barra inferior sobre o nome da codificação ativa. Quando você clica uma janela pop-up é aberta com diversas opções. Clique em "Save with encoding"("Salvar com Codificação") e selecione o encoding desejado da lista que irá aparecer.

E.2.4 É possível trabalhar com UTF-8 sem um editor com suporte a UTF-8 ?

Sim e não.

Essa informação é tão importante que resolvemos deixar em uma sub-seção a parte. O que iremos dizer pode lhe confundir, mas é melhor que você saiba logo: às vezes você codifica o programa em um editor sem suporte a UTF-8 e ele é exibido corretamente. Em quais casos isso ocorre ?

Se você não estiver "exibindo"caracteres diretamente escritos no editor de código, esses caracteres podem ser exibidos corretamente. No exemplo a seguir temos uma pequena ilustração.

Vamos supor que você usou um editor antigo, sem suporte a UTF-8, para digitar o seguinte código :

```
REQUEST HB_CODEPAGE_UTF8
PROCEDURE MAIN

    hb_cdpSelect("UTF8")
    ? DisplayInfo()

RETURN
```

Vamos supor, também, que a função DisplayInfo() lê dados de um banco MySQL em UTF-8 e retorna uma string acentuada. Como eu **não usei** o editor de códigos para escrever algum caractere acentuado (ela veio direto do Banco de Dados em tempo de execução), a informação será exibida corretamente.

Importante: Você **sempre** precisa ativar o suporte a codificação no Harbour (REQUEST e hb_cdpSelect) se quiser imprimir alguma em coisa usando outras páginas de código[2].

[2] Na verdade o Harbour já vem com duas codificações inclusas: a UTF8 e a EN, que é o padrão do Clipper e também do Harbour. Nesse caso você só vai precisar usar hb_CdpSelect(). Contudo, habitue-se com a ideia do REQUEST, porque existem dezenas de codificações que precisam ser explicitamente declaradas em tempo de compilação através de REQUEST. Em diversos exemplos de código você verá um comando REQUEST HB_CODEPAGE_UTF8, contudo, ele foi colocado aí apenas para que você se acostume com o uso do REQUEST HB_CODEPAGE_ ...

Vamos ver só mais um exemplo para encerrar. Suponha que você usou o seu velho editor, sem suporte a UTF-8, para digitar o código abaixo:

```
REQUEST HB_CODEPAGE_UTF8
PROCEDURE MAIN

hb_cdpSelect("UTF8")
? "Olá pessoal"

RETURN
```

Nesse caso, como eu estou usando o editor, sem suporte a UTF-8, para digitar caracteres acentuados, os mesmos não serão exibidos corretamente. Nesse caso você vai precisar abrir o arquivo usando um editor com suporte a UTF-8 e realizar a conversão. O uso de um editor moderno com suporte a UTF-8 e uma boa prática que deve ser seguida, mas você é livre para decidir. Existem casos e casos.

E.3 Funções usadas para dar suporte a codepages distintas

E.3.1 hb_CdpSelect()

O próprio nome da função já nos dá uma pista sobre o seu objetivo. "Cdp"significa "Codepage", portanto CdpSelect quer dizer, em bom português : "Seleciona a Codepage"(Página de código). Na verdade, essa função pode ser usada apenas para consultar qual o codepage atualmente selecionado, basta usar ela sem passar parâmetro algum.

```
? hb_CdpSelect() // Retorna o ID do codepage
```

O exemmplo E.2 ilustra o uso da função.

Listagem E.2: Uso da função hb_CdpSelect()
Fonte: codigos/cdpselect.prg

```
REQUEST HB_CODEPAGE_PT850
PROCEDURE MAIN

    hb_cdpSelect( "EN" )
    ? hb_cdpSelect()
    ? hb_UTF8ToStr( "É < G =" ), hb_BChar( 144 ) < "G" // É < G é .F.
    hb_cdpSelect( "PT850" )
    ? hb_cdpSelect()
    ? hb_UTF8ToStr( "É < G =" ), hb_BChar( 144 ) < "G" // É < G é .T.

RETURN
```

Interessante notar que a ordenação fica afetada se a página não for selecionada corretamente. Por exemplo, em uma página de código do idioma inglês, um caractere acentuado vem após todos os caracteres do alfabeto, mas em uma página de código latina, a ordem alfabética é respeitada também quando o caractere é acentuado.

.:Resultado:.

```
EN
É < G = .F.
PT850
É < G = .T.
```

As funções hb_Utf8ToStr() e hb_BChar() são explicadas adiante. Basicamente hb_Utf8ToStr() é usada para poder exibir um caractere padrão UTF-8 quando a página de código selecionado não é UTF-8. Já hb_BChar() nos retorna o caractere correspondente ao código informado.

O quadro sintático da função hb_CdpSelect() está descrito logo a seguir:

Descrição sintática 57

1. Nome : hb_CdpSelect()
2. Classificação : função.
3. Descrição : Seleciona uma nova codepage ou informa o valor da codepage atual.
4. Sintaxe

```
hb_CdpSelect( [<cCodePage]> ) -> cRetCodePage

cRetCodePage : Se <cCodePage> for definida, cRetCodePage
refere-se a página de código anterior (a que estava vi-
gente antes da substituição). Se <cCodePage> não for
definida, então cRetCodePage refere-se ao valor da página
de código atual.
```

5. Parâmetros
   - [<cCodePage>] : Página de código que você deseja selecionar.
6. Observações : a página de código deve existir e estar a disposição. Isso é feito mediante um comando REQUEST. Caso você deseje incluir todas as páginas de código, inclua o cabeçalho "hbextcdp.ch".
7. Fonte : `https://harbour.github.io/doc/harbour.html#hb_cdpselect`

E.3.2 hb_CdpList()

Essa função nos retorna um array com todas as páginas de código disponíveis no seu programa. A listagem E.3 nos mostra, através de um exemplo simples, uma possível aplicação dessa função.

Listagem E.3: Listando todas as codepages.
Fonte: codigos/cdplist.prg

```
#include "hbextcdp.ch"
PROCEDURE MAIN

    LOCAL cCodePage

    FOR EACH cCodePage IN hb_cdpList()
        ?  cCodePage
    NEXT

RETURN
```

O exemplo acima lista todas as páginas disponíveis no Harbour, porque o arquivo "hbextdcp.ch"foi informado. Esse arquivo contém uma série de comandos REQUEST (DYNAMIC) para todas as páginas de código.

A seguir um pequeno trecho do que é incluído na sua aplicação através de "hbextdcp.ch".

```
DYNAMIC HB_CODEPAGE_UA1125
DYNAMIC HB_CODEPAGE_UA1251
DYNAMIC HB_CODEPAGE_UA866
DYNAMIC HB_CODEPAGE_UAKOI8
DYNAMIC HB_CODEPAGE_UTF16LE
DYNAMIC HB_CODEPAGE_UTF8
DYNAMIC HB_CODEPAGE_UTF8EX
```

Dica 178

Os comandos REQUEST mais usados por aplicativos da lingua portuguesa são :

1. REQUEST HB_CODEPAGE_PT850 : usada por aplicações antigas, compatíveis com o Clipper/dBase em ambiente MS-DOS. A página de código selecionada é a 850.
2. REQUEST HB_CODEPAGE_PTISO : bastante usada por aplicações desenvolvidas em ambiente Windows. Os ambientes de desenvolvimento Visual Basic e Delphi utilizam muito essa página, que é a ISO-8859-1.
3. REQUEST HB_CODEPAGE_UTF8 : é a página de código mais usada atualmente, principalmente por aplicativos WEB e Mobile. Acabou também sendo a página de código padrão usada pelos bancos de dados atuais, como as últimas versões do PostgreSQL e MySQL. A página de código usada é a UTF-8, que é uma implementação do padrão UNICODE. Você não precisa declarar esse REQUEST explicitamente porque ele já é inserido automaticamente.

Para encerrar, veja um pequeno exemplo de como o UTF-8 já vem disponibilizado no Harbour sem a necessidade de REQUEST, embora o a página selecionada por padrão seja a EN (padrão Clipper).

Listagem E.4: Não precisamos de REQUEST para inserir suporte a UTF-8. Mas precisamos de hb_CdpSelect() para ativá-lo.
Fonte: codigos/cdputf8.prg

```
PROCEDURE MAIN                                                          1
                                                                        2
    LOCAL cResp                                                         3
    ACCEPT "Ativo suporte a UTF-8 ? (S/N) " TO cResp                    4
    IF cResp $ "Ss"                                                     5
        hb_cdpSelect("UTF8")                                            6
    ENDIF                                                               7
    ? "áéíióúç"                                                         8
                                                                        9
RETURN                                                                  10
```

Se teclar "S", a acentuação aparece corretamente, porque hb_CdpSelect() foi usada para selecionar "UTF8"como ativa.

E.3.3 hb_CdpUniID()

A página de código selecionada por hb_CdpSelect() é tudo o que necessitamos para criar aplicativos que usem múltiplas páginas de código, contudo elas pertencem a um grupo maior, que é mantido pela ISO (Organização Internacional de Normalização). Para saber qual grupo pertence a sua página, use a função hb_CdpUniID(). A listagem E.5 ilustra um uso da função hb_CdpUniID().

Listagem E.5: Descobrindo a qual mapa de caracteres ISO a nossa página de código pertence.
Fonte: codigos/cdpuniid.prg

```
#include "hbextcdp.ch"                                                  1
PROCEDURE MAIN                                                          2
                                                                        3
    LOCAL cCodePage                                                     4
                                                                        5
    FOR EACH cCodePage IN hb_cdpList()                                  6
        ? cCodePage , " = " , hb_cdpUniID( cCodePage )                  7
    NEXT                                                                8
                                                                        9
RETURN                                                                  10
```

Um trecho selecionado do programa acima mostra o mapa de caracteres ISO que abrange as páginas de código disponíveis. Note que determinados idiomas pertencem ao mesmo mapa de caracteres (iso8859-1), como o Português, o Francês e o Italiano.

.:Resultado:.

```
 PTISO  =  iso8859-1 // Português
 ITISO  =  iso8859-1 // Italiano
 FRISO  =  iso8859-1 // Francês
```

E.3.4 hb_Translate()

Quando queremos traduzir uma string de um código para outro usamos a função hb_Translate(). O funcionamento dela é bem simples, como mostra o quadro sintático abaixo :

Descrição sintática 58

1.Nome : hb_Translate()

2.Classificação : função.

3.Descrição : traduz uma string de uma página de código para outra

4.Sintaxe

```
hb_Translate( <cStr> ,;
              [<cOrigem>] ,;
              [<cDestino>] ) -> cStrTraduzida

cStrTraduzida : string no novo formato de codepage
```

5.Parâmetros

- •<cStr> : String a ser traduzida
- •[<cOrigem>] : Identificador da página de origem
- •[cDestino>] : Identificador da página de destino

6.Fonte : `https://harbour.github.io/doc/harbour.html#hb_translate` - Acessada em 20-Set-2021.

Observações adicionais

1.Se você omitir o ID da página de origem, o ID corrente será usado;

2.se você omitir o ID da página de destino, o ID corrente será usado;

3.você precisa requisitar o ID se quiser usar essas funções. Use REQUEST antes de Main (consulte a próxima listagem).

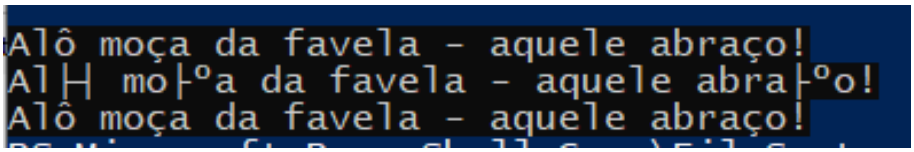
O exemplo a seguir é muito comum quando queremos importar um código legado para o padrão UTF-8. As antigas aplicações usavam a codificação OEM850, e as atuais costumam usar UTF-8. Veja na listagem a seguir um exemplo simples de conversão usando hb_Translate().

```
REQUEST HB_CODEPAGE_UTF8                                                    1
REQUEST HB_CODEPAGE_PT850                                                   2
PROCEDURE MAIN                                                              3
                                                                            4
LOCAL cStr := "Alô moça da favela - aquele abraço!"                         5
? cStr                                                                      6
```

```
    ? hb_Translate( cStr , "PT850" , "UTF8" )
    hb_cdpSelect("UTF8")
    ? hb_Translate( cStr , "PT850" , "UTF8" )

    RETURN
```

O programa acima foi digitado em um editor antigo, usando a página de código 850. Ao ser executado ele exibe a seguinte saída

Figura E.3: Saída do teste de codepage.



Note algumas observações importantes a seguir (lembre-se: nos exemplos a seguir o editor de textos foi configurado com a página de código 850)

Observação 1: Página de código do editor igual a página de código do Harbour

```
? cStr
```

exibe a string corretamente.

.:Resultado:.

```
    Alô moça da favela - aquele abraço!
```

Isso ocorre porque a página de código do editor de textos é a 850, e a página que o Harbour está utilizando é PT850, que é compatível com grupo 850.

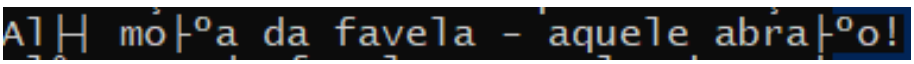
Observação 2: Conversão efetuada para UTF-8, mas o resultado foi exibido na página PT850

Agora, usando hb_Translate() para traduzir:

```
? hb_Translate( cStr , "PT850" , "UTF8" )
```

Dessa vez a string **não foi exibida corretamente**.

Figura E.4: Saída do teste de codepage (II).



Isso ocorre porque eu estou tentando exibir uma string no formato UTF-8, mas a página do Harbour ativa é PT850. Ou seja, a hb_Translate() trabalhou perfeitamente, mas o seu retorno foi exibido em uma outra página de código. Não esqueça: uma string em uma determinada página de código só será exibida se essa página estiver ativa.

Observação 3: Conversão efetuada para UTF-8, mas o resultado foi exibido na página UTF8

```
hb_cdpSelect("UTF8")
? hb_Translate( cStr , "PT850" , "UTF8" )
```

exibe a string corretamente.

.:Resultado:.

```
    Alô moça da favela - aquele abraço!
```

E.3.5 hb_CdpTerm()

Os Prompts de comandos, como o DOS, o Powershell ou o Shell Bash, também possuem suporte a diversas páginas de código. Para descobrir qual a página de código do terminal, onde o seu executável está, use a função hh_CdpTerm(). O programa da listagem E.6 nos mostra um exemplo de uso.

Listagem E.6: Obtendo a página de código do terminal.
Fonte: codigos/cdpterm.prg

```
1
2
3
4
5
6
7
8
9
```

```
PROCEDURE MAIN

    ? hb_cdpTerm()
    hb_cdpSelect("UTF8")
    ? hb_cdpTerm()


RETURN
```

Na listagem acima, procuramos mostrar que o retorno da função hb_CdpTerm() independe da página de código vigente. Note que hb_CdpSelect() não altera o retorno de hb_CdpTerm().

No Windows :

.:Resultado:.

```
PT850
PT850
```

No Linux :

.:Resultado:.

```
UTF8
UTF8
```

F O futuro

F.1 Por que meus acentos não aparecem corretamente ?

F.1.1 Inicialização com tipos de dados

Agora que nós já sabemos os principais tipos de dados do Harbour (Caractere, numérico, data e lógico), vamos ver uma nova forma ainda não totalmente implementada (atualmente uso o harbour 3.2) de se inicializar variáveis que promete conferir uma segurança maior ao nosso programa. Trata-se da inicialização com o tipo de dado associado. Até agora nós aconselhamos (ainda não explicamos) que você inicie as suas variáveis com LOCAL. Agora veremos uma sintaxe que torna essa inicialização mais segura: a inicialização com o tipo e dado associado a variável. Primeiro veremos mais uma vez a forma com que temos inicializado as nossas variáveis :

Listagem F.1: Até agora estamos criando variáveis assim.

```
/*
Inicialização perigosa!
*/

PROCEDURE Main
LOCAL cNome := "Programador"

   hb_cdpSelect( "UTF8" ) // Seleciona o suporte a UTF8


   ? "O valor de cNome é  : ", cNome

   cNome := 12

   ? "O valor de cNome é agora " , cNome


RETURN
```

O Harbour é um projeto em constante evolução, talvez quando você ler esse documento esse recurso já esteja totalmente implementado na linguagem. Atualmente, se você criar uma variável e atribuir a ela uma data, nada impede de você aproveitar essa mesma variável e atribuir a ela um número. Ou seja, nada impede que você atribua um tipo de dado diferente a uma variável anteriormente criado com outro tipo de dado. Na listagem F.1 a variável **cNome** foi criada com o intuito de ser do tipo caractere, mas ela assumiu um valor numérico na linha 13. Existem muitos códigos que

funcionam perfeitamente utilizando essa técnica de “reaproveitamento” de variáveis, mas ela não é considerada uma técnica segura.

Para evitar problemas futuros, o Harbour está evoluindo[1] para uma forma mais segura de se criar variáveis. Confira esse mesmo código na listagem F.2.

Listagem F.2: Formato com tipagem forte.

```
PROCEDURE Main
local a as numeric, b as character
a = 1
b = 2 //erro em tempo de compilação se a opção de verificação estática de tipo
b = "x"
? a + b //também gerará erro em tempo de compilação


RETURN
```

Mantendo o seu compromisso de compatibilidade com os códigos antigos, a equipe que desenvolve o Harbour não irá “mudar as regras do jogo”. Você sempre poderá atribuir um número a uma variável data se assim desejar. Simplesmente essa nova forma de declaração fará parte da linguagem e conviverá com a antiga. Se você quiser que o Harbour trabalhe da forma tradicional apenas inicialize sem o tipo de dado associado conforme a listagem F.1, mas se você já quer ir logo se acostumando ao jeito novo, o Harbour já aceita a sintaxe da listagem F.2, embora o recurso ainda não funcione. Quando esse recurso estiver funcionando você não poderá mudar o tipo de dado de uma variável declarada como (AS NUMERIC) para outro tipo de dado (se tentar irá causar um erro de execução).

Acrescente os seguintes comandos que aparecem na segunda listagem.

Primeira listagem : codificação padrão (a que nós usamos).

```
PROCEDURE Main

   ? "João, que horas são ?"

RETURN
```

.:Resultado:.

```
Jo|o, que horas s|o ?
```

Segunda listagem : problema resolvido.

```
REQUEST HB_CODEPAGE_PT850
PROCEDURE Main

   hb_cdpSelect( "PT850" )
   ? "João, que horas são ?"
```

[1] Essa característica ainda não está implementada

```
RETURN
```

.:Resultado:.

```
João, que horas são ?
```

Nós evitamos o uso do til nas nossas listagens iniciais. Preferimos esse pequeno inconveniente a ter que incluir símbolos sem a devida e clara explicação.

A dica é : prefira a primeira listagem pois lá você saberá exatamente a razão de tudo o que está lá.

G Instalando o Harbour

Esse apêndice tem por objetivo principal ensinar alguns passos básicos para a instalação do Harbour. Atualmente (Setembro de 2016) o Harbour possui duas versões : a 3.2 e a 3.4. A versão 3.2 é a versão oficial por enquanto, mas esse quadro com certeza irá mudar em breve. Resolvemos abordar duas formas de ter o Harbour na sua máquina, a primeira delas para usuários do sistema operacional Windows, e a segunda forma é para usuários Linux. A instalação para Linux tornou-se bem tranquila nos últimos anos, mas você ainda precisa ter conhecimentos básicos da linha de comando do Linux. Na instalação para Linux nós abordaremos o processo de compilação do Harbour, não da sua instalação.

G.1 Instalando o Harbour 3.2 na sua máquina com Windows

O projeto Harbour atualmente está hospedado em dois sites : o GitHub e o SourceForge. O código fonte da linguagem está hospedado no GitHub no seguinte endereço: `https://github.com/harbour/core/`. Essa versão não é a que nós queremos, pois não é o nosso objetivo compilar o Harbour. A versão pronta para ser instalada em máquinas com Windows pode ser baixada do site SourceForge. Enquanto escrevo, o endereço exato onde o Harbour para Windows se encontra é esse : `https://sourceforge.net/projects/harbour-project/files/binaries-windows/nightly/`.

Caso você deseje chegar a esse endereço sem clicar no link indicado anteriormente, siga os seguintes passos :

1. Abra o seu navegador e vá para o endereço `https://sourceforge.net/projects/harbour-project/`
2. **Não baixe** a versão que ele sugeriu. Embaixo do botão verde tem um link com o nome “Browse All Files”. Clique nele. (Figura G.1).
3. Irá aparecer uma lista com o nome de diversos sistemas operacionais. Clique no link “binaries-window”. (Figura G.2).
4. Clique agora na pasta “nightly”. Não tenha medo de baixar uma versão recém compilada. Ela é perfeitamente funcional. (Figura G.3).
5. Clique agora no arquivo executável. Nesse exato momento o nome dele é “harbour-nightly-win-mingw.exe.” (Figura G.4).

Figura G.1: Baixando o Harbour (Passo 1 de 4)

Figura G.2: Baixando o Harbour (Passo 2 de 4)

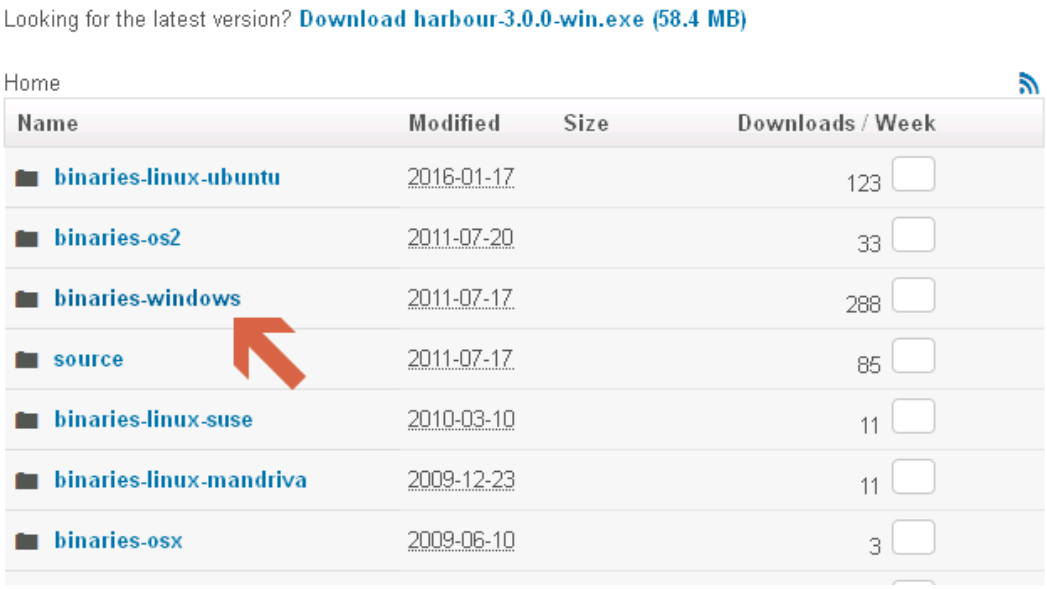


Figura G.3: Baixando o Harbour (Passo 3 de 4)

Figura G.4: Baixando o Harbour (Passo 4 de 4)

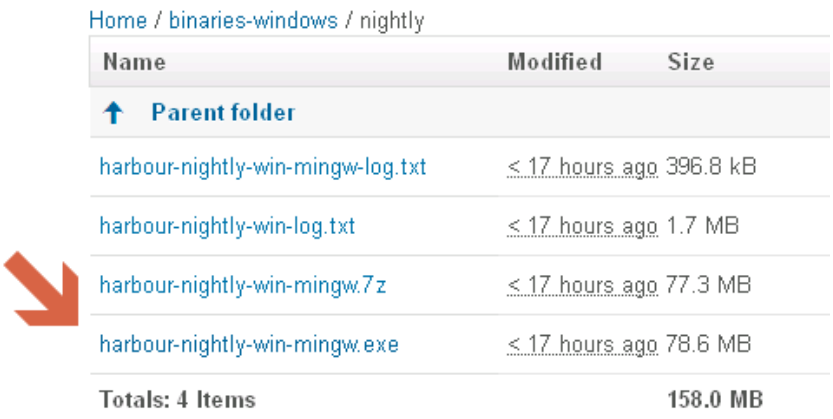


Figura G.5: Instalando o Harbour

Esse arquivo é a versão mais recente do Harbour juntamente com um compilador C chamado Mingw[1]. Ele tem aproximadamente 80 Mb.

O processo de instalação do Harbour é bem tranquilo, conforme mostram as figuras G.5, G.6, G.7 e G.8.

G.2 Compilando o Harbour 3.4 no Linux Ubuntu

Essa seção descreve a compilação do Harbour no Linux. Um processo qualquer de compilação exige um grau de conhecimento maior do que um processo de instalação. Mesmo sendo um processo mais complexo, nós detalhamos todo

[1] O Mingw é uma versão para Windows do compilador gcc (usado pelo Linux para compilar os programas desenvolvidos para Linux, inclusive o próprio kernel do Linux).

Figura G.6: Instalando o Harbour

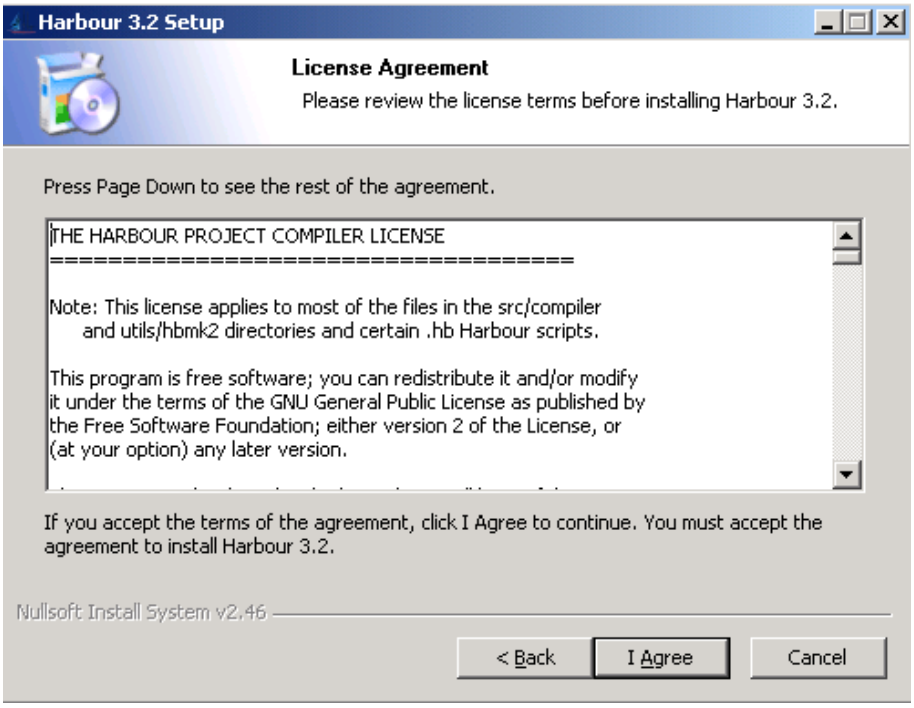


Figura G.7: Instalando o Harbour

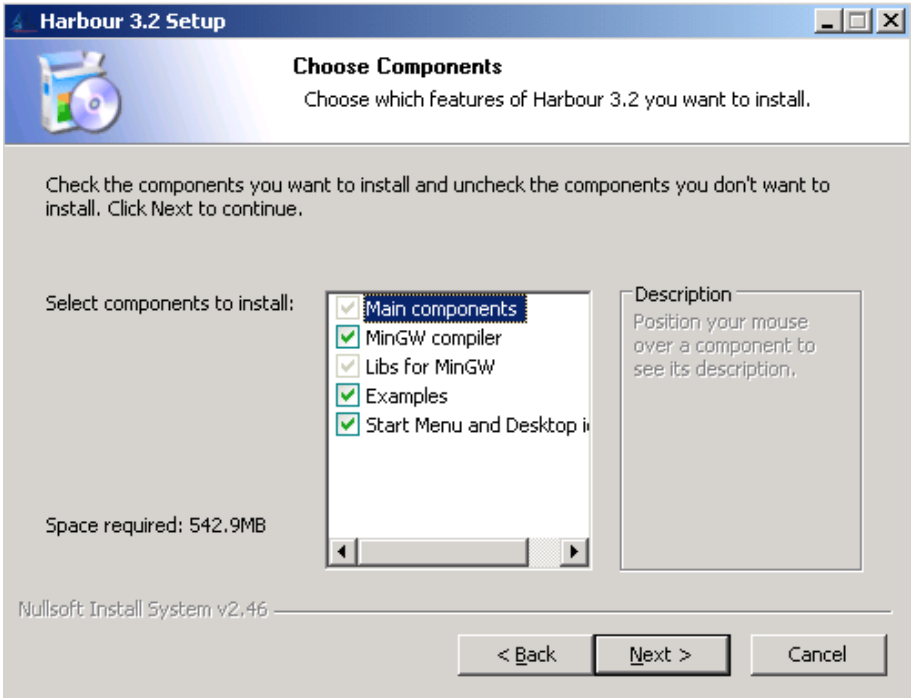
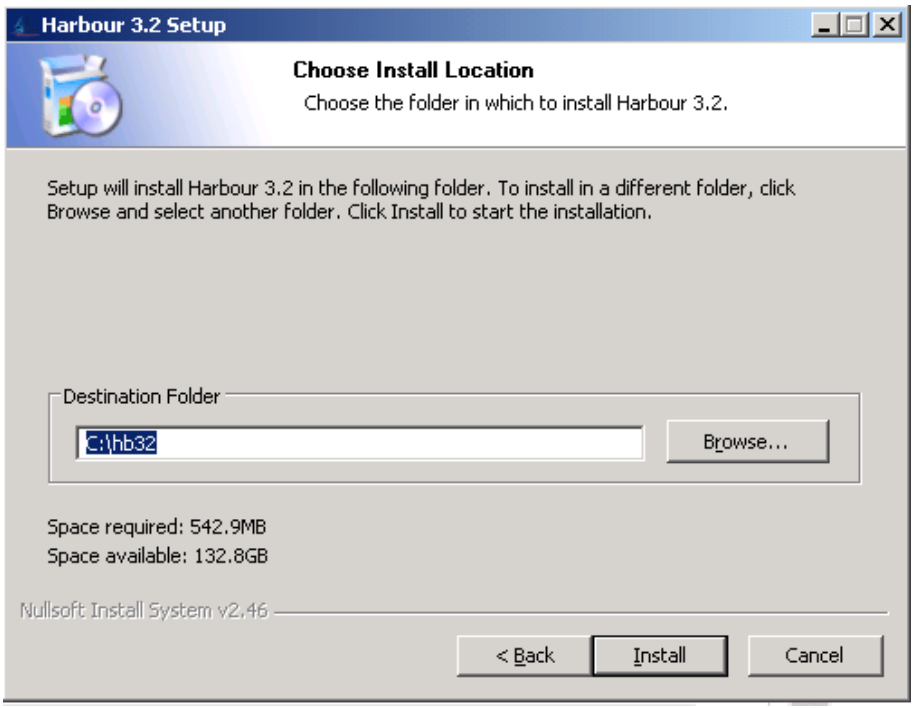


Figura G.8: Instalando o Harbour



o processo de compilação para que você não tenha surpresas. A distribuição do Linux que a compilação foi testada é a Ubuntu 16.04 LTS, que pode ser baixada de `http://www.ubuntu.com/download/desktop`. Nós pressupomos que você já possui uma máquina LINUX (pode ser uma máquina virtual) devidamente configurada e conectada a internet. Vamos agora aos passos :

1.Baixe a versão do harbour usando o cliente do git :

.:Resultado:.

```
git clone https://github.com/vszakats/harbour-core.git
```

Uma pasta chamada “harbour-core” será criada.

2.Para evitar ficar digitando sudo <nome dos comandos> se transforme logo em root com sudo -s

.:Resultado:.

```
sudo -s
```

3.MySQL: Abra o terminal e execute o comando :

.:Resultado:.

```
apt-get install  libmysqlclient-dev
```

4.PostgreSQL : Abra o terminal e execute o comando :

.:Resultado:.

```
apt-get install libpq-dev
```

Os passos 3 e 4 correspondem aos arquivos usados pelo gcc, durante a compilação do Harbour, para a configuração do acesso aos dois bancos de dados.

5. Pronto! Você agora tem o fonte do Harbour e os arquivos includes do PostgreSQL e MySQL. Quando você for compilar o Harbour você precisa informar onde esses arquivos (PostgreSQL e MySQL) estão localizados. Para fazer isso proceda assim :

   Primeiro localizando os arquivos do PostgreSQL

   .:Resultado:.

```
find / -name libpq

/usr/include/postgresql/8.4/server/libpq
/usr/include/postgresql/libpq
/usr/include/postgresql/internal/libpq
```

   Dos retornos acima o que você irá usar é /usr/include/postgresql/libpq

   Agora os arquivos libmysql

```
find / -name *mysql*h*

/usr/include/mysql/mysqlx_error.h
/usr/include/mysql/mysql_time.h
/usr/include/mysql/mysql_com_server.h
/usr/include/mysql/mysql.h
/usr/include/mysql/mysqld_error.h
/usr/include/mysql/mysql/service_mysql_pas
... CONTINUA A LISTAGEM
```

   O diretório é /usr/include/mysql

6. Entre na pasta “harbour-core”

   .:Resultado:.

```
make HB_WITH_PGSQL=/usr/include/postgresql/libpq
   HB_WITH_MYSQL=/usr/include/mysql
```

   (Note que os parâmetros HB... são passados pela linha de comando)

7. Se deu tudo certo deve dar uma mensagem tipo a mensagem abaixo nas últimas linhas :

   .:Resultado:.

```
! Built: Harbour 3.4.0dev (cba3a79) (2016-08-01 16:40) using C
   compiler: GNU C 5.3.1 (64-bit)
! postinst script finished
```

8. Agora digite make install para instalar os binários nas suas respectivas pastas.

.:Resultado:.

```
make install
```

9. Agora digite um programa para teste, como o abaixo e compile

   Conteúdo de teste.prg

```
FUNCTION MAIN()

  ? ""
  ? "Harbour funcionando"
  ? ""

RETURN NIL
```

10. Execute o programa gerado

.:Resultado:.

```
 teste
```

Quando fui executar irá aparecer uma mensagem de erro :

.:Resultado:.

```
./teste: error while loading shared libraries:
   libharbour.so.3.4: cannot open shared object file: No such
   file or directory
```

Não se preocupe, nós só precisamos achar o local onde as libs foram instaladas e avisar ao Linux que elas estão lá. Passo 1 : Encontrar, Passo 2: Registrar.

leftmargin=1cmEncontrando as libs do Harbour. Execute o comando find para achar o diretório onde as libs foram geradas.

.:Resultado:.

```
    find / -name libharbour.so.3.4
```

O comando me retorna 3 LINHAS RELEVANTES :

.:Resultado:.

```
/home/bolsista/src/harbour-core/lib/linux/gcc/libharbour.so.3
/usr/local/lib/libharbour.so.3.4
/usr/local/lib/harbour/libharbour.so.3.4
```

Vamos comentar as saídas: A primeira linha é a própria pasta onde o harbour foi compilado, NÃO É ELA, pois o comando make install copiou essas libs para outro local

Portando, ou é a segunda linha ou é a terceira linha o local onde as libs estão.

Na segunda linha dou ls -la /usr/local/lib para ver se é nessa pasta. O sistema me retorna :

.:Resultado:.

```
drwxr-xr-x  6 root root  4096 Ago  1 14:52 .
drwxr-xr-x 11 root root  4096 Ago  1 14:52 ..
drwxr-xr-x  2 root root  4096 Ago  1 14:52 harbour
lrwxrwxrwx  1 root root    27 Ago  1 14:52 libharbour.so ->
   harbour/libharbour.so.3.4.0
lrwxrwxrwx  1 root root    27 Ago  1 14:52
   libharbour.so.3.4 -> harbour/libharbour.so.3.4.0
lrwxrwxrwx  1 root root    27 Ago  1 14:52
   libharbour.so.3.4.0 -> harbour/libharbour.so.3.4.0
```

Não é nesse diretório, pois são apenas links simbólicos para o diretório /usr/local/lib/harbour (por isso usei ls -la)

Provavelmente é a terceira. Confirme com ls /usr/local/lib/harbour . O sistema retorna algo como :

.:Resultado:.

```
libbz2.a             libhbcommon.a  libhblang.a
   libhbpipeio.a  libhbvm.a      librddnsx.a
libed25519.a         libhbcpage.a   libhblzf.a
   libhbpp.a      libhbvmmt.a    librddntx.a
libexpat.a           libhbcplr.a    libhbmacro.a
   libhbrdd.a     libhbxdiff.a   librddsql.a
libgtcgi.a           libhbcrypto.a  libhbmemio.a
   libhbrtl.a     libhbxpp.a     librdduser.a
... ETC ...
```

Pronto, achei o local onde as libs estão : /usr/local/lib/harbour

leftmargin=1cmRegistrando as libs do Harbour

Pronto, confirmei que é o diretório /usr/local/lib/harbour agora devo acrescentar esse diretório nas buscas que o linux pelas libs dinâmicas.

Crie o arquivo /etc/ld.so.conf.d/harbour.conf O conteúdo do arquivo deve ser :

```
#harbour                                                    1
/usr/local/lib/harbour                                      2
```

Agora execute o comando ldconfig na linha de comando

Agora tente executar novamente o teste :

.:Resultado:.

```
./teste

Harbour funcionando
```

Agora sim, o Harbour está funcionando. Se quiser saber o porque disso leia nas próximas linhas :

As distribuições do linux diferem um pouco uma da outra e os executáveis Harbour (na verdade qualquer executável C) necessitam registrar o local

onde as suas libs ficam instaladas. A distribuição Ubuntu mantem isso no diretório /etc/ldconfig . A localização desse arquivo difere de distribuição para distribuição, por isso temos que criar nós mesmos. O mais importante é registrar com o comando ldconfig (ele só precisa ser executado uma vez).

Em resumo temos :

Binários em /usr/local/bin Libs em /usr/local/lib/harbour Includes em /usr/local/include Contribs em /usr/local/share/harbour/contrib

H Exercícios : constantes e variáveis

> Somos o que repetidamente fazemos. A excelência, portanto, não é um feito, mas um hábito.
>
> Aristóteles

Objetivos do capítulo

- Praticar o que foi visto até agora.

H.1 Resposta aos exercícios de fixação sobre variáveis - Capítulo 4

1.Escreva um programa que declare 3 variáveis e atribua a elas valores numéricos através do comando INPUT; depois mais 3 variáveis e atribua a elas valores caracteres através do comando ACCEPT; finalmente imprima na tela os valores.

Listagem H.1: Resposta

```
/*
Entrada : 3 variáveis numéricas e 3 variáveis caracteres
Saída : As variáveis mostradas
*/
PROCEDURE Main
LOCAL nA, nB, nC // Variáveis numéricas
LOCAL cA, cB, cC // Variáveis caracteres

   INPUT "Insira o primeiro valor numérico : " TO nA
   INPUT "Insira o segundo valor numérico : " TO nB
   INPUT "Insira o terceiro valor numérico : " TO nC

   ACCEPT "Insira o primeiro valor caractere : " TO cA
   ACCEPT "Insira o segundo valor caractere : " TO cB
   ACCEPT "Insira o terceiro valor caractere : " TO cC

   ? "Os valores numéricos : " , nA, nB, nC
   ? "Os valores caracteres : " , cA, cB, cC

RETURN
```

2.[Horstman 2005, p. 47] Escreva um programa que exibe a mensagem “Oi, meu nome é Hal!”. Então, em uma nova linha, o programa deve imprimir a mensagem “O que você gostaria que eu fizesse ?”. Então é a vez do usuário digitar uma entrada. [...] Finalmente, o programa deve ignorar a entrada do usuário e imprimir uma mensagem “Sinto muito, eu não posso fazer isto.”. Aqui está uma execução típica :

.:Resultado:.

```
Oi, meu nome é Hal!
O que você gostaria que eu fizesse ?
A limpeza do meu quarto.
Sinto muito, eu não posso fazer isto.
```

Comentário : nesse exemplo você deve ter recebido o aviso de warning :

.:Resultado:.

```
r0402.prg(16) Warning W0032  Variable 'CENTRADA' is assigned
   but not used in function 'MAIN(11)'
```

Esse aviso, nós já vimos, é um “warning”. Ele não impedirá o seu programa de ser gerado, mas avisa que algo possivelmente está errado. Você consegue identificar o “possível erro” ? Pense um pouco a respeito. A resposta está nessa nota de rodapé[1].

Listagem H.2: Resposta

```
/*
Entrada : 3 variáveis numéricas e 3 variáveis caracteres
Saída : As variáveis mostradas
*/
PROCEDURE Main
LOCAL cEntrada // Valor a ser digitado

   ? "Oi, meu nome é Hal!"
   ? "O que você gostaria que eu fizesse ?"
   ?
   ACCEPT TO cEntrada

   ? "Sinto muito, eu näo posso fazer isso"

RETURN
```

3.[Horstman 2005, p. 47] Escreva um programa que imprima uma mensagem “Oi, meu nome é Hal!”. Então, em uma nova linha, o programa deve imprimir a mensagem “Qual o seu nome ?” [...] Finalmente, o programa deve imprimir a mensagem “Oi, *nome do usuário*. Prazer em conhecê-lo!” Aqui está uma execução típica :

.:Resultado:.

```
Oi, meu nome é Hal!
Qual é o seu nome ?
Dave
Oi, Dave. Prazer em conhecê-lo.
```

Listagem H.3: Resposta

```
/*
Entrada : 3 variáveis numéricas e 3 variáveis caracteres
Saída : As variáveis mostradas
*/
PROCEDURE Main
LOCAL cEntrada  // Valor a ser digitado

   ? "Oi, meu nome é Hal!"
```

[1]O “possível erro” é o seguinte : você declarou uma variável (cEntrada), depois a inicializou com o comando ACCEPT, mas não fez nada com ela. Isso é muito estranho. Por que alguém iria criar uma variável e não fazer nada com ela ? Por isso o compilador emitiu um aviso informando que ela (a variável) recebeu (assigned) um valor mas ele não foi usado. Mas, como o nosso exercício não previa uso para a variável cEntrada, podemos ignorar esse aviso.

```
   ? "Qual é o seu nome ?"                                          9
   ? // Pula uma linha (fica melhor para o usuário)                 10
   ACCEPT TO cEntrada                                               11
                                                                    12
   ? "Oi," , cEntrada, "prazer em conhecê-lo."                      13
                                                                    14
RETURN                                                              15
```

4.Escreva um programa que receba quatro números e exiba a soma desses números.

Listagem H.4: Resposta

```
/*                                                                  1
Descr. : Programa que recebe quatro números,                        2
         calcula e mostra a soma desses números.                    3
Entrada: quatro números.                                            4
Saída  : a soma desses números exibida.                             5
*/                                                                  6
PROCEDURE Main                                                      7
LOCAL n1, n2, n3, n4 // Parcelas                                    8
LOCAL nSoma // Soma                                                 9
                                                                    10
   INPUT "Entre com valor 1 : " TO n1                               11
   INPUT "Entre com valor 2 : " TO n2                               12
   INPUT "Entre com valor 3 : " TO n3                               13
   INPUT "Entre com valor 4 : " TO n4                               14
                                                                    15
   nSoma := n1 + n2 + n3 + n4                                       16
   ? "A soma desses números é " , nSoma                             17
                                                                    18
RETURN                                                              19
```

5.Escreva um programa que receba três notas e exiba a média dessas notas.

Listagem H.5: Resposta

```
/*                                                                  1
Descr. : Programa que recebe três notas,                            2
         calcula e mostra a média desses números.                   3
Entrada: três números.                                              4
Saída  : a média desses números exibida.                            5
*/                                                                  6
PROCEDURE Main                                                      7
LOCAL n1, n2, n3 // Valores                                         8
LOCAL nMedia // Média dos valores                                   9
                                                                    10
   INPUT "Entre com valor 1 : " TO n1                               11
   INPUT "Entre com valor 2 : " TO n2                               12
   INPUT "Entre com valor 3 : " TO n3                               13
                                                                    14
   nMedia := (( n1 + n2 + n3 ) / 3 )                                15
```

```
   ? "A média desses números é " , nMedia

RETURN
```

6. Escreva um programa que receba três números, três pesos e mostre a média ponderada desses números.

Listagem H.6: Resposta

```
/*
Descr. : Programa que recebe três notas e seus pesos,
         calcula e mostra a média ponderada desses números.
Entrada: três números e três pesos.
Saída  : a média ponderada desses números exibida.
*/
PROCEDURE Main
LOCAL n1, n2, n3 // Valores
LOCAL p1, p2, p3 // Pesos
LOCAL nMedia // Média

   INPUT "Entre com valor 1 : " TO n1
   INPUT "Entre com peso 1 : " TO p1
   INPUT "Entre com valor 2 : " TO n2
   INPUT "Entre com peso 2 : " TO p2
   INPUT "Entre com valor 3 : " TO n3
   INPUT "Entre com peso 3 : " TO p3


   nMedia := ( (( n1 * p1 ) + ( n2 * p2 ) + ( n3 * p3 ) ) / ( p1 + p2 + p3
   ? "A média ponderada desses números é " , nMedia

RETURN
```

7. Escreva um programa que receba o salário de um funcionário, receba o percentual de aumento (por exemplo 15 se for 15%) e exiba o novo salário do funcionário.

Listagem H.7: Resposta

```
/*
Descr. : Programa que recebe o salário do funcionário, o percentual de rea
         calcule e mostre o novo salário.
Entrada: Salário.
Saída  : O salário reajustado.
*/
PROCEDURE Main
LOCAL nSalario
LOCAL nAumento
LOCAL nSalarioNovo

   INPUT "Entre com valor do salário : " TO nSalario

   INPUT "Entre com o percentual de aumento (Ex: 15 se for 15%) : " TO nAu
```

```
                                                                    14
   nSalarioNovo := nSalario + ( nSalario * nAumento ) / 100         15
   ? "O novo salário é " , nSalarioNovo                             16
                                                                    17
RETURN                                                              18
```

8. Modifique o programa anterior para exibir também o valor do aumento que o funcionário recebeu.

Listagem H.8: Resposta

```
/*                                                                  1
Descr. : Programa que recebe o salário do funcionário,              2
         o percentual de aumento,                                   3
         calcule e mostre o novo salário, e o valor do aumento.     4
Entrada: Salário e percentual de aumento.                           5
Saída  : O novo salário e o valor do aumento.                       6
*/                                                                  7
PROCEDURE Main                                                      8
LOCAL nSalario                                                      9
LOCAL nPercAumento                                                  10
LOCAL nValorAumento                                                 11
LOCAL nSalarioNovo                                                  12
                                                                    13
   INPUT "Entre com valor do salário : " TO nSalario                14
                                                                    15
   INPUT "Entre com o percentual de aumento (Ex: 15) : " TO nPercAumento
                                                                    16
   nValorAumento := ( nSalario * nPercAumento ) / 100               17
   nSalarioNovo := nSalario + nValorAumento                         18
   ? "O novo salário é " , nSalarioNovo                             19
   ? "O aumento foi de " , nValorAumento                            20
                                                                    21
RETURN                                                              22
```

9. Modifique o programa anterior para receber também o percentual do imposto a ser descontado do salário novo do funcionário, e quando for exibir os dados (salário novo e valor do aumento), mostre também o valor do imposto que foi descontado.

Listagem H.9: Resposta

```
/*                                                                  1
Descr. : Programa que recebe o salário do funcionário,              2
         o percentual de aumento, o percentual do imposto           3
                                                                    4
   calcule e mostre o novo salário, o percentual do imposto
         e o valor do aumento.                                      5
Entrada: Salário, percentual do imposto e percentual de aumento.    6
Saída  : O novo salário, o valor do aumento e o valor do imposto.   7
*/                                                                  8
PROCEDURE Main                                                      9
```

```
LOCAL nSalario
LOCAL nPercAumento
LOCAL nPercImposto
LOCAL nValorAumento
LOCAL nValorDoImposto
LOCAL nSalarioNovo

   INPUT "Entre com valor do salário : " TO nSalario

   INPUT "Entre com o percentual de aumento (Ex: 25) : " TO nPercAumento

   INPUT "Entre com o percentual de imposto (Ex: 10) : " TO nPercImposto

   nValorAumento := ( nSalario * nPercAumento ) / 100
   nSalarioNovo := nSalario + nValorAumento
   nValorDoImposto := ( nSalarioNovo * nPercImposto ) / 100
   nSalarioNovo := nSalarioNovo - nValorDoImposto
   ? "O novo salário é " , nSalarioNovo
   ? "O aumento foi de " , nValorAumento
   ? "O imposto foi de " , nValorDoImposto

RETURN
```

10. Escreva um programa que receba um valor a ser investido e o valor da taxa de juros. O programa deve calcular e mostrar o rendimento e o valor total depois do rendimento.

Listagem H.10: Resposta

```
/*
Descr. : Recebe o valor de um depósito e o valor da taxa de juros,
         calcule e mostre o valor do rendimento e o valor total
         depois do rendimento.
Entrada: Depósito e taxa de juros.
Saída  : O valor do rendimento e o valor total depois do rendimento.
*/
PROCEDURE Main
LOCAL nDeposito // Valor depositado
LOCAL nTaxaJuros // Taxa de juros
LOCAL nRendimento // Rendimento
LOCAL nValorFinal // Valor final

   INPUT "Entre com valor do depósito : " TO nDeposito
   INPUT "Entre com a taxa de juros (Ex: 25) : " TO nTaxaJuros

   nRendimento := ( nDeposito * nTaxaJuros ) / 100
   nValorFinal := nDeposito + nRendimento
   ? "O valor do rendimento foi " , nRendimento
   ? "O valor total depois do rendimento foi " , nValorFinal

RETURN
```

11. Calcule a área de um triangulo. O usuário deve informar a base e a altura e o programa deve retornar a área.

    Dica : $nArea = \frac{nBase * nAltura}{2}$

Listagem H.11: Resposta

```
/*
Descr. : Calcula a área de um triangulo sabendo que
          Area = ( Base * Altura ) / 2
Entrada: Base e altura.
Saída  : O valor da área de um triangulo.
*/
PROCEDURE Main
LOCAL nBase // Base
LOCAL nAltura // Altura
LOCAL nArea // Area

   INPUT "Entre com a base : " TO nBase
   INPUT "Entre com a altura : " TO nAltura

   nArea := ( nBase * nAltura ) / 2
   ? "O valor da área é  " , nArea

RETURN
```

12. Escreva um programa que receba o ano do nascimento do usuário e retorne a sua idade e quantos anos essa pessoa terá em 2045.

    Dica : YEAR(DATE()) é uma combinação de duas funções que retorna o ano corrente.

Listagem H.12: Resposta

```
/*
Entrada: ano de nascimento e o ano atual
Saída  : a idade da pessoa, quantos anos ela terá em 2045

Dica : YEAR( DATE() ) retorna o ano corrente.
*/
#define ANO_FUTURO 2045
PROCEDURE Main
LOCAL nAnoNasc, nAno := YEAR( DATE() )
LOCAL nIdade, nIdadeFuturo

   INPUT "Entre com o ano do seu nascimento : " TO nAnoNasc

   nIdade := nAno - nAnoNasc
   nIdadeFuturo := ANO_FUTURO - nAnoNasc

   ? "Sua idade atual é " , nIdade
   ? "Em" , ANO_FUTURO, "você terá " , nIdadeFuturo , "anos"

RETURN
```

13. Escreva um programa que receba uma medida em pés e converta essa medida para polegadas, jardas e milhas.

    Dicas

    - 1 pé = 12 polegadas
    - 1 jarda = 3 pés
    - 1 milha = 1,76 jardas

Listagem H.13: Resposta

```
/*
Descr. : Recebe uma medida em pés e converte a medida
         em polegada, jarda, milha
Entrada: O número em pés
Saída  : O valor da entrada em pés e converte a medida
         em polegada, jarda, milha


Nota : pé = 12 polegadas
       1 jarda = 3 pés
       1 milha = 1,760 jarda

*/
PROCEDURE Main
LOCAL nPe
LOCAL nPolegada, nJarda, nMilha

   INPUT "Entre com a medida em pés : " TO nPe

   nPolegada := nPe * 12
   nJarda := nPe / 3
   nMilha := nJarda / 1760

   ? "O valor em polegadas é " , nPolegada
   ? "O valor em jardas é " , nJarda
   ? "O valor em milhas é " , nMilha

RETURN
```

14. Escreva um programa que receba dois números (nBase e nExpoente) e mostre o valor da potência do primeiro elevado ao segundo.

Listagem H.14: Resposta

```
/*
Entrada: Base e expoente
Saída  : a potência
*/
PROCEDURE Main
LOCAL nBase, nExpoente
LOCAL nPotencia
```

```
                                                                    8
   INPUT "Entre com a base : " TO nBase                             9
   INPUT "Entre com o expoente : " TO nExpoente                     10
                                                                    11
   nPotencia := nBase ^ nExpoente                                   12
                                                                    13
   ? "O valor da potência é " , nPotencia                           14
                                                                    15
RETURN                                                              16
```

15. Escreva um programa que receba do usuário o nome e a idade dele . Depois de receber esses dados o programa deve exibir o nome e a idade do usuário convertida em meses. Use o exemplo abaixo como modelo :

.:Resultado:.

```
    Digite o seu nome : Paulo
    Digite quantos anos você tem : 20

    Seu nome é Paulo e você tem aproximadamente 240 meses de
  vida.
```

Dica : Lembre-se que o sinal de multiplicação é um “*”.

Listagem H.15: Resposta

```
/*                                                                  1
Entrada: Nome e idade                                               2
Saída  : Meses de vida aproximados                                  3
*/                                                                  4
#define MESES_POR_ANO 12                                            5
PROCEDURE Main                                                      6
LOCAL cNome                                                         7
LOCAL nIdade, nMeses                                                8
                                                                    9
   ACCEPT "Informe o seu nome : " TO cNome                          10
   INPUT "Entre com a idade : " TO nIdade                           11
                                                                    12
   nMeses := nIdade * MESES_POR_ANO                                 13
                                                                    14
                                                                    15
   ? "O seu nome é " , cNome, " e você tem aproximadamente " , nMeses , "
                                                                    16
RETURN                                                              17
```

16. Escreva um programa que receba do usuário um valor em horas e exiba esse valor convertido em segundos. Conforme o exemplo abaixo :

.:Resultado:.

```
    Digite um valor em horas : 3
```

```
     3    horas tem       10800      segundos
```

Listagem H.16: Resposta

```
/*
Entrada: Valor em horas
Saída  : Valor convertido para segundos
*/
#define SEGUNDOS_POR_HORA 60*60 // 1 hora = 60 minutos * 60 segundos
PROCEDURE Main
LOCAL nHora, nSegundos

    INPUT "Entre com o valor em horas : " TO nHora

   nSegundos := nHora * SEGUNDOS_POR_HORA

   ? nHora, "horas tem" , nSegundos, "segundos."

RETURN
```

17. Faça um programa que informe o consumo em quilômetros por litro de um veículo. O usuário deve entrar com os seguintes dados : o valor da quilometragem inicial, o valor da quilometragem final e a quantidade de combustível consumida.

Listagem H.17: Resposta

```
/*
Entrada: Quilometragem inicial e quilometragem final, e quantidade de comb
Saída  : Valor do consumo
*/
PROCEDURE Main
LOCAL nKmIni, nKmFim // Quilometragem
LOCAL nQtd // Quantidade consumida
LOCAL nConsumo // Consumo


   INPUT "Entre com o valor da quilometragem inicial : " TO nKmIni
   INPUT "Entre com o valor da quilometragem final : " TO nKmFim
   INPUT "Entre com a quantidade consumida : " TO nQtd

   nConsumo := ( nKmFim - nKmIni ) / nQtd

   ? "O consumo é de " , nConsumo , " Km/l"

RETURN
```

H.2 Respostas aos desafios - Capítulo 4

H.2.1 Identifique o erro de compilação no programa abaixo.

Listagem H.18: Erro 1

```
/*
Onde está o erro ?
*/
PROCEDURE Main
LOCAL x, y, x // Número a ser inserido

   x := 5
   y := 10
   x := 20

RETURN
```

Resposta : A variável x foi declarada duas vezes.

H.2.2 Identifique o erro de lógica no programa abaixo.

Um erro de lógica é quando o programa consegue ser compilado mas ele não funciona como o esperado. Esse tipo de erro é muito perigoso pois ele não impede que o programa seja gerado. Dessa forma, os erros de lógica só podem ser descoberto durante a seção de testes ou (pior ainda) pelo cliente durante a execução. Um erro de lógica quase sempre é chamado de *bug*.

Listagem H.19: Erro de lógica

```
/*
Onde está o erro ?
*/
PROCEDURE Main
LOCAL x, y // Número a ser inserido

   x := 5
   y := 10
   ACCEPT "Informe o primeiro número : " TO x
   ACCEPT "Informe o segundo número : " TO y
   ? "A soma  é ", x + y

RETURN
```

Resposta : O comando ACCEPT foi usado para receber valores numéricos.

H.2.3 Valor total das moedas

[Horstman 2005, p. 50] Eu tenho 8 moedas de 1 centavo, 4 de 10 centavos e 3 de 25 centavos em minha carteira. Qual o valor total de moedas ? Faça um programa que calcule o valor total para qualquer quantidade de moedas informadas.

Siga o modelo :

.:Resultado:.

```
Informe quantas moedas você tem :
```

```
Quantidade de moedas de 1 centavo :  10
Quantidade de moedas de 10 centavos : 3
Quantidade de moedas de 25 centavos : 4

Você tem     1.40 em moedas.
```

Listagem H.20: Resposta

```
/*
Entrada: Quantidade de moedas (25, 10 e 1 centavos)
Saída  : Valor em moedas
*/
#define FATOR_25 0.25  // Fator de conversäo para 25 centavos
#define FATOR_10 0.1 // Fator de conversão para 10 centavos
#define FATOR_01 0.01 // Fator de conversão para 1 centavo
PROCEDURE Main
LOCAL nQtd1Cent, nQtd10Cent, nQtd25Cent // Quantidades
LOCAL nValor // Valor em moedas

   INPUT "Quantidade de moedas de um centavo : " TO nQtd1Cent
   INPUT "Quantidade de moedas de dez centavos : " TO nQtd10Cent

   INPUT "Quantidade de moedas de vinte e cinco centavos : " TO nQtd25Cent


   nValor := ( nQtd25Cent * FATOR_25 ) + ( nQtd10Cent * FATOR_10 ) + ( nQtd1Ce

   ? "Você tem R$" , nValor, "em moedas"

RETURN
```

H.2.4 O comerciante maluco

Adaptado de [Forbellone e Eberspacher 2005, p. 62]. Um dado comerciante maluco cobra 10% de acréscimo para cada prestação em atraso e depois dá um desconto de 10% sobre esse valor. Faça um programa que solicite o valor da prestação em atraso e apresente o valor final a pagar, assim como o prejuízo do comerciante na operação.

Listagem H.21: Resposta

```
/*
Entrada : O valor da parcela
Saída : O valor final após o acréscimo, o valor final após o desconto e o prej
*/
#define DESCONTO 0.10
#define ACRESCIMO 0.10
PROCEDURE Main
LOCAL nParcela // nValor inicial
LOCAL nValorAcr // nValor final com acréscimo
LOCAL nValorDesc // Valor final após o desconto

   INPUT "Informe o valor da parcela em atraso : " TO nParcela
```

```
                                                                    13
   nValorAcr := nParcela + ( nParcela * ACRESCIMO )                 14
                                                                    15
   ? "O valor da parcela acrescida de ", ACRESCIMO ,"por cento é " , nValorAcr
   nValorDesc := nValorAcr - ( nValorAcr * DESCONTO )               16
   ? "O valor descontado ", DESCONTO, "por cento é " , nValorDesc   17
   ? "O prejuízo é de " , nParcela - nValorDesc                     18
                                                                    19
RETURN                                                              20
```

Nota: Na resposta eu acrescentei, além do valor final a pagar (nValorDesc) e o valor do prejuízo, o valor da parcela acrescida. Isso torna o problema mais claro. Mas se você não mostrou o valor da parcela acrescida (nValorAcr) não tem problema.

H.3 Resposta aos exercícios de fixação sobre variáveis - Capítulo 5

1.Calcule a área de um círculo. O usuário deve informar o valor do raio.

Dica : $nArea = PI * nRaio^2$.

Lembrete : Não esqueça de criar a constante PI (Onde PI = 3.1415) .

Listagem H.22: Resposta

```
/*                                                                  1
Descr. : Calcula a área de um circulo sabendo que                   2
          Area = PI * Raio ^ 2                                      3
Entrada: O raio.                                                    4
Saída  : O valor da área de um círculo.                             5
*/                                                                  6
#define PI_VALOR 3.1415                                             7
PROCEDURE Main                                                      8
LOCAL nRaio  // Raio da circunferência                              9
LOCAL nArea // Área                                                 10
                                                                    11
   INPUT "Entre com o raio : " TO nRaio                             12
                                                                    13
   nArea := PI_VALOR * nRaio ^ 2                                    14
   ? "O valor da área é  " , nArea                                  15
                                                                    16
RETURN                                                              17
```

2.Escreva um programa que receba um valor numérico e mostre :

- •O valor do número ao quadrado.
- •O valor do número ao cubo.
- •O valor da raiz quadrada do número.
- •O valor da raiz cúbica do número.

Nota : suponha que o usuário só irá informar números positivos.

Dica : lembre-se que $\sqrt[2]{100} = 100^{\frac{1}{2}}$

Listagem H.23: Resposta

```
/*
Descr. : Calcula o número ao quadrado, ao cubo, a raiz quadrada
         e a raiz cúbica do número
Entrada: O número.
Saída  : o número ao quadrado, ao cubo, a raiz quadrada
         e a raiz cúbica do número

Nota : A raiz é o número elevado a 1 / potência
*/
PROCEDURE Main
LOCAL nNumero

   INPUT "Entre com o raio : " TO nNumero

   ? "O quadrado é  " , nNumero ^ 2
   ? "O cubo é  " , nNumero ^ 3
   ? "A raiz quadrada é  " , nNumero ^ ( 1 / 2 )
   ? "A raiz cúbica é  " , nNumero ^ ( 1 / 3 )

RETURN
```

3. Construa um programa para calcular o volume de uma esfera de raio R, em que R é um dado fornecido pelo usuário.

   Dica : o volume V de uma esfera é dado por $V = \frac{4*PI*Raio^3}{3}$.

Listagem H.24: Resposta

```
/*
Resposta
*/
#define PI 3.14
PROCEDURE Main
LOCAL nRaio

   CLS
   ? "Cálculo do volume de uma esfera"
   INPUT "Informe o valor do raio da esfera " TO nRaio
   ? "O volume da esfera é : " , ( ( 4 * PI * nRaio ) ^ 3 ) / 3

RETURN
```

   Comentário: note o uso de parênteses para evitar que a fórmula seja aplicada erroneamente.

4. Construa programas para reproduzir as equações abaixo. Considere que o lado esquerdo da equação seja a variável que queremos saber o valor. As variáveis

do lado direito devem ser informadas pelo usuário. Siga o exemplo no modelo abaixo :

IMPORTANTE: Estamos pressupondo que o usuário é um ser humano perfeito e que não irá inserir letras nem irá forçar uma divisão por zero.

•$z = \frac{1}{x+y}$
Modelo :

```
PROCEDURE Main
LOCAL x,y,z

     INPUT "Insira o valor de x : " TO x
     INPUT "Insira o valor de y : " TO y
     z = ( 1 / ( x + y ) )
     ? "O valor de z é : " , z


RETURN
```

.:Resultado:.

```
Insira o valor de x : 10
Insira o valor de y : 20
O valor de z é :          0.03
```

Agora faça o mesmo com as fórmulas seguintes.

•$x = \frac{y-3}{z}$

Listagem H.25: Resposta

```
/*
Resposta
*/
//$ x = \frac{y-3}{z}$
PROCEDURE Main
LOCAL nY, nZ

   CLS

   INPUT "Informe o valor de y : " TO nY
   INPUT "Informe o valor de z : " TO nZ
   ? ( nY - 3 ) / nZ

RETURN
```

•$k = \frac{x^3+z/5}{y^2+8}$

Listagem H.26: Resposta

```
/*
Resposta
```

```
*/
PROCEDURE Main
LOCAL nY, nZ, nX

   CLS

   INPUT "Informe o valor de x : " TO nX
   INPUT "Informe o valor de y : " TO nY
   INPUT "Informe o valor de z : " TO nZ
   ? ( nX^3 + nZ/5 ) / ( nY^2 + 8 )

RETURN
```

•$y = \frac{x^3 z}{r} - \frac{4n}{h}$

Listagem H.27: Resposta

```
/*
Resposta
*/
PROCEDURE Main
LOCAL nH, nZ, nX, nN, nR

   CLS

   INPUT "Informe o valor de x : " TO nX
   INPUT "Informe o valor de z : " TO nZ
   INPUT "Informe o valor de r : " TO nR
   INPUT "Informe o valor de h : " TO nH
   INPUT "Informe o valor de n : " TO nN
   ? ( ( (nX ^ 3) * nZ )/ nR ) - ( ( 4*nN ) * nH  )

RETURN
```

•$t = \frac{y}{2} + \frac{3k^2}{4n}$

Listagem H.28: Resposta

```
/*
Resposta
*/
PROCEDURE Main
LOCAL nY, nK, nN

   CLS

   INPUT "Informe o valor de y : " TO nY
   INPUT "Informe o valor de k : " TO nK
   INPUT "Informe o valor de n : " TO nN
   ? ( (nY) / 2 ) + ( ( (3*nK)^2 ) / 4*nN )

RETURN
```

5. Crie um programa que leia dois valores para as variáveis cA e cB, e efetue a troca dos valores de forma que o valor de cA passe a possuir o valor de cB e o valor de cB passe a possuir o valor de cA. Apresente os valores trocados.

   Dica : Utilize uma variável auxiliar cAux.

6. São dadas três variáveis nA,nB e nC. Escreva um programa para trocar seus valores da maneira a seguir:

   - nB recebe o valor de nA
   - nC recebe o valor de nB
   - nA recebe o valor de nC

   Dica : Utilize uma variável auxiliar nAux.

7. Leia uma temperatura em graus Fahrenheit e apresentá-la convertida em graus Celsius. A fórmula de conversão é $nC = (nF - 32) * (5/9)$, onde nC é o valor em Celsius e nF é o valor em Fahrenheit.

8. Uma oficina mecânica precisa de um programa que calcule os custos de reparo de um motor a diesel padrão NVA-456. O custo é calculado com a fórmula $nCusto = \frac{nMecanicos}{0.4} * 1000$. O programa deve ler um valor para o número de mecânicos (nMecanicos) e apresentar o valor de custo (nCusto). Os componentes da equação do custo estão listados abaixo :

   - nCusto = Preço de custo de reparo do motor.
   - nMecanicos = Número de mecânicos envolvidos.
   - 0.4 = Constante universal de elasticidade do cabeçote.
   - 1000 = Constante para conversão em valor monetário.

9. Escreva um programa que receba um valor (par ou ímpar) e imprima na tela os 3 próximos sequenciais pares ou ímpares.

   Por exemplo

   .:Resultado:.

```
Informe o número : 4
     6,     8 e    10.
```

   outro exemplo (com um valor ímpar)

   .:Resultado:.

```
Informe o número : 5
    7,    9 e    11.
```

10. No final do capítulo anterior nós mostramos um exemplo de um programa que calcula a quantidade de números entre dois valores quaisquer (incluindo esses valores). A listagem está reproduzida a seguir :

codigos/pratica_var.prg

```
/*
Descrição: Calcula quantos números existem entre dois intervalos
           (incluídos os números extremos)
Entrada: Limite inferior (número inicial) e limite superior (número final)
Saída: Quantidade de número (incluídos os extremos)
*/
#define UNIDADE_COMPLEMENTAR 1 // Deve ser adicionada ao resultado final
PROCEDURE Main
LOCAL nIni, nFim // Limite inferior e superior
LOCAL nQtd // Quantidade

   ? "Informa quantos números existem entre dois intervalos"
   ? "(Incluídos os números extremos)"
   INPUT "Informe o número inicial : " TO nIni
   INPUT "Informe o número final : " TO nFim
   nQtd := nFim - nIni + UNIDADE_COMPLEMENTAR

   ? "Entre os números " , nIni, " e " , nFim, " existem ", nQtd, " número

RETURN
```

Modifique esse programa (salve-o como excluidos.prg) para que ele passe a calcular a quantidade entre dois valores quaisquer, excluíndo esses valores limites.

H.4 Resposta aos exercícios de fixação sobre condicionais - Capítulo 7

1.[Ascencio e Campos 2014, p. 45] Faça um programa que receba o número de horas trabalhadas e o valor do salário mínimo, calcule e mostre o salário a receber, seguindo essas regras :

a)a hora trabalhada vale a metade do salário mínimo.

b)o salário bruto equivale ao número de horas trabalhadas multiplicado pelo valor da hora trabalhada.

c)o imposto equivale a 3% do salário bruto.

d)o salário a receber equivale ao salário bruto menos o imposto

Listagem H.29: Resposta

```
/*
Resposta
*/
#define PI 3.14
PROCEDURE Main
LOCAL nRaio
```

```
   CLS
   ? "Cálculo do volume de uma esfera"
   INPUT "Informe o valor do raio da esfera " TO nRaio
   ? "O volume da esfera é : " , ( ( 4 * PI * nRaio ) ^ 3 ) / 3

RETURN
```

2.[Ascencio e Campos 2014, p. 64] Faça um programa que receba três números e mostre-os em ordem crescente. Suponha que o usuário digitará três números diferentes.

3.Altere o programa anterior para que ele mesmo não permita que o usuário digite pelo menos dois números iguais.

4.[Ascencio e Campos 2014, p. 78] Faça um programa para resolver equações de segundo grau.

a)O usuário deve informar os valores de *a*, *b* e *c*.

b)O valor de *a* deve ser diferente de zero.

c)O programa deve informar o valor de delta.

d)O programa deve informar o valor das duas raízes.

5.[Forbellone e Eberspacher 2005, p. 46] Faça um programa que leia o ano de nascimento de uma pessoa, calcule e mostre sua idade e verifique se ela já tem idade para votar (16 anos ou mais) e para conseguir a carteira de habilitação (18 anos ou mais).

6.[Forbellone e Eberspacher 2005, p. 47] O IMC (Índice de massa corporal) indica a condição de peso de uma pessoa adulta. A fórmula do IMC é igual ao peso dividido pelo quadrado da altura. Elabore um programa que leia o peso e a altura de um adulto e mostre a sua condição de acordo com a tabela abaixo :

a)abaixo de 18.5 : abaixo do peso

b)entre 18.5 : peso normal

c)entre 25 e 30 : acima do peso

d)acima de 30 : obeso

H.5 Resposta aos exercícios de fixação sobre estruturas de repetição - Capítulo 9

1.[Forbellone e Eberspacher 2005, p. 65] Construa um programa que leia um conjunto de dados contendo altura e sexo (“M” para masculino e “F” para feminino) de 10 pessoas e, depois, calcule e escreva :

a)a maior e a menor altura do grupo

b)a média de altura das mulheres

c)o número de homens e a diferença porcentual entre eles e as mulheres

2.[Forbellone e Eberspacher 2005, p. 66] Anacleto tem 1,50 metro e cresce 2 centímetros por ano, enquanto Felisberto tem 1,10 metro e cresce 3 centímetros por ano. Construa um programa que calcule e imprima quantos anos serão necessários para que Felisberto seja maior do que Anacleto.

3.Adaptado de [Deitel e Deitel 2001, p. 173] Identifique e corrija os erros em cada um dos comandos (Dica: desenvolva pequenos programinhas de teste com eles para ajudar a descobrir os erros.) :

a)
```
nProduto := 10
c := 1
DO WHILE ( c <= 5 )
    nProduto *= c
ENDDO
++c
```

b)
```
x := 1
DO WHILE ( x <= 10 )
    x := x - 1
ENDDO
```

c)O código seguinte deve imprimir os valores de 1 a 10.

```
x := 1
DO WHILE ( x < 10 )
    ? x++
ENDDO
```