



VLADEMIRO LANDIM JUNIOR

**INTRODUÇÃO A PROGRAMAÇÃO USANDO A LINGUAGEM
HARBOUR.**

FORTALEZA, CEARÁ

2016

VLADEMIRO LANDIM JUNIOR

**INTRODUÇÃO A PROGRAMAÇÃO USANDO A LINGUAGEM
HARBOUR.**

Uma introdução aos aspectos básicos da programação de computadores usando a linguagem Harbour como ferramenta de aplicação dos conceitos aprendidos.

Área de concentração: Introdução a Programação, Linguagem Harbour, Computação, Informática, Algoritmos.

FORTALEZA, CEARÁ

2016

LANDIM, Vlademiro.

Introdução a programação usando a linguagem Harbour.
/ Vlademiro Landim Junior. 2016.

143p.;il. color. enc.

VLADEMIRO LANDIM JUNIOR

**INTRODUÇÃO A PROGRAMAÇÃO USANDO A LINGUAGEM
HARBOUR.**

IMPORTANTE: 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 vlad@altersoft.net 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.

Área de concentração: Introdução a Programação, Linguagem Harbour, Computação, Informática, Algoritmos.

Aos meus Pais.

Agradecimentos

Agradeço a Paulo César Toledo e a todos que participam do fórum Clipper On Line (<http://www.pctoledo.com.br/forum>). A solidariedade e a atenção de todos vocês foi essencial para a conclusão desse trabalho. **Com certeza não vou poder citar todos** pois muitos me ajudaram mesmo sem saber, e na pressa do momento eu acabei esquecendo o nome da pessoa ou até mesmo nem lendo o nome. Segue abaixo uma lista muito parcial (a ordem não reflete a importância, fui me lembrando e digitando):

- Paulo César Toledo
- José Quintas
- “Maligno”
- Fladimir
- Rochinha
- “Asimoes”
- Claudio Soto
- Vailton
- Pablo Cesar
- Janio
- Stanis Luksys
- Jairo Maia
- Itamar M. Lins Jr.
- “RobertoLinux”

- porter
- “alxsts”
- Eolo
- “paiva_dbdc”
- “rbonotto”
- “vagucs”
- “sygecon”
- “Imatech”

Caso alguém encontre algum erro nesse material envie um e-mail com as correções a serem feitas para vlad@altersoft.net 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.

“Suponham que um de vocês tenha um amigo e que recorra a ele à meia-noite e diga ‘Amigo, empreste-me três pães, porque um amigo meu chegou de viagem, e não tenho nada para lhe oferecer’. E o que estiver dentro responda: ‘Não me incomode. A porta já está fechada, e eu e meus filhos já estamos deitados. Não posso me levantar e lhe dar o que me pede’. Eu lhes digo: Embora ele não se levante para dar-lhe o pão por ser seu amigo, por causa da importunação se levantará e lhe dará tudo o que precisar. Por isso lhes digo: Peçam, e lhes será dado; busquem, e encontrarão; batam, e a porta lhes será aberta”

(Jesus Cristo, Filho do Deus vivo - Evangelho de Lucas 11:5-9)

Resumo

Esse livro busca ensinar os princípios de programação utilizando a linguagem Harbour. Ele aborda os conceitos básicos de qualquer linguagem de programação, variáveis, tipos de dados, estruturas sequenciais, estruturas de decisão, estruturas de controle de fluxo, tipos de dados complexos, funções, escopo e tempo de vida de variáveis. Como a linguagem utilizada é a linguagem Harbour, o presente estudo busca também apresentar algumas particularidades dessa linguagem como comparação de *strings* e macro substituição. Palavras-chave: Introdução a Programação, Linguagem de programação, Linguagem Harbour, Sistemas de Informação, xBase, Clipper.

Abstract

This book seeks to teach the principles of programming using the language Harbour . It covers the basics of any programming language , variables , data types , sequential structures , decision structures , flow control structures , complex data types , functions, scope and lifetime of variables. As the language is the language Harbour , this study also seeks to present some peculiarities of this language as a comparison `textit` `string` and macro replacement.

Keywords: . . .

Lista de Figuras

Figura 2.1	O processo de compilação	26
Figura 2.2	Processo de geração de um programa escrito em Harbour	28
Figura 2.3	Adicionando as variáveis de ambiente	29
Figura 2.4	Adicionando as variáveis de ambiente	29
Figura 2.5	Adicionando as variáveis de ambiente	29
Figura 2.6	Abrindo o prompt de comando do windows	30
Figura 2.7	Criando um atalho na área de trabalho	31
Figura 2.8	Abrindo o prompt de comando do windows pelo xDevStudio	32
Figura 3.1	Criando o arquivo hello.prg pelo xDevStudio	38
Figura 3.2	Salvando o arquivo	38
Figura 3.3	Salvando o arquivo (Parte II)	39

Lista de Tabelas

Tabela 5.1	Operadores matemáticos	91
Tabela 5.2	Operadores de atribuição compostos	99

Sumário

1	INTRODUÇÃO	16
1.1	A quem se destina esse livro	16
1.2	Pré-requisitos necessários	17
1.3	Metodologia utilizada	18
1.4	Material de Apoio	19
1.5	Plano da obra	19
1.6	Porque Harbour ?	20
2	O PROCESSO DE CRIAÇÃO DE UM PROGRAMA DE COMPUTADOR ..	23
2.1	O que é um programa de computador ?	23
2.2	Linguagens de programação	24
2.3	Tipos de linguagens de programação	26
2.3.1	O Harbour pertence a qual categoria ?	27
2.3.2	Instalando o Harbour no seu computador	28
2.3.2.1	Testando se o Harbour foi instalado corretamente	30
2.3.3	Instalando o xDevStudio	31
2.3.4	O processo de compilação	32
2.3.5	Conclusão	34
3	MEU PRIMEIRO PROGRAMA EM HARBOUR	35
3.1	De que é feito um programa de computador ?	36
3.2	O mínimo necessário	36

3.3	O primeiro programa	37
3.3.1	Classificação dos elementos da linguagem Harbour	44
3.3.1.1	As regras de sintaxe da linguagem	44
3.3.2	Uma pequena pausa para a acentuação correta	46
3.4	As strings	47
3.4.1	Quebra de linha	49
3.5	Números	50
3.6	Uma pequena introdução as Funções	51
3.7	Comentários	56
3.8	Praticando	58
3.9	Desafio	59
3.10	Exercícios	60
4	CONSTANTES E VARIÁVEIS	61
4.1	Constantes	62
4.1.1	Grupos de constantes	64
4.2	Variáveis	66
4.2.1	Criação de variáveis de memória	66
4.2.2	Escolhendo nomes apropriados para as suas variáveis	68
4.2.3	Seja claro ao nomear suas variáveis	70
4.2.4	Atribuição	72
4.3	Variáveis: declaração e inicialização	73
4.4	Recebendo dados do usuário	76
4.5	Exemplos	79
4.5.1	Realizando as quatro operações	79
4.5.2	Calculando o antecessor e o sucessor de um número	80
4.6	Exercícios de fixação	82
4.7	Desafios	84
4.7.1	Identifique o erro de compilação no programa abaixo.	84
4.7.2	Identifique o erro de lógica no programa abaixo.	85
4.7.3	Valor total das moedas	85

4.7.4	O comerciante maluco	86
5	TIPOS DE DADOS : VALORES E OPERADORES	87
5.1	Definições iniciais	88
5.2	Variáveis do tipo numérico	90
5.2.1	As quatro operações	91
5.2.2	O operador %	94
5.2.3	O operador de exponenciação (^)	94
5.2.4	Os operadores unários	95
5.2.5	A precedência dos operadores numéricos	95
5.2.6	Novas formas de atribuição	97
5.2.6.1	Operadores de incremento e decremento	99
5.3	Variáveis do tipo caractere	102
5.4	Variáveis do tipo data	105
5.4.1	Inicializando uma variável com a função CTOD()	105
5.4.2	Inicializar usando a função STOD()	108
5.4.3	Inicialização sem funções, usando o novo formato 0d	109
5.4.4	Os operadores do tipo data	112
5.4.5	Os operadores de atribuição com o tipo de dado data	115
5.5	Variáveis do tipo Lógico	117
5.6	De novo os operadores de atribuição	117
5.7	Operadores especiais	119
5.8	O estranho valor NIL	120
5.9	Um tipo especial de variável : SETs	120
5.10	Exercícios de fixação	122
	REFERÊNCIAS BIBLIOGRÁFICAS	126
A	EXERCÍCIOS : CONSTANTES E VARIÁVEIS	128
A.1	Resposta aos exercícios de fixação sobre variáveis - Capítulo 4	129
A.2	Respostas aos desafios - Capítulo 4	139
A.2.1	Identifique o erro de compilação no programa abaixo.	139
A.2.2	Identifique o erro de lógica no programa abaixo.	140

A.2.3	Valor total das moedas	140
A.2.4	O comerciante maluco.....	141
A.3	Resposta aos exercícios de fixação sobre variáveis - Capítulo 5	142

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.

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

1.3 Metodologia utilizada

Durante a escrita deste livro, procurei seguir a sequência 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 sequência, 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 se demorar demais a exibir as cenas que envolvem batidas de carro e perseguições. Por outro lado, 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 Kernighan). 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 sequência dos cursos introdutórios de programação. Se por acaso você não tem experiência alguma com programação e deseja iniciar o aprendizado, esse 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 desestimele 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, opera-

dores, 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 de 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. Procurei também incluir no material de apoio o compilador Harbour que eu usei para escrever os exemplos desse livro, a linguagem C que acompanha o compilador Harbour e o editor de códigos xDevStudio, desenvolvido por Vailton Renato.

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** aborda um assunto central em qualquer linguagem de programação: as estruturas de controle. Ele é um capítulo longo porque ele aborda juntamente com tais estruturas os conceitos de algoritmos e fluxogramas. Também lhe dá algumas técnicas de como transformar um problema do mundo real em um programa de computador. Ao finalizar esse capítulo você já poderá escrever pequenos programas realmente funcionais. O **capítulo 7** introduz o uso de funções. Ele divide-se em duas partes: a primeira trás uma pequena listagem das funções do Harbour e vários exemplos que devem ser praticados. A segunda parte desse capítulo lhe ensina a criar as suas próprias funções. O **capítulo 8** aborda as estruturas complexas da linguagem, como arrays e hashes. Se você acompanhou tudo direitinho até o capítulo 7 não haverá problemas aqui, pois a maioria dos tópicos que ele aborda já foram vistos. O **capítulo 9** aborda as classes de variáveis e o **capítulo 10** conclui o livro com alguns aspectos avançados da linguagem. 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. Caso você não queira um executável autônomo, e dispensar o compilador C, o Harbour pode ser interpretado² através do seu utilitário *hbrun*.
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.

²Nota técnica: o Harbour não pode ser considerada uma linguagem interpretada, pois um código intermediário é gerado para que ela possa ser executada. O código fonte não fica disponível como em outras linguagens.

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 gráfica, 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 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, se comparada com linguagens mais populares como o Java e o C#, 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. Sejam 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.

³Hacker 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 estereótipo 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.

O processo de criação de um programa de computador

Qualquer coisa é fácil, se você puder incluí-las em sua coleção de modelos.

Seymour Papert

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 sequência 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 sequência de instruções baseado no resultado.
5. **repetir** : executa uma ação repetidamente.

Isso é praticamente tudo o que você precisa saber por enquanto. Nos próximos capítulos nós abordaremos cada um desses 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 sequência, 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 gerados diretamente por seres humanos e introduzidos diretamente na memória da máquina, através da alteração na posição de chaves ou “jumpers”. Era uma tarefa tediosa e sujeita a erros.

Com o tempo, os cientistas desenvolveram um programa de computador cuja função era simplificar a criação desses códigos. Esse programa de computador, recebeu o nome de

“montador” (*assembler*). Os montadores representaram um importante avanço sobre a programação em código de máquina puro, e a linguagem desse programa recebeu o nome de “Assembler”. A sequência abaixo representa um código em Assembler.

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

A linguagem Assembler é classificada como uma linguagem de baixo nível. Essa classificação significa “o quão próximo da linguagem de máquina pura” uma linguagem de programação está. No caso do Assembler, a relação entre uma instrução em Assembler e uma instrução em código de máquina é praticamente de um para um. Os montadores 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, o programa gerado em Assembler não irá funcionar.

Com o passar dos anos as linguagens de alto nível surgiram, tornando o processo de programar cada vez mais simples e acessível. O nome do programa usado para gerar o código de máquina através de uma linguagem não era mais montador, pois a relação entre o código gerado e o código de máquina não era mais de um para um, mas de um para “n”. O nome do programa usado passou a ser “compilador”, pois uma linha de código representa a compilação de várias linhas em linguagem de baixo nível. A linguagem Harbour é uma linguagem de alto nível, por exemplo a seguinte instrução :

```
a := 10
```

gera um código que equivale a várias linhas em uma linguagem de baixo nível. 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 a Linguagem C é a única largamente utilizada¹.

O programador, sem saber, acaba desenvolvendo sempre em linguagem de máquina (que é a única linguagem que um computador entende), mas quando utiliza uma linguagem de alto nível, muitas operações são simplificadas e até ocultadas. É por isso que o termo “programa compilado em Assembler” não é tecnicamente correto, pois o nível com que um montador trabalha é diferente do nível com que um compilador trabalha.

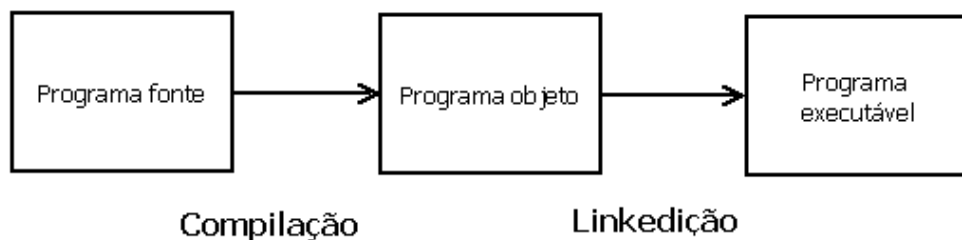
¹Essa classificação de médio nível para a Linguagem C foi dada pelos criadores da linguagem. A Linguagem C, portanto, abrange duas categorias : alto nível e médio nível.

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 interpretada². 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 2.1.

Figura 2.1: 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

²Existem 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

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 final³.

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 ⁴ entre plataformas. É mais fácil escrever um programa PHP que funcione em servidores Windows e em Linux do que escrever um programa em C que funcione nessas duas plataformas citadas. Quando o programa é pequeno a dificuldade praticamente inexistente, 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 ?

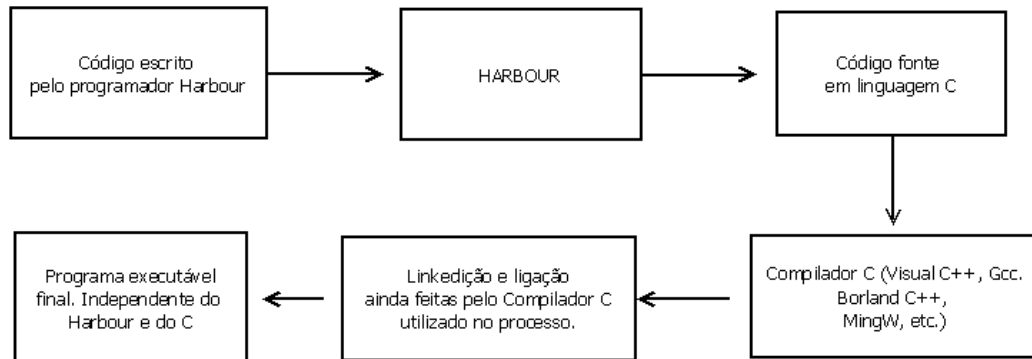
O lema da linguagem Harbour é : “escreva uma vez, compile em qualquer lugar”, então o Harbour é um compilador certo ? Mais ou menos. Na verdade o Harbour não é classificado tecnicamente como um compilador puro, pois ele depende de um compilador externo para poder gerar o seu programa executável independente. 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.

Todo esse processo de geração é simplificado para o programador, mas pode ser visualizado através da figura 2.2.

³Existem 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

⁴Um programa portátil pode funcionar sem problemas em diversos sistemas operacionais, como Windows, Linux, MAC, Android, etc.

Figura 2.2: Processo de geração de um programa escrito em Harbour



Se você não tiver a linguagem C na sua máquina você ainda assim vai conseguir gerar um programa em Harbour que funcione, mas ele vai depender de um programa externo chamado *hbrun* para poder funcionar. É mais ou menos o mesmo processo que acontece com as linguagens interpretadas, a diferença é que o código que irá ser lido pelo *hbrun* não é o código fonte original do programador, mas um código objeto gerado pelo Harbour, dessa forma o código fonte de uma aplicação não precisa ser distribuída ⁵.

2.3.2 Instalando o Harbour no seu computador

O material que acompanha o livro pode ser baixado de www.altersoft.net. Ele vem em formato zip e o seu nome é “Curso_Harbour.zip”. Descompacte esse conteúdo em uma pasta qualquer do seu sistema windows ⁶. Como vamos seguir um roteiro de instalação, vamos descompactar essa pasta na raiz do seu sistema de arquivos windows, assim :

C:\Curso_Harbour.

Quando você descompactar o arquivo verifique se as seguintes pastas existem :

1. C:\Curso_Harbour\harbour : Pasta onde está o compilador harbour;
2. C:\Curso_Harbour\xDevStudio : Pasta onde está o editor xDevStudio;
3. C:\Curso_Harbour\pratica : Pasta onde você irá trabalhar digitando os códigos;

Para que o Harbour funcione corretamente tudo o que você tem a 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, que é C:\Curso_Harbour , então os PATHs são os seguintes :

⁵Semelhante a forma com a qual um programa escrito em Java é gerado

⁶Se você usa o Linux, no apêndice XXX tem um roteiro de instalação no Linux

- C:\Curso_Harbour\harbour\bin
- C:\Curso_Harbour\harbour\comp\mingw\bin

Para adicionar essas variáveis de ambiente faça conforme as figuras 2.3, 2.4 e 2.5

Figura 2.3: Adicionando as variáveis de ambiente

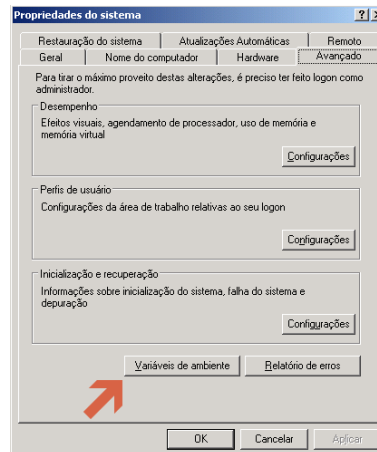


Figura 2.4: Adicionando as variáveis de ambiente

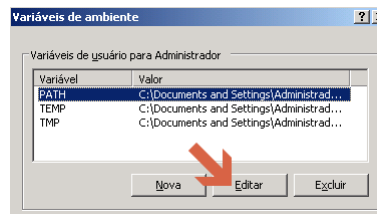
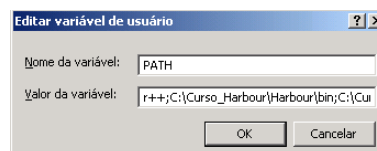


Figura 2.5: Adicionando as variáveis de ambiente



Quando for adicionar (figura 2.5) você deve ter o cuidado de não apagar as variáveis que estão lá. Vá até o final da linha, **digite um ponto e vírgula no final da linha** e digite o conteúdo baixo :

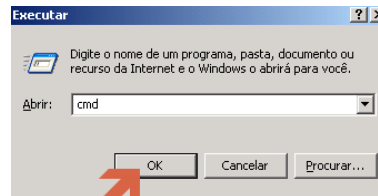
C:\Curso_Harbour\Harbour\bin;C:\Curso_Harbour\Harbour\comp\mingw\bin

Clique em Ok para salvar. Pronto, as variáveis estão configuradas.

2.3.2.1 Testando se o Harbour foi instalado corretamente

Para testar faça o seguinte: abra o prompt do sistema operacional, conforme a figura 2.6.

Figura 2.6: Abrindo o prompt de comando do windows



No prompt 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 (r1507030922)
Copyright (c) 1999-2015, http://harbour-project.org/

Syntax:  harbour <file[s][.prg]|@file> [options]

Options:  -a                automatic memvar declaration
          -b                debug info
          -build            display detailed version info
          -credits          display credits
          -d<id>[=<val>]    #define <id>
          -es[<level>]     set exit severity
          -fn[:[l|u]|-]     set filename casing (l=lower u=upper)
          -fd[:[l|u]|-]     set directory casing (l=lower u=upper)
          -fp[:<char>]      set path separator
          -fs[-]            turn filename space trimming on or off
                           (default)
```

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

.:Resultado:.

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

Pronto, o compilador gcc também está instalado.

2.3.3 Instalando o xDevStudio

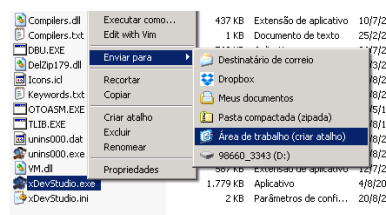
O xDevStudio é um editor desenvolvido por Vailton Renato (WebNet do Brasil) especialmente para os desenvolvedores Clipper, Harbour e outros da família xBase. Nós insistimos para que você use esse editor para digitar os códigos desse livro. Por vários motivos :

1. Esse foi o mesmo editor utilizado para a digitação dos códigos de exemplo.
2. Usando esse editor você não terá problemas com a exibição de acentos e cedilhas.
3. Esse editor foi especialmente configurado para suportar todos os comandos do Harbour. Por exemplo, quando você digitar alguma instrução de programação conhecida, essa instrução irá ser destacada com uma cor, e as vezes com um formato diferente.
4. O editor é totalmente em português, desenvolvido por um programador brasileiro e que não deve em nada aos similares estrangeiros.
5. O editor possui ótimos recursos de documentação da linguagem Clipper / Harbour. Acompanha os manuais do Clipper (que são todos válidos para o Harbour também), além de outras facilidades. Nessa parte alguns documentos podem estar em Inglês.

Portanto, o xDevStudio é o editor recomendado para que você use para digitar os códigos desse livro. Nós não chegaremos a utilizar todos os recursos do xDevStudio, por exemplo, a compilação através do editor não será usada, pois nós queremos que você aprenda a compilar um programa pelo prompt de comando do windows. Depois, quando você entender o que está ocorrendo você pode partir, em uma etapa posterior a leitura desse livro, para estudar o xDevStudio a fundo. É bom ressaltar que existem também outras opções free para o desenvolvimento de programas em Harbour, mas por enquanto, você deve usar o xDevStudio.

Para facilitar o acesso ao editor faça conforme a figura 2.7

Figura 2.7: Criando um atalho na área de trabalho



Clique com o botão direito do mouse sobre o ícone do xDevStudio e crie um atalho na área de trabalho. Vamos agora seguir adiante no nosso aprendizado.

Dê agora um duplo click sobre o ícone do xDevStudio. Aguarde o carregamento do editor. Após o carregamento do editor, acesse o prompt do windows através do xDevStudio.

Clique no menu “Ferramentas” e selecione a opção “Prompt de Comando” conforme a figura 2.8. Aguarde um pouco e a janela do Prompt de comando irá aparecer.

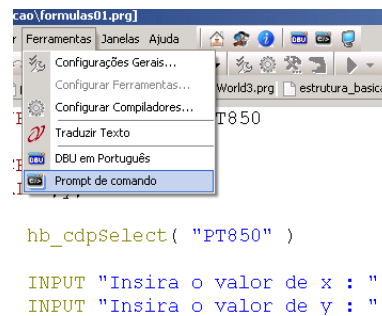
Observe que o prompt de comando abre na mesma pasta onde está o xDevStudio. Tudo o que você tem a fazer é navegar, através de comandos do prompt até a pasta de exemplos para poder compilar alguns exemplos de teste. Faça assim, dentro do prompt de comando (após cada linha tecle enter) :

.:Resultado:.

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

Pronto, você agora está na pasta pratica. Nessa pasta nós iremos testar o compilador. Utilize também essa pasta para digitar os seus códigos.

Figura 2.8: Abrindo o prompt de comando do windows pelo xDevStudio



2.3.4 O processo de compilação

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

1. O compilador Harbour instalado
2. O compilador C (gcc) instalado
3. O xDevStudio com um atalho na área de trabalho
4. O prompt de comando aberto e na pasta pratica

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

Para compilar um programa em Harbour existe um utilitário chamado *hbm2* que faz todo o processo descrito no capítulo anterior. Esse programa não faz a compilação, mas gerencia o Harbour e todos os programas envolvidos durante o processo. Digite no prompt de comando *hbm2* e tecle enter. Algo semelhante com a figura abaixo deve surgir (não mostramos tudo na figura abaixo, apenas as linhas iniciais).

.:Resultado:.

```
Harbour Make (hbm2) 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 “pratica” existe um arquivo chamado exemplo.prg, agora iremos compilar esse arquivo.

Digite hbm2 exemplo e tecle enter ⁷

.:Resultado:.

```
hbm2: Processando script local: hbm.hbm
hbm2: Harbour: Compilando módulos...
Harbour 3.2.0dev (r1507030922)
Copyright (c) 1999-2015, http://harbour-project.org/
Compiling 'exemplo.prg'...
Lines 15, Functions/Procedures 1
Generating C source output to '.hbm\win\mingw\exemplo.c'... Done.
hbm2: Compilando...
hbm2: Linkando... exemplo.exe
```

O processo de compilação é simples e rápido, mas durante esse processo, muitas operações aconteceram. Vamos analisar a saída do comando hbm2.

- **hbm2: Processando script local: hbm.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 *hbm.hbm*. Nós já providenciamos esse arquivo com algumas das opções mais usadas, por isso o compilador exibiu essa mensagem.
- **Compiling 'exemplo.prg'...** : A fase de compilação do Harbour.
- **Lines 15, Functions/Procedures 1** : Ele analisou o arquivo e encontrou 15 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.
- **hbm2: Compilando...** : A fase onde o compilador C entra em ação.

⁷Com o passar dos exemplos nós omitiremos detalhes básicos, como teclar enter após a digitação no prompt de comando.

- **hbm2: Linkando... exemplo.exe** : A fase final, de linkagem e consequente geração do executável.

2.3.5 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 geração do seu primeiro programa em Harbour.

3

Meu primeiro programa em Harbour

Geralmente erra mais quem decide cedo do que quem decide tarde; mas, depois de tomada a decisão, é necessário recuperar o atraso da sua execução.

Francesco Guicciardini

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, vestir a 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. 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 pequena rotina acima, retirado de (FORBELLONE; EBERSPACHER, 2005, p. 4), nos mostra a ideia simples por detrás 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 todas os comandos de um programa de computador. Uma analogia interessante é comparar o desenvolvimento de um programa a leitura de um romance: a grande maioria dos romances está organizada em capítulos. Quando nós vamos iniciar a leitura começamos pelo capítulo 1 e seguimos em sequência 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.

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 terá que 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.

Listing 3.1: O primeiro programa

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

O programa escrito em 3.1 é completamente funcional do ponto de vista sintático, porém ele não executa nada. Com isso queremos dizer que o mínimo necessário para se ter um programa sintaticamente correto é a procedure Main³.

3.3 O primeiro programa

Vamos agora criar o nosso primeiro programa em Harbour. O exemplo a seguir (listagem 3.2) 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.

Listing 3.2: Hello World

```
1  PROCEDURE Main
2
3      ? "Hello World"
4
5  RETURN
```

Digite o programa usando o xDevStudio e compile o programa usando o *hbmk2*.

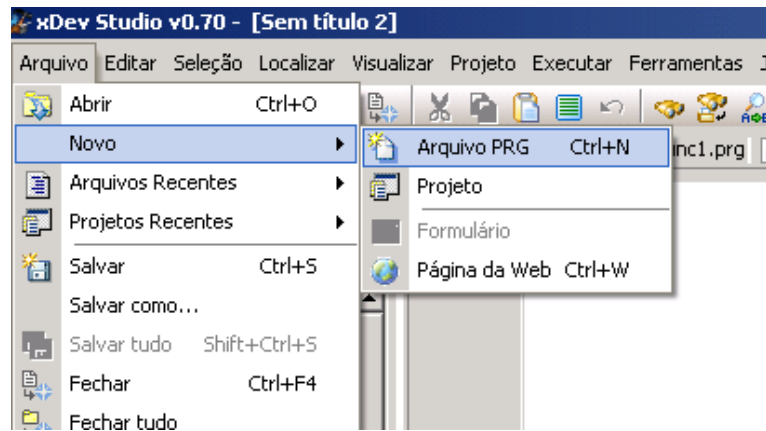
Abra o xDevStudio, clique no menu Arquivo, e selecione a opção Novo e em seguida Arquivo PRG. Todo programa escrito em Harbour possui a extensão PRG. Inicie a digitação do programa da listagem 3.2, conforme a figura 3.1.

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

²Note que nos exemplos nós usamos a palavra reservada PROCEDURE para iniciar a nossa rotina.

³De agora em diante nós chamaremos os procedimentos de “procedures”.

Figura 3.1: Criando o arquivo hello.prg pelo xDevStudio



Quando concluir a digitação salve o arquivo. Como é a primeira vez que você salva esse arquivo você deve informar o nome e o local onde ele ficará. **Salve-o na pasta pratica**, conforme a figura 3.2 e 3.3. Quando for salvar dê-lhe o nome “hello.prg”. 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 3.2: Salvando o arquivo

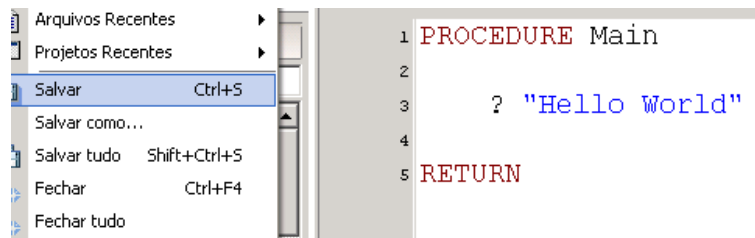
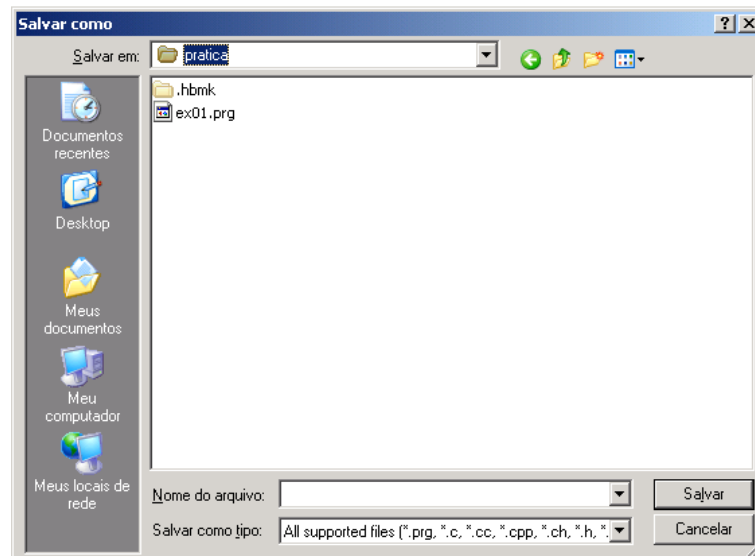


Figura 3.3: Salvando o arquivo (Parte II)



Agora abra o Prompt de comando de dentro do xDevStudio e vá para a pasta pratica. Lembre-se, no xDevStudio clique no menu Ferramentas e em seguida na opção “Prompt de Comando”. Depois digite os comandos abaixo para ir para a pasta “pratica”.

..:Resultado:..

```
C:\Curso_Harbour\xDevStudio> cd ..
C:\Curso_Harbour> cd pratica
C:\Curso_Harbour\pratica> hbm2 hello
```

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

..:Resultado:..

```
C:\Curso_Harbour\pratica> hbm2 hello
```

Após o processo de compilação, que não deve demorar, 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 símbolo **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 3.2 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; 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**, isto é, a linguagem **não faz** diferenciação entre letras maiúsculas e minúsculas, 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. É uma exceção da observação anterior.
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 lê⁴

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 enquanto 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.

Dica 2

Note que o código da listagem 3.2 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

⁴Essa 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. Hoje em dia não precisamos mais dessa economia porque os dispositivos de armazenamento de hoje possuem uma capacidade muito superior aos antigos disquetes.

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.

^aEssa 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_\(typesetting\)](https://en.wikipedia.org/wiki/Indentation_(typesetting)) (Acessado em 20-Ago-2016)

^bNa 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 3.2.

Listing 3.3: O comando ??

```
1 PROCEDURE Main()
2
3     ?? "Hello "
4     ?? "World"
5
6 RETU
```

.: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.

Listing 3.4: O comando ? e ??

```

1  /*
2  Comandos ? e ??
3  */
4  PROCEDURE Main
5
6      ? "O aniversário de Claudia será hoje às", "dez horas."
7      ? "O valor da compra foi de R$" , "4 reais"
8
9  RETURN

```

.:Resultado:.

```

O aniversário de Claudia será hoje às dez horas.
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* :

```

1  PROCEDURE Main
2
3
4  RETURN

```

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

^aA 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 :

Listing 3.5: `pratica_hello.prg`

```

1
2  PROCEDURE Main

```

```

3
4 ? "===== "
5 ? "= PEDIDO DE VENDA N. 12 = "
6 ? "===== "
7 ? " PRODUTO          QTD      VALOR          TOTAL  "
8 ? " =====          =====          =====  "
9 ? " Blusa GP Gde      12      150,00        1800,00  "
10 ? " Camisa Regat      2       25,00         50,00  "
11 ? "                                     -----  "
12 ? "                                TOTAL        1850,00  "
13 ? "===== "
14
15 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, pois o nosso intelecto consegue contextualizar e deduzir a mensagem. Já um computador não possui esse poder, portanto as suas instruções precisam ser escritas corretamente e algumas regras devem ser seguidas.

3.3.1.1 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 <> : São elementos fornecidos pelo usuário. No caso, o usuário pode fornecer uma lista de expressões.

A notação acima contempla todos os casos, por exemplo :

```

2
3      ?
4      ? "Claudiana"
5      ? "Roberto" , 12
6
7 RETURN

```

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

Se você digitou, compilou e executou o programa da listagem 3.4 e não utilizou o editor que recomendamos no capítulo anterior (o xDevStudio), provavelmente você teve problemas com a exibição de palavras acentuadas. Você deve ter notado que a acentuação não apareceu corretamente e no lugar da palavra 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, mas nós não queremos abordar agora assuntos que só serão plenamente compreendidos mais adiante. Então, sem prejuízo para o aprendizado, vamos usar o editor indicado. 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 (os idiomas português, francês, alemão, etc. possuem seus próprios símbolos e seus próprios padrões). 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 (por exemplo, o formato UTF-8). Nos exemplos desse livro nós estamos usando a codificação padrão do Harbour, por isso fique atento pois existe apenas um caractere que a codificação que nós usamos não consegue imprimir: o til ^a.

Resolver esse problema é fácil, mas como dissemos, não queremos complicar muito com símbolos sem a devida explicação. Mas, se mesmo assim, você quer resolver esse problema, leia o apêndice ??, mas nós insistimos para que você não faça isso caso você não tenha os mínimos conhecimentos de programação.

Por enquanto, vamos nos concentrar em aprender a linguagem e só depois cuidar facilmente desse detalhe. Não vamos abordar esse assunto agora nem antecipar outros problemas. Não se esqueça de usar o xDevStudio para digitar seus códigos também.

^aOs antigos programadores resolviam esse problema colocando um trema na vogal que iria receber o til. Admitimos que essa não é uma solução elegante, mas era a única que estava facilmente ao alcance do programador da década de 1980.

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 ⁵, 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 3.6 está errado, porque eu não posso usar uma aspa do mesmo tipo para ser impressa, isso confunde o compilador.

Listing 3.6: Erro com aspas

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

Já o código da listagem 3.7 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 :

Listing 3.7: Uso correto de delimitadores de *string*

```
1  /*
2  Delimitando strings
3  */
4  PROCEDURE Main
5
6      ? "A praia estava 'legal'."
7      ? 'A praia estava "legal".'
8      ? [Colchetes podem ser usados para delimitar strings!]
9      ? [Você deram um "nó em pingo d'água"]
10     ? "Mas mesmo assim evite o uso de colchetes."
11
12  RETURN
```

⁵O Harbour também utiliza colchetes para delimitar *strings*, mas esse recurso é bem menos usado. Prefira delimitar *string* com aspas (simples ou duplas).

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 :

```
1 PROCEDURE Main
2
3     ? "Olá pessoal." "Hoje vamos sair mais cedo."
4
5 RETURN
```

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

```
1 PROCEDURE Main
2
3     ? "Olá pessoal." , "Hoje vamos sair mais cedo."
4
5 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.

```
1 PROCEDURE Main
2
3     ? "Olá pessoal." + "Hoje vamos sair mais cedo."
4
5 RETURN
```

.: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.

Listing 3.8: A linha não acabou

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

.: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 “+”⁶ 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.

Listing 3.9: A linha não acabou (ERRO)

```
1 PROCEDURE Main
```

⁶O termo técnico para esse sinal é “operador” (veremos o que é isso mais adiante)

```

2
3      ? "Essa linha está dividida de " ; // <= Faltou o operador
4      "forma errada."
5
6      ? "Essa linha está dividida de + ; // <= Faltou a aspa
7      forma errada."
8
9
10 RETURN

```

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

```

1
2      ? 12; ? "Olá, pessoal"

```

equivale a

```

1
2      ? 12
3      ? "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 :

```

1 PROCEDURE Main
2
3      ? "2" + "2" // Ir   exibir 22 e n  o 4.
4
5 RETURN

```

..Resultado:..

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.

```
1 PROCEDURE Main
2
3     ? 2 + 2
4
5 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 :

```
1 PROCEDURE Main
2
3     ? 2 + "2"
4
5 RETURN
```

.:Resultado:.

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

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, nos capítulos posteriores ele será visto detalhadamente. Por enquanto vamos ficar com o seguinte conceito pouco formal : **uma função é um símbolo que possui 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.

```

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

```

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

Digite 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 :

```

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

```

.: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 :

```
1 PROCEDURE Main
2
3     ? "TIME()" // Mostra o nome TIME()
4     ? "A hora atual é " TIME() // Gera um erro
5
6 RETURN
```

Para exibir a mensagem você deve fazer assim :

```
1 PROCEDURE Main
2
3     ? "Agora são ", TIME() // Use a vírgula do comando ‘?’
4
5 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 3.10 ilustra o uso da função SECONDS(), a “função” dela é “retornar” quantos segundos transcorreram desde a meia-noite.

Listing 3.10: Funções simples

```

1  /*
2  Função
3  */
4  PROCEDURE Main
5
6      ? "Desde a meia-note até agora transcorreram ", SECONDS(), "
          segundos."
7
8  RETURN

```

.: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 espécie de “oráculo”: você faz a pergunta

(chamando a função) e ela retorna a resposta na forma de um valor ⁷.

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

```
1  PROCEDURE Main
2
3      ? SQRT( 64 )
4
5  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)

⁷Você 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.

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.

Listing 3.11: Uso correto de comentários

```

1  PROCEDURE Main
2
3      ? "Olá, pessoal" // Útil para explicar a linha corrente.
4      // ? "Essa linha TODA será ignorada."
5      * ? "Essa linha também será ignorada."
6      && ? "E essa também..."
7
8      ? "O valor : " , 1200 // Útil para comentar após
9                          // o final da linha.
10
11  /*
12
13      Note que esse comentário

```

```

14      possui várias linhas.
15
16  */
17
18  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 */.

Listing 3.12: Uso INCORRETO de comentários

```

1  PROCEDURE Main
2
3      /*
4          ERRO !
5              Os comentários não podem ser aninhados, ou
6              seja, não posso colocar um dentro do outro.
7
8              /* Aqui está o erro */
9
10         */
11
12     ? "Olá pessoal, esse programa não irá compilar."
13
14  RETURN

```

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

```

1
2      ? 12 /* preço */ + 3 /* imposto */ + 4 // Imprime o custo

```

.:Resultado:.

19

Dica 8

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; 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.

Listing 3.13: Erro 1

```

1  /*
2  * Aprendendo Harbour
3  */
4  PROCEDURE Main
5
6      ? "Hello" "World"
7
8
9  RETURN

```

Listing 3.14: Erro 2

```

1  /*
2  * Aprendendo Harbour
3  /*
4  PROCEDURE Main
5
6      ? "Hello World"
7
8
9  RETURN

```

Listing 3.15: Erro 3

```

1  /*
2  /* Aprendendo Harbour
3  */
4  PROCEDURE Main
5
6      ? "Hello World"
7
8
9  RETURN

```

Listing 3.16: Erro 4

```

1  /*
2  Aprendendo Harbour
3  */
4  PROCEDURE Main
5
6      ? Hello World
7
8
9  RETURN

```

Listing 3.17: Erro 5

```

1  /*
2  Aprendendo Harbour
3  */
4  PROCEDURE Main
5
6      ? "Hello ";
7      ?? "World"
8
9
10
11 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

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

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 :

```

1  PROCEDURE Main
2
3      ? "Hello World" // ‘‘Hello World’’ é uma constante caractere
4      ? 2 // 2 é uma constante numérica
5
6  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 costuma 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; 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 simbólica” ou “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 simbólica chamada VOLUME que vale 1000.

```

1  #define VOLUME 10000
2  PROCEDURE Main
3

```

```

4      ? VOLUME // Vai imprimir 10000
5
6 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 9

A expressão “**tempo de compilação**” refere-se ao momento em a instrução foi definida pelo computador. Nesse caso específico, as constantes simbólicas 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 simbólica. Vamos tomar como ilustração a listagem abaixo :

```

1 #define VOLUME 12
2 PROCEDURE Main
3
4     ? VOLUME
5
6 RETURN

```

Quando você executa o hbm2 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 :

```

1
2 PROCEDURE Main
3
4     ? 12
5
6 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 10

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. 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 simbólica **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 11

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.

^aAlém de números e strings posso criar constantes simbólicas 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 simbólicas.

Dica 12

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 Grupos de constantes

As vezes é importante você ter todas as suas constantes agrupadas em um único arquivo. Isso evita ter que ficar redigindo 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 :

```

1 #define BRANCO "FFFFFF"
2 PROCEDURE Main
3
4     ? BRANCO // Imprime FFFFFFF
5
6 RETURN

```

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

```

1 #define BRANCO "FFFFFF"
2 #define VERMELHO "FF0000"
3 #define AZUL "0000FF"
4 #define AMARELO "FFFF00"
5 PROCEDURE Main
6
7     ? BRANCO
8     ? VERMELHO
9     ? AZUL
10    ? AMARELO
11
12 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”.

```

1 #define BRANCO "FFFFFF"
2 #define VERMELHO "FF0000"
3 #define AZUL "0000FF"
4 #define AMARELO "FFFF00"

```

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

```

1 #include "cores.ch"
2 PROCEDURE Main
3
4     ? BRANCO
5     ? VERMELHO
6     ? AZUL
7     ? AMARELO
8
9 RETURN

```

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

Dica 13

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 arquivos “`include`” próprios que você pode utilizar. 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 precisamos 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.

Listing 4.1: Criação de variáveis de memória

```

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

```

.: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:.

```

> hbm2 var0
hbm2: Processando script local: hbm.hbm
hbm2: 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 '.hbm\win\mingw\var0.c'... Done.
hbm2: Compilando...
hbm2: 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 item 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 :

Listing 4.2: Nomes válidos para variáveis

```
1
2  /*
3     Nomes válidos
4  */
5  PROCEDURE Main
6
7     nOpc := 100
8     _iK := 200  // Válido, porém não aconselhado (Começa com
9                 underline).
10    nTotal32 := nOpc + _iK
11    ? "nOpc = " , nOpc
12    ? "_iK = " , _iK
13    ? "nTotal32 = " , nTotal32
14  RETURN
```

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

.:Resultado:.

```
nOpC =          100
_iK =          200
nTotal32 =        300
```

Dica 14

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

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 15

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.

Listing 4.3: Uso de palavras reservadas

```
1
2 /*
3     Evite usar palavras reservadas
4 */
5 PROCEDURE Main
6
7     PRIVATE := 100 // PRIVATE é uma palavra reservada
8     REPLACE := 12  // REPLACE também é
9     TOTAL := PRIVATE + REPLACE // Total também é
10    ? TOTAL
11
12 RETURN
```

.:Resultado:.

112

Dica 16

(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.

Listing 4.4: Nomes inválidos para variáveis

```

1  /*
2    Nomes inválidos
3  */
4  PROCEDURE Main
5
6    90pc := 100 // Inicializa com um número
7
8  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 minúsculas².

Listing 4.5: Notações

```

1  /*
2    Notações utilizadas
3  */

```

²Essa notação é conhecida como Notação Hungara.

```

4  PROCEDURE Main
5
6      Tot_Nota := 1200 // Variável numérica (Notação com underscores)
7                      // que representa o total da nota fiscal
8
9      NomCli  := "Rob Pike" // Variável caracter (Notação hungara)
10                      // que representa o nome do cliente
11
12
13  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.

Dica 17

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; PIKE, 2000, p. 3). Procure usar nomes descritivos, além disso, procure manter esses nomes curtos, conforme a listagem 4.6.

Listing 4.6: Nomes curtos e concisos para variáveis

```

1  /*
2      NOMES CONSIGOS
3
4      Adaptado de (KERNIGHAN, p.3, 2000)
5  */
6  PROCEDURE Main
7
8      /*
9      Use comentários "//" para acrescentar informações sobre
10     a variável, em vez de deixar o nome da variável grande
11     demais.
12     */
12     nElem := 1 // Elemento corrente
13     nTotElem := 1200 // Total de elementos

```

³Essa notação é chamada por Rick Spence de “Notação Hungara Modificada”


```

14
15 RETURN

```

Evite detalhar demais conforme a listagem 4.7.

Listing 4.7: Nomes longos e detalhados demais

```

1  /*
2    NOMES GRANDES DEMAIS PARA VARI?VEIS. EVITE ISSO!!!
3
4    Adaptado de (KERNIGHAN, p.3, 2000)
5  */
6  PROCEDURE Main
7
8      nNumeroDoElementoCorrente := 1
9      nNumeroTotalDeElementos := 1200
10
11 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 `:=`.

Listing 4.8: Outras forma de atribuição

```

1
2  /*
3    Atribuição
4  */
5  PROCEDURE Main
6
7      x = 1200  // Atribui 1200 ao valor x
8      STORE 100 TO y, k, z // Atribui 100 ao valor y, k e z
9
10     ? x + y + k + z
11
12 RETURN

```

..Resultado:..

1500

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

Listing 4.9: Atribuição simultânea

```

1  /*
2  Atribuição
3  */
4  PROCEDURE Main
5
6      x := y := z := k := 100
7      ? x + y + z + k
8
9  RETURN

```

.:Resultado:.

400

Dica 18

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 19

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.
```

^aVocê só poderá entender completamente esse conceito no final do livro

Dica 20

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) :

```
1  PROCEDURE Main
2  LOCAL nValor := 12
3
4      ? nValor
5      // O restante das instruções ...
6
7  RETURN
```

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

```

1  PROCEDURE Main
2  LOCAL nValor
3
4      nValor := 12
5      ? nValor
6      // O restante das instruções ...
7
8  RETURN

```

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

```

1  PROCEDURE Main
2
3      nValor := 12
4      ? nValor
5      // O restante das instruções ...
6
7  RETURN

```

DICA IMPORTANTE!

Você pode designar o próprio Harbour para monitorar essa prática em tempo de compilação. Isso é muito bom pois evita que você possa declarar 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 hbmh.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 hbmh.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 hbmh.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.

Listing 4.10: Recebendo dados externos digitados pelo usuário

```

1
2  /*
3  Uso do comando INPUT
4  */
5  PROCEDURE Main
6  LOCAL nVal1 := 0 // Declara a variável parcela de pagamento
7  LOCAL nVal2 := 0 // Declara a variável parcela de pagamento
8
9      INPUT "Informe o primeiro valor : " TO nVal1
10     INPUT "Informa o segundo valor : " TO nVal2
11
12     ? "A soma dos dois valores é : ", nVal1 + nVal2
13
14
15 RETURN

```

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 21

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

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

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 4.11). Observe que o *ACCEPT* funciona da mesma forma que o *INPUT*.

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

```
1
2 /*
3  Uso do comando ACCEPT
4  */
5  PROCEDURE Main
6  LOCAL cNome // Seu nome
7
8      /* Pede e exibe o nome do usuário */
9      ACCEPT "Informe o seu nome : " TO cNome
10     ? "O seu nome é : ", cNome
11
12 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 22

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.

^aVeremos 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 23

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 4.14 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.

Listing 4.12: As quatro operações

```
1  /*
2  As quatro operações
3  Entrada : dois números
4  Saída : As quatro operações realizadas com esses dois números
5  */
6  PROCEDURE Main
7  LOCAL nValor1, nValor2 // Valores a serem calculados
8
9      // Recebendo os dados
10     ? "Introduza dois números para que eu realize as quatro oper.: "
```

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


```

11     INPUT "Introduza o primeiro valor : " TO nValor1
12     INPUT "Introduza o segundo valor : " TO nValor2
13
14     // Calculando e exibindo
15     ? "Soma..... : " , nValor1 + nValor2
16     ? "Subtração..... : " , nValor1 - nValor2
17     ? "Multiplicação.... : " , nValor1 * nValor2
18     ? "Divisão..... : " , nValor1 / nValor2
19
20     RETURN

```

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.

Listing 4.13: Descobrindo o antecessor e o sucessor

```

1
2  /*
3  Descobre o antecessor e o sucessor
4  */
5  PROCEDURE Main
6  LOCAL nValor // Número a ser inserido
7
8      // Recebendo os dados
9      ? ""
10     ? "**** Descobrindo o antecessor e o sucessor ****"
11     ? ""
12     INPUT "Introduza o número : " TO nValor
13

```

```

14      // Calculando e exibindo
15      ? "Antecessor..... : " , nValor - 1
16      ? "Sucessor..... : " , nValor + 1
17
18 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.

Listing 4.14: Calculando quantos números existem entre dois números

```

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

```

```

18     ? "Entre os números " , nIni, " e " , nFim, " existem ",
        nQtd, " números"
19
20 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

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 triângulo. O usuário deve informar a base e a altura e o programa deve retornar a área.
Dica : $nArea = \frac{nBase * nAltura}{2}$
12. 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 $PI = 3.1415$) .
13. 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 $\sqrt[2]{100} = 100^{\frac{1}{2}}$

14. 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.

15. 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

16. Escreva um programa que receba dois números (nBase e nExpoente) e mostre o valor da potência do primeiro elevado ao segundo.
17. 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 “*”.

18. 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
```

19. 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.

Listing 4.15: Erro 1

```
1 /*
2 Onde está o erro ?
```

```

3  */
4  PROCEDURE Main
5  LOCAL x, y, x // Número a ser inserido
6
7      x := 5
8      y := 10
9      x := 20
10
11 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*.

Listing 4.16: Erro de lógica

```

1  /*
2  Onde está o erro ?
3  */
4  PROCEDURE Main
5  LOCAL x, y // Número a ser inserido
6
7      x := 5
8      y := 10
9      ACCEPT "Informe o primeiro número : " TO x
10     ACCEPT "Informe o segundo número : " TO y
11     ? "A soma é ", x + y
12
13 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; 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

Somente a moralidade das nossas ações
pode nos dar a beleza e a dignidade de
viver.

Albert Einstein

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 sequência 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; AUGENSTEIN, 1995, p. 18)

.

Portanto, o conceito de tipo de dado está 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 que Luis Damas se referiu 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 // 0.66666666666666666667
```

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.

Listing 5.1: Operadores e tipo de dado

```

1  /*
2  Tipo de dado e operador
3  */
4  PROCEDURE Main
5  LOCAL x, y
6
7      x := "Feliz "
8      y := "Natal"
9
10     ? x + y // Exibirá "Feliz Natal"
11     ? x / y // Erro de execução (Operador / não pode ser usado em
           strings)
12
13 RETURN

```

.: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 valores¹ numérico. Mas não foi isso o que aconteceu, os valores que foram passados foram caracteres, daí a mensagem “erro de argumento” (“argument error”).

Dica 24

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.

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

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.

Listing 5.2: Operador +.

```

1  /*
2  Exemplos de operadores numéricos
3  */
4  PROCEDURE Main
5  LOCAL nNumero
6
7      ? "Tabuada (Soma)"
8      INPUT "Informe um número entre 1 e 9 : " TO nNumero
9
10     ? "Soma"
11     ? "----"
12     ? nNumero, " + 1 = " , nNumero + 1
13     ? nNumero, " + 2 = " , nNumero + 2
14     ? nNumero, " + 3 = " , nNumero + 3
15     ? nNumero, " + 4 = " , nNumero + 4
16     ? nNumero, " + 5 = " , nNumero + 5
17     ? nNumero, " + 6 = " , nNumero + 6
18     ? nNumero, " + 7 = " , nNumero + 7
19     ? nNumero, " + 8 = " , nNumero + 8
20     ? nNumero, " + 9 = " , nNumero + 9
21
22 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).

Listing 5.3: Operador -.

```

1  /*
2  Exemplos de operadores numéricos
3  */
4  PROCEDURE Main
5  LOCAL nNumero
6
7      ? "Tabuada (Subtração)"
8      INPUT "Informe um número entre 1 e 9 : " TO nNumero
9
10     ? "Subtração"
11     ? "-----"
12     ? nNumero, " - " , nNumero, " = " , nNumero - nNumero
13     ? nNumero + 1, " - " , nNumero, " = " , (nNumero + 1) - nNumero
14     ? nNumero + 2, " - " , nNumero, " = " , (nNumero + 2) - nNumero
15     ? nNumero + 3, " - " , nNumero, " = " , (nNumero + 3) - nNumero
16     ? nNumero + 4, " - " , nNumero, " = " , (nNumero + 4) - nNumero
17     ? nNumero + 5, " - " , nNumero, " = " , (nNumero + 5) - nNumero
18     ? nNumero + 6, " - " , nNumero, " = " , (nNumero + 6) - nNumero
19     ? nNumero + 7, " - " , nNumero, " = " , (nNumero + 7) - nNumero
20     ? nNumero + 8, " - " , nNumero, " = " , (nNumero + 8) - nNumero
21     ? nNumero + 9, " - " , nNumero, " = " , (nNumero + 9) - nNumero
22
23 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 subtracao.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

```

13

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 25

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
? 2 ^ 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 pode facilmente passar despercebidos 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. Em outras palavras, basta se lembrar das suas aulas de matemática quando o professor dizia que tem que resolver primeiro a multiplicação para depois resolver a adição. 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 (-)

Listing 5.4: Precedência de operadores matemáticos

```
1
2 /*
3  Tipo de dado e operador
4  */
5 PROCEDURE Main
6 LOCAL x, y, z
7
```



```

8      x := 2
9      y := 3
10     z := 4
11
12     ? "Precedência de operadores matemáticos"
13     ?
14     ? "Dados x = ", x
15     ? "          y = ", y
16     ? "          z = ", z
17     ?
18     ? "Exemplos"
19     ?
20     ? "x + y * z = " , x + y * z
21     ? "x / y ^ 2 = " , x / y ^ 2
22     ? " -x ^ 3     = " , -x ^ 3
23
24 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 26

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
- Decorar a ordem com que os operadores de mesma precedência são avaliados. Por exemplo $1 + 2 + 3$ possuem a mesma precedência, mas como o programa os avalia ? Ele soma 1 com 2 primeiro, ou soma 2 com 3 primeiro ? Isso as vezes faz toda a diferença em expressões complexas.

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.

No apêndice ?? temos a tabela com a precedência de todos os operadores.

5.2.6 Novas formas de atribuição

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

Listing 5.5: Uma nova forma de atribuição bastante usada.

```
1
2 /*
3  Atribuição de dados a variáveis
4  */
5 PROCEDURE Main
6 LOCAL x    // Valores numéricos
```

```

7
8   ? "Uma forma bastante usada de atribuição"
9   x := 10
10  ? "x vale ", x
11
12  x := x + 10
13  ? "agora vale ", x
14  x := x + 10
15  ? "agora vale ", x
16
17  x := x * 10
18  ? "agora vale ", x
19
20  x := x - 10
21  ? "agora vale ", x
22
23  x := x / 10
24  ? "agora vale ", x
25
26 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 feitos os cálculos no lado direito da atribuição para, só então, o resultado ser gravado na variável que está 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 :

Tabela 5.2: Operadores de atribuição compostos

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

```
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.

5.2.6.1 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 :

Listing 5.6: Operadores de incremento e decremento.

```
1  /*
2  Incremento e decremento
3  */
4  PROCEDURE Main
5  LOCAL nNum1 , nNum2  // Valores numéricos
6
```

```

7
8      // Operador de pré-incremento ++
9      nNum1 := 0
10     nNum2 := ++nNum1
11     ? nNum1 // Vale 1
12     ? nNum2 // Vale 1
13
14     // Operador de pós-incremento ++
15     //inicio
16     nNum1 := 0
17     nNum2 := nNum1++
18     ? nNum1 // Vale 1
19     ? nNum2 // Vale 0
20     //fim
21
22     // Operador de pré-decremento --
23     nNum1 := 1
24     nNum2 := --nNum1
25     ? nNum1 // Vale 0
26     ? nNum2 // Vale 0
27
28     // Operador de pós-decremento --
29     //inicio
30     nNum1 := 1
31     nNum2 := nNum1--
32     ? nNum1 // Vale 0
33     ? nNum2 // Vale 1
34     //fim
35
36     // Os dois operadores em conjunto
37     //inicio
38     nNum1 := 1
39     nNum2 := 5
40     ? nNum1-- * 2 + ( ++nNum2 ) // Mostra 8
41     // O cálculo efetuado foi :
42     // 1 * 2 + 6
43     // Mostra 8
44     ? nNum1 // Vale 0
45     ? nNum2 // Vale 6
46     //fim
47
48 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 27

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, ou cadeias de caracteres. Vimos também que elas servem para armazenar letras, números, datas, etc. Apesar dessa versatilidade ela não pode realizar somas com números nem outras operações, porque ela somente realiza 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.

```

1  PROCEDURE Main
2  LOCAL cNome := " Algoritmo "
3
4      ? LEN( cNome )  // Imprime o número 11 (Conta com os espaços)
5
6  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.

Listing 5.7: Operadores de concatenação

```

1  /*
2  Operadores de strings + e -
3
4  Note que todas as strings tem, propositadamente, um
5  espaço em branco no seu início e outro no seu final.
6
7  */
8  PROCEDURE Main
9  LOCAL cPre , cPos
10 LOCAL cNome
11
12     cPre := "    Harbour    " // String tamanho 16
13     cPos := "    Project    " // String tamanho 16
14
15     ? "Exemplo 1 : Concatenando com +"
16     ? cPre + cPos
17     ?? "X" // Vai ser impresso após o término da linha acima
18     ? "O tamanho da string acima é de " , Len( cPre + cPos )
19     ?
20     ? "Exemplo 2 : Concatenando com -"
21     ? cPre - cPos
22     ?? "X" // Vai ser impresso após o término da linha acima
23     ? "O tamanho da string acima é de " , Len( cPre - cPos )
24
25     //inicio
26     cNome := " (www.harbour-project.org) "
27     ?
28     ? "Exemplo 3 : Concatenando três strings com +"
29     ? cPre + cPos + cNome
30     ?? "X" // Vai ser impresso após o término da linha acima
31     ? "O tamanho da string acima é de " , Len( cPre + cPos + cNome)
32     ?
33     ? "Exemplo 4 : Concatenando três strings com -"
34     ? cPre - cPos - cNome
35     ?? "X" // Vai ser impresso após o término da linha acima
36     ? "O tamanho da string acima é de " , Len( cPre - cPos - cNome)
37     //fim
38

```


Prática número 15

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

.:Resultado:.

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

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

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

Exemplo 4 : Concatenando três strings com -
  Harbour      Project (www.harbour-project.org)      X
0 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 28

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), isso porque um operador é executado mais rapidamente do que uma função.

^aVeremos 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().

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)

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

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

Listing 5.8: Variável do tipo data

```

1
2  /*
3  Variável Data
4  */
5  PROCEDURE Main
6
7
8      dvencimento := CTOD( "08/12/2011")
9      ? dvencimento
10
11 RETURN

```

.:Resultado:.

08/12/11

Um detalhe importante que com certeza passou despercebido : 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 início da procedure **Main()**. O exemplo da listagem 5.9 mostra a forma certa de se inicializar uma data.

Listing 5.9: Variável do tipo data iniciadas corretamente

```

1
2  /*
3  Variável Data INICIADA COM O SET CORRETO
4  */
5  PROCEDURE Main
6
7      SET DATE BRITISH
8      dvencimento := CTOD( "08/12/2011")
9      ? dvencimento
10
11 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 29

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 30

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/dia/ano),
                        // 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.

^aAinda 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 inconveniente 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 :

```

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

³STOD significa “string to data”.

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 2011

0d -> Informa que é um valor do tipo data

2016 -> Ano

08 -> Mês

11 -> Dia

Veja um exemplo na listagem 5.10.

Listing 5.10: Iniciando datas

```

1
2
3
4  PROCEDURE Main()
5  LOCAL dPagamento
6
7
8      dPagamento := 0d20161201 // Primeiro de dezembro de 2016
9      ? dPagamento
10
11
12  RETURN

```

..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.

```

1 #define HOJE stod("20029999") // Data inválida

```

```

2 PROCEDURE Main
3
4     ? HOJE // Exibir uma data nula, não um erro.
5
6 RETURN

```

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).

```

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

```

.: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 :

```

1 #define DATA_LIMITE_COPIA 0d20173109 // Essa data está errada
2 PROCEDURE Main
3
4     ? DATA_LIMITE_COPIA
5     SET DATE BRITISH
6     ? DATA_LIMITE_COPIA
7
8 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.

Listing 5.11: Configurações de data

```

1
2  /*
3  Variável Data exibida corretamente
4  */
5  PROCEDURE Main
6  LOCAL dVenc // Data de vencimento
7
8      dVenc := CTOD( "12/31/2021" ) // 31 de dezembro de 2021
          (mês/dia/ano)
9
10     ? "Exibindo a data 31 de dezembro de 2021."
11     ? "O formato padrão é o americano (mês/dia/ano)"
12     ? dVenc
13
14     SET DATE BRITISH // Exibe as datas no formato dia/mês/ano
15     ? "Exibe as datas no formato dia/mês/ano"
16     ? "O vencimento da última parcela é em " , dVenc
17
18     SET CENTURY ON // Ativa a exibição do ano com 4 dígitos
19     ? "Ativa a exibição do ano com 4 dígitos"
20     ? "A mesma data acima com o ano com 4 dígitos : ", dVenc
21
22     ? "Outros exemplos com outras datas : "
23     ? CTOD("26/08/1970")
24     dVenc := CTOD( "26/09/1996" )
25     ? dVenc
26     dVenc := CTOD( "15/05/1999" )
27     ? dVenc
28
29     SET CENTURY OFF // Data volta a ter o ano com 2 dígitos
30     ? "Data volta a ter o ano com 2 dígitos"
31     ? dVenc
32
33

```


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.

Listing 5.12: Operadores de data

```

1
2  /*
3  Variável Data exibida corretamente
4  */
5  PROCEDURE Main
6  LOCAL dCompra // Data da compra
7  LOCAL dVenc // Data de vencimento
8
9      SET DATE BRITISH
10     dCompra := CTOD( "01/02/2015" ) // Data da compra
11     dVenc := CTOD( "05/02/2015" ) // Data do vencimento
12
13     ? "Data menor (compra) :", dCompra
14     ? "Data maior (vencimento) :", dVenc
15     ? "Maior menos a menor (dias entre)"
16     ? "Data de vencimento menos a Data da compra"
17     ? dVenc - dCompra // Resulta em 4
18     ? "Menor menos a maior (dias entre)"
19     ? "Data da compra menos a Data de vencimento"
20     ? dCompra - dVenc // Resulta em -4
21     ? "Subtrai dois dias"
22     ? "Data de vencimento menos dois dias"
23     ? dVenc - 2 // Resulta em 03/02/15
24     ? "Soma dois dias (data + número)"
25     ? "Data de vencimento mais dois dias"
26     ? dVenc + 2 // Resulta em 07/02/15
27     ? "Soma dois dias (número + data ) "
28     ? "2 + dVenc"
29     ? 2 + dVenc // Resulta em 07/02/15 (a mesma coisa)
30
31
32 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 31

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/ano),
                        // 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 mantém 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.

```

1  PROCEDURE Main
2  LOCAL dVenc := dPag := Od20161211 // Atribuição múltipla
3
4      SET DATE BRITISH
5      ? dVenc, dPag
6
7  RETURN

```

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).

Listing 5.13: Operadores `+=` e `-=`

```

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

```

.: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.

Listing 5.14: Operadores -=

```
1  /*
2  Variável Data e operadores -=
3  */
4  PROCEDURE Main
5  LOCAL dVenc // Data de vencimento
6  LOCAL dPag // Data de pagamento
7
8      SET DATE BRITISH
9      dVenc := CTOD( "08/12/2011")
10     dPag := CTOD( "02/12/2011")
11     dVenc -= dPag
12     ? "A diferença entre o vencimento e o pagamento foi de ",
        dVenc , "dias"
13
14 RETURN
```

.: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 datas
```

Dica 32

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, operadores 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 minúsculas.

```
.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. Veremos o uso de valores lógicos nos próximos capítulos.

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.

```

1  PROCEDURE Main
2  LOCAL xValor
3
4      xValor := 12
5      ? xValor
6      xValor := "Cliente inadimplente"
7      ? xValor
8      SET DATE BRITISH
9      xValor := DATE()
10     ? xValor
11     xValor := .f.
12     ? xValor
13     // O restante das instruções ...
14
15 RETURN

```

.: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 33

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*.

Listing 5.15: O valor NIL.

```

1  /*
2  O valor NIL
3  */
4  PROCEDURE Main
5  LOCAL x
6
7  ? "O valor de x é : ", x      // O valor de x é : NIL
8
9  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*. No apêndice XXX nós veremos todos os *SETs* com os seus respectivos valores válidos, você não precisa se apressar em decorá-los pois nós iremos aprender os tipos principais no decorrer do livro.
2. Não existem *SETs* sem valores. Todos eles possuem valores criados durante a inicialização do programa ou seja, um valor padrão⁵. 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 34

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
//
```

⁵O termo técnico para “valor padrão” é valor *default*.

```
... Continua...
```

```
RETURN NIL
```

5.10 Exercícios de fixação

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 $V = \frac{4*PI*Raio^3}{3}$.

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 :

```
1 PROCEDURE Main
2 LOCAL x,y,z
```

```

3
4     INPUT "Insira o valor de x : " TO x
5     INPUT "Insira o valor de y : " TO y
6     z = ( 1 / ( x + y ) )
7     ? "O valor de z é : " , z
8
9
10  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.

- $x = \frac{y-3}{z}$
- $k = \frac{x^3+z/5}{y^2+8}$
- $y = \frac{x^3z}{r} - \frac{4n}{h}$
- $t = \frac{y}{2} + \frac{3k^2}{4n}$

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 :

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

```
18      ? "Entre os números " , nIni, " e " , nFim, " existem ",  
      nQtd, " números"  
19  
20 RETURN
```

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

Referências Bibliográficas

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, L. **Linguagem C**. LTC, Rio de Janeiro, 2013.

DEITEL, H. M.; DEITEL, P. J. **C++ Como programar**. Bookman, Porto Alegre, 2001.

FORBELLONE, A. L.; EBERSPACHER, H. F. **Lógica de programação: a construção de algoritmos e estrutura de dados**. Prentice Hall, São Paulo, 2005.

HORSTMAN, C. **Conceitos de computação com o essencial de C++**. Bookman, Porto Alegre, 2005.

KERNIGHAN, B. W.; PIKE, R. **A prática da programação**. Campus, Rio de Janeiro, 2000.

KERNIGHAN, B. W.; RITCHIE, D. M. **C: a linguagem de programação**. Campus, Rio de Janeiro, 1986.

KRESIN, A. *Harbour for beginners*. 2016. <http://www.kresin.ru/en/hrbfaq.html/>. [Online; accessed 13-Ago-2016].

MIZRAHI, V. V. **Treinamento em linguagem C++**. Pearson, São Paulo, 2004.

NANTUCKET. **Clipper 5.0 - manual de referência**. Nantucket Corporation, 1990.

RAMALHO, J. A. A. **Clipper 5.0 - Básico**. McGraw-Hill, São paulo, 1991.

SPENCE, R. **Clipper 5.0 - release 5.01**. Makron, Rio de Janeiro, 1991.

SPENCE, R. **Clipper 5.2**. Makron, Rio de Janeiro, 1994.

STROUSTRUP, B. **Princípios e práticas de programação com C++**. Bookman, Porto Alegre, 2012.

SWAN, T. **Tecle e aprenda C**. Berkeley Brasil editora, São Paulo, 1994.

TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de dados usando C**. Makron Books, São Paulo, 1995.

VIDAL, A. G. d. R. **Clipper**. LTC - Livros Técnicos e Científicos, Rio de Janeiro, 1989.

VIDAL, A. G. d. R. **Clipper 5**. LTC - Livros Técnicos e Científicos, Rio de Janeiro, 1991.



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.

A.1 Resposta aos exercícios de fixação sobre variáveis - Capítulo 4

1.

2. 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.

Listing A.1: Resposta

```

1  /*
2  Entrada : 3 variáveis numéricas e 3 variáveis caracteres
3  Saída : As variáveis mostradas
4  */
5  PROCEDURE Main
6  LOCAL nA, nB, nC // Variáveis numéricas
7  LOCAL cA, cB, cC // Variáveis caracteres
8
9      INPUT "Insira o primeiro valor numérico : " TO nA
10     INPUT "Insira o segundo valor numérico : " TO nB
11     INPUT "Insira o terceiro valor numérico : " TO nC
12
13     ACCEPT "Insira o primeiro valor caractere : " TO cA
14     ACCEPT "Insira o segundo valor caractere : " TO cB
15     ACCEPT "Insira o terceiro valor caractere : " TO cC
16
17     ? "Os valores numéricos : " , nA, nB, nC
18     ? "Os valores caracteres : " , cA, cB, cC
19
20 RETURN

```

3. (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 rodapé¹.

Listing A.2: Resposta

```
1  /*
2  Entrada : 3 variáveis numéricas e 3 variáveis caracteres
3  Saída : As variáveis mostradas
4  */
5  PROCEDURE Main
6  LOCAL cEntrada // Valor a ser digitado
7
8      ? "Oi, meu nome é Hal!"
9      ? "O que você gostaria que eu fizesse ?"
10     ?
11     ACCEPT TO cEntrada
12
13     ? "Sinto muito, eu não posso fazer isso"
14
15  RETURN
```

- 4.(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.
```

Listing A.3: Resposta

```
1  /*
2  Entrada : 3 variáveis numéricas e 3 variáveis caracteres
3  Saída : As variáveis mostradas
4  */
```

¹O “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.

```

5 PROCEDURE Main
6 LOCAL cEntrada // Valor a ser digitado
7
8     ? "Oi, meu nome é Hal!"
9     ? "Qual é o seu nome ?"
10    ? // Pula uma linha (fica melhor para o usuário)
11    ACCEPT TO cEntrada
12
13    ? "Oi," , cEntrada, "prazer em conhecê-lo."
14
15 RETURN

```

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

Listing A.4: Resposta

```

1  /*
2  Descr. : Programa que recebe quatro números,
3           calcula e mostra a soma desses números.
4  Entrada: quatro números.
5  Saída  : a soma desses números exibida.
6  */
7  PROCEDURE Main
8  LOCAL n1, n2, n3, n4 // Parcelas
9  LOCAL nSoma // Soma
10
11     INPUT "Entre com valor 1 : " TO n1
12     INPUT "Entre com valor 2 : " TO n2
13     INPUT "Entre com valor 3 : " TO n3
14     INPUT "Entre com valor 4 : " TO n4
15
16     nSoma := n1 + n2 + n3 + n4
17     ? "A soma desses números é " , nSoma
18
19 RETURN

```

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

Listing A.5: Resposta

```

1  /*
2  Descr. : Programa que recebe três notas,
3           calcula e mostra a média desses números.
4  Entrada: três números.
5  Saída  : a média desses números exibida.
6  */
7  PROCEDURE Main
8  LOCAL n1, n2, n3 // Valores

```

```

9 LOCAL nMedia // Média dos valores
10
11     INPUT "Entre com valor 1 : " TO n1
12     INPUT "Entre com valor 2 : " TO n2
13     INPUT "Entre com valor 3 : " TO n3
14
15     nMedia := (( n1 + n2 + n3 ) / 3 )
16     ? "A média desses números é " , nMedia
17
18 RETURN

```

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

Listing A.6: Resposta

```

1  /*
2  Descr. : Programa que recebe três notas e seus pesos,
3           calcula e mostra a média ponderada desses números.
4  Entrada: três números e três pesos.
5  Saída  : a média ponderada desses números exibida.
6  */
7  PROCEDURE Main
8  LOCAL n1, n2, n3 // Valores
9  LOCAL p1, p2, p3 // Pesos
10 LOCAL nMedia // Média
11
12     INPUT "Entre com valor 1 : " TO n1
13     INPUT "Entre com peso 1 : " TO p1
14     INPUT "Entre com valor 2 : " TO n2
15     INPUT "Entre com peso 2 : " TO p2
16     INPUT "Entre com valor 3 : " TO n3
17     INPUT "Entre com peso 3 : " TO p3
18
19     nMedia := ( ( ( n1 * p1 ) + ( n2 * p2 ) + ( n3 * p3 ) ) / (
20         p1 + p2 + p3 ) )
21     ? "A média ponderada desses números é " , nMedia
22
23 RETURN

```

8. 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.

Listing A.7: Resposta

```

1  /*
2  Descr. : Programa que recebe o salário do funcionário, o
           percentual de reajuste

```

```

3          calcule e mostre o novo salário.
4  Entrada: Salário.
5  Saída  : O salário reajustado.
6  */
7  PROCEDURE Main
8  LOCAL nSalario
9  LOCAL nAumento
10 LOCAL nSalarioNovo
11
12     INPUT "Entre com valor do salário : " TO nSalario
13     INPUT "Entre com o percentual de aumento (Ex: 15 se for 15%)
        : " TO nAumento
14
15     nSalarioNovo := nSalario + ( nSalario * nAumento ) / 100
16     ? "O novo salário é " , nSalarioNovo
17
18 RETURN

```

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

Listing A.8: Resposta

```

1  /*
2  Descr. : Programa que recebe o salário do funcionário,
3          o percentual de aumento,
4          calcule e mostre o novo salário, e o valor do aumento.
5  Entrada: Salário e percentual de aumento.
6  Saída  : O novo salário e o valor do aumento.
7  */
8  PROCEDURE Main
9  LOCAL nSalario
10 LOCAL nPercAumento
11 LOCAL nValorAumento
12 LOCAL nSalarioNovo
13
14     INPUT "Entre com valor do salário : " TO nSalario
15     INPUT "Entre com o percentual de aumento (Ex: 15) : " TO
        nPercAumento
16
17     nValorAumento := ( nSalario * nPercAumento ) / 100
18     nSalarioNovo := nSalario + nValorAumento
19     ? "O novo salário é " , nSalarioNovo
20     ? "O aumento foi de " , nValorAumento
21
22 RETURN

```

10. 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.

Listing A.9: Resposta

```

1  /*
2  Descr. : Programa que recebe o salário do funcionário,
3           o percentual de aumento, o percentual do imposto
4           calcule e mostre o novo salário, o percentual do
5           imposto
6           e o valor do aumento.
7  Entrada: Salário, percentual do imposto e percentual de aumento.
8  Saída : O novo salário, o valor do aumento e o valor do
9           imposto.
10 */
11 PROCEDURE Main
12 LOCAL nSalario
13 LOCAL nPercAumento
14 LOCAL nPercImposto
15 LOCAL nValorAumento
16 LOCAL nValorDoImposto
17 LOCAL nSalarioNovo
18
19 INPUT "Entre com valor do salário : " TO nSalario
20 INPUT "Entre com o percentual de aumento (Ex: 25) : " TO
21     nPercAumento
22 INPUT "Entre com o percentual de imposto (Ex: 10) : " TO
23     nPercImposto
24
25 nValorAumento := ( nSalario * nPercAumento ) / 100
26 nSalarioNovo := nSalario + nValorAumento
27 nValorDoImposto := ( nSalarioNovo * nPercImposto ) / 100
28 nSalarioNovo := nSalarioNovo - nValorDoImposto
29 ? "O novo salário é " , nSalarioNovo
30 ? "O aumento foi de " , nValorAumento
31 ? "O imposto foi de " , nValorDoImposto
32
33 RETURN

```

11. 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.

Listing A.10: Resposta

```

1  /*
2  Descr. : Recebe o valor de um depósito e o valor da taxa de
3           juros,

```

```

3          calcule e mostre o valor do rendimento e o valor total
4          depois do rendimento.
5  Entrada: Depósito e taxa de juros.
6  Saída  : O valor do rendimento e o valor total depois do
          rendimento.
7  */
8  PROCEDURE Main
9  LOCAL nDeposito // Valor depositado
10 LOCAL nTaxaJuros // Taxa de juros
11 LOCAL nRendimento // Rendimento
12 LOCAL nValorFinal // Valor final
13
14     INPUT "Entre com valor do depósito : " TO nDeposito
15     INPUT "Entre com a taxa de juros (Ex: 25) : " TO nTaxaJuros
16
17     nRendimento := ( nDeposito * nTaxaJuros ) / 100
18     nValorFinal := nDeposito + nRendimento
19     ? "O valor do rendimento foi " , nRendimento
20     ? "O valor total depois do rendimento foi " , nValorFinal
21
22 RETURN

```

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

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

Listing A.11: Resposta

```

1  /*
2  Descr. : Calcula a área de um triangulo sabendo que
3          Area = ( Base * Altura ) / 2
4  Entrada: Base e altura.
5  Saída  : O valor da área de um triangulo.
6  */
7  PROCEDURE Main
8  LOCAL nBase // Base
9  LOCAL nAltura // Altura
10 LOCAL nArea // Area
11
12     INPUT "Entre com a base : " TO nBase
13     INPUT "Entre com a altura : " TO nAltura
14
15     nArea := ( nBase * nAltura ) / 2
16     ? "O valor da área é " , nArea
17
18 RETURN

```


13. 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.

Listing A.12: Resposta

```

1  /*
2  Entrada: ano de nascimento e o ano atual
3  Saída : a idade da pessoa, quantos anos ela terá em 2045
4
5  Dica : YEAR( DATE() ) retorna o ano corrente.
6  */
7  #define ANO_FUTURO 2045
8  PROCEDURE Main
9  LOCAL nAnoNasc, nAno := YEAR( DATE() )
10 LOCAL nIdade, nIdadeFuturo
11
12     INPUT "Entre com o ano do seu nascimento : " TO nAnoNasc
13
14     nIdade := nAno - nAnoNasc
15     nIdadeFuturo := ANO_FUTURO - nAnoNasc
16
17     ? "Sua idade atual é " , nIdade
18     ? "Em" , ANO_FUTURO, "você terá " , nIdadeFuturo , "anos"
19
20 RETURN

```

14. 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

Listing A.13: Resposta

```

1  /*
2  Descr. : Recebe uma medida em pés e converte a medida
3           em polegada, jarda, milha
4  Entrada: O número em pés
5  Saída : O valor da entrada em pés e converte a medida
6           em polegada, jarda, milha
7
8
9  Nota : pé = 12 polegadas
10       1 jarda = 3 pés

```

```

11      1 milha = 1,760 jarda
12
13  */
14  PROCEDURE Main
15  LOCAL nPe
16  LOCAL nPolegada, nJarda, nMilha
17
18      INPUT "Entre com a medida em pés : " TO nPe
19
20      nPolegada := nPe * 12
21      nJarda := nPe / 3
22      nMilha := nJarda / 1760
23
24      ? "O valor em polegadas é " , nPolegada
25      ? "O valor em jardas é " , nJarda
26      ? "O valor em milhas é " , nMilha
27
28  RETURN

```

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

Listing A.14: Resposta

```

1  /*
2  Entrada: Base e expoente
3  Saída : a potência
4  */
5  PROCEDURE Main
6  LOCAL nBase, nExpoente
7  LOCAL nPotencia
8
9      INPUT "Entre com a base : " TO nBase
10     INPUT "Entre com o expoente : " TO nExpoente
11
12     nPotencia := nBase ^ nExpoente
13
14     ? "O valor da potência é " , nPotencia
15
16  RETURN

```

16. 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 “*”.

Listing A.15: Resposta

```

1  /*
2  Entrada: Nome e idade
3  Saída  : Meses de vida aproximados
4  */
5  #define MESES_POR_ANO 12
6  PROCEDURE Main
7  LOCAL cNome
8  LOCAL nIdade, nMeses
9
10     ACCEPT "Informe o seu nome : " TO cNome
11     INPUT "Entre com a idade : " TO nIdade
12
13     nMeses := nIdade * MESES_POR_ANO
14
15     ? "O seu nome é " , cNome, " e você tem aproximadamente " ,
        nMeses , " meses de vida"
16
17 RETURN

```

17. 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

```

Listing A.16: Resposta

```

1  /*
2  Entrada: Valor em horas
3  Saída  : Valor convertido para segundos
4  */
5  #define SEGUNDOS_POR_HORA 60*60 // 1 hora = 60 minutos * 60
        segundos
6  PROCEDURE Main
7  LOCAL nHora, nSegundos
8

```

```

9      INPUT "Entre com o valor em horas : " TO nHora
10
11      nSegundos := nHora * SEGUNDOS_POR_HORA
12
13      ? nHora, "horas tem" , nSegundos, "segundos."
14
15  RETURN

```

18. 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.

Listing A.17: Resposta

```

1  /*
2  Entrada: Quilometragem inicial e quilometragem final, e
   quantidade de combustível
3  Saída : Valor do consumo
4  */
5  PROCEDURE Main
6  LOCAL nKmIni, nKmFim // Quilometragem
7  LOCAL nQtd // Quantidade consumida
8  LOCAL nConsumo // Consumo
9
10     INPUT "Entre com o valor da quilometragem inicial : " TO
        nKmIni
11     INPUT "Entre com o valor da quilometragem final : " TO nKmFim
12     INPUT "Entre com a quantidade consumida : " TO nQtd
13
14     nConsumo := ( nKmFim - nKmIni ) / nQtd
15
16     ? "O consumo é de " , nConsumo , " Km/l"
17
18  RETURN

```

A.2 Respostas aos desafios - Capítulo 4

A.2.1 Identifique o erro de compilação no programa abaixo.

Listing A.18: Erro 1

```

1  /*
2  Onde está o erro ?
3  */
4  PROCEDURE Main
5  LOCAL x, y, x // Número a ser inserido

```

```

6
7     x := 5
8     y := 10
9     x := 20
10
11 RETURN

```

Resposta : A variável x foi declarada duas vezes.

A.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*.

Listing A.19: Erro de lógica

```

1  /*
2  Onde está o erro ?
3  */
4  PROCEDURE Main
5  LOCAL x, y // Número a ser inserido
6
7     x := 5
8     y := 10
9     ACCEPT "Informe o primeiro número : " TO x
10    ACCEPT "Informe o segundo número : " TO y
11    ? "A soma é ", x + y
12
13 RETURN

```

Resposta : O comando ACCEPT foi usado para receber valores numéricos.

A.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.

```

Listing A.20: Resposta

```

1  /*
2  Entrada: Quantidade de moedas (25, 10 e 1 centavos)
3  Saída  : Valor em moedas
4  */
5  #define FATOR_25 0.25 // Fator de conversão para 25 centavos
6  #define FATOR_10 0.1 // Fator de conversão para 10 centavos
7  #define FATOR_01 0.01 // Fator de conversão para 1 centavo
8  PROCEDURE Main
9  LOCAL nQtd1Cent, nQtd10Cent, nQtd25Cent // Quantidades
10 LOCAL nValor // Valor em moedas
11
12 INPUT "Quantidade de moedas de um centavo : " TO nQtd1Cent
13 INPUT "Quantidade de moedas de dez centavos : " TO nQtd10Cent
14 INPUT "Quantidade de moedas de vinte e cinco centavos : " TO
    nQtd25Cent
15
16 nValor := ( nQtd25Cent * FATOR_25 ) + ( nQtd10Cent * FATOR_10 )
    + ( nQtd1Cent * FATOR_01 )
17
18 ? "Você tem R$" , nValor, "em moedas"
19
20 RETURN

```

A.2.4 O comerciante maluco

Adaptado de (FORBELLONE; 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.

Listing A.21: Resposta

```

1  /*
2  Entrada : O valor da parcela
3  Saída  : O valor final após o acréscimo, o valor final após o
    desconto e o prejuízo.
4  */
5  #define DESCONTO 0.10
6  #define ACRESCIMO 0.10

```

```

7  PROCEDURE Main
8  LOCAL nParcela // nValor inicial
9  LOCAL nValorAcr // nValor final com acréscimo
10 LOCAL nValorDesc // Valor final após o desconto
11
12     INPUT "Informe o valor da parcela em atraso : " TO nParcela
13
14     nValorAcr := nParcela + ( nParcela * ACRESCIMO )
15     ? "O valor da parcela acrescida de ", ACRESCIMO, "por cento é "
16     , nValorAcr
17     nValorDesc := nValorAcr - ( nValorAcr * DESCONTO )
18     ? "O valor descontado ", DESCONTO, "por cento é " , nValorDesc
19     ? "O prejuízo é de " , nParcela - nValorDesc
20
21 RETURN

```

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.

A.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 $PI = 3.1415$).

Listing A.22: Resposta

```

1  /*
2  Descr. : Calcula a área de um círculo sabendo que
3           Area = PI * Raio ^ 2
4  Entrada: O raio.
5  Saída : O valor da área de um círculo.
6  */
7  #define PI_VALOR 3.1415
8  PROCEDURE Main
9  LOCAL nRaio // Raio da circunferência
10 LOCAL nArea // Área
11
12     INPUT "Entre com o raio : " TO nRaio
13
14     nArea := PI_VALOR * nRaio ^ 2
15     ? "O valor da área é " , nArea
16
17 RETURN

```

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 $\sqrt[2]{100} = 100^{\frac{1}{2}}$

Listing A.23: Resposta

```

1  /*
2  Descr. : Calcula o número ao quadrado, ao cubo, a raiz quadrada
3           e a raiz cúbica do número
4  Entrada: O número.
5  Saída  : o número ao quadrado, ao cubo, a raiz quadrada
6           e a raiz cúbica do número
7
8  Nota : A raiz é o número elevado a 1 / potência
9  */
10 PROCEDURE Main
11 LOCAL nNumero
12
13     INPUT "Entre com o raio : " TO nNumero
14
15     ? "O quadrado é " , nNumero ^ 2
16     ? "O cubo é " , nNumero ^ 3
17     ? "A raiz quadrada é " , nNumero ^ ( 1 / 2 )
18     ? "A raiz cúbica é " , nNumero ^ ( 1 / 3 )
19
20 RETURN

```