



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.

95p.;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 a responsabilidade é minha. Por favor envie um e-mail com as correções a serem feitas para vlad@altersoft.net com o título “E-book Harbour”.

“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	25
Figura 2.2	Processo de geração de um programa escrito em Harbour	27
Figura 2.3	Adicionando as variáveis de ambiente	28
Figura 2.4	Adicionando as variáveis de ambiente	28
Figura 2.5	Adicionando as variáveis de ambiente	28
Figura 2.6	Abrindo o prompt de comando do windows	29
Figura 2.7	Criando um atalho na área de trabalho	30
Figura 2.8	Abrindo o prompt de comando do windows pelo xDevStudio	31
Figura 3.1	Criando o arquivo hello.prg pelo xDevStudio	37
Figura 3.2	Salvando o arquivo	37
Figura 3.3	Salvando o arquivo (Parte II)	38

Lista de Tabelas

Sumário

1	INTRODUÇÃO	15
1.1	A quem se destina esse livro	15
1.2	Pré-requisitos necessários	16
1.3	Metodologia utilizada	17
1.4	Material de Apoio	18
1.5	Plano da obra	18
1.6	Porque Harbour ?	19
2	O PROCESSO DE CRIAÇÃO DE UM PROGRAMA DE COMPUTADOR ..	22
2.1	O que é um programa de computador ?	22
2.2	Linguagens de programação	23
2.3	Tipos de linguagens de programação	25
2.3.1	O Harbour pertence a qual categoria ?	26
2.3.2	Instalando o Harbour no seu computador	27
2.3.2.1	Testando se o Harbour foi instalado corretamente	29
2.3.3	Instalando o xDevStudio	30
2.3.4	O processo de compilação	31
2.3.5	Conclusão	33
3	MEU PRIMEIRO PROGRAMA EM HARBOUR	34
3.1	De que é feito um programa de computador ?	35
3.2	O mínimo necessário	35

3.3	O primeiro programa	36
3.3.1	Uma pequena pausa para a acentuação correta	43
3.4	As strings	44
3.4.1	Quebra de linha	47
3.5	Números	48
3.6	Uma pequena introdução as Funções	49
3.7	Comentários	51
3.8	Praticando	53
3.9	Desafio	55
3.10	Exercícios	55
4	CONSTANTES E VARIÁVEIS	57
4.1	Constantes	58
4.2	Variáveis	60
4.2.1	Criação de variáveis de memória	60
4.2.2	Escolhendo nomes apropriados para as suas variáveis	61
4.2.3	Seja claro ao nomear suas variáveis	64
4.2.4	Atribuição	66
4.3	Variáveis: declaração e inicialização	67
4.4	Recebendo dados do usuário	69
4.5	Exemplos	72
4.5.1	Realizando as quatro operações	72
4.5.2	Calculando o antecessor e o sucessor de um número	73
4.6	Exercícios de fixação	74
4.7	Desafios	76
4.7.1	Identifique o erro de compilação no programa abaixo.	76
4.7.2	Identifique o erro de lógica no programa abaixo.	77
4.7.3	Valor total das moedas	77
4.7.4	O comerciante maluco	77
	REFERÊNCIAS BIBLIOGRÁFICAS	79
A	EXERCÍCIOS : CONSTANTES E VARIÁVEIS	80

A.1	Resposta aos exercícios de fixação sobre variáveis - Capítulo 4	81
A.2	Desafios	93
A.2.1	Identifique o erro de compilação no programa abaixo.	93
A.2.2	Identifique o erro de lógica no programa abaixo.	93
A.2.3	Valor total das moedas	94
A.2.4	O comerciante maluco	94

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 sequencia da maioria dos livros de introdução a programação e algoritmos. Contudo, verifiquei que existem pelo menos dois grupos de livros de informática: o primeiro grupo, o mais tradicional, procura apresentar os tópicos obedecendo uma estrutura sequencial, iniciando com as estruturas mais básicas até chegar nas estruturas mais complexas da linguagem. Pertencem a essa categoria livros como “Conceitos de computação usando C++” (Cay Horstman), “C++ como programar” (Deitel e Deitel), “Princípios e práticas de programação com C++” (Bjarne Stroustrup) , “Clipper 5.0 Básico” (José Antonio Ramalho) e a coleção “Clipper 5” (Geraldo A. da Rocha Vidal). Nesses livros não existem muitas quebras de sequencia, por exemplo: os códigos envolvendo variáveis são muito poucos, até que o tópico variável seja abordado completamente. Essa abordagem pode não agradar a algumas pessoas, da mesma forma que um filme de ação pode entediar o expectador se demorar demais a exhibir 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 sequencia dos cursos introdutórios de programação. Se por acaso você não tem experiência alguma com programação e deseja iniciar o aprendizado, essa livro será de grande ajuda para você, de modo que você não terá muitas surpresas com os códigos apresentados. Se você praticar os códigos apresentados, o aprendizado será um pouco lento, mas você não correrá o risco de ter surpresas no seu futuro profissional por usar um comando de uma forma errada. Caso você se 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. Sejamos claros : se você deseja aprender programação com o único objetivo de arranjar um emprego em pouco tempo então não estude Harbour, parta para outras linguagens mais populares como Java, PHP e C#. Harbour é indicado para os seguintes casos :

1. **O programador da linguagem Clipper** que deseja migrar para outra plataforma (o Clipper gera aplicativos somente para a plataforma MS-DOS) sem efetuar mudanças significativas no seu código fonte.
2. **Profissional que quer ser o dono do próprio negócio desenvolvendo aplicativos comerciais** : Sim, nós dissemos que você pode até desenvolver jogos e aplicativos gráficos com Harbour. Mas você vai encontrar pouco sobre esses assuntos nos fóruns da linguagem. O assunto predominante, depois de dúvidas básicas, são aqueles relacionados ao dia-a-dia de um sistema de informação comercial, como comunicação com impressoras fiscais e balanças eletrônicas, acesso a determinado banco de dados, particularidades de alguma interface gráfica, etc. O assunto pode, até mesmo, resvalar para normas fiscais e de tributação, pois o programador de aplicativos empresariais precisa conhecer um pouco sobre esses assuntos.
3. **Profissional que quer aprender rapidamente** e produzir logo o seu próprio software com rapidez e versatilidade.
4. **O usuário com poucos conhecimentos técnicos de programação** mas que já meche com softwares de produtividade (Excel, Calc, MS-Access, etc.) podem se utilizar do Harbour para adentrar no mundo da programação pela porta da frente em pouco tempo.
5. **O estudante** que quer ir além do que é ensinado nos cursos superiores. Com Harbour você poderá integrar seus códigos C e C++ (Biblioteca QT, wxWidgets, etc.) além de fuçar as “entranhas” de inúmeros projetos livres desenvolvidos em Harbour. Por trás do código simples do Harbour existem montanhas de código em C puro, ponteiros, estruturas de dados, integração com C++, etc.
6. **O aprendiz** ou o *hobbista* que quer aprender a programar computadores mas não quer adentrar em uma infinidade de detalhes técnicos pode ter no Harbour uma excelente ferramenta.
7. **O Hacker**³ que quer desenvolver ferramentas de segurança (e outras ferramentas de análise) pode obter com o Harbour os subsídios necessários para as suas pesquisas.

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

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 sequencia e de decisão que o computador executa para realizar uma tarefa. Todos os computadores que você já utilizou, desde um relógio digital até um sofisticado smartphone são controlados por programas que executam essas operações extremamente primitivas. Parece impossível, mas todos os programas de computador que você já usou, todos mesmo, utilizam apenas poucas operações primitivas :

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

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

2.2 Linguagens de programação

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

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

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

Ainda hoje os computadores, todos eles, funcionam através desses códigos. O que mudou foi a forma com a qual eles são gerados. Antes eles eram 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 sequencia 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.

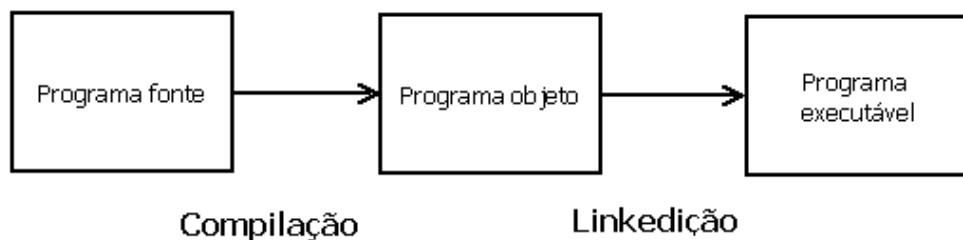
¹Essa classificação de médio nível para a Linguagem C foi dada pelos criadores da linguagem. A Linguagem C, portanto, abrange duas categorias : alto nível e médio nível.

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 ?

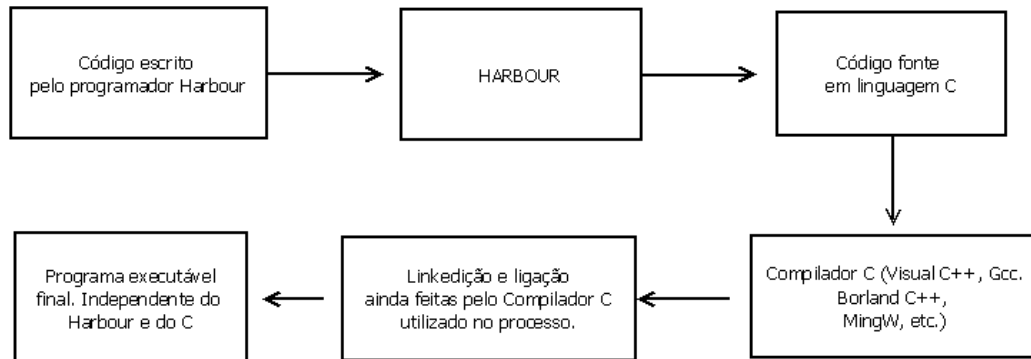
O lema da linguagem Harbour é : “escreva uma vez, compile em qualquer lugar”, então o Harbour é um compilador certo ? Mais ou menos. Na verdade o Harbour não é classificado tecnicamente como um compilador puro, pois ele depende de um compilador externo para poder gerar o seu programa executável independente.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

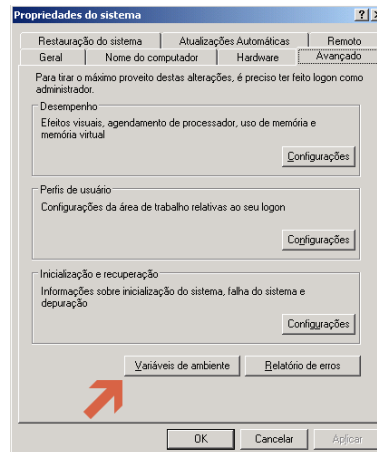


Figura 2.4: Adicionando as variáveis de ambiente

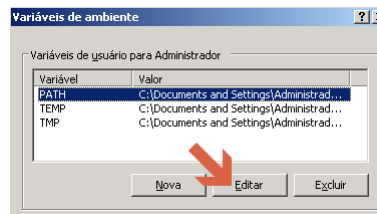
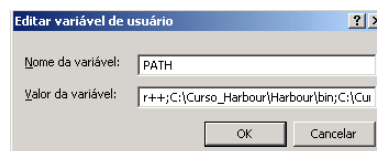


Figura 2.5: Adicionando as variáveis de ambiente



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 :

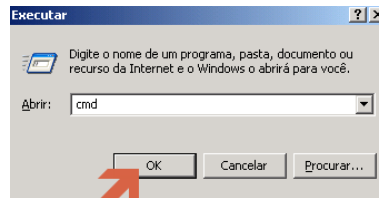
C:\Curso_Harbour\Harbour\bin;C:\Curso_Harbour\Harbour\comp\mingw\bin

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

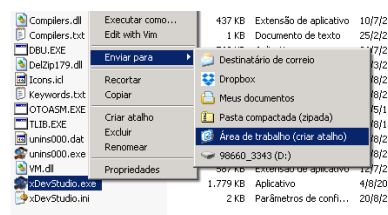
O xDevStudio é um editor desenvolvido por Vailton Renato (WebNet do Brasil) especialmente para os desenvolvedores Clipper, Harbour e outros da família xBase. Nós insistimos para que você use esse editor para digitar os códigos desse livro. Por vários motivos :

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 clique 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.

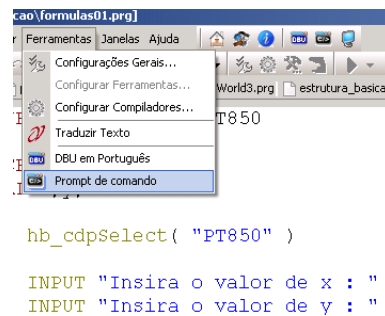
Observe que o prompt de comando abre na mesma pasta onde está o xDevStudio. Tudo o que você tem a fazer é navegar, através de comandos do prompt até a pasta de exemplos para poder compilar alguns exemplos de teste. Faça assim, dentro do prompt de comando (após cada linha tecler enter) :

.: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 tecler 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 tecla 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

3.2 O mínimo necessário

A linguagem Harbour possui dois tipos de rotinas : as *procedures* 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 ?

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

No caso do Harbour, existe uma rotina (função ou procedure) 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 cabeçalho (PROCEDURE Main) e o comando Return.

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 Main.

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 “?” é um comando que 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 *hbm2*.

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.

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”.

²Note que nos exemplos nós usamos a palavra reservada PROCEDURE para iniciar a nossa rotina. Nos capítulos seguintes nós veremos a diferença entre uma procedure e uma função.

Figura 3.1: Criando o arquivo hello.prg pelo xDevStudio

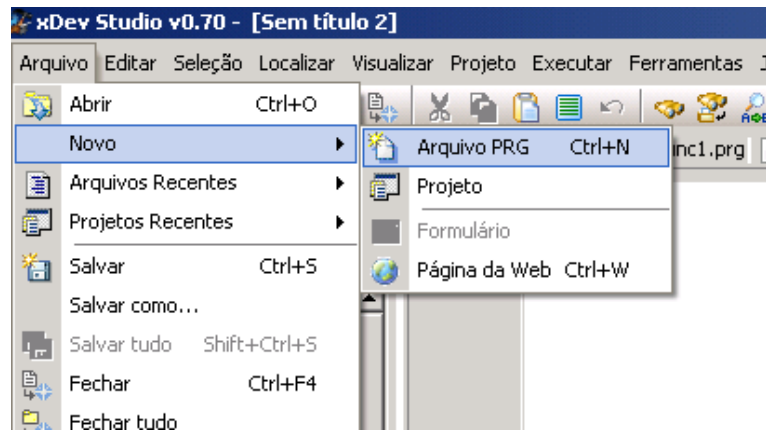
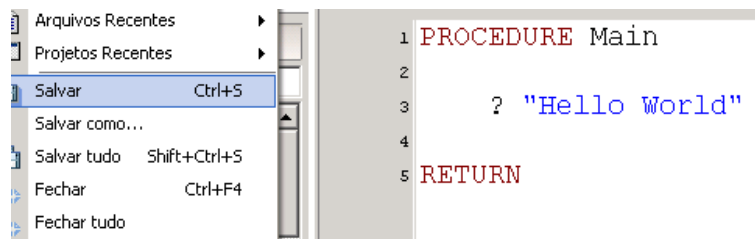


Figura 3.2: Salvando o arquivo



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:.

```

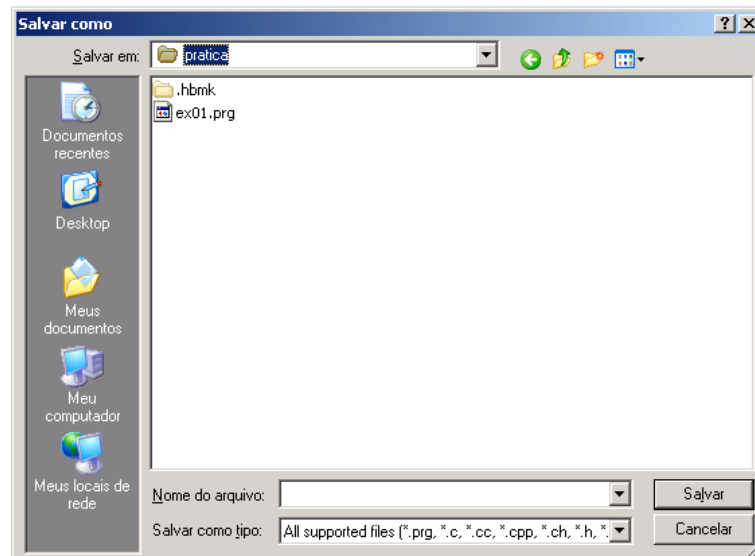
C:\Curso_Harbour\pratica> hbm2 hello
  
```

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:.



Figura 3.3: Salvando o arquivo (Parte II)



```
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 parênteses 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 a única exceção da observação anterior.
3. **Teste 3:** Se você abriu um parenteses mas não fechou (não importa se foi o primeiro ou o último) ou se você esqueceu os parênteses, 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 lê³

³Essa 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.

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 conseguiu explicar a causa. 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 prática não é nenhuma regra obrigatória, mas ajuda a manter o seu código legível. Tome cuidado com o editor de texto que você utiliza para programar, pois os espaços entre as tabulações podem variar de editor para editor. Fique atento ao tamanho das tabulações no seu editor de texto, as tabulações são úteis para facilitar a marcação das indentações. Geralmente os editores possuem uma opção^b que permite alterar o tamanho da tabulação. Alguns editores possuem ainda uma opção extra que converte as tabulações automaticamente em espaços, e nós consideramos uma boa ideia habilitar essa opção, caso ela exista. Nos exemplos desse livro as tabulações possuem o tamanho igual a três espaços.

^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
```



3.3.1 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 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 uma resposta “rápida” para o problema do til. Acrescente os seguintes comandos que aparecem na segunda listagem.

Primeira listagem : codificação padrão (a que nós usamos).

```

1
2 PROCEDURE Main
3
4     ? "João, que horas são ?"
5
6 RETURN
```

.:Resultado:.



Segunda listagem : problema resolvido.

```

1 REQUEST HB_CODEPAGE_PT850
2 PROCEDURE Main
3
4     hb_cdpSelect( "PT850" )
5     ? "João, que horas são ?"
6
7 RETURN

```

.:Resultado:.

```

1 João, que horas são ?

```

Nós evitamos o uso do til nas nossas listagens iniciais. Preferimos esse pequeno inconveniente a ter que incluir símbolos sem a devida e clara explicação.

A dica é : prefira a primeira listagem pois lá você saberá exatamente a razão de tudo o que está lá. 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 ⁴, 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’ mas e gostou do preço”. Nesse caso, devemos

⁴O Harbour também utiliza colchetes para delimitar *strings*, mas esse recurso é bem menos usado. Prefira delimitar *string* com aspas (simples ou duplas).

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

```

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 :

```
PROCEDURE Main
```

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

```
RETURN
```

.:Resultado:.

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

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

```

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 “+”⁵ 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
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 "
```

⁵O termo técnico para esse sinal é “operador” (veremos o que é isso mais adiante)

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" // Irá exibir 22 e não 4.
4
5 RETURN
```

.:Resultado:.

22

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

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

```
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
```

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 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.
```

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 : /* Início /* Início2 */ 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* : “Tecla 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.

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 “constante de pré-processador”. Ele recebe esse nome porque as constantes recebem um nome especial 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.

```

1 #define VOLUME 10000
2 PROCEDURE Main
3
4     ? VOLUME // Vai imprimir 10000
5

```

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

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 **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 10

Use as constantes de pré-processador para evitar números mágicos. Esse recurso não necessariamente se aplica somente a números. Posso definir strings também.

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.

Dica 11

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.

4.2 Variáveis

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

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

4.2.1 Criação de variáveis de memória

Uma variável deve ser definida antes de ser usada. O Harbour permite que uma variável seja definida (o jargão técnico é “declarada”) de várias formas, a primeira delas é simplesmente criar um nome válido e definir a ele um valor, conforme a listagem 4.1. Note que a variável fica no lado esquerdo e o seu valor fica no lado direito. Entre a variável e o seu valor existe um sinal (:=) que representa uma atribuição de valor. Mais adiante veremos as outras formas de atribuição, mas habitue-se a usar esse sinal (o nome técnico é “operador”) pois ele confere ao seu código uma clareza maior.

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

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 ítem 1.
3. Não utilize letras acentuadas ou espaços para compor o nome da sua variável.

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

Listing 4.2: Nomes válidos para variáveis

```

1  /*
2  Nomes válidos
3  */
4  PROCEDURE Main
5
6
7     n0pc := 100
8     _iK := 200 // Válido, porém não aconselhado (Começa com
9                underline).
10    nTotal32 := n0pc + _iK
11    ? "n0pc = " , n0pc
12    ? "_iK = " , _iK
13    ? "nTotal32 = " , nTotal32
14 RETURN

```

.:Resultado:.

```

n0pc =          100
_iK =           200
nTotal32 =      300

```

Dica 12

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”.

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

Dica 13

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 14

(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 minúsculas ².

Listing 4.5: Notações

```

1  /*
2   Notações utilizadas
3  */
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 character (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**

²Essa notação é conhecida como Notação Hungara.

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

bom usar o bom senso : se você vai criar um código pequeno para exemplificar uma situação, você pode abdicar dessa prática.

Dica 15

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    */
13    nElem := 1 // Elemento corrente
14    nTotElem := 1200 // Total de elementos
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 16

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 17

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.

“Você só poderá entender completamente esse conceito no final do livro

Dica 18

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     // 0 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     // 0 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     // 0 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 hbmk.hbm ? Ele foi citado no início do livro durante o aprendizado do processo de compilação (capítulo 2). Você pode acrescentar nesse arquivo a opção -w3. Essa opção eleva para três (o nível máximo) de “warnings” (avisos de cuidado) durante o processo de compilação. No nosso diretório de prática nós temos um arquivo hbmk.hbm já com a opção -w3, por isso se você compilar o terceiro exemplo dessa dica, então ele irá exibir um aviso, mas **não deixará de gerar o programa**.

As mensagens geradas durante a compilação do nosso exemplo acima são :

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

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

4.4 Recebendo dados do usuário

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

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
```

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

```

9      INPUT "Informe o primeiro valor : " TO nVal1
10     INPUT "Informe o segundo valor : " TO nVal2
11
12     ? "A soma dos dois valores é : ", nVal1 + nVal2
13
14
15 RETURN

```

Após digitar o número tecla *ENTER*.

.:Resultado:.

```

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

```

Dica 19

Note que, na listagem 4.10 as variáveis foram declaradas em uma linha separada, enquanto que na listagem 4.9 elas foram declaradas em uma única linha. Não existe uma regra fixa com relação a isso, mas aconselhamos você a organizar as variáveis de acordo com o comentário (//) que provavelmente você fará após a declaração. Por exemplo, se existem duas variáveis que representam coisas que podem ser comentadas em conjunto, então elas devem ser declaradas em apenas uma linha. A listagem 4.10 ficaria mais clara se as duas linhas de declaração fossem aglutinadas em apenas uma linha. Isso porque as duas variáveis (*nVal1* e *nVal2*) possuem uma forte relação entre si. O ideal seria :

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

ou ainda recorrendo a atribuição simultânea de valor :

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

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

Se os dados digitados forem do tipo caractere você deve usar o comando *ACCEPT*. (Veja um exemplo na listagem 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

```

Após digitar a *string* tecla *ENTER*.

.:Resultado:.

```

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

```

Dica 20

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

4.5.1 Realizando as quatro operações

O exemplo da listagem 4.12 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 /*
3 As quatro operações
4 */
5 PROCEDURE Main
6 LOCAL nValor1, nValor2 // Valores a serem calculados
7
8 // Recebendo os dados
9 ? "Introduza dois números para que eu realize as quatro oper.: "
10 INPUT "Introduza o primeiro valor : " TO nValor1
11 INPUT "Introduza o segundo valor : " TO nValor2
12
13 // Calculando e exibindo
```

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

```

14 ? "Adição..... : " , nValor1 + nValor2
15 ? "Subtração..... : " , nValor1 - nValor2
16 ? "Multiplicação.... : " , nValor1 * nValor2
17 ? "Divisão..... : " , nValor1 / nValor2
18
19 RETURN

```

.: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: Descobrimdo o antecessor e o sucessor

```

1
2 /*
3  Descubre o antecessor e o sucessor
4  */
5 PROCEDURE Main
6 LOCAL nValor // Número a ser inserido
7
8 // Recebendo os dados
9 ? ""
10 ? "**** Descobrimdo 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

```

.:Resultado:.

```

**** Descobrimo o antecessor e o sucessor ****

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

```

4.6 Exercícios de fixação

1. Escreva um programa que declare 3 variáveis e atribua os valores 10,20 e 30 a elas; depois mais 3 variáveis e atribua a elas as letras v,l e j; finalmente imprima na tela :

.:Resultado:.

```

As variáveis inteiras contem os números 10, 20 e 30.
As variáveis caracteres contem os valores v, l e j.

```

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 4.14: 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 4.15: 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.

Referências Bibliográficas

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

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.

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

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

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

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



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. Escreva um programa que declare 3 variáveis e atribua os valores 10, 20 e 30 a elas; depois mais 3 variáveis e atribua a elas as letras v, l e j; finalmente imprima na tela :

.:Resultado:.

```
As variáveis inteiras contem os números 10, 20 e 30.
As variáveis caracteres contem os valores v, l e j.
```

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
```

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

.:Resultado:.

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

Comentário : nesse exemplo você deve ter recebido o aviso de warning :

.:Resultado:.

```
r0402.prg(16) Warning W0032 Variable 'CENTRADA' is assigned
but not used in function 'MAIN(11)'
```

Esse aviso, nós já vimos, é um “warning”. Ele não impedirá o seu programa de ser gerado, mas avisa que algo possivelmente está errado. Você consegue identificar o “possível erro”? Pense um pouco a respeito. A resposta está nessa nota de 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
```

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.
```

Listing A.3: Resposta

```
1 /*
2  Entrada : 3 variáveis numéricas e 3 variáveis caracteres
```

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.

```

3 Saída : As variáveis mostradas
4 */
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

```

4. 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

```

5. 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

```

6. 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 RETURN

```

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

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

```

8. 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

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.

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

```

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.

Listing A.10: Resposta

1 /*

```

2  Descr. : Recebe o valor de um depósito e o valor da taxa de
        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

```

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

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

Listing A.11: Resposta

```

1  /*
2  Descr. : Calcula a área de um triângulo sabendo que
3          Area = ( Base * Altura ) / 2
4  Entrada: Base e altura.
5  Saída : O valor da área de um triângulo.
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

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 $PI = 3.1415$) .

Listing A.12: 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

```

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

Listing A.13: 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

```

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.

Listing A.14: 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

```

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

Listing A.15: 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

```

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

Listing A.16: 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

```

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 “*”.

Listing A.17: 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 " ,
16         nMeses , " meses de vida"
17 RETURN

```

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

```

Listing A.18: 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

```

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.

Listing A.19: Resposta

```

1  /*
2  Entrada: Quilometragem inicial e quilometragem final, e
   quantidade de combustivel
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 Desafios

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

Listing A.20: 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.21: 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.22: 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
15         nQtd25Cent
16
17     nValor := ( nQtd25Cent * FATOR_25 ) + ( nQtd10Cent * FATOR_10 )
18             + ( nQtd1Cent * FATOR_01 )
19
20     ? "Você tem R$" , nValor, "em moedas"
21
22 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.23: 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 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.