
**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.
125p.;il. color. enc.

**VLADEMIRO LANDIM JUNIOR**

# INTRODUÇÃO A PROGRAMAÇÃO USANDO A LINGUAGEM HARBOUR.



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):

1. Paulo César Toledo
2. José Quintas
3. “Maligno”
4. Fladimir
5. Rochinha
6. “Asimoes”
7. Claudio Soto
8. Vailton
9. Pablo Cesar
10. janio
11. Stanis Luksys
12. Jairo Maia
13. Itamar M. Lins Jr.
14. RobertoLinux

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

## LISTA DE TABELAS

## SUMÁRIO

# 1 INTRODUÇÃO

## 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. Não se engane, programar pode ser 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 Material de Apoio

Antes de apresentar o material de apoio desse livro, vamos abordar um pouco o método que ele utiliza. Esse livro baseia-se em uma simples ideia : programar é uma atividade que demanda muita prática, portanto, você é 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 com-

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

pletar a listagem. Cada fragmento que falta no código diz respeito ao assunto que está sendo tratado no momento.

## 1.3 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 interpretado[2] 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.

[2]Nota 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 gráficas, biblioteca de jogos, desenvolvimento web, funções de rede, programação concorrente, etc.

10. **Multi-banco** : Apesar de possuir o seu próprio banco de dados, o Harbour pode se comunicar com os principais bancos de dados relacionais do mercado (SQLite, MySQL, PostgreSQL, Oracle, Firebird, MSAccess, etc.). Além disso, todo o aprendizado obtido com o banco de dados do Harbour pode ser usado no bancos relacionais. Um objeto de banco de dados pode ser manipulado diretamente através de RDDs, tornando a linguagem mais clara. Você escolhe a melhor forma de conexão.

11. **Programação Windows** : Apesar de criticado por muitos, o Microsoft Windows repousa veladamente nos notebooks de muitos mestres e doutores em Ciência da Computação. A Microsoft está sentindo a forte concorrência dos smartphones e da web, mas ela ainda domina o mundo desktop. Com Harbour você tem total facilidade para o desenvolvimento de um software para windows. Além das libs gráficas para criação de aplicações o Harbour possui acesso a dlls, fontes ODBC, ADO e outros componentes (OLE e ActiveX). Automatizar uma planilha em Excel, comunicar-se com o outros aplicativos que possuem suporte a windows é uma tarefa simples de ser realizada com o Harbour.

Existem pontos negativos ? Claro que sim. A falta de documentação atualizada e a baixa base instalada, 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 e C#. Se você quer desenvolver para a Web rapidamente existem inúmeras linguagens que foram desenvolvidas especialmente para esse fim. 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 "manja" de outros softwares de produtividade, como uma planilha eletrônica 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** que quer desenvolver ferramentas de segurança (e outras ferramentas) 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.

# 2 MEU PRIMEIRO PROGRAMA EM HARBOUR

## 2.1 Um programa simples em Harbour

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.

No caso do Harbour, existe uma função em que são colocadas todas as instruções que devem ser executadas inicialmente. Essa função[1] chama-se Main(), e todo bloco a executar fica entre o seu cabeçalho e o comando Return.

**Dica 1**

Se você for do tipo que testa tudo (o ideal para quem quer realmente aprender programação), você deve ter percebido que o programa é gerado mesmo sem a função Main(). Mesmo assim você deve se habituar a criar a função ou procedimento Main(), isso irá garantir que o programa iniciará por esse ponto. Caso contrário, você correrá o risco de não saber por onde o seu programa começou caso ele fique com muitos arquivos. Portanto : crie sempre uma função ou procedimento Main() nos seus programas, mesmo que eles tenham apenas poucas linhas, isso irá ajudar a criar um bom hábito de programação e lhe poupará dores de cabeça futuras.

Vamos então escrever o primeiro programa em Harbour.

Listing 2.1: O primeiro programa

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

O programa escrito em 2.1 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 função Main().

Geralmente uma função, (por exemplo: **Main**) é seguida com parênteses - **Main()** .Quando os parênteses não tem mais nada entre eles (como é o caso dos exemplos) isso significa que eles não recebem qualquer informação do mundo exterior. Se a função ou procedimento não tiverem informações a serem recebidas os parênteses não são obrigatórios, todavia, é incomum (a não ser em códigos antigos) encontrarmos funções ou procedimentos sem parênteses. Existe também uma forma antiga (derivada do dBase II) que permite que informações sejam passadas sem o uso de parênteses, nós iremos ver essa forma nos próximos capítulos, mas não daremos ênfase a essa forma antiga de se programar.

[1] Note que nos exemplos a palavra reservada FUNCTION foi trocada por PROCEDURE. Nos capítulos seguintes nós veremos a diferença entre esses dois blocos de instruções.

**Dica 2**

Habitue-se a colocar parênteses em funções ou procedimentos, mesmo que eles não recebam valores externos.

É bom ressaltar que Harbour é **Case Insensitive**, isto é, ele **não faz** diferenciação entre maiúsculas e minúsculas, sendo portanto a mesma coisa escrever main(), Main(), mAin() ou MAIN().

## 2.2 Exibição de dados

O exemplo a seguir (listagem 2.2) exibe a frase "Hello World" na tela. O símbolo "?" é um comando que avalia o conteúdo da expressão especificada e o exibe.

Listing 2.2: Hello World

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

**Dica 3**

Note que o código da listagem 2.2 possui parte a parte central "recuada" em relação ao início (*PROCEDURE Main()*) e o fim do bloco (*RETURN*). Esse recuo chama-se "indentação" e cada marca de tabulação (obtida com a tecla Tab) representa um nível de indentação. Essa pratica não é nenhuma regra obrigatória, mas ajuda a manter o seu código legível. Tome cuidado com o editor de texto que você utiliza para programar, pois os espaços entre as tabulações podem variar de editor para editor. Procure sempre configurar o tamanho desse espaço em três.

A seguir (código 2.3) 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.

Listing 2.3: O comando ??

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

Usaremos esses dois comandos para exibir os dados durante esse documento 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 2.4 (linha 7).

Listing 2.4: O comando ? e ??

```
1  /*
2  Comandos ? e  ??
3  */
4  PROCEDURE MAIN()
5
6
7     ? "A versão do Harbour que eu utilizo é a ", 3.2, " !!"
8     ? "Mais na frente resolveremos os problemas com"
9     ?? " a acentuação quando o arquivo estiver "
10    ?? " usando a codificação utf8."
11
12
13 RETURN
```

### 2.2.1 Uma pequena pausa para a acentuação correta

Se você compilou e executou o programa da listagem 2.4 (o que aconselhamos) você deve ter notado que a acentuação não apareceu corretamente. Isso acontece porque a codificação do arquivo utilizado segue o padrão UTF8[2] e o Harbour não suporta diretamente esse padrão[3]. Veja o mesmo código com suporte a UTF8 na listagem 2.5.

Listing 2.5: O comando ? e ??

```
1  /*
2  Comandos ? e  ??
3  Codificação UTF8
4  */
5  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
6
7  PROCEDURE MAIN()
8
9     hb_cdpSelect( "UTF8EX" )  // Seleciona o suporte a UTF8
10
11    ? "A versão do Harbour que eu utilizo é a ", 3.2, " !!"
12    ? "Agora sim, os acentos apareceram!"
13
14
15 RETURN
```

[2]UTF8 é um padrão de codificação que permite a representação de qualquer caractere universal, como por exemplo o árabe e o chinês. Essa mudança se deveu por causa da globalização dos sistemas de informação e a necessidade de inclusão desses caracteres.

[3]A linguagem Harbour assumiu o compromisso de manter a compatibilidade com antigos programas desenvolvidos em Clipper. Naquela época o padrão de codificação UTF8 ainda não tinha sido criado e o padrão vigente era o ASCII, por isso você precisa ativar o suporte na sua aplicação a UTF8.

É importante ressaltar que o seu editor de texto deve ter suporte a UTF8 (todos os modernos editores possuem esse suporte) . Além disso o suporte a UTF8 deve estar selecionado como o ativo para que os caracteres sejam escritos nesse padrão.

**Dica 4**

Habitue-se a digitar os seus códigos utilizando um editor com suporte a UTF8. Atualmente todas as aplicações estão utilizando esse padrão, inclusive os modernos banco de dados. Se a sua aplicação não tiver suporte a UTF8 ela poderá ter problemas com acentuação quando for se conectar com aplicativos de terceiros, como banco de dados por exemplo. Você precisará de um editor de texto com suporte a UTF8 e deve selecionar esse modo no seu editor.

## 2.3 Uma pequena introdução as Funções

O símbolo *hb_cdpSelect*, que você viu na listagem 2.5, possui um significado especial para o Harbour. Não iremos nos deter muito na explicação dele no momento, mas é interessante você aprender um pouco sobre ele para que os próximos exercícios sejam mais interessantes. Esse símbolo chama-se *função*.

Iremos, nos capítulos seguintes, estudar as funções em detalhes, por enquanto é necessário que você entenda que uma função é um símbolo que realiza uma tarefa particular. Por exemplo : a função *hb_cdpSelect*, a grosso modo, realiza a tarefa de permitir que o programa possa exibir os acentos e outros símbolos corretamente. Nos próximos capítulos, você estará construindo as suas próprias funções, por ora nós iremos trabalhar com algumas funções simples disponibilizadas pelo Harbour.

A seguir temos alguns exemplos de funções, note que a presença de parênteses após o seu nome é obrigatória para que o Harbour saiba que são funções.

```
? DATE()  // Imprime o dia de hoje
? TIME()  //  Imprime a hora
```

A listagem 2.6 ilustra o uso de algumas funções. Alguns termos que aparecem na listagem serão vistos apenas no próximo capítulo.

Listing 2.6: Funções simples

```
1 ï»¿/*
2 Função
3 */
4 REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6 PROCEDURE MAIN()
```

```
7
8     hb_cdpSelect( "UTF8EX" )  // Seleciona o suporte a UTF8
9     ? "Desde a meia-note até agora transcorreram ", SECONDS(), "
          segundos."
10
11  RETURN
```

### 2.3.1 Os delimitadores de strings

Um conjunto de caracteres delimitados por aspas recebem 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 aspa dentro de um texto, como por exemplo : “Ivo viu a ‘uva”’. Nesse caso, devemos ter cuidado quando formos usar aspas dentro de strings. O seguinte código da listagem 2.7 está errado.

Listing 2.7: 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 2.8 está correto :

Listing 2.8: Uso correto de delimitadores de *string*

```
1  /*
2  Delimitando strings
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE Main()
7
8      hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
9
10     ? "A praia estava 'legal'."
11     ? 'A praia estava "legal".'
12     ? [Colchetes podem ser usados delimitador de strings!]
13     ? [Você deram um "nó em pingo d'água"]
14     ? "Mas mesmo assim evite o uso de colchetes."
```

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

```
15
16 RETURN
```

## 2.4 Comentários

Os comentários são notas ou observações que você coloca dentro do seu programa mas que são 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. "*" : Da mesma forma, só pode ser usado antes da linha iniciar. (Formato antigo derivado do Dbase II, evite esse formato. ).
3. && : Pode ser usado antes da linha iniciar ou após ela finalizar. (Formato antigo derivado do Dbase II, evite esse formato. ).
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 2.9: Uso correto de comentários

```
1
2  PROCEDURE MAIN()
3  LOCAL nVal // Útil para explicar a linha corrente.
4  // LOCAL cBox  <=== Essa linha TODA será ignorada.
5  * Essa linha também será ignorada.
6  && E essa também...
7
8      nVal := 1200
9      ? nVal * 2  // Útil para comentar após
10                    // o final da linha.
11
12 /*
13
14     Note que esse comentário
```

```
15      possui várias linhas.
16
17  */
18
19  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 não permite a existência de comentários dentro de comentários. Por exemplo : /* Inicio /* Inicio2 */ Fim */.

Listing 2.10: 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 :

```
x := a + b + c
```

é equivalente a :

```
x := a /* valor inicial */ + b /* valor bruto */ + c // Calculo de parcelas
```

**Dica 5**

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 ser até 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. Sempre comente as funções, os dados globais e as classes. No caso de funções e procedimentos um comentário simples na abertura do bloco de código já é suficiente na maioria dos casos.
3. 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).
4. Escolha nomes auto-explicativos para as suas variáveis, evite comentar variáveis com nomes esquisitos. Veremos mais a esse respeito adiante.
5. Não contradiga o código. Quando alterar o seu trabalho atualize também os seus comentários ou apague-os.

## 2.5 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 2.11: A linha não acabou

```
1  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
2
3  PROCEDURE MAIN()
4
5      hb_cdpSelect( "UTF8EX" )  // Seleciona o suporte a UTF8
6
7      ? "Essa linha está dividida aqui mas " + ;
8        "será impressa apenas em uma linha"
9
10 RETURN
```

Uma string não pode ser dividida sem ser finalizada no código. Veja exemplo acima. Observe que utilizamos um operador de "+" (veremos o que é isso mais adiante) para poder concatenar a string que foi dividida.

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

Listing 2.12: 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 :

```
a := 1; ?  a; ? a + 20

equivale a

a := 1
? a
? a + 20
```

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.

## 2.6 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 2.13: Erro 1

```
1  /*
2  * Aprendendo Harbour
```

```
3 */
4 PROCEDURE MAIN()
5
6     ? "Hello" "World"
7
8
9 RETURN
```

Listing 2.14: Erro 2

```
1 /*
2 * Aprendendo Harbour
3 /*
4 PROCEDURE MAIN()
5
6     ? "Hello World"
7
8
9 RETURN
```

Listing 2.15: Erro 3

```
1 /*
2 /* Aprendendo Harbour
3 */
4 PROCEDURE MAIN()
5
6     ? "Hello World"
7
8
9 RETURN
```

Listing 2.16: Erro 4

```
1 /*
2  Aprendendo Harbour
3 */
4 PROCEDURE MAIN()
5
6     ? Hello World
7
8
9 RETURN
```

Listing 2.17: Erro 5

```
1 /*
2   Aprendendo Harbour
3 */
```

```
4  PROCEDURE MAIN()
5
6      ?  "Hello ";
7          ?? "World"
8
9
10
11 RETURN
```

## 2.7 Desafio

Escreva um programa que coloque na tela a seguinte saída :

```
LOJAO ANFISA
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.

 I
```

## 2.8 Exercícios

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

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

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

2. Escreva um programa que apresente na tela :

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

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

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

# 3 VARIÁVEIS E TIPOS DE DADOS

## 3.1 Variáveis

Variáveis são pedaços de memória 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".

### 3.1.1 Criação de variáveis de memória

Uma variável deve ser definida antes de ser usada. O Harbour permite que uma variável seja definida de várias formas, a primeira delas é simplesmente criar um nome válido e definir a ele um valor, conforme a listagem 3.1. 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 operador (:=) que representa uma atribuição de valor. Mais adiante veremos as outras formas de atribuição, mas habitue-se a usar esse operador (Inspirado na linguagem Pascal) pois ele confere ao seu código uma clareza maior.

Listing 3.1: Criação de variáveis de memória

```
1 /*
2    Criação de variáveis de memória (forma 1)
3 */
4 PROCEDURE MAIN()
5
6    cNomeQueVoceEscolher := "Vlademiro Landim Junior"
7    ? cNomeQueVoceEscolher
8
9 RETURN
```

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.

Listing 3.2: Nomes válidos para variáveis

```
1 /*
2    Nomes válidos
```

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

```
3  */
4  PROCEDURE MAIN()
5
6     nOpc := 100
7     _iK := 200
8     nTotal32 := nOpc + _iK
9
10 RETURN
```

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 da linguagem, como TOTAL, USE, REPLACE, etc. A lista de todas as palavras reservadas encontra-se no Apêndice X, mas você não deve se preocupar em decorar todas as palavras reservadas. O que você precisa é seguir algumas regras simples para se nomear uma variável, conforme veremos a seguir.

**Dica 7**

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 [a] não reclama do uso de palavras reservadas, mas reforçamos que você **não** deve adotar essa prática.

Listing 3.3: Uso de palavras reservadas

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

[a] A versão que nós usamos para desenvolver esse documento foi o 3.2.0

**Dica 8**

(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 3.4 ) exemplifica uma atribuição de nomes inválidos à uma variável de memória.

Listing 3.4: Nomes inválidos para variáveis

```
1 /*
2   Nomes inválidos
3 */
4 PROCEDURE MAIN()
5
6    90pc := 100 // Inicializa com um número
7
8 RETURN
```

### 3.1.2 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 3.5: 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 caracter (Notação hungara)
```

[2] Essa notação é conhecida como 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.

**Dica 9**

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 3.6.

Listing 3.6: Nomes curtos e concisos para variáveis

```
1  /*
2    NOMES CONSISOS
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
          demais.
11    */
12    nElem := 1 // Elemento corrente
13    nTotElem := 1200 // Total de elementos
14
15 RETURN
```

Evite detalhar demais conforme a listagem 3.7.

Listing 3.7: Nomes longos e detalhados demais

```
1  /*
2    NOMES GRANDES DEMAIS PARA VARIÁVEIS. EVITE ISSO!!!
3
```

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

```
4    Adaptado de (KERNIGHAN, p.3, 2000)
5  */
6  PROCEDURE MAIN()
7
8     nNumeroDoElementoCorrente := 1
9     nNumeroTotalDeElementos := 1200
10
11 RETURN
```

### 3.1.3 Atribuição

Já vimos como atribuir dados a uma variável através do operador *:=* que foi inspirado na Linguagem Pascal. Agora iremos detalhar essa forma e ver outras formas de atribuição.

O código listado em 3.8 nos mostra a atribuição com o operador ‘‘=’’ e com o comando STORE. Conforme já dissemos, prefira o operador *:=*.

Listing 3.8: Outras forma de atribuição

```
1  /*
2  Atribuição
3  */
4  PROCEDURE MAIN()
5
6     x = 1200  // Atribuí 1200 ao valor x
7     STORE 100 TO y // Atribuí 100 ao valor y
8
9     ? x + y
10
11 RETURN
```

A atribuição de variáveis também pode ser feita de forma simultânea, conforme a listagem 3.9.

Listing 3.9: Atribuição simultânea

```
1  /*
2  Atribuição
3  */
4  PROCEDURE MAIN()
5  LOCAL x := y := z := k := 100
6
7     ? x + y + z + k
8
9
10
```

```
11 RETURN
```

**Dica 10**

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 um tipo de variável chamada de *LOCAL*. Você não precisa se preocupar em entender o que esse tipo 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 3.9). 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.
```

**Dica 11**

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.

## 3.2 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 3.10.

Listing 3.10: Recebendo dados externos digitados pelo usuário

```
1  /*
2  Uso do comando INPUT
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5  PROCEDURE MAIN()
6  LOCAL nVal1 := 0 // Declara a variável parcela de pagamento
7  LOCAL nVal2 := 0 // Declara a variável parcela de pagamento
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    INPUT "Informe o primeiro valor : " TO nVal1
12    INPUT "Informa o segundo valor : " TO nVal2
13
14    ? "A soma dos dois valores é : ", nVal1 + nVal2
15
16
17 RETURN
```

Após digitar o número tecle *ENTER*.

**Dica 12**

Note que, na listagem 3.10 as variáveis foram declaradas em uma linha separada, enquanto que na listagem 3.9 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 3.10 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 o seu tipo numérico (no caso do exemplo acima, o zero informa que as variáveis devem ser tratadas

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

como numéricas).

Se os dados digitados forem do tipo caractere você deve usar o comando *ACCEPT*. (Veja um exemplo na listagem 3.11). Observe que o *ACCEPT* funciona da mesma forma que o *INPUT*.

Listing 3.11: Recebendo dados externos digitados pelo usuário (Tipo caractere)

```
1  /*
2  Uso do comando ACCEPT
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5  PROCEDURE MAIN()
6  LOCAL cNome := "" // Seu nome
7
8     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
9
10    /* Pede e exibe o nome do usuário */
11    ACCEPT "Informe o seu nome : " TO cNome
12    ? "O seu nome é : ", cNome
13
14
15 RETURN
```

Após digitar a *string* tecle *ENTER*.

**Dica 13**

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

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

Se mesmo assim você quer saber o porque dessa diferença continue lendo, senão pode pular o trecho a seguir e ir para a seção de exemplos.
Na primeira metade da década de 1980 o dBase II foi lançado. Naquela época não tínhamos os recursos de hardware e de software que temos hoje, tudo era muito simples. Os primeiros comandos de entrada de dados usados pelo dBase foram o *INPUT* e o *ACCEPT*. O *ACCEPT* serve para receber apenas variáveis do tipo caracter. Você não precisa nem declarar a variável, pois se ela não existir o comando irá criá-la para você. O *INPUT*

funciona da mesma forma, mas serve também para receber dados numéricos, caracteres, data e lógico. Porém você deve formatar a entrada de dados convenientemente, e é isso torna o seu uso confuso para outros tipos de dados que não sejam os numéricos.
Exemplos de uso do comando INPUT :

```
Usando com variáveis numéricas :

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

Usando com variáveis caracteres :

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

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

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

## 3.3 Exemplos

### 3.3.1 Realizando as quatro operações

O exemplo da listagem 3.12 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 3.12: As quatro operações

```
1 /*
2 As quatro operações
3 */
4 REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6 PROCEDURE MAIN()
7 LOCAL nValor1 := nValor2 := 0 // Valores a serem calculados
```

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

```
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebendo os dados
12    ? "Introduza dois números para que eu realize as quatro oper.: "
13    INPUT "Introduza o primeiro valor : " TO nValor1
14    INPUT "Introduza o segundo valor : " TO nValor2
15
16    // Calculando e exibindo
17    ? "Adição.......... : " , nValor1 + nValor2
18    ? "Subtração....... : " , nValor1 - nValor2
19    ? "Multiplicação.... : " , nValor1 * nValor2
20    ? "Divisão......... : " , nValor1 / nValor2
21
22
23  RETURN
```

### 3.3.2 Calculando o antecessor e o sucessor de um número

O exemplo da listagem 3.12 gera um pequeno programa que descobre qual é o antecessor e o sucessor de um número qualquer inserido pelo usuário.

Listing 3.13: Descobrindo o antecessor e o sucessor

```
1  /*
2  Descobre o  antecessor e o sucessor
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nValor := 0 // Número a ser inserido
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebendo os dados
12    ? ""
13    ? "**** Descobrindo o antecessor e o sucessor ****"
14    ? ""
15    INPUT "Introduza o número : " TO nValor
16
17    // Calculando e exibindo
18    ? "Antecessor........ : " , nValor - 1
19    ? "Sucessor.......... : " , nValor + 1
20
21 RETURN
```

## 3.4 Exercícios

1. (MIZRAHI, 1990, p. 25) - Escreva um programa que declare 3 variáveis numéricas e atribua os valores 1,2 e 3 a elas; 3 variáveis caracteres e atribua a elas as letras a,b e c; finalmente imprima na tela :

   As variáveis inteiras contem os números 1, 2 e 3. As variáveis caracteres contem os valores a, b e c.

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 :

```
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 :

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

4. 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 :

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

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

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

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

## 3.5 Desafios

### 3.5.1 Identifique o erro de compilação no programa abaixo.

Listing 3.14: 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
```

### 3.5.2 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 3.15: 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
```

### 3.5.3 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 :

```
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 TIPOS DE DADOS : VALORES E OPERADORES

## 4.1 Definições iniciais

De acordo com Tenenbaum,

> um tipo de dado significa um conjunto de valores e uma sequencia de operações sobre esses valores. Este conjunto e essas operações formam uma construção matemática que pode ser implementada usando determinada estrutura de hardware e software (TENENBAUM; LANGSAM; AUGENSTEIN, 1995, p. 18)

.

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

De acordo com Damas,

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

.

Por exemplo, uma variável que recebe o valor 10 é uma variável do tipo numérica. Esse tipo só está plenamente implementado quando nós definimos para esse tipo os seus respectivos operadores. Alguns exemplos de operadores são “+”, “-”, “*” e “/” que nós já vimos em alguns códigos anteriores.

```
a := 2
b := 3

? a + b  // Exibe 5
? a - b   // Exibe -1
? a * b   // Exibe 6
? a / b   //  0.6666666666666666667
```

Da mesma forma, uma variável que recebe o valor “Aprovado com sucesso” é uma variável do tipo caractere. O tipo caractere possui alguns operadores em comum com o tipo

numérico, um deles é o “+” que assume o papel de “concatenar” strings, conforme o exemplo abaixo :

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

O conceito de tipo de dado e seus operadores, portanto, são as duas faces de uma mesma moeda. Uma vez definido uma variável como caractere, os tipos de operadores disponíveis são diferentes dos operadores disponíveis para uma variável data, por exemplo. O exemplo da listagem 4.1 serve para ilustrar o que acontece quando nós utilizamos operadores incompatíveis com o tipo de dado definido.

Listing 4.1: Operadores e tipo de dado

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

## 4.2 Os tipos básicos de dados e seus operadores

### 4.2.1 O tipo caractere e seus operadores + e -

As variáveis do tipo caractere possuem operadores “+” e “-”. O operador de “+” é largamente usado, mas o operador de “-” é praticamente um desconhecido. Esses operadores não realizam as mesmas operações que eles realizariam se as variáveis fossem numéricas, afinal de contas, não podemos somar nem subtrair strings. Veremos na listagem 4.2 o uso de ambos os operadores. Note que usamos uma função chamada *LEN*, cujo propósito é informar a quantidade de caracteres de uma string.

Listing 4.2: Operadores de concatenação

```
ï»¿/*
Operadores de strings + e -

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

*/
REQUEST HB_CODEPAGE_UTF8EX

PROCEDURE MAIN()
LOCAL cPre , cPos AS CHARACTER
LOCAL cNome AS CHARACTER

   hb_cdpSelect( "UTF8EX" )
   cPre := " Harbour "
   cPos := " Project "

   ? "Exemplo 1 : Concatenando com +"
   ? cPre + cPos
   ?? "X" // Vai ser concatenado imediatamente após o término da
      linha acima
   ? "O tamanho da string acima é de " , Len( cPre + cPos ) //
      Exibe 18
   ?
   ? "Exemplo 2 : Concatenando com -"
   ? cPre - cPos
   ?? "X" // Vai ser concatenado imediatamente após o término da
      linha acima
   ? "O tamanho da string acima é de " , Len( cPre - cPos )  //
      Exibe 18


   cNome := " (www.harbour-project.org) "
   ?
   ? "Exemplo 3 : Concatenando três strings com +"
   ? cPre + cPos + cNome
   ?? "X" // Vai ser concatenado imediatamente após o término da
      linha acima
   ? "O tamanho da string acima é de " , Len( cPre + cPos + cNome)
       // Exibe 45
   ?
   ? "Exemplo 4 : Concatenando três strings com -"
   ? cPre - cPos - cNome
   ?? "X" // Vai ser concatenado imediatamente após o término da
      linha acima
   ? "O tamanho da string acima é de " , Len( cPre - cPos - cNome)
```

Tabela 4.1: Operadores matemáticos

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

```
        // Exibe 45
40
41  RETURN
```

Quando o "+" e o "-" são usados com strings, o termo correto é "concatenação" de strings, e não "soma" ou "subtração" de strings. Os operadores "+" e "-" realizam diferentes tipos de concatenações, o operador de "+" é largamente usado com strings, ele simplesmente une as duas strings e não altera os espaços em branco, mas o operador "-" possui uma característica que pode confundir o programador. De acordo com Vidal, "o operador menos realiza a concatenação sem brancos, pois ela remove o espaço em branco do início da string da direita, contudo os espaços em branco que precedem o operador são movidos para o final da cadeia de caracteres" (VIDAL, 1991, p. 91).

**Dica 14**

Evite concatenar strings com o operador menos, existem funções que podem realizar o seu intento de forma mais clara para o programador.

### 4.2.2 O tipo numérico e seus operadores matemáticos

Nós já vimos em códigos anteriores alguns operadores do tipo de dado numérico, a tabela 4.1 apresenta uma listagem completa dos operadores matemáticos para tipos numéricos.

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

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

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

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

Outro detalhe a ser levado em conta é a precedência dos operadores. Em outras palavras, basta se lembrar das suas aulas de matemática quando o professor dizia que tem que resolver primeiro a multiplicação para depois resolver a adição. A ordem dos operadores matemáticos é :

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

Listing 4.3: Precedência de operadores matemáticos

```
/*
Tipo de dado e operador
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL x, y, z

   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

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

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

RETURN
```

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

**Dica 15**

Você não precisa decorar a tabela de precedência dos operadores pois ela é muito grande. Ainda iremos ver outros operadores e a tabela final não se restringe a tabela 4.1. Se habitue a utilizar parênteses para ordenar os cálculos, mesmo que você não precise deles. Parênteses são úteis também para poder tornar a expressão mais clara :

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

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

### 4.2.3 O tipo data e seus operadores

As antigas linguagens que compõem o padrão xBase foram criadas para suprir a demanda do mercado por aplicativos comerciais, essa característica levou os seus desenvolvedores a criarem um tipo de dado básico para se trabalhar com datas (e dessa forma facilitar a geração de parcelas, vencimentos, etc.). O Harbour, como sucessor dessas linguagens, possui um tipo de dado primitivo que permite cálculos com datas. A forma usual de se criar uma variável data depende de uma função chamada **CTOD()** [1]. Veja um exemplo da criação de uma variável data no código 4.4.

Listing 4.4: Variável do tipo data

```
1  /*
2  Variável Data
3  */
4  PROCEDURE MAIN()
5
6
7     dvencimento := CTOD( "01/01/2021")
8     ? dvencimento
9
10 RETURN
```

Uma outra forma de se iniciar uma data (listagem 4.5) é através de um formato especial criado especialmente para esse fim. A sintaxe dele é :

[1] Abreviação da frase (em inglês) “Character to Date” (“Caracter para Data”)

```
0d20160811  equivale à 11 de Agosto de 2011

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

Listing 4.5: Iniciando datas

```
ï»¿

 PROCEDURE Main()
 LOCAL dPagamento


   dPagamento := 0d20161201 // Primeiro de dezembro de 2016
   ? dPagamento


 RETURN
```

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

Antes de prosseguirmos com o estudo das datas vamos fazer uma pequena parada para configurar o formato da data a ser exibida. Como a nossa data obedece ao padrão dia/mês/ano (o padrão britânico), nós iremos configurar a exibição conforme a listagem abaixo usando *SET DATE BRITISH* . Se você quiser exibir o ano com quatro dígitos use *SET CENTURY ON* . Você só precisa usar esses dois comandos uma vez no início do seu programa, geralmente no inicio da função **Main()**. Note que na listagem 4.6 as datas após a linha já obedecem ao padrão configurado (formato britânico e exibição de ano com quatro dígitos ativado).

Listing 4.6: Configurações de data

```
/*
Variável Data exibida corretamente
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL dVenc // Data de vencimento

```

```
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10    dVenc := CTOD( "12/31/2021") // 31 de dezembro de 2021
          (mês/dia/ano)
11
12    ? "Exibindo a data 31 de dezembro de 2021."
13    ? "O formato padrão é o americano (mês/dia/ano)"
14    ? dVenc
15    SET DATE BRITISH // Exibe as datas no formato dia/mês/ano
16    ? "Exibe as datas no formato dia/mês/ano"
17    ? "O vencimento da última parcela é em " , dVenc
18
19    SET CENTURY ON // Ativa a exibição do ano com 4 dígitos
20    ? "Ativa a exibição do ano com 4 dígitos"
21    ? "A mesma data acima com o ano com 4 dígitos : ", dVenc
22
23    ? "Outros exemplos com outras datas : "
24    ? CTOD("26/08/1970")
25    dVenc := CTOD( "26/09/1996")
26    ? dVenc
27    dVenc := CTOD( "15/05/1999")
28    ? dVenc
29
30    SET CENTURY OFF // Data volta a ter o ano com 2 dígitos
31    ? "Data volta a ter o ano com 2 dígitos"
32    ? dVenc
33
34
35 RETURN
```

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

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

Quando subtraímos duas datas o valor resultante é o número de dias entre elas. Veja o exemplo ilustrado na listagem 4.7.

Listing 4.7: Operadores de data

```
1 /*
```

```
2  Variável Data exibida corretamente
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL dVenc // Data de vencimento
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    SET DATE BRITISH
12    dCompra := CTOD( "01/02/2015" ) // Data da compra
13    dVenc := CTOD( "05/02/2015" ) // Data do vencimento
14
15    ? "Data menor (compra) :", dCompra
16    ? "Data maior (vencimento) :" , dVenc
17    ? "Maior menos a menor (dias entre)"
18    ? dVenc - dCompra  // Resulta em 4
19    ? "Menor menos a maior (dias entre)"
20    ? dCompra - dVenc  // Resulta em -4
21    ? "Subtrai dois dias"
22    ? dVenc - 2 // Resulta em 03/02/15
23    ? "Soma dois dias (data + número)"
24    ? dVenc + 2 // Resulta em 07/02/15
25    ? "Soma dois dias (número + data )  "
26    ? 2 + dVenc // Resulta em 07/02/15 (a mesma coisa)
27
28
29 RETURN
```

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

**Dica 16**

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

```
PROCEDURE Main()

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

RETURN
```

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

```
PROCEDURE Main()

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

RETURN
```

### 4.2.4 Variáveis do tipo Lógico e os operadores relacionais e lógicos

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

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

**Dica 17**

Procure criar as variáveis lógicas usando o prefixo l (de lógica). Adicione também o prefixo “Eh” ou “Estah” (O “H” simula o acento). Esse prefixo adicional simula uma resposta a uma pergunta e torna o código mais claro. Por exemplo: se você deseja criar uma variável para determinar se a matricula do aluno está ativa crie a seguinte variável : *lEhAtiva* ou *lEstahAtiva*. Se você está usando o idioma inglês (que recomendamos, por ter nomes menores e por causa da globalização) para nomear suas variáveis troque o *Eh* por *Is* e crie com esse nome : *lIsActive*[a]. Essa dica, retirada de (KERNIGHAN; PIKE, 2000, p. 5) tem o seu sentido quando você for aprender as estruturas de seleção no próximo capítulo seguinte. Por ora, veja que o exemplo abaixo torna a leitura mais fácil :

```
IF  lIsActive  // Leia assim: ‘‘Se está ativo então’’
     // Execute operation   ‘‘execute a operação’’
ENDIF
```

Tabela 4.2: Operadores relacionais

| Operação | Operador |
|---|---|
| Menor que | < |
| Maior que | > |
| Igual a | = |
| Duplo igual a | == |
| Diferente de | <> |
| Diferente de | # |
| Diferente de | != |
| Menor ou igual a | <= |
| Maior ou igual a | >= |
| Comparação de strings (Está contido) | $ |

```
IF lIsOk  // ‘‘Se está Ok então’’
     // Execute operation   ‘‘execute a operação’’
ENDIF
```

Tenho minhas dúvidas se as formas no nosso idioma “lEhAtivo” ou “lEstahAtivo” e equivalentes tornam melhor a leitura do código. Mas resolvi colocar pois traduzem a dica de Kernighan para o português.

[a]Procure se familiarizar ao máximo com o idioma inglês, além de facilitar na pesquisa por informações ele também pode ser usado para nomear as suas variáveis. Em alguns casos é essencial, caso o seu código venha a ser compartilhado com pessoas de outras nacionalidades.

#### 4.2.4.1 Operadores relacionais

Os operadores relacionais são binários e geram resultados lógicos (verdadeiro ou falso) a partir do relacionanento (comparação) de duas expressões (VIDAL, 1991, p. 93). Os operadores relacionais do Harbour estão listados na tabela 4.2.

A expressão da direita é comparada com a da esquerda, resultando um valor verdadeiro (.t.) ou falso (.f.). Você só pode comparar valores que sejam do mesmo tipo de dado. Uma relação do tipo 2 > “Ana” irá resultar em um erro pois não faz sentido. Esses operadores estão ilustrados na listagem 4.8.

Listing 4.8: Operadores relacionais

```
1 /*
2 Operadores relacionais
3 */
4 REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6 PROCEDURE MAIN()
```

```
7  LOCAL x, y // Valores numéricos
8  LOCAL dData1, dData2 // Datas
9  LOCAL cString1, cString2 // Caracteres
10
11    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
12    SET DATE BRIT // Data dd/mm/aa
13    SET CENTURY ON // Ano com 4 dígitos
14
15    ? "Comparando variáveis numéricas"
16    ?
17    x := 2
18    y := 3
19    ? 'x := 2'
20    ? 'y := 3'
21    ? "x > y : " , x > y // Retorna falso (.f.)
22    ? "y = 3 : " , y = 3 // Retorna verdadeiro (.t.)
23    ? "y <> x : ", y <> x // Retorna verdadeiro (.t.)
24    ?
25    ? "Comparação de datas"
26    ?
27    dData1 := CTOD("01/09/2015")
28    dData2 := CTOD("02/09/2015")
29    ? 'dData1 := CTOD("01/09/2015")'
30    ? 'dData2 := CTOD("02/09/2015")'
31    ? "dData1 >= dData2 : ", dData1 >= dData2 // Retorna falso (.f.)
32    ? "( dData1 + 1 ) = dData2 : ", ( dData1 + 1 ) = dData2 //
         Retorna verdadeiro (.t.)
33    ?
34    ? "Comparação de strings"
35    ?
36    cString1 := "Vlademiro"
37    cString2 := "Vlad"
38
39    ? 'cString1 := "Vlademiro"'
40    ? 'cString2 := "Vlad"'
41
42    ?  " cString1 == cString2 : " , cString1 == cString2 // .f.
43
44    /* Cuidado !! */
45    /* Leia com atenção o tópico sobre comparação de strings */
46    ?  "Cuidado !! com a comparação abaixo. Elas são confusas"
47    ?  " cString1 = cString2 : " , cString1 = cString2  // .t.
48    ?
49
50 RETURN
```

Tabela 4.3: Os operadores lógicos

| Operação | Operador |
|---|---|
| Fornece .T. se ambos os operandos forem verdadeiros | .AND. |
| Fornece .T. se um dos operandos forem verdadeiros | .OR. |
| Fornece verdadeiro se o seu operando for falso | .NOT. |
| Outra notação para o operador .NOT. | ! |

**Dica 18**

O operador "=" é um operador "sobrecarregado". Isso significa que ele está sendo usado para duas coisas diferentes. Ele tanto serve para comparar quanto serve para atribuir. Por isso nós aconselhamos que você evite o seu uso.
Para comparar use o operador "==" (Duplo igual) Para atribuir use o operador ":=" (Atribuição)

Outro operador que merece destaque é o operador "$" (Está contido). Ele só funciona com variáveis do tipo caractere e serve para informar se uma cadeia de string está contida na outra.

```
// Leia assim : ‘‘Natal’’ está contido em ‘‘Feliz Natal’’ ?
? ‘‘Natal’’ $ ‘‘Feliz Natal’’  // Retorna .t.

? ‘‘natal’’ $ ‘‘Feliz Natal’’  // Retorna .f. (‘‘Natal’’ é diferente de ‘‘natal’
```

O conteúdo de strings é comparado levando-se em consideração as diferenças entre maiúsculas e minúsculas ("Natal" é diferente de "natal").

#### 4.2.4.2 Operadores lógicos

Os operadores lógicos atuam nas variáveis lógicas, nas comparações (vistas na seção anterior) e na negação de expressões. Os operadores estão descritos na tabela 4.3.

Os operadores lógicos permitem concatenar várias comparações efetuadas entre operadores relacionais. Por exemplo :

```
a := 1
b := 2
c := 3
? b > a .AND. c > b // Verdadeiro
```

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

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

A avaliação de expressões com .OR. é diferente da avaliação de expressões de com .AND. porque no primeiro caso todos os casos tem que ser avaliados pois se um deles for verdadeiro a expressão toda retornará verdade. Por exemplo: se eu avalio "x == 2 .OR. y == 2", se x for 4 mas y for 2 então a expressão será verdadeira. Já no caso de .AND. se eu avalio "x == 2 .AND. y == 2" e x for 4 o programa não irá mais avaliar y pois seria uma perda de tempo. Esse tipo de avaliação é chamada de avaliação de curto circuito (SPENCE, 1991, p. 66).

#### 4.2.4.3 Comparação entre strings

A comparação entre strings é um aspecto confuso da linguagem Harbour, devido ao seu compromisso de otimização de buscas no seu banco de dados interno[2]. É de extrema importância que essa parte fique completamente entendida.

Por padrão temos o seguinte comportamento : **"Se a string da esquerda conter a string da direita então elas são iguais"**.

Abaixo temos duas strings que são consideradas iguais :

```
? ‘‘Vlademiro’’ = ‘‘Vlad’’  // A comparação retorna .t.
```

Agora, se nós preenchermos com espaços em branco a string da direita (de modo que elas duas tenham o mesmo tamanho), então elas são consideradas diferentes

```
? ‘‘Vlademiro’’ = ‘‘Vlad     ’’  // A comparação retorna .f.
```

Esse comportamento estranho tem a sua explicação em um das características da linguagem Harbour : a existência do seu próprio banco de dados. Segundo Spence : "na prática nós necessitamos de comparações não exatas. [...] Ao procurar os registros em uma base de

[2]O banco de dados internos do Harbour, os arquivos DBF não serão abordados nesse trabalho.

dados que comecem com S (digamos, procurando por Skorasczewski), queremos comparações não exatas" (SPENCE, 1991, p. 62).

Essa característica também vale para a negação da igualdade quando elas envolvem strings.

```
? ‘‘Vlademiro’’ != ‘‘Vlad’’       // A comparação retorna .f. (iguais)
? ‘‘Vlademiro’’ != ‘‘Vlad     ’’  // A comparação retorna .t. (diferentes)
```

**Dica 19**

Cuidado quando for comparar variáveis do tipo caractere. Com certeza você percebeu que as regras de igualdade não se aplicam a elas da mesma forma que se aplicam as outras variáveis. Quando for comparar strings use o operador "==" (Duplo igual). Ele evita as comparações esquisitas que você acabou de ver.

```
? ‘‘Vlademiro’’ == ‘‘Vlad’’  // (.f.)
? ‘‘Vlademiro’’ == ‘‘Vlad     ’’  //  (.f.)
? ‘‘Vlademiro’’ == ‘‘Vlademiro ‘‘ // (.f.) A string da direita
                                        //    tem um espaço a mais
? ‘‘Vlademiro’’ == ‘‘Vlademiro’’ // Só retorna verdadeiro nesse caso
```

Para verificar a diferença use o operador de negação combinado com o operador de duplo igual, conforme o exemplo a seguir :

```
? .NOT. ( ‘‘Vlademiro’’ == ‘‘Vlad’’ ) // Retorna .t. (diferentes)

ou

? !( ‘‘Vlademiro’’ == ‘‘Vlad’’ )
```

Deixe para usar o operador "=" quando você estiver buscando valores aproximados no banco de dados do Harbour [a].

**EM RESUMO :** Use o operador duplo igual ("==") para comparar strings entre si e a combinação dos operadores duplo igual e de negação para negar uma comparação. O ideal mesmo é você usar essa dica para todos os tipos de variáveis e não usar o operador "=".

[a] O banco de dados do Harbour não segue o padrão SQL. O Harbour pode acessar os bancos de dados relacionais baseados em SQL, mas não diretamente. Esses tópicos estão além do escopo desse livro.

### 4.2.5 Operadores de atribuição

Os operadores de atribuição são utilizados para atribuir um tipo de dado a uma variável. Nós já vimos o principal operador de atribuição ( := ) e aconselhamos você a utilizá-lo no lugar do operador normal ( = ). Esses operadores citados podem ser usados em qualquer tipo de dado e se a variável não tiver sido declara com *LOCAL* esses operadores tem o poder de criar a variável. Ainda não chegamos a estudar o significado de *LOCAL*, mas gostaríamos de reforçar que o seu uso é um ótimo hábito de programação. O código da listagem 4.9 ilustra essas formas já vistas de atribuição.

Listing 4.9: Formas simples de atribuição.

```
/*
Atribuição de dados a variáveis
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL x, y // Valores numéricos


   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
   ? "Operação convencional de atribuição"
   x := 10
   y := 20
   ? "x = ", x
   ? "y = ", y
   ?
   ? "Operação em linha de atribuição"
   x := y := 10
   ? "x = ", x
   ? "y = ", y
   ?
   y := x * 2
   x := "Agora x é caracter"
   ? "x = ", x
   ? "y = ", y
   ?


RETURN
```

Existe uma forma ainda não vista de atribuição. Observe atentamente o código 4.10.

Listing 4.10: Uma nova forma de atribuição bastante usada.

```
/*
Atribuição de dados a variáveis
```

```
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL x  // Valores numéricos
8
9
10    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
11    ? "Uma forma bastante usada de atribuição"
12    x := 10
13    ? "x vale ", x
14
15    x := x + 10
16    ? "agora vale ", x
17
18    x := x + 10
19    ? "agora vale ", x
20
21    x := x * 10
22    ? "agora vale ", x
23
24    x := x - 10
25    ? "agora vale ", x
26
27    x := x / 10
28    ? "agora vale ", x
29
30
31 RETURN
```

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

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

Deveríamos fazer :

```
x := 10
```

Tabela 4.4: Operadores de atribuição compostos

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

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

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

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

Equivale a

x := 10
x *= 2
```

A tabela 4.4, retirada de (VIDAL, 1991, p. 98), resume bem esses operadores e seu modo de funcionamento.

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

```
x := ‘‘Muito‘‘
x := x + ‘‘ obrigado’’

Equivale a

x := ‘‘Muito‘‘
x += ‘‘ obrigado’’
```

#### 4.2.5.1 Operadores de incremento e decremento

Se você entendeu a utilização dos operadores de atribuição compostos não será difícil para você entender os operadores de incremento e decremento. Eles funcionam da mesma

forma que os operadores de atribuição já vistos, mas eles apenas somam ou subtraem uma unidade da variável. O exemplo a seguir irá ilustrar o funcionamento desses operadores.

O operador de incremento funciona assim :

```
Essa operação

    x := 10
    x := x + 1

 Equivale a

    x := 10
    ++x

 e também equivale a

    x := 10
    x++
```

A mesma coisa vale para o operador de decremento.

```
Essa operação

    x := 10
    x := x - 1

 Equivale a

    x := 10
    --x

 e também equivale a

    x := 10
    x--
```

Existe uma sutil diferença entre ++x e x++ e também entre –x e x–. Utilizados como prefixo (–x ou ++x) esses operadores alteram o operando antes de efetuar a atribuição. Utilizados como sufixo (x– ou x++) o operando é alterado e só depois a atribuição é efetuada. A listagem 4.11 exemplifica o que foi dito :

Listing 4.11: Operadores de incremento e decremento.

```
/*
Incremento e decremento
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL nNum1 , nNum2  // Valores numéricos

   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

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

   nNum1 := 0
   nNum2 := nNum1++
   ? nNum1 // Vale 1
   ? nNum2 // Vale 0


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

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

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


```

```
46 RETURN
```

**Dica 20**

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

### 4.2.6 Operadores especiais

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

1. Função ou agrupamento () : Já vimos o uso de parênteses para agrupar expressões e para a escrita da função Main(). Ele também é usado como uma sintaxe alternativa para o operador & (ainda não visto).
2. Colchetes [] : Também já vimos o seu uso como um delimitador de *strings*, mas ele também é usado para especificar um elemento de uma matriz (veremos o que é matriz em um capítulo a parte).
3. Chaves : Definição literal de uma matriz ou de um bloco de código. Será visto nos próximos capítulos.
4. Identificador de Alias -> : Não será visto nesse documento por fugir do seu escopo.
5. Passagem de parâmetro por referência : Será visto no tópico especial sobre funções.
6. Macro & : Será visto mais adiante.

### 4.2.7 O estranho valor NIL

O Harbour possui um valor chamado NIL [3]. Quando você cria uma variável e não atribui valor a ela, ela tem um tipo de dado indefinido e um valor indefinido. Esse valor é chamado de *NIL*.

Listing 4.12: O valor NIL.

```
1 /*
2 O valor NIL
3 */
```

[3] *NIL* é uma palavra do latim antigo que significa "nada"

```
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5  PROCEDURE MAIN()
6  LOCAL x
7
8     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
9
10
11    ? "O valor de x é  : ", x     // O valor de x é  :  NIL
12
13
14 RETURN
```

Existem duas formas de uma variável receber o valor NIL.

1. Atribuindo o valor NIL a variável. Por exemplo : x := NIL
2. Inicializando-a sem atribuir valor. Por exemplo LOCAL x

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

### 4.2.8 Um tipo especial de variável : *SETs*

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

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

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

1. Você não pode criar um *SET*. No apêndice XXX nós veremos todos os *SETs* com os seus respectivos valores válidos, você não precisa se apressar em decorá-los pois nós iremos aprender os tipos principais no decorrer do livro.

2. Não existem *SETs* sem valores. Todos eles possuem valores criados durante a inicialização do programa ou seja, um valor padrão[4]. Por exemplo, no caso de *SET DATE* (que altera a forma como a data é vista) o valor padrão é *AMERICAN*, que mostra a data no formato mês, dia e ano.

3. Você não pode atribuir qualquer valor a um *SET* porque ele possui uma lista de valores válidos. Cada *SET* possui a sua lista de valores válidos.

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

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

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

**Dica 21**

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

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

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

   ... Continua...

RETURN NIL
```

### 4.2.9 Inicialização com tipos de dados

Agora que nós já sabemos os principais tipos de dados do Harbour (Caractere, numérico, data e lógico), vamos ver uma nova forma ainda não totalmente implementada (atual-

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

mente uso o harbour 3.2) de se inicializar variáveis que promete conferir uma segurança maior ao nosso programa. Trata-se da inicialização com o tipo de dado associado. Até agora nós aconselhamos (ainda não explicamos) que você inicie as suas variáveis com LOCAL. Agora veremos uma sintaxe que torna essa inicialização mais segura: a inicialização com o tipo e dado associado a variável. Primeiro veremos mais uma vez a forma com que temos inicializado as nossas variáveis :

Listing 4.13: Até agora estamos criando variáveis assim.

```
1  /*
2  Inicialização perigosa!
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5  PROCEDURE MAIN()
6  LOCAL cNome := "Programador"
7
8     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
9
10
11    ? "O valor de cNome é  : ", cNome    // O valor de cNome é  :
          Programador
12
13    cNome := 12
14
15    ? "O valor de cNome é agora " , cNome  // O valor de cNome é 12
16
17
18 RETURN
```

O Harbour é um projeto em constante evolução, talvez quando você ler esse documento esse recurso já esteja totalmente implementado na linguagem. Atualmente, se você criar uma variável e atribuir a ela uma data, nada impede de você aproveitar essa mesma variável e atribuir a ela um número. Ou seja, nada impede que você atribua um tipo de dado diferente a uma variável anteriormente criado com outro tipo de dado. Na listagem 4.13 a variável **cNome** foi criada com o intuito de ser do tipo caractere, mas ela assumiu um valor numérico na linha 13. Existem muitos códigos que funcionam perfeitamente utilizando essa técnica de "reaproveitamento" de variáveis, mas ela não é considerada uma técnica segura.

Para evitar problemas futuros, o Harbour está evoluindo[5] para uma forma mais segura de se criar variáveis. Confira esse mesmo código na listagem 4.14.

Listing 4.14: Formato com tipagem forte.

```
1
2  PROCEDURE MAIN()
3  local a as numeric, b as character
4  a = 1
```

[5]Essa característica ainda não está implementada

```
5  b = 2 //erro em tempo de compilação se a opção de verificação
       estática de tipos estiver ligada
6  b = "x"
7  ? a + b //também gerará erro em tempo de compilação
8
9
10 RETURN
```

Mantendo o seu compromisso de compatibilidade com os códigos antigos, a equipe que desenvolve o Harbour não irá "mudar as regras do jogo". Você sempre poderá atribuir um número a uma variável data se assim desejar. Simplesmente essa nova forma de declaração fará parte da linguagem e conviverá com a antiga. Se você quiser que o Harbour trabalhe da forma tradicional apenas inicialize sem o tipo de dado associado conforme a listagem 4.13, mas se você já quer ir logo se acostumando ao jeito novo, o Harbour já aceita a sintaxe da listagem 4.14, embora o recurso ainda não funcione. Quando esse recurso estiver funcionando você não poderá mudar o tipo de dado de uma variável declarada como (AS NUMERIC) para outro tipo de dado (se tentar irá causar um erro de execução).

## 4.3 Exercícios de revisão geral

1. (ASCENCIO; CAMPOS, 2014, p. 36) Faça um programa que receba quatro números inteiros, calcule e mostre a soma desses números.

2. (FORBELLONE; EBERSPACHER, 2005, p. 33) Construa um algoritmo para calcular o volume de uma esfera de raio R, em que R é um dado fornecido pelo usuário.

   ```
   O volume de uma esfera é dado por V = ( 4 * PI * R ^ 3 ) / 3.
   ```

3. (ASCENCIO; CAMPOS, 2014, p. 36) Faça um programa que receba três notas, calcule e mostre a média aritmética.

4. O comando *SET DECIMALS TO numero* serve para alterar a exibição do número de casas decimais, onde *numero* é um valor inteiro. Altere o programa anterior para exibir a média com apenas uma casa decimal.

5. Altere o programa anterior para exibir a média ponderada além da média aritmética. Dica: a média ponderada exige a criação de mais 3 variáveis representando os pesos.

6. (ASCENCIO; CAMPOS, 2014, p. 39) Faça um programa que receba o salário de um funcionário, calcule e mostre o novo salário, sabendo-se que este sofreu um aumento de 25%.

7. Altere o programa anterior para calcular de quanto foi o aumento.

8. Faça um programa que calcule e mostre a área de um retângulo.
9. (ASCENCIO; CAMPOS, 2014, p. 42) Faça um programa que receba um número positivo e maior que zero, calcule e mostre :
    (a) o número digitado ao quadrado;
    (b) o número digitado ao cubo;
    (c) a raiz quadrada do número digitado;
    (d) a raiz cúbica do número digitado;

# 5 ESTRUTURAS DE CONTROLE

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

Até agora os programas que foram desenvolvidos obedecem a uma estrutura chamada linear, basicamente resumida em entrada, cálculo e saída. Mas vários comandos, que veremos no decorrer desse capítulo, permitem que o computador tome decisões baseado nos valores de variáveis, e que também execute um processamento de centenas de linhas sem ter que digitar essas linhas uma a uma. Deitel chama isso de *transferência de controle*.

Essas estruturas foram desenvolvidas na década de 1960 por Brian e Jacopini. O trabalho desses dois pesquisadores demonstrou que **todo e qualquer** programa de computador podia ser escrito em termos de somente três estruturas de controle : sequência, controle de fluxo e repetição (DEITEL; DEITEL, 2001, p. 101).

Nós já estudamos as estruturas de sequência. Nesse capítulo veremos as estruturas de controle de fluxo e de repetição com a ajuda de fluxogramas e pseudocódigo.

## 5.1 Algoritmos : Pseudocódigos e Fluxogramas

Antes de prosseguirmos iremos introduzir um conceito novo : o algoritmo. O conceito central da programação e da ciência da computação é o de algoritmo (GUIMARãES; LAGES, 1994, p. 2). Programar é construir algoritmos.

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

> A formulação de um algoritmo geralmente consiste em um texto contendo comandos (instruções) que devem ser executados numa ordem prescrita. Esse texto é uma representação concreta do algoritmo e [...] é expandido somente no espaço da folha de papel (GUIMARãES; LAGES, 1994, p. 2)

Não é nosso objetivo parar o estudo da linguagem Harbour para adentrar no estudo dos algoritmos, nós iremos apenas nessa seção introduzir alguns conceitos e ir escrevendo os algoritmos juntamente com os códigos futuros que comporão o nosso estudo. Nas próximas sub-seções nós abordaremos duas ferramentas simples que são usadas na confecção de algoritmos : o pseudo-código e o fluxograma. Os pseudo-códigos e os fluxogramas estão listados no apêndice A para não dificultar a leitura desse capítulo, nós aconselhamos a leitura desse apêndice mas em um instante posterior.

### 5.1.1 Pseudo-código

O pseudo-código[1] como o próprio nome sugere é uma versão na nossa língua da linguagem de programação. A ideia é permitir que com um conjunto básico de primitivas seja possível ao projetista pensar **apenas** no problema, e não na linguagem de programação. Apenas tome cuidado para não ficar muito distante da linguagem. Os elementos do pseudo-código são poucos e serão apresentados juntamente com os códigos em Harbour. Até agora os elementos que nós vimos são :

1. Variáveis
2. Operadores
3. Comandos de exibição

### 5.1.2 Fluxograma

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

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

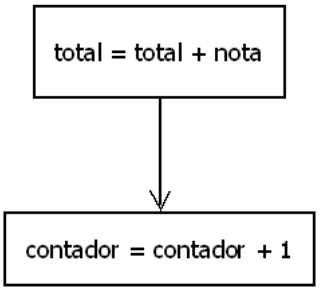
Nós usaremos basicamente apenas três componentes :

[1] Pseudo é uma palavra de origem grega que significa falso.

1. Um quadro retangular que representa uma instrução qualquer ou um conjunto de instruções.
2. Um quadro em forma de losango que representa uma decisão (estudaremos o que é isso em Harbour na próxima seção).
3. Setas que interligam esses quadros

Se nós fossemos representar através de fluxogramas os códigos que nós desenvolvemos até agora só teríamos uma sequencia de retangulos interligados conforme a figura 5.1.

Figura 5.1: Fluxograma de uma estrutura sequencial




Na próxima seção nós apresentaremos os comandos do Harbour para controle de fluxo com suas respectivas representações em fluxograma.

## 5.2 Comandos de controle de fluxo

De acordo com Swan,

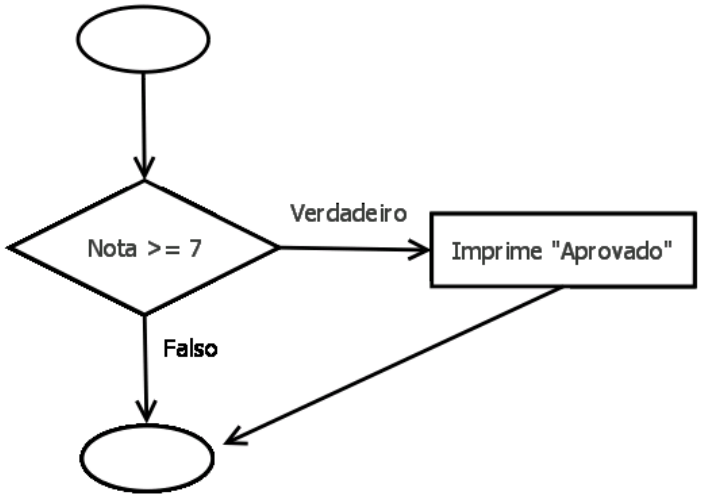
> os comandos de controle de fluxo utilizam os operadores relacionais de igualdade e lógicos que comparam operandos de diversas maneiras. Todos esses operadores tem o mesmo objetivo - produzir um valor representando verdadeiro ou falso que possa ser usado para direcionar o *fluxo* de um programa (SWAN, 1994, p. 93).

O Harbour dispõe das seguintes estruturas de controle de fluxo

1. IF ... ENDIF
2. DO CASE ... ENDCASE

Uma estrutura de seleção em um Fluxograma é representada por um losango, conforme a figura 5.2

Figura 5.2: Representação de uma estrutura de seleção em um fluxograma




### 5.2.1 A estrutura de seleção IF ... ENDIF

A estrutura IF (Se) permite que o programa tome uma decisão “se” uma determinada comparação ou variável retornar verdadeiro ou falso. A estrutura de seleção IF ... ENDIF funciona conforme a listagem 5.1.

Listing 5.1: Estrutura de seleção IF

```
1  /*
2  Estruturas de controle de fluxo
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nNota as NUMERIC // Nota
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebe o valor da nota
12    nNota := 0
13    INPUT "Informe a nota do aluno : " TO nNota
14
15    // Decide se foi aprovado ou não de acordo com a média
16    IF ( nNota >= 7 )
17       ? "Aprovado"
18    ENDIF
19
20
21 RETURN
```

Um ponto importante a destacar é o recuo (indentação) que é feito no interior do bloco de código do IF (linha 17). Esses recuos não são obrigatórios, mas você deve se habituar a fazê-los para que o seu código fique claro.

**Dica 22**

Da mesma forma que você deve "indentar" o interior do bloco principal, você deve aplicar recuos nos blocos internos de todas as suas estruturas de controle. Nos exemplos a seguir você verá que as estruturas de controle podem ficar uma dentro da outra[a], habitue-se a aplicar os recuos nelas também, quantos níveis forem necessários.

[a]Quando uma estrutura fica uma dentro da outra, o termo correto é estruturas "aninhadas"

O código listado em 5.1 está correto, mas se o aluno for reprovado ele não exibe mensagem alguma. Essa situação é resolvida usando um ELSE (Senão). Veja a sua aplicação em 5.2.

Listing 5.2: Estrutura de seleção IF ... ELSE ... ENDIF

```
1  /*
2  Estruturas de controle de fluxo
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nNota as NUMERIC // Nota
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebe o valor da nota
12    nNota := 0
13    INPUT "Informe a nota do aluno : " TO nNota
14
15    // Decide se foi aprovado ou não de acordo com a média
16    IF ( nNota >= 7 )
17       ? "Aprovado"
18    ELSE
19       ? "Reprovado"
20    ENDIF
21
22
23 RETURN
```

Agora, imagine a seguinte mudança : o diretor da escola lhe pede para incluir uma mudança no código: se o aluno tirar menos de 7 e mais de 3 ele vai para recuperação e não é reprovado imediatamente. Esse problema pode ser resolvido de duas formas diferentes : inserindo uma nova estrutura IF dentro do ELSE da primeira estrutura ou usando um ELSEIF. As duas respostas resolvem o problema.

A listagem 5.3 mostra a primeira forma, que é inserindo uma estrutura dentro da outra (observe a indentação).

Listing 5.3: Estrutura de seleção IF ... ELSE ... ENDIF com uma estrutura dentro de outra.

```
1  /*
2  Estruturas de controle de fluxo
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nNota as NUMERIC // Nota
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebe o valor da nota
12    nNota := 0
13    INPUT "Informe a nota do aluno : " TO nNota
14
15    // Decide se foi aprovado ou não de acordo com a média
16    IF ( nNota >= 7 )
17       ? "Aprovado"
18    ELSE
19       IF ( nNota >= 5 )
20          ? "Recuperação"
21       ELSE
22          ? "Reprovado"
23       ENDIF
24    ENDIF
25
26
27 RETURN
```

A listagem 5.4 aborda a segunda forma, que é com a inclusão de um ELSEIF (que faz parte do comando IF principal e não deve sofrer indentação).

Listing 5.4: Estrutura de seleção IF ...ELSEIF ... ELSE ... ENDIF

```
1  /*
2  Estruturas de controle de fluxo
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nNota as NUMERIC // Nota
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebe o valor da nota
```

```
12    nNota := 0
13    INPUT "Informe a nota do aluno : " TO nNota
14
15    // Decide se foi aprovado ou não de acordo com a média
16    IF ( nNota >= 7 )
17       ? "Aprovado"
18    ELSEIF ( nNota >= 5 )
19          ? "Recuperação"
20    ELSE
21          ? "Reprovado"
22    ENDIF
23
24
25 RETURN
```

**Dica 23**

O código da listagem 5.4 é mais claro do que o código da listagem 5.3. Quando você se deparar com uma situação que envolve múltiplos controles de fluxo e apenas uma variável prefira a estrutura IF ... ELSE ... ELSEIF. Múltiplos níveis de indentação tornam o código mais difícil de ler.

#### 5.2.1.1 A estrutura de seleção DO CASE ... ENDCASE

A estrutura de seleção DO CASE ... ENDCASE é uma variante da estrutura IF ... ELSE ... ELSEIF. Vamos continuar do mesmo exemplo anterior : suponha agora que o operador se enganou na hora de introduzir a nota do aluno e digitou 11 em vez de 1. O programa não pode aceitar valores inválidos: o menor valor possível é zero e o maior valor possível é dez.

As duas listagens a seguir resolvem o problema. A primeira utiliza IF ... ELSEIF ... ELSE e a segunda utiliza DO CASE ... ENDCASE. Note que as duas estruturas são equivalentes.

Listing 5.5: Estrutura de seleção IF ... ELSE ...ELSEIF

```
1  /*
2  Estruturas de controle de fluxo
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nNota as NUMERIC // Nota
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebe o valor da nota
12    nNota := 0
```

```
13    INPUT "Informe a nota do aluno : " TO nNota
14
15    // Decide se foi aprovado ou não de acordo com a média
16    IF ( nNota > 10 ) .OR. ( nNota < 0 )
17       ? "Valor inválido"
18    ELSEIF ( nNota >= 7 )
19       ? "Aprovado"
20    ELSEIF ( nNota >= 5 )
21       ? "Recuperação"
22    ELSEIF ( nNota >= 0 )
23       ? "Reprovado"
24    ELSE
25       ? "Valor inválido"
26    ENDIF
27
28
29 RETURN
```

Agora usando DO CASE ... ENDCASE na listagem 5.6. O fluxograma da correspondente está representado pela figura A.2 do apêndice A.

Listing 5.6: Estrutura de seleção DO CASE ... ENDCASE

```
1  /*
2  Estruturas de controle de fluxo
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL nNota AS NUMERIC// Nota
8
9     hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
10
11    // Recebe o valor da nota
12    nNota := 0
13    INPUT "Informe a nota do aluno : " TO nNota
14
15    // Decide se foi aprovado ou não de acordo com a média
16    DO CASE
17    CASE ( nNota > 10 ) .OR. ( nNota < 0 )
18       ? "Valor inválido"
19    CASE ( nNota >= 7 )
20       ? "Aprovado"
21    CASE ( nNota >= 5 )
22       ? "Recuperação"
23    CASE ( nNota >= 0 )
24       ? "Reprovado"
```

```
25      OTHERWISE
26         ? "Valor inválido"
27      ENDCASE
28
29
30  RETURN
```

**Dica 24**

Ao testar múltiplas condições, uma declaração CASE é preferível a uma longa sequencia de IF/ELSEIFs. [...] A sentença OTHERWISE serve como a última condição; cuida de qualquer situação não prevista por um CASE (SPENCE, 1991, p. 70).

Existe mais um problema que pode ocorrer. Imagine agora que o usuário em vez de digitar a nota 6 digite um Y e passe um caractere para ser avaliado, quando a variável foi definida como numérica. Isso é um erro que faz com que o programa pare no meio da execução (erros de execução). Vamos abordar essa situação quando estivermos estudando o controle de erros do Harbour.

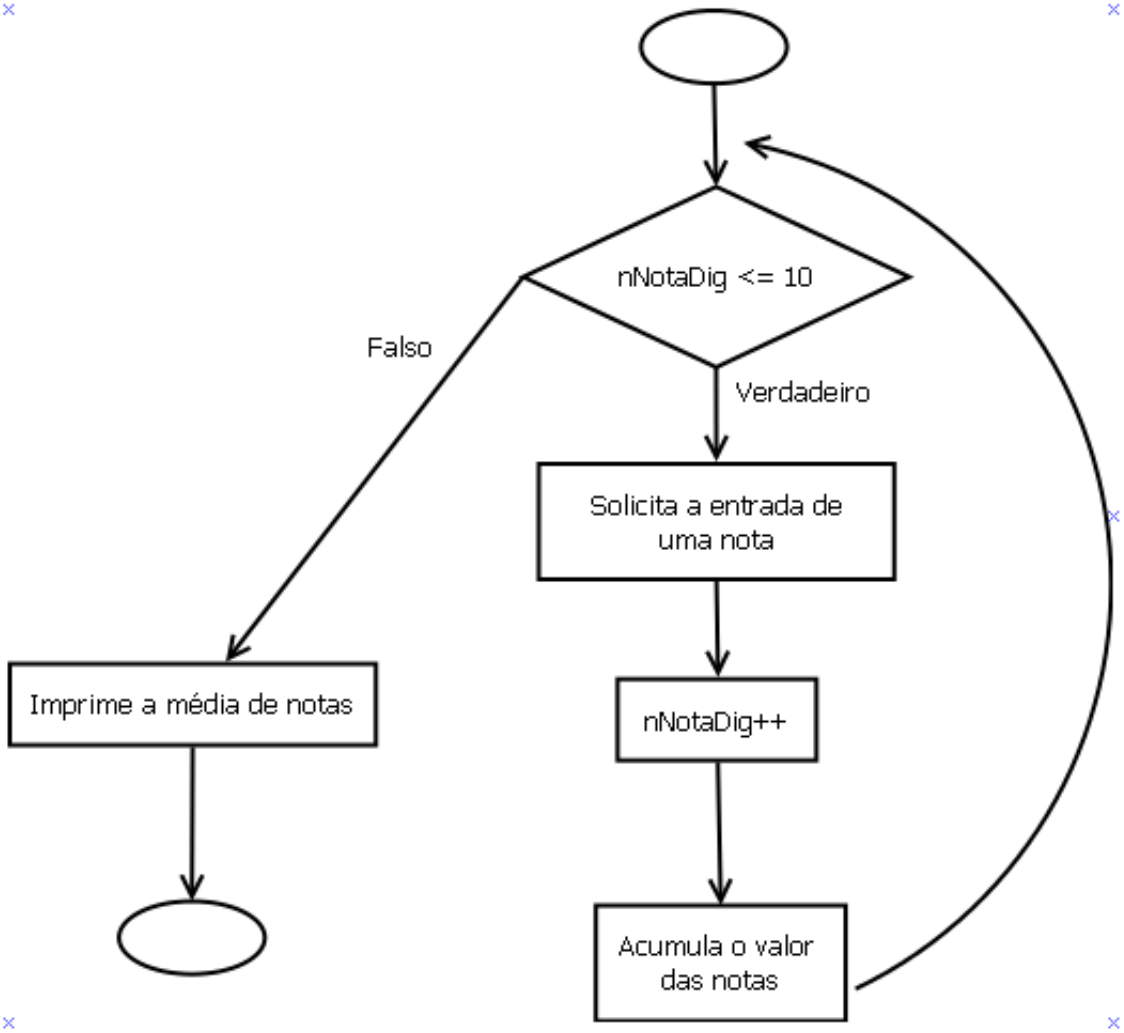
## 5.3 Estruturas de repetição

Estrutura de repetição (ou *loops*) são estruturas que repetem uma mesma sequência um dado numero de vezes. O número de vezes com que essa sequência é repetida pode ser fixa (conhecida antecipadamente) ou pode ser variável (baseada em uma condição de saída). O Harbour possui esses dois tipos de estruturas de repetição. Segundo Swan,

> "os loops permitem programar processos repetitivos, economizando espaço em código (você tem que digitar o comando repetitivo apenas uma vez não importando quantas vezes ele seja necessário) e executando inúmeras tarefas.[...] Sem loops a programação de computador como a conhecemos não existiria (SWAN, 1994, p. 107).

A figura 5.3 mostra como um fluxograma representa uma estrutura de repetição. Note que o fluxograma não possui nenhuma notação para estruturas de repetição, mas ela pode facilmente ser notada porque existe um bloco de código associado a uma estrutura de decisão. No final desse bloco de código uma seta retorna para a estrutura de decisão para verificar a condição de permanência na estrutura.

Figura 5.3: Fluxograma de uma estrutura de repetição




### 5.3.1 DO WHILE

A estrutura DO WHILE (Faça enquanto) é a mais básica de todas, nela o programa entra em *loop* e fica repetindo enquanto for verdade a condição de permanência. Conforme a listagem 5.7.

Listing 5.7: DO WHILE

```
1 /*
2 Estruturas de controle de fluxo
3 */
4 REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6 PROCEDURE MAIN()
7 LOCAL nNota AS NUMERIC// Nota
8
9    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
```

```
10
11    // Recebe o valor da nota
12    nNota := 0
13    DO WHILE ( nNota < 10 ) // Condição de permanência
14       // O programa ficará aqui enquanto
15       // nNota for menor do que 10
16       ? nNota++
17    ENDDO
18    ? "O valor de nNota é ", nNota  // Exibe 10
19
20
21 RETURN
```

**Dica 25**

Cuidado com a condição de saída do laço. No caso da listagem 5.7 essa condição é satisfeita graças ao incremento da variável **nNota** na linha 16. Sem esse incremento o programa nunca iria finalizar normalmente (o programa nunca iria sair do *loop*).

**Dica 26**

Cuidado com a condição de entrada no laço. No caso da listagem 5.7 essa condição é satisfeita graças ao valor de **nNota** que é inferior a 10. Se a variável fosse pelo menos igual a 10 o *loop* nunca seria executado. Em determinadas situações isso é certo, mas em outras situações não, tudo irá depender do problema a ser resolvido. O que queremos dizer é que os laços exigem que você tenha um controle sobre as condições de entrada e saída dele.

#### 5.3.1.1 Repetição controlada por um contador

O programa da listagem 5.7 foi desenvolvido usando uma técnica conhecida como "repetição controlada por contador". Nela, nós sabemos antecipadamente quantas iterações [2] o programa irá realizar.

Listing 5.8: DO WHILE (Digitação de notas)

```
1 /*
2 Estruturas de controle de fluxo
3 Adaptada de (DEITEL; DEITEL, 2001, p.111)
4 */
5 REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
6
7 PROCEDURE MAIN()
```

[2] Iteração significa : "repetição de alguma coisa ou de algum processo". No nosso contexto o termo iteração pode ser interpretado como : "o número de vezes que o interior do laço é repetido". Não confunda iteração com interação. Interação significa "ação mútua" ou "influência recíproca entre dois elementos".

```
8  LOCAL nTotNotas AS NUMERIC,; // Soma das notas
9        nNotaDig AS NUMERIC,; // Número de notas digitadas
10       nNota AS NUMERIC,; // O valor de uma nota
11       nMedia AS NUMERIC // Médias de notas
12
13    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
14
15    // Inicialização
16    nTotNotas := 0
17    nNotaDig := 1 // Condição de permanência no laço
18
19    // Processamento
20    DO WHILE ( nNotaDig <= 10 ) // Repete 10 vezes
21       INPUT "Forneça a nota : " TO nNota
22       nNotaDig++
23       nTotNotas += nNota
24    ENDDO
25
26    // Fase de término
27    nMedia := nTotNotas / 10
28    ? "A média geral é ", nMedia  // Exibe a média
29
30 RETURN
```

#### 5.3.1.2 Repetição controlada por um sentinela ou Repetição indefinida

Uma repetição controlada por um sentinela (ou repetição indefinida) permite que o laço seja repetido um número indefinido de vezes, dessa forma o usuário é quem define a condição de saida do laço. Ele vale tanto para a digitação de uma nota quanto para a digitação de um número muito grande de notas. Essa técnica envolve uma variável numérica conhecida como sentinela, cujo valor é usado para armazenar a nota do aluno corrente, mas também serve para sair do laço quando o valor for igual a um número pré-determinado. Segundo Deitel, “o valor da sentinela deve ser escolhido de forma que não possa ser confundido com um valor aceitável fornecido como entrada” (DEITEL; DEITEL, 2001, p. 114).

Listing 5.9: DO WHILE (Digitação de notas com sentinela)

```
1  /*
2  Estruturas de controle de fluxo
3  Programa para cálculo da média da turma com repetição controlada
      por sentinela
4  Adaptada de (DEITEL; DEITEL, 2001, p.117)
5  */
6  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
7
8  PROCEDURE MAIN()
```

```
9  LOCAL nTotNotas AS NUMERIC,; // Soma das notas
10       nNotaDig AS NUMERIC,; // Número de notas digitadas
11       nNota AS NUMERIC,; // O valor de uma nota
12       nMedia AS NUMERIC // Médias de notas
13
14    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
15
16    // Inicialização
17    nTotNotas := 0
18    nNotaDig := 0
19    nNota := 0
20    ? "Forneça a nota ou tecle -1 para finalizar"
21    INPUT "Forneça a nota : " TO nNota
22
23    // Processamento
24    DO WHILE ( nNota <> -1 ) // Repete enquanto não for -1
25       nTotNotas += nNota
26       nNotaDig++
27       INPUT "Forneça a nota : " TO nNota
28    ENDDO
29
30    // Fase de término
31    IF ( nNotaDig <> 0)
32       nMedia := nTotNotas / nNotaDig
33       ? "A média geral é ", nMedia  // Exibe a média
34    ELSE
35       ? "Nenhuma nota foi digitada"
36    ENDIF
37
38 RETURN
```

**Dica 27**

Quando for criar uma repetição indefinida (baseada em sentinela) procure escolher um valor de saída fora do intervalo válido, dessa forma o usuário não corre o risco de digitar esse valor pensando que ele é um valor válido. Alguns exemplos de valores de sentinela : -1, 9999, -9999, etc.

**Dica 28**

Boas práticas para se criar uma repetição indefinida :

1. O primeiro valor deverá ser lido antes da entrada do laço.
2. O próximo valor deverá ser lido antes de ser avaliado.

Na nossa listagem 5.9 esses valores são lidos pelo comando *INPUT*.

## 5.4 Implementando

Vamos agora simular através de exemplos as etapas no desenvolvimento de um programa de computador. Não é a nossa intenção criar processos rígidos para que você os siga sem questionar. O nosso objetivo é lhe dar uma direção a ser seguida quando lhe pedirem para desenvolver programas simples de computador. O seguinte processo é baseado no estudo de caso formulado por Deitel e Deitel (DEITEL; DEITEL, 2001, p. 120). O estudo de caso envolve as seguintes etapas :

1. Definição do problema
2. Lista de observações
3. Criação do pseudo-código através de refinamento top-down
4. Criação do programa

### 5.4.1 Definição do problema

Considere o seguinte problema :

*Uma escola oferece um curso preparatório para o Exame Nacional do Ensino Médio (ENEM). A escola deseja saber qual foi o desempenho dos seus estudantes. Você é o responsável pelo desenvolvimento do programa que irá resumir os resultados. Uma lista de dez estudantes com os resultados lhe foi entregue e na lista está escrito 1 se o estudante passou ou um 2 se o estudante foi reprovado. Se mais de 8 estudante passaram imprima a mensagem : "Aumente o preço do curso".*

### 5.4.2 Lista de observações

Temos agora as seguintes observações tiradas da descrição do problema :

1. O programa irá processar 10 resultados. Sempre que surgirem listas a serem processadas considere a possibilidade de criar um laço (WHILE).
2. Cada resultado é 1 ou 2. Devemos determinar se o resultado é 1 ou se é 2. Note que teremos que usar uma estrutura de seleção. Sempre que surgirem decisões a serem tomadas ou dados a serem classificados (se é 1 ou se é 2) considere usar uma estrutura de seleção (IF).
3. Devemos contar o número de aprovações e o número de reprovações. Considere usar um contador (uma variável que é incrementada dentro da estrutura de seleção).

4. Por fim, devemos testar se mais de oito passaram no exame e exibir uma mensagem. Novamente deveremos utilizar uma estrutura de seleção (IF) para exibir uma mensagem (“Aumente o preço do curso”).

### 5.4.3 Criação do pseudo-código através do refinamento top-down

O refinamento top-down é uma técnica de análise que pode ser usada na confecção de procedimentos como o desse estudo de caso. Essa técnica consiste em definir o problema principal e ir subdividindo-o até chegar nos níveis mais baixos do problema. Os itens a seguir exemplificam essa técnica :

- Representação do pseudo-código do topo : *Analise os resultados do exame e decida se o preço deve ser aumentado*
- Faça o primeiro refinamento
    1. Inicialize as variáveis
    2. Obtenha as dez notas e conte as aprovações e reprovações
    3. Imprima um resumo dos resultados e decida se o preço deve ser aumentado
- Faça o segundo refinamento

Listing 5.10: Pseudo-código da análise dos resultados do teste

```
1  algoritmo "Analisando os resultados do teste"
2  // Seção de Declarações
3  inicio
4  // Seção de Comandos
5
6     // Inicialize as variáveis
7     Inicialize as aprovações com 0
8     Inicialize as reprovações com 0
9     Inicialize o contador com 1
10
11    // Obtenha as 10 notas
12    Enquando o contador <= 10
13       Leia o próximo resultado
14
15       Se o estudante foi aprovado
16          Some 1 as aprovações
17       senão
18          Some 1 as reprovações
19
20
21    // Imprima o resultado e decida se o curso
```

```
22      // deve ser aumentado
23      Imprima o número de aprovações
24      Imprima o número de reprovações
25
26      Se mais de 8 estudantes foram aprovados
27          Imprima "Aumente o preço do curso"
28
29  fimalgoritmo
```

- Faça o terceiro e último refinamento

Listing 5.11: Pseudo-código da análise dos resultados do teste

```
1   algoritmo "Analisando os resultados do teste"
2   // Seção de Declarações
3   var
4       aprova // Número de aprovações
5       reprova // Número de reprovações
6       contador // contador de estudantes
7   inicio
8   // Seção de Comandos
9       aprova := 0
10      reprova := 0
11      contador := 1
12
13      enquanto contador <= 10
14          Leia o próximo resultado do teste
15
16          Se o estudante foi aprovado
17              aprova++
18          senão
19              reprova++
20          fimse
21
22      fimenquanto
23
24      Imprime aprovações
25      Imprime reprovações
26
27      se contador > 8
28          Imprime "Aumente o preço do curso"
29      fimse
30
31  fimalgoritmo
```

Mais uma vez enfatizamos : você não precisa se ater precisamente a esses passos. O que queremos dizer é que o código do programa antes de ser concluído deve passar por

sucessivos refinamentos.

### 5.4.4 Criação do programa

Feito todos esses passos, a codificação do programa fica muito mais fácil porque muitos questionamentos foram respondidos nas fases anteriores. Quando muitas dúvidas surgirem enquanto você estiver codificando, e essas dúvidas não se relacionarem com algum detalhe técnico, isso é um forte indício de que essas dúvidas não são propriamente dúvidas técnicas de programação, mas questionamentos referentes a fases anteriores a da codificação. Hoje em dia, é cada vez menor o número de pessoas que apenas sentam e desenvolvem um programa sem passar por etapas anteriores que definem realmente o problema.

Listing 5.12: Análise dos resultados do teste

```
1  /*
2  Estruturas de controle de fluxo
3  Análise dos resultados do teste
4  Adaptada de (DEITEL; DEITEL, 2001, p.122)
5  */
6  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
7
8  PROCEDURE MAIN()
9  LOCAL nPasses AS NUMERIC,; // Número de aprovações
10       nFailures AS NUMERIC,; // Número de reprovações
11       nStudentCounter AS NUMERIC,; // Contador de estudantes
12       nResult AS NUMERIC // Resultado do teste
13
14    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
15
16    // Inicialização
17    nPasses := 0
18    nFailures := 0
19    nStudentCounter := 1
20    nResult := 0
21
22    // Processamento
23    DO WHILE ( nStudentCounter <= 10 ) // Repete enquanto não for
          <= 10
24       INPUT "Forneça o resultado (1=aprovado, 2=reprovado) : " TO
             nResult
25
26       IF ( nResult == 1 )
27          nPasses++
28       ELSE
29          nFailures++
30       ENDIF
31
```

```
32          nStudentCounter++
33
34      ENDDO
35
36      // Fase de término
37      ? "Aprovados : ", nPasses
38      ? "Reprovados: ", nFailures
39
40      IF ( nPasses > 8 )
41          ?
42          ? "Aumente o preço do curso."
43      ENDIF
44
45  RETURN
```

## 5.5 Um tipo especial de laço controlado por contador

Os laços controlados por contador são muito usados por programadores de diversas linguagens. De tão usado, esse laço ganhou uma estrutura especial chamada de *FOR*. Quando a estrutura *FOR* começa a ser executada, a variável **nContador** é declarada e inicializada ao mesmo tempo. Abaixo segue um exemplo simples dessa estrutura :

```
FOR x := 1 TO 10
      ? x
NEXT
```

A variável contador (no caso o **x**) é automaticamente inicializada quando o laço é iniciado. Nos nossos códigos de exemplo essa variável será inicializada fora do laço através de *LOCAL*, conforme já frisamos nos capítulos anteriores. Ainda não podemos revelar a razão de inicializar com *LOCAL*, apenas podemos adiantar que é uma boa prática de programação e que, além disso, o seu código fica mais claro.

Na listagem 5.13 vemos uma comparação entre o laço *WHILE* e o laço FOR. Note que o laço *WHILE* é mais genérico do que *FOR*, embora você deva usar *FOR* nas situações que envolvem laço com contador conhecido.

Listing 5.13: Comparação entre os laços FOR e WHILE

```
1  /*
2  Estruturas de controle de fluxo
3  Comparação entre os laços FOR e WHILE
4  */
5  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
```

```
6
7  PROCEDURE MAIN()
8  LOCAL nContador AS NUMERIC
9
10    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
11
12    // Processamento while
13    ? "Processando com o laço WHILE"
14    nContador := 1 // Incialização obrigatória
15    DO WHILE ( nContador <= 10 )
16       ? nContador
17       ++nContador
18    ENDDO
19    ? "Após o laço WHILE o valor de nContador agora é " , nContador
20
21
22    // Processamento
23    ? "Processando com o laço FOR "
24    FOR nContador := 1 TO 10
25        ? nContador
26    NEXT
27    ? "Após o laço FOR o valor de nContador agora é " , nContador
28
29 RETURN
```

A estrutura *FOR* ainda possui uma interessante variação, observe a seguir :

```
FOR x := 1 TO 10 STEP 2
      ? x
NEXT
```

A cláusula *STEP* da estrutura *FOR* determina a quantidade a ser mudada para cada iteração do loop. Esse valor pode ser positivo ou negativo. Se essa cláusula não for especificada o valor do incremento é de um para cada iteração.

**Dica 29**

Quando a situação exigir uma quantidade determinada de iterações dê preferência ao laço *FOR*.

**Dica 30**

Deitel e Deitel nos advertem : “evite colocar expressões cujos valores não mudam dentro de laços” (DEITEL; DEITEL, 2001, p. 137). Essa advertência tem o seguinte sentido :

evite realizar cálculos cujo valor será sempre o mesmo dentro de laços. Por exemplo, se você colocar a expressão ( 12 * 43000 ) dentro de um laço que se repetirá mil vezes então essa expressão será avaliada mil vezes sem necessidade! Os autores acrescentam que "os modernos compiladores otimizados colocarão automaticamente tais expressões fora dos laços no código gerado em linguagem de máquina [...] porém, ainda é melhor escrever um bom código desde o início."

O Harbour aceita um tipo especial de comentário na estrutura *FOR*. Esse comentário, caso você queira usá-lo, se restringe a **apenas uma palavra** após o *NEXT* da estrutura *FOR*. Veja um exemplo abaixo :

```
FOR x := 1 TO 10
      ? x
NEXT x
```

Se fosse...

```
NEXT final do processamento de x
```

...geraria um erro, por causa do espaço (tem que ser apenas uma palavra).

Essa forma de comentário foi inspirado nas primeiras versões de uma linguagem chamada BASIC e é usado por alguns programadores, embora não tenha se popularizado. A função desse comentário é indicar, no final do laço, qual foi a variável que foi incrementada no início do mesmo laço. Isso é útil quando o laço é grande demais e ocupa mais de uma tela de computador, ficando difícil para o programador saber qual é a variável que pertence ao inicio do seu respectivo laço. Por exemplo :

```
FOR x := 1 TO 10

      ... Várias linhas de código


      FOR y := 1 TO 100

            ... Várias linhas de código


      NEXT y

      ? x
```

```
NEXT x
```

**Dica 31**

Você pode se deparar com laços que são tão grandes que não cabem no espaço de uma tela no seu computador. Dessa forma, habitue-se a informar no final do laço qual foi a variável que iniciou esse laço ou algum comentário adicional. Utilize comentários (//) para informar a condição no final do laço. Por exemplo :

```
DO WHILE nNotas <= 100

    // Vários comandos

ENDDO // Final do processamento de Notas  (nNotas)


Ou


FOR nNotas := 1 TO 100

      // Vários comandos

NEXT  // Final do processamento de Notas (nNotas)
```

Se o laço for grande você deve comentar o seu final. Prefira usar comentários no final de cada laço a usar esses comentários especiais já citados. Por exemplo :

```
FOR x := 1 TO 10

    ... Muitas linhas de código

NEXT  // x

ou

NEXT // Um comentário sucinto

no lugar de

NEXT x  // <=== Evite esse formato
```

**Dica 32**

Não altere o conteúdo de uma variável contadora de um laço *FOR*. O código abaixo funciona mas pode confundir o programador, além de ser difícil de ser lido.

```
FOR x := 1 TO 100
      x := x + 1
NEXT
```

**Dica 33**

Kernighan nos adverte sobre a importância da consistência de um idioma:

> Assim como os idiomas naturais, as linguagens de programação tem idiomas, maneiras convencionais pelas quais os programadores experientes escrevem um código comum. Um aprendizado central de qualquer linguagem é o desenvolvimento da familiaridade com os seus idiomas. Um dos idiomas mais comuns é a forma de um loop (KERNIGHAN; PIKE, 2000, p. 13)

A listagem 5.14 exibe um programa sintaticamente correto, mas confuso devido a inconsistência do idioma (*loop* fora do padrão).

Listing 5.14: Programa confuso devido a forma não usual do uso dos seus laços.

```
ï»¿/*
Consistência
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL nContador AS NUMERIC

   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

   ? "Processando com o laço WHILE "
   nContador := 1
   DO ;
   WHILE ;
   nContador < 100
       ? nContador
       nContador += 15
   ENDDO
   ? "O valor após a saída é " , nContador

   FOR ;
```

```
   nContador ;
   := 1 ;
   TO;
   100
       ? nContador
   NEXT


RETURN
```

## 5.6 Os comandos especiais : EXIT e LOOP

O Harbour dispõe de comandos que só funcionam no interior de estruturas de repetição, são eles : EXIT e LOOP. Esses dois comandos funcionam somente no interior dos laços WHILE e FOR.

O comando EXIT força uma saída prematura do laço, e o comando LOOP desvia a sequencia para o topo do laço.

Listing 5.15: Saída prematura com EXIT.

```
ï»¿/*
Desafio : Comando EXIT
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL nContador AS NUMERIC
LOCAL cResposta AS CHARACTER

   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

   ? "Processando com o laço WHILE "
   nContador := 1
   DO WHILE nContador < 100
       ? nContador
       ACCEPT "Deseja contar mais um ? (S/N) " TO cResposta
       IF !( cResposta $ "Ss" )  // Leia cResposta não está
          contido em "Ss"
          EXIT
       ENDIF
       nContador += 1
   ENDDO
   ? "O valor após a saída é " , nContador

```

```
24    ? "Processando com o laço FOR "
25    FOR nContador := 1 TO 100
26        ? nContador
27        ACCEPT "Deseja contar mais um ? (S/N) " TO cResposta
28        IF !( cResposta $ "Ss" ) // Leia cResposta não está contido
              em "Ss"
29           EXIT
30        ENDIF
31    NEXT
32    ? "O valor após a saída é " , nContador
33
34
35
36
37
38 RETURN
```

Listing 5.16: Desvio para o topo do laço com LOOP.

```
1  ï»¿/*
2  Desafio : Comando LOOP
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL cResposta AS CHARACTER
8  LOCAL nContador AS NUMERIC
9
10    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
11
12    ? "Processando com o laço WHILE "
13    DO WHILE .T.
14       ACCEPT "Confirma a operação ? (S/N) " TO cResposta
15       IF !( cResposta $ "SsNn" ) // Leia cResposta não está
             contido em "SsNn"
16          ? "Digite 'S' para sim ou 'N' para não"
17          LOOP
18       ELSE
19          EXIT
20       ENDIF
21    ENDDO
22
23    ? "Processando com o laço FOR "
24    FOR nContador := 1 TO 5
25        ? nContador
26        ACCEPT "Deseja prosseguir para a etapa seguinte (S/N) " TO
              cResposta
```

```
27          IF ( cResposta $ "nN" ) // Leia cResposta está contido em
               "Nn"
28            ? "Pulando sem executar o restante da operação " ,
                 nContador
29            LOOP
30          ENDIF
31          ? "RESTANTE DA OPERAÇÃO " , nContador
32       NEXT
33       ? "O valor após a saída é " , nContador
34
35    RETURN
```

**Dica 34**

Os comandos EXIT e LOOP são usados frequentemente por programadores Harbour, mas você deve evitar o uso desses comandos caso você tenha a opção de colocar a condição de permanência no topo do laço (lembra-se da dica sobre idiomas) . Quando for usá-los não esqueça de usar a indentação e, se possível, colocar comentários adicionais.

Os comandos EXIT e LOOP possuem uma sutil diferença dentro de um laço FOR (PAPAS; MURRAY, 1994, p. 203). O EXIT causa o final da execução do laço. Em contraste, o LOOP faz com que todos os comandos seguintes a ele sejam ignorados, **mas não evita o incremento da variável de controle de laço**. A listagem 5.17 ilustra essa situação.

Listing 5.17: Diferenças entre EXIT e LOOP.

```
1  ï»¿/*
2  Diferenças entre loop e continue
3  */
4  REQUEST HB_CODEPAGE_UTF8EX
5
6  PROCEDURE MAIN()
7  LOCAL nCont AS NUMERIC
8  LOCAL cResp AS CHARACTER
9
10    hb_cdpSelect( "UTF8EX" )
11
12    ? "LOOP"
13    FOR nCont := 1 TO 10
14       ? "O valor de nCont é " , nCont
15       IF nCont == 5
16          ? "VOU PARA O TOPO DO LAÇO MAS nCont SERÁ INCREMENTADA"
17          LOOP
18       ENDIF
19    NEXT
20    ? "O valor fora do laço é " , nCont // nCont vale 11
21
```

```
22    ? "EXIT"
23    FOR nCont := 1 TO 10
24       ? "O valor de nCont é " , nCont
25       IF nCont == 5
26          ? "VOU SAIR IMEDIATAMENTE SEM INCREMENTAR nCont"
27          EXIT
28       ENDIF
29    NEXT
30    ? "O valor fora do laço é " , nCont // nCont vale 5
31
32 RETURN
```

Esse comportamento se explica da seguinte forma : o incremento de uma variável se dá no topo do laço (FOR), e não na base dele (NEXT). O topo do laço (FOR) tem três funções : inicializar a variável, atribuir valor de incremento a variável e verificar se a condição ainda é satisfeita. Se a condição não for satisfeita o laço é abandonado e a variável sempre terá um valor superior ao limite superior do laço FOR. O NEXT funciona como uma espécie de LOOP, ele só faz mandar para o topo do laço.

## 5.7 O laço REPITA ATÉ

O laço do tipo "REPITA ATÉ" [3] é outro tipo especial de laço *WHILE*[4]. O Harbour não tem um laço desse tipo, mas ele pode ser facilmente obtido através do exemplo abaixo :

```
lCondicaoDeSaida := .f.
DO WHILE .T.  // Sempre é verdade (Sempre entra no laço)
     ... Processamento
     IF lCondicaoDeSaida  // Condição de saída
         EXIT  // Sai
     ENDIF
ENDDO
```

Esse laço difere dos demais porque ele **sempre** será executado **pelo menos** uma vez. Isso porque a sua condição de saída não está no início do laço, mas no seu final. No exemplo acima em algum ponto do processamento a variável **lCondicaoDeSaida** deve receber o valor verdadeiro para que o processamento saia do laço.

**Dica 35**

Em laços do tipo **REPITA ATÉ** você deve ter cuidado somente com a condição de saída do laço. A condição de entrada dele sempre é verdade.

[3] Esse laço é um velho conhecido dos programadores da linguagem PASCAL. O laço chama-se *REPEAT ... UNTIL*. Os programadores C tem também a sua própria versão no laço *do ... while*.

[4] Note que o laço *FOR* é o primeiro tipo especial de laço *WHILE*.

**Dica 36**

Se você tem certeza de que o processamento do laço será executado no mínimo uma vez use a construção **REPITA ATÉ**.

## 5.8 Estruturas especiais

Até esse ponto poderíamos encerrar o aprendizado das estruturas de repetição e de decisão, pois o mínimo necessário para você desenvolver um código estruturado já foi visto. São elas :

1. IF ... ELSE ... ENDIF
2. DO WHILE ... ENDDO

Nós aprendemos também as estruturas adicionais que auxiliam na clareza e no desenvolvimento de um programa :

1. IF ... ELSE ... ELSEIF ... ENDIF
2. DO CASE ... ENDCASE
3. FOR ... NEXT
4. A forma "REPITA ATÉ" do laço WHILE

Veremos agora algumas estruturas especiais que facilitam bastante a vida do programador, mas que só devem ser usadas em casos especiais, quando a situação exigir que você abra mão das estruturas acima.

### 5.8.1 BEGIN SEQUENCE

A estrutura BEGIN SEQUENCE se enquadra em uma estrutura de especial de sequencia. Ela não é uma estrutura de seleção pura nem um loop. Ela foi criada pelo antigo desenvolvedor da linguagem Clipper 5.0 que a define como *uma estrutura de controle frequentemente usada para controle de erros em tempo de execução* (NANTUCKET, 1990, p. 1-18). O controle de erros do Harbour só será visto após o capítulo referente a funções, porque muitos exemplos dependem do correto entendimento do que é uma função. Por enquanto nós vamos aplicar o BEGIN SEQUENCE em casos mais básicos mas que nos permitirão entender o funcionamento dessa interessante estrutura de sequencia.

Imagine a seguinte situação ilustrada na listagem 5.18. Note que existe uma condição no interior do laço mais interno (o que incrementa o valor de y) que requer a saída dos dois laços. Porém o comando EXIT irá sair apenas do laço mais interno para o laço externo (o

que incrementa o valor de x). Esse é um problema que de vez em quando ocorre quando nos deparamos com laços aninhados.

Listing 5.18: Situação problema

```
ï»¿/*
BEGIN SEQUENCE
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL x,y AS NUMERIC


   hb_cdpSelect( "UTF8EX" )


   FOR x := 1 TO 15
      FOR y := 1 TO 15
         ? x , y
         IF x == 3 .AND. y == 10 // Condição de saída
            ? "DEVO SAIR DOS DOIS LAÇOS"
            EXIT
         ENDIF
      NEXT

   NEXT
   ? "Final do programa"
   ? " x vale " , x , " e y vale " , y

RETURN
```

A solução imediata para o nosso problema é botar uma segunda verificação na saída do laço externo (o que incrementa o valor de x), conforme a listagem 5.19.

Listing 5.19: Situação problema: solução 1

```
ï»¿/*
BEGIN SEQUENCE
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL x,y AS NUMERIC


   hb_cdpSelect( "UTF8EX" )


```

```
13    FOR x := 1 TO 15
14       FOR y := 1 TO 15
15          ? x , y
16          IF x == 3 .AND. y == 10 // Condição de saída
17             ? "DEVO SAIR DOS DOIS LAÇOS"
18             EXIT
19          ENDIF
20       NEXT
21       IF x == 3 .AND. y == 10
22          ? "DEVO SAIR DESSE LAÇO"
23          EXIT
24       ENDIF
25    NEXT
26    ? "Final do programa"
27    ? " x vale " , x , " e y vale " , y
28
29 RETURN
```

Agora que você entendeu o problema e uma solução, vamos agora implementar a solução usando BEGIN SEQUENCE no código 5.20

Listing 5.20: Situação problema: solução 2 (Usando BEGIN SEQUENCE)

```
1  ï»¿/*
2  BEGIN SEQUENCE
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL x,y AS NUMERIC
8
9
10    hb_cdpSelect( "UTF8EX" )
11
12    BEGIN SEQUENCE
13       FOR x := 1 TO 15
14          FOR y := 1 TO 15
15             ? x , y
16             IF x == 3 .AND. y == 10 // Condição de saída
17                ? "DEVO SAIR DOS DOIS LAÇOS"
18                BREAK // Me manda para fora da sequencia
19             ENDIF
20          NEXT
21       NEXT
22    END SEQUENCE
23    ? "Final do programa"
24    ? " x vale " , x , " e y vale " , y
```

```
25
26 RETURN
```

Essa estrutura especial de quebra de sequencia é poderosa e pode tornar um código mais claro, mas todo esse poder tem um preço. Ela pode gerar códigos confusos pois a sequencia de um fluxo pode ser quebrada de forma confusa. No exemplo da listagem 5.20 nós vimos que BEGIN SEQUENCE nos ajudou, mas se você abusar demais dessa estrutura ela pode tornar seu código confuso. Spence nos alerta para termos cuidado ao usar BEGIN SEQUENCE e BREAK; pois nós podemos criar uma codificação ilegível com o uso excessivo desse par (SPENCE, 1994, p. 21).

**Dica 37**

Se você usar muito a estrutura de sequencia BEGIN SEQUENCE nos seus programas é melhor ficar atento: você pode estar escrevendo códigos difíceis de serem lidos posteriormente.

Quando nós formos estudar o controle de erros nós voltaremos a abordar essa estrutura especial de sequencia.

**Dica 38**

O uso básico de BEGIN SEQUENCE é manipular exceções [erros]. Ele fornece um local adequado para saltar quando ocorre um erro. Você pode usá-lo como um ponto de interrupção para a lógica profundamente aninhada.(SPENCE, 1994, p. 21)

### 5.8.2 FOR EACH

O laço *FOR EACH* será visto parcialmente aqui porque ele depende do entendimento de estruturas de dados ainda não vistas. Esse laço é usado para percorrer todos os itens de :

1. Uma string
2. Um array
3. Um hash

Como ainda não vimos o que é array nem o que é um hash deixaremos os respectivos exemplos para quando abordarmos as estruturas de dados complexas da linguagem Harbour. Por enquanto vamos demonstrar o uso desse poderoso laço com as strings, através da listagem 5.21.

Listing 5.21: Um exemplo de FOR EACH com strings

```
1 ï»¿/*
2 For each
3 */
```

```
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL cCliente AS CHARACTER
8  LOCAL cEl AS CHARACTER
9
10    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
11
12    ? "Processando com o laço FOREACH "
13    ? "Ele pegará a string e processará letra por letra"
14    ACCEPT "Informe seu nome : " TO cCliente
15    FOR EACH cEl IN cCliente
16       ? cEl
17    NEXT
18
19
20 RETURN
```

A listagem 5.22 exibe a string de “trás para frente” com a cláusula *DESCEND*.

Listing 5.22: Um exemplo de FOR EACH com strings (Reverso)

```
1  ï»¿/*
2  For each
3  */
4  REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6  PROCEDURE MAIN()
7  LOCAL cNome AS CHARACTER
8  LOCAL cEl AS CHARACTER
9
10
11    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
12
13    ? "Processando com o laço FOREACH "
14    ? "Ele pegará a string e processará letra por letra"
15    ? "MAS IRÁ EXIBIR AO CONTRÁRIO"
16    ACCEPT "Informe seu nome : " TO cNome
17    FOR EACH cEl IN cNome DESCEND
18       ? cEl
19    NEXT
20
21
22 RETURN
```

**Dica 39**

Sempre que você precisar processar o interior de uma string use o laço FOR EACH ou uma função de manipulação de strings. No capítulo referente a funções você verá que o Harbour possui inúmeras funções prontas para o processamento de strings, a correta combinação desses dois recursos (o laço FOR EACH e as funções de manipulação de strings) tornarão os seus programas mais eficientes e mais fáceis de serem lidos.

### 5.8.3 SWITCH

A estrutura de seleção SWITCH pode ser exemplificada na listagem 5.23.

Listing 5.23: Um exemplo de SWITCH

```
ï»¿/*
Switch
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL nOpc AS NUMERIC


   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

   ? "Uso de SWITCH "
   INPUT "Informe a sua opção ( 1 , 2 ou 3)  : " TO nOpc

   SWITCH nOpc
   CASE 1
      ? "Escolheu 1"
   CASE 2
      ? "Escolheu 2"
      EXIT
   CASE 3
      ? "Escolheu 3"
   OTHERWISE
      ? "Opção inválida"
   END
   ? "Terminou!"

RETURN
```

Essa estrutura é uma das implementações novas do Harbour, ela foi inspirada na estrutura de seleção switch da linguagem C. Apesar de simples, essa estrutura possui algumas particularidades que listaremos a seguir :

1. A estrutura irá analisar se o valor da variável é igual aos valores informados nos respectivos CASEs, quando o valor for igual então o processamento irá “entrar” no CASE correspondente.

2. Cuidado, uma vez que o processamento entre no CASE correspondente, você deve informar para ele sair com a instrução EXIT. Se você não digitar EXIT então o processamento irá “entrar” nos CASEs subsequentes, mesmo que o valor da variável não seja igual ao valor do CASE.

A listagem 5.23 ilustra isso. Se o usuário digitar 1, então o programa irá entrar nos CASEs 1 e 2 até que encontra um EXIT para sair. Caso o usuário digite 2, então o programa irá entrar no CASE 2, encontrar um EXIT e sair. Se, por sua vez, o usuário digitar 3 o programa irá entrar no CASE 3 e na cláusula OTHERWISE. Esse comportamento estranho é verificado na Linguagem C e em todas as linguagens derivadas dela, como C++, PHP e Java. Inclusive, existe um termo para esse comportamento : “fall through”, que significa aproximadamente “cair em”. Kernighan e Pike (KERNIGHAN; PIKE, 2000, p. 18) recomendam não criarem *fall through* dentro de um SWITCH. Apenas um caso de *fall through* é tido como claro, que é quando vários cases tem um código idêntico, conforme abaixo :

```
SWITCH nOpc
CASE 1
CASE 2
CASE 3
   ...
   EXIT
CASE 4
   ...
   EXIT
END
```

Similar ao DO CASE ... END CASE e ao IF .. ELSEIF ... ELSE ... ENDIF, a estrutura de seleção SWITCH ainda possui a desvantagem de não aceitar expressões em sua condição, por exemplo, o código da listagem 5.24 **está errado** pois contém uma expressão nas avaliações de condição :

Listing 5.24: Um exemplo de SWITCH COM ERRO

```
1 ï»¿/*
2 Switch
3 */
4 REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8
5
6 PROCEDURE MAIN()
```

```
7  LOCAL nAltura AS NUMERIC
8
9
10    hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8
11
12    ? "SIMULAÇÃO DE ERRO COM SWITCH "
13    INPUT "Informe a sua altura (Ex: 1.70)  : " TO nAltura
14
15    SWITCH nAltura
16    CASE nAltura > 2.0
17       ? "ALTO"
18    CASE nAltura > 1.60 .AND. nAltura < 2.0
19       ? "ESTATURA MEDIANA"
20    CASE nAltura <= 1.6
21       ? "BAIXA ESTATURA"
22    END
23
24 RETURN
```

Com tantas limitações, qual a vantagem de se usar SWITCH ? Não bastaria um DO CASE ... END CASE ou um IF .. ELSEIF ... ELSE ... ENDIF ?

Três motivos justificam o uso do SWITCH :

1. Ele é muito mais rápido do que as outras estruturas de seleção. Se você for colocar uma estrutura de seleção dentro de um laço que se repetirá milhares de vezes, a performance obtida com um SWITCH pode justificar o seu uso.
2. O SWITCH é uma estrutura que existe em outras linguagens, como já dissemos, por isso ela não é nenhuma característica esquisita da linguagem Harbour. Muitos programadores usam SWITCH em seus códigos em C, C++, etc.
3. O SWITCH obriga a análise de apenas uma variável. Isso torna a programação mais segura quando se quer analisar apenas um valor.

**Dica 40**

Não esqueça do EXIT dentro de um caso de um SWITCH. Só omita o EXIT (*fall through*) quando múltiplos casos devam ser tratados da mesma maneira.

## 5.9 Conclusão

Encerramos uma importante etapa no aprendizado de qualquer linguagem de programação, as estruturas de seleção e de repetição constituem a base de qualquer linguagem de programação. As próximas seções envolvem desafios e exercícios, tem também um resumo das estruturas vistas em forma de fluxograma no apêndice XXX.

## 5.10 Desafios

### 5.10.1 Compile, execute e descubra

O programa da listagem 5.25 possui um erro de lógica. Veja se descobre qual é.

Listing 5.25: Onde está o erro de lógica ?

```
ï»¿/*
Desafio : Erro de lógica
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL nContador AS NUMERIC

   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

   ? "Processando com o laço WHILE "
   nContador := 1
   DO WHILE nContador <> 100
       ? nContador
       nContador += 15
   ENDDO
   ? "O valor após a saída é " , nContador


RETURN
```

O programa da listagem 5.26 possui um erro de lógica. Veja se descobre qual é.

Listing 5.26: Onde está o erro de lógica ?

```
ï»¿/*
Desafio : Erro de logica
*/
REQUEST HB_CODEPAGE_UTF8EX // Disponibiliza o suporte a UTF8

PROCEDURE MAIN()
LOCAL cResposta AS CHARACTER
LOCAL nContador AS NUMERIC

   hb_cdpSelect( "UTF8EX" ) // Seleciona o suporte a UTF8

   ? "Processando com o laço WHILE "
   ACCEPT "Confirma a operação ? (S/N) " TO cResposta
   DO WHILE .T.
      IF !( cResposta $ "SsNn" ) // Leia cResposta não está
         contido em "SsNn"
```

```
16          ? "Digite 'S' para sim ou 'N' para não"
17          LOOP
18       ELSE
19          EXIT
20       ENDIF
21    ENDDO
22
23
24 RETURN
```

### 5.10.2 Compile, execute e MELHORE

O programa da listagem 5.27 (adaptado de (PAPAS; MURRAY, 1994, p. 204)) nos apresenta um joguinho no qual o usuário precisa adivinhar um número. Esse programa está funcionando sem erros de lógica. O desafio agora é melhorar esse programa. Eis aqui algumas dicas :

1. Quando o usuário informar o palpite, o programa deve informar se o mesmo está acima do valor ou abaixo do valor sorteado.
2. Se o valor que o usuário informar diferir do valor sorteado em menos ou mais de três unidades exiba a mensagem adicional : "Está quente!".
3. Pense em outras formas de melhorar o programa. Por exemplo : os dois intervalor (10 e 40) podem ser informados por um segundo jogador. E por aí vai. O mais importante é "mecher" no código.

Listing 5.27: Jogo da adivinhação

```
1  ï»¿/*
2  Exemplos de loop e continue
3  Adaptado de PAPAS and MURRAY (1994)
4
5  Esse código sorteia um número com a função HB_RAND
6
7  HB_RandomInt( 10 , 40 ) // Sorteia um número entre 10 e 40
8
9  */
10 REQUEST HB_CODEPAGE_UTF8EX
11
12 PROCEDURE MAIN()
13 LOCAL nRand, nPalpite, nTentativa AS NUMERIC
14 LOCAL lSorte AS LOGICAL
15
16    hb_cdpSelect( "UTF8EX" )
```

```
17    nRand := HB_RandomInt( 10 , 40 ) // Realiza o sorteio
18    lSorte := .f. // Inicia lSorte com falso
19    nTentativa := 0
20
21    ? "Tente adivinhar um número entre 10 e 40"
22    DO WHILE .NOT. lSorte // Faça enquanto não tiver sorte
23       ++nTentativa
24       INPUT "Tente adivinhar meu número : " TO nPalpite
25       IF nPalpite == nRand
26          lSorte := .t.
27       ELSE
28          LOOP
29       ENDIF
30       ? "Acertou! Você precisou de " , nTentativa , " tentativas"
31    ENDDO
32
33 RETURN
```

## 5.11 Exercícios de revisão geral

1. (ASCENCIO; CAMPOS, 2014, p. 45) Faça um programa que receba o número de horas trabalhadas e o valor do salário mínimo, calcule e mostre o salário a receber, seguindo essas regras :

   (a) a hora trabalhada vale a metade do salário mínimo.

   (b) o salário bruto equivale ao número de horas trabalhadas multiplicado pelo valor da hora trabalhada.

   (c) o imposto equivale a 3% do salário bruto.

   (d) o salário a receber equivale ao salário bruto menos o imposto

2. (ASCENCIO; CAMPOS, 2014, p. 64) Faça um programa que receba três números e mostre-os em ordem crescente. Suponha que o usuário digitará três números diferentes.

3. Altere o programa anterior para que ele mesmo não permita que o usuário digite pelo menos dois números iguais.

4. (ASCENCIO; CAMPOS, 2014, p. 78) Faça um programa para resolver equações de segundo grau.

   (a) O usuário deve informar os valores de $a$, $b$ e $c$.

   (b) O valor de $a$ deve ser diferente de zero.

   (c) O programa deve informar o valor de delta.

(d) O programa deve informar o valor das duas raízes.

5. (FORBELLONE; EBERSPACHER, 2005, p. 46) Faça um programa que leia o ano de nascimento de uma pessoa, calcule e mostre sua idade e verifique se ela já tem idade para votar (16 anos ou mais) e para conseguir a carteira de habilitação (18 anos ou mais).

6. (FORBELLONE; EBERSPACHER, 2005, p. 47) O IMC (Índice de massa corporal) indica a condição de peso de uma pessoa adulta. A fórmula do IMC é igual ao peso dividido pelo quadrado da altura. Elabore um programa que leia o peso e a altura de um adulto e mostre a sua condição de acordo com a tabela abaixo :

   (a) abaixo de 18.5 : abaixo do peso

   (b) entre 18.5 : peso normal

   (c) entre 25 e 30 : acima do peso

   (d) acima de 30 : obeso

7. (FORBELLONE; EBERSPACHER, 2005, p. 65) Construa um programa que leia um conjunto de dados contendo altura e sexo (“M” para masculino e “F” para feminino) de 10 pessoas e, depois, calcule e escreva :

   (a) a maior e a menor altura do grupo

   (b) a média de altura das mulheres

   (c) o número de homens e a diferença porcentual entre eles e as mulheres

8. (FORBELLONE; EBERSPACHER, 2005, p. 66) Anacleto tem 1,50 metro e cresce 2 centímetros por ano, enquanto Felisberto tem 1,10 metro e cresce 3 centímetros por ano. Construa um programa que calcule e imprima quantos anos serão necessários para que Felisberto seja maior do que Anacleto.

9. Adaptado de (DEITEL; DEITEL, 2001, p. 173) Identifique e corrija os erros em cada um dos comandos (Dica: desenvolva pequenos programinhas de teste com eles para ajudar a descobrir os erros.) :

   (a)
   ```
   nProduto := 10
   c := 1
   DO WHILE ( c <= 5 )
       nProduto *= c
   ENDDO
   ++c
   ```

   (b)
   ```
   x := 1
   DO WHILE ( x <= 10 )
       x := x - 1
   ```

```
ENDDO
```

(c) O código seguinte deve imprimir os valores de 1 a 10.

```
x := 1
DO WHILE ( x < 10 )
    ? x++
ENDDO
```

# 6 TIPOS DERIVADOS

# 7 FUNÇÕES

Todas as funções em Harbour (isso vale para as outras linguagens também) foram construídas em um momento posterior a criação da linguagem. Outros programadores as desenvolveram, as testaram e, por fim, elas acabaram sendo distribuídas juntamente com a linguagem.

Ritchie nos adverte que uma função não faz parte essencial de uma linguagem de programação (KERNIGHAN; RITCHIE, 1986, p. 24).

#### 7.0.0.1 A anatomia de uma função

Uma função é composta basicamente por :

1. Um nome
2. Um par de parenteses
3. Uma lista de valores de entrada
4. Um retorno

Exemplos de funções

```
DATE()  // Imprime o dia de hoje
TIME()  //  Imprime a hora
```

Em muitos casos você necessita passar um "valor" para uma função (o nome técnico para esse "valor" é parâmetro). Quase sempre uma função retorna um valor que resulta do tipo de processamento que ela executa (esse outro "valor" recebe o nome de "valor de retorno da função" [1] ).

Observe o código a seguir, por exemplo :

```
x := 12.456
? INT( x )  // Imprime 12
```

Nesse exemplo, a função INT recebe como parâmetro um valor (no exemplo esse valor é 12.456, através da variável x) e retorna a sua parte inteira (12).

[1] É comum expressões como "a função *CalculaSalario* **retornou** um valor muito alto" ou "A função *Xyx* **retorna** nulo".

# 8 CLASSES DE VARIÁVEIS

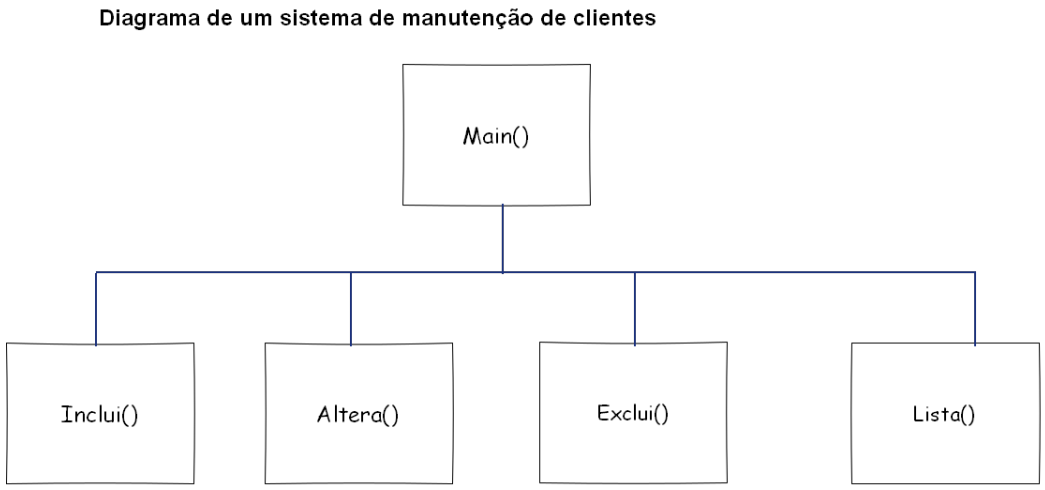
## 8.1 Classes básicas

Antes de passarmos para os outros tipos de variáveis (como o bloco de código, o array, o memorando, etc.) iremos ver as quatro classes de variáveis. Esse assunto é muito importante e, caso não seja bem compreendido, poderá influenciar negativamente todo o aprendizado a seguir. O Harbour possui quatro tipo de classe de variáveis :

1. Privada
2. Pública
3. Local
4. Estática

Antes de iniciarmos o estudo dessas classes, vamos criar um esboço de um programa relativamente complexo (figura 8.1). É bom ressaltar que um programa raramente é composto por apenas um bloco, mas devido a sua complexidade ele acaba sendo subdividido em inúmeras partes. Por exemplo, um cadastro de clientes pode possuir 5 blocos : menu principal, inclusão de clientes, alteração de clientes, exclusão de clientes e listagem de clientes.

Figura 8.1: Manuteção de clientes




Listing 8.1: Manutenção de clientes (Fonte)

```
1 /*
```

```
2  Manutenção de clientes (menu principal)
3  */
4  PROCEDURE MAIN()
5
6
7     ? "Manutenção de clientes"
8     cTitSis := "Manutenção de Clientes"
9
10 RETURN
11
12 /*
13 Inclusão de clientes
14 */
15 PROCEDURE INCLUI()
16
17
18    ? "Inclusão de clientes"
19    ? cTitSis
20
21 RETURN
22
23 /*
24 Alteração de clientes
25 */
26 PROCEDURE ALTERA()
27
28
29    ? "Alteração de clientes"
30    cTitSis
31
32 RETURN
33
34 /*
35 Exclusão de clientes
36 */
37 PROCEDURE EXCLUI()
38
39
40    ? "Exclusão de clientes"
41    cTitSis
42
43 RETURN
44
45 /*
46 Listagem de clientes
47 */
```

```
48 PROCEDURE LISTA()
49
50    ? "Listagem de clientes"
51    cTitSis
52
53 RETURN
```

### 8.1.1 Privada

Uma variável é da classe Privada quando o seu valor é

# 9 CONCLUSÃO

# REFERÊNCIAS BIBLIOGRÁFICAS


ASCENCIO, A. F. G.; CAMPOS, E. A. V. de. **Fundamentos da programação de computadores: algoritmos, PASCAL, C/C++ (padrão ANSI) e JAVA**. Pearson, São Paulo, 2014.

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

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

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

GUIMARãES, n. d. M.; LAGES, N. A. d. C. **Algoritmos e estruturas de dados**. LTC - Livros Técnicos e Científicos, Rio de Janeiro, 1994.

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.

MIZRAHI, V. V. **Treinamento em linguagem C.** McGraw-Hill, São Paulo, 1990.

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

ONLINE, F. C. **Fórum Clipper OnLine** . 2016.

PAPAS, C. H.; MURRAY, W. H. **Borland C++ 4**. Makron, Rio de Janeiro, 1994.

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

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

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

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

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

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

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

YOURDON, E. **Análise estruturada moderna**. Campus, Rio de Janeiro, 1992.

# APÊNDICE A – FLUXOGRAMAS

## A.1 Estruturas de controle

Figura A.1: Estrutura de seleção IF (Fluxograma da listagem 5.1)

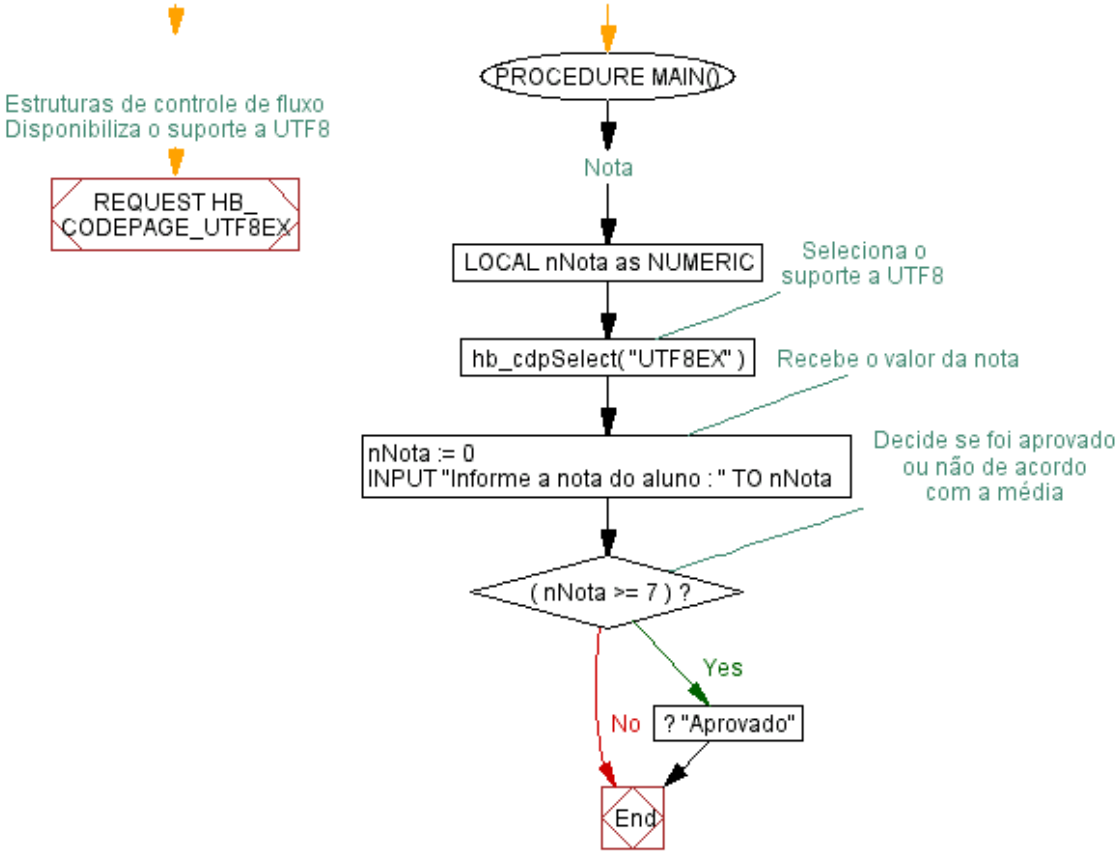
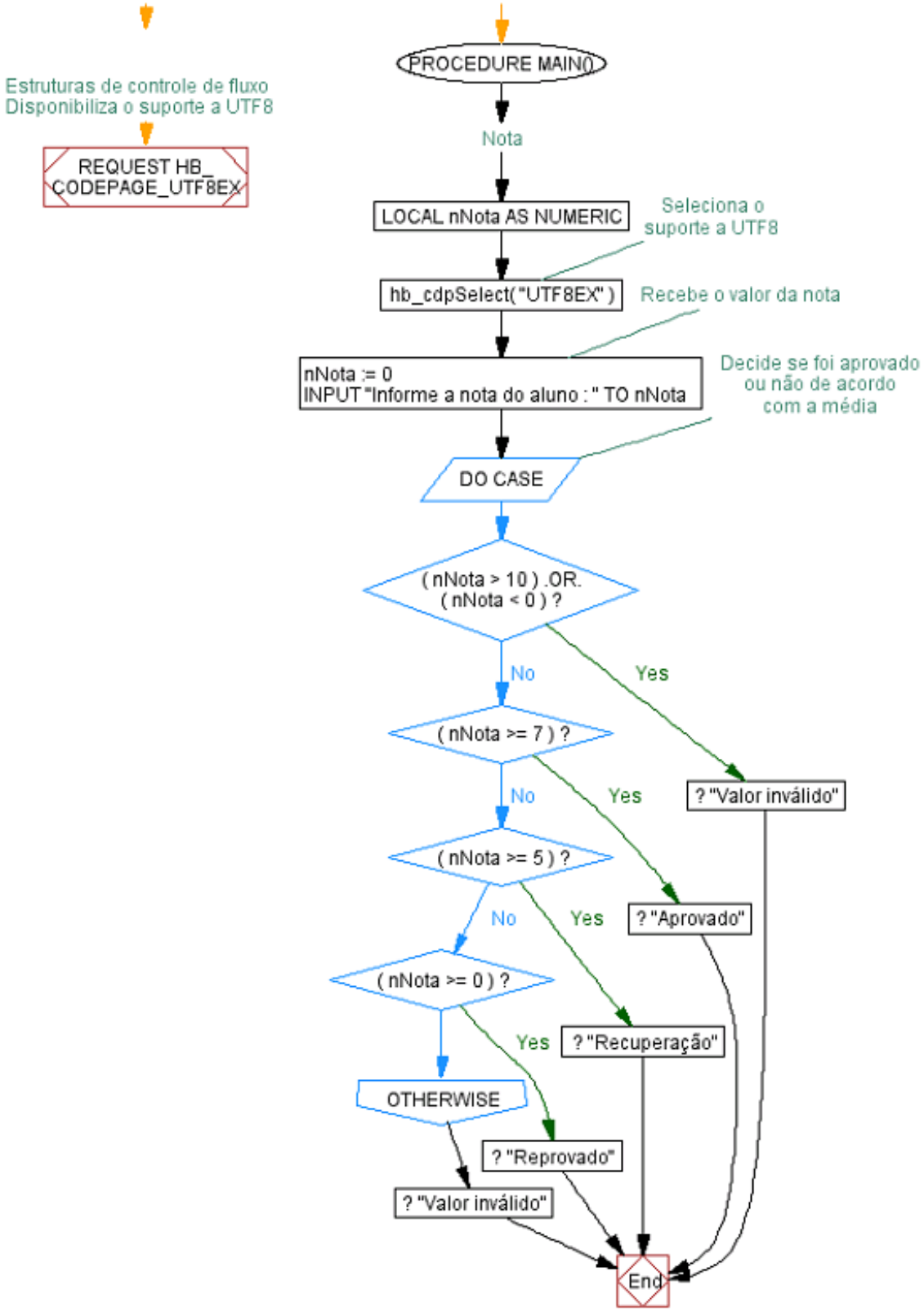


Figura A.2: Fluxograma de uma estrutura de seleção DO CASE ... ENDCASE da listagem 5.6)



# APÊNDICE B – PSEUDO-CÓDIGOS

## B.1 Estruturas de controle

Listing B.1: Estrutura de seleção IF (Algoritmo da listagem 5.1)

```
1  algoritmo "Estrutura de seleção IF"
2  // Seção de Declarações
3  var
4     nNota:real // Real significa que é um número real (não inteiro)
5  inicio
6  // Seção de Comandos
7     escreva( "Informe a nota do aluno" )
8     leia( nNota )
9     se nNota > 7
10       escreva "Aprovado"
11    fimse
12
13 fimalgoritmo
```

# APÊNDICE C – OS *SETS*

## C.1 Estruturas de controle

```
\#include "set.ch"
FUNCTION Main()

xAnt := Set(_SET_CONSOLE,.T.)  //SET CONSOLE ON

 ... Operações ...
Set(_SET_CONSOLE,xAnt )

... Restante do programa
```

(ONLINE, 2016)

# APÊNDICE D – RESUMO DE PROGRAMAÇÃO ESTRUTURADA

## D.1 Fluxogramas

Figura D.1: Estrutura de seleção IF (Seleção única)

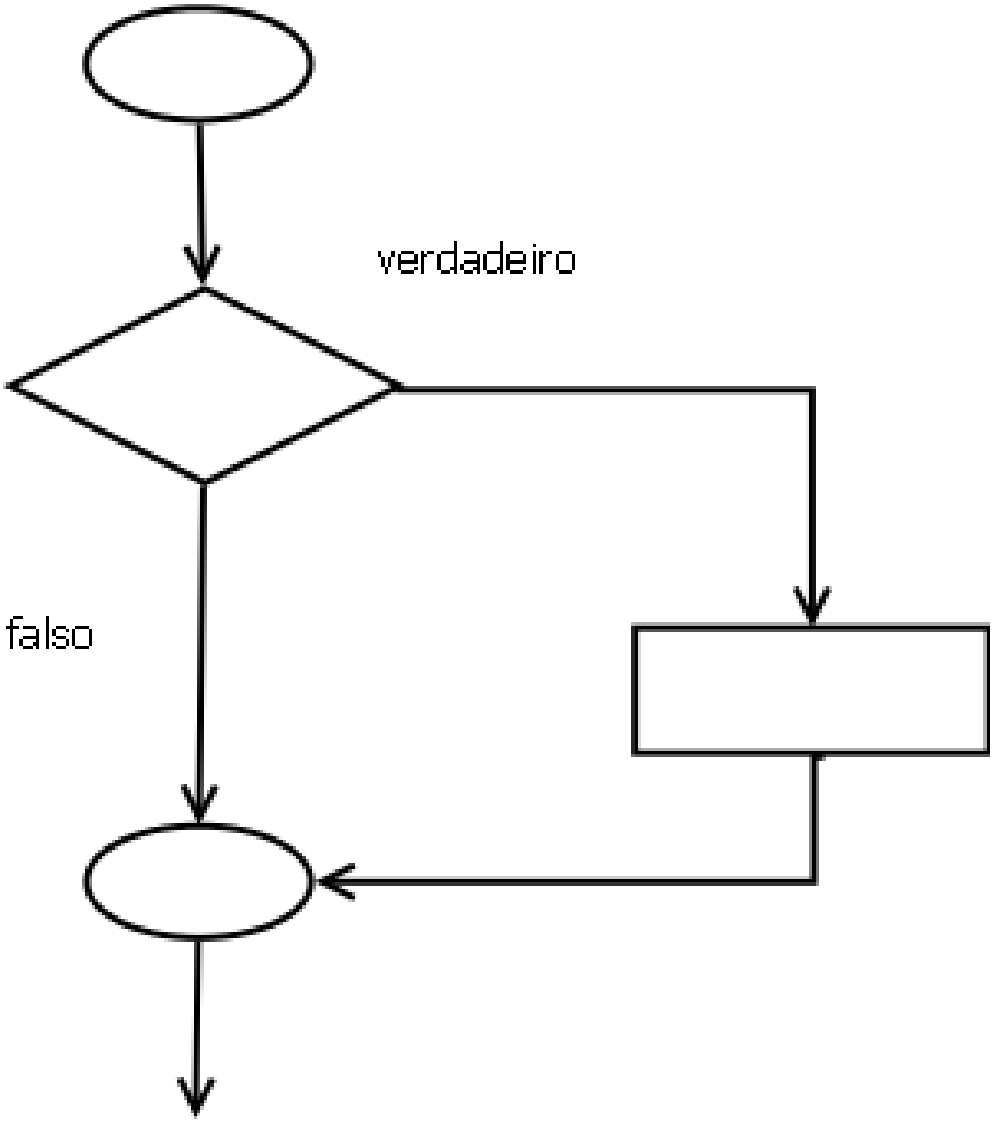


Figura D.2: Estrutura de seleção IF (Seleção dupla)

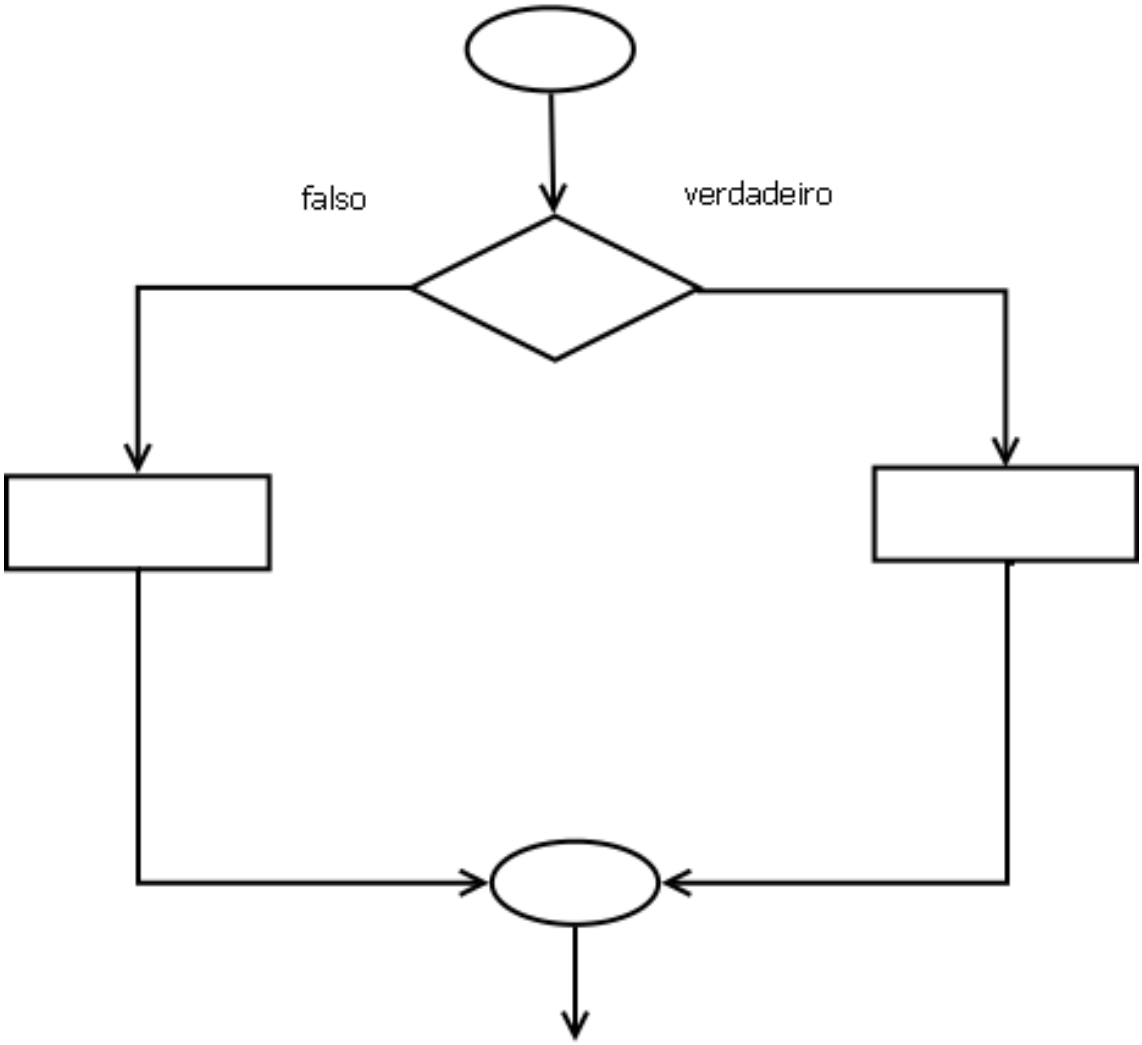


Figura D.3: Estrutura de seleção CASE (Seleção múltipla)

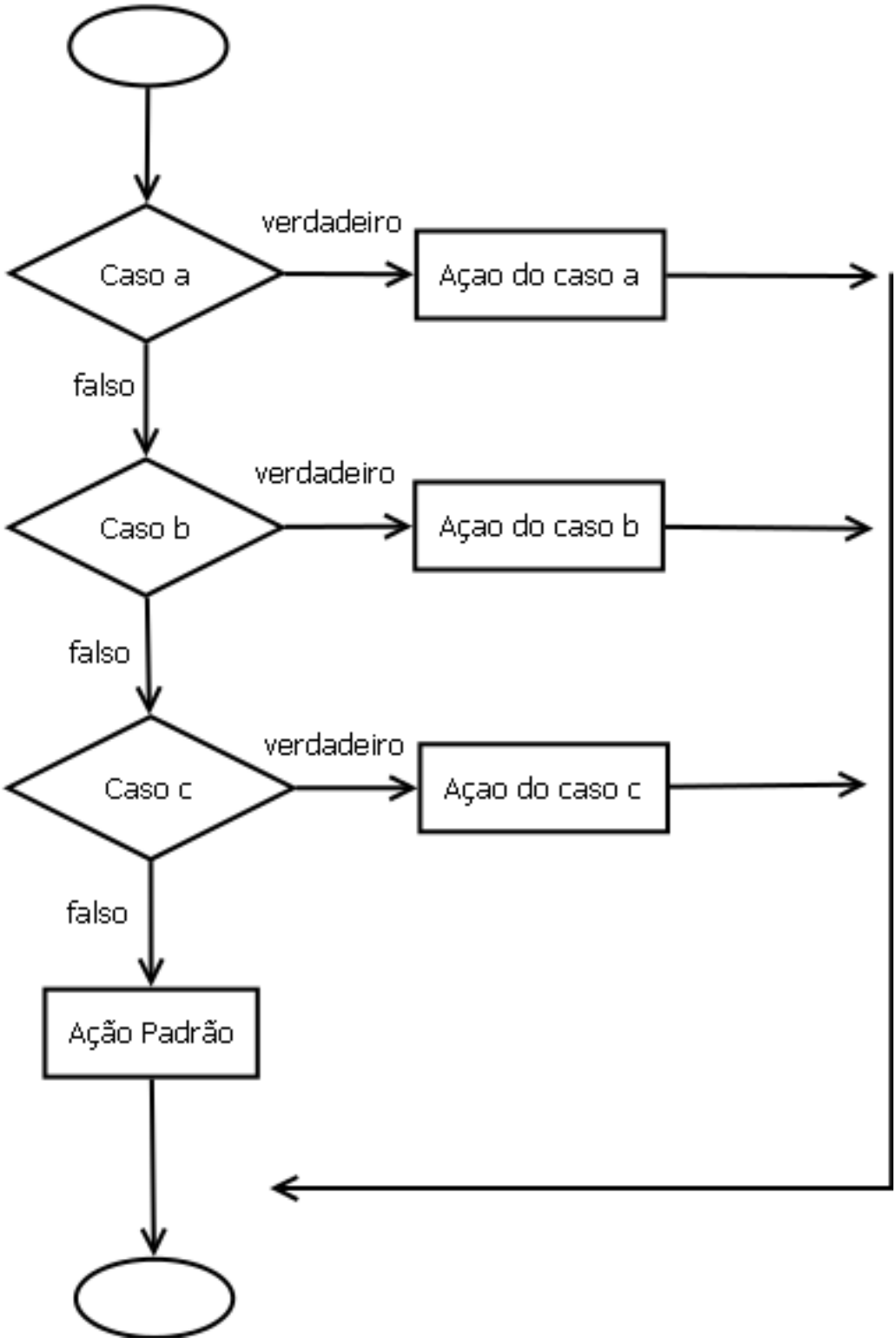


Figura D.4: Estrutura de repetição WHILE

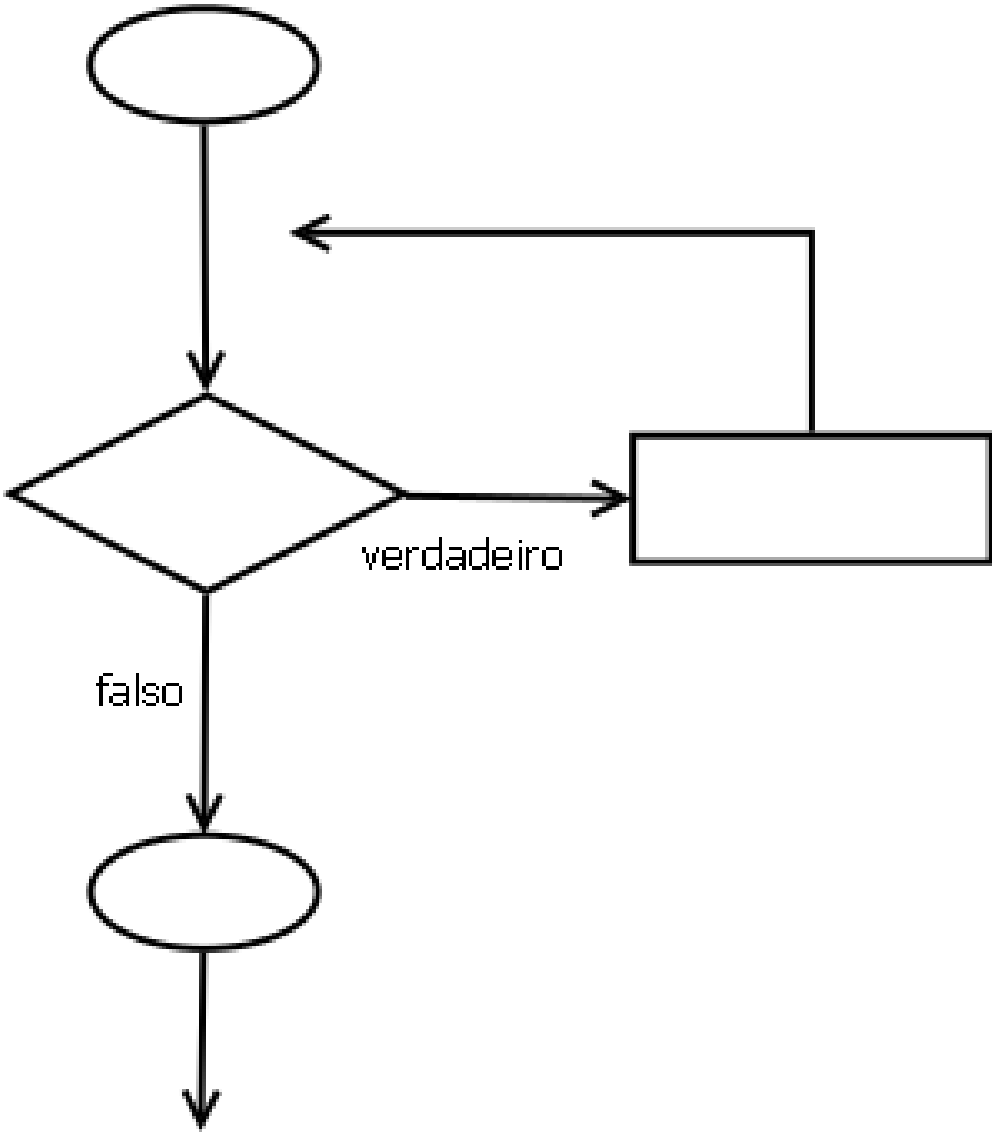


Figura D.5: Estrutura de repetição FOR

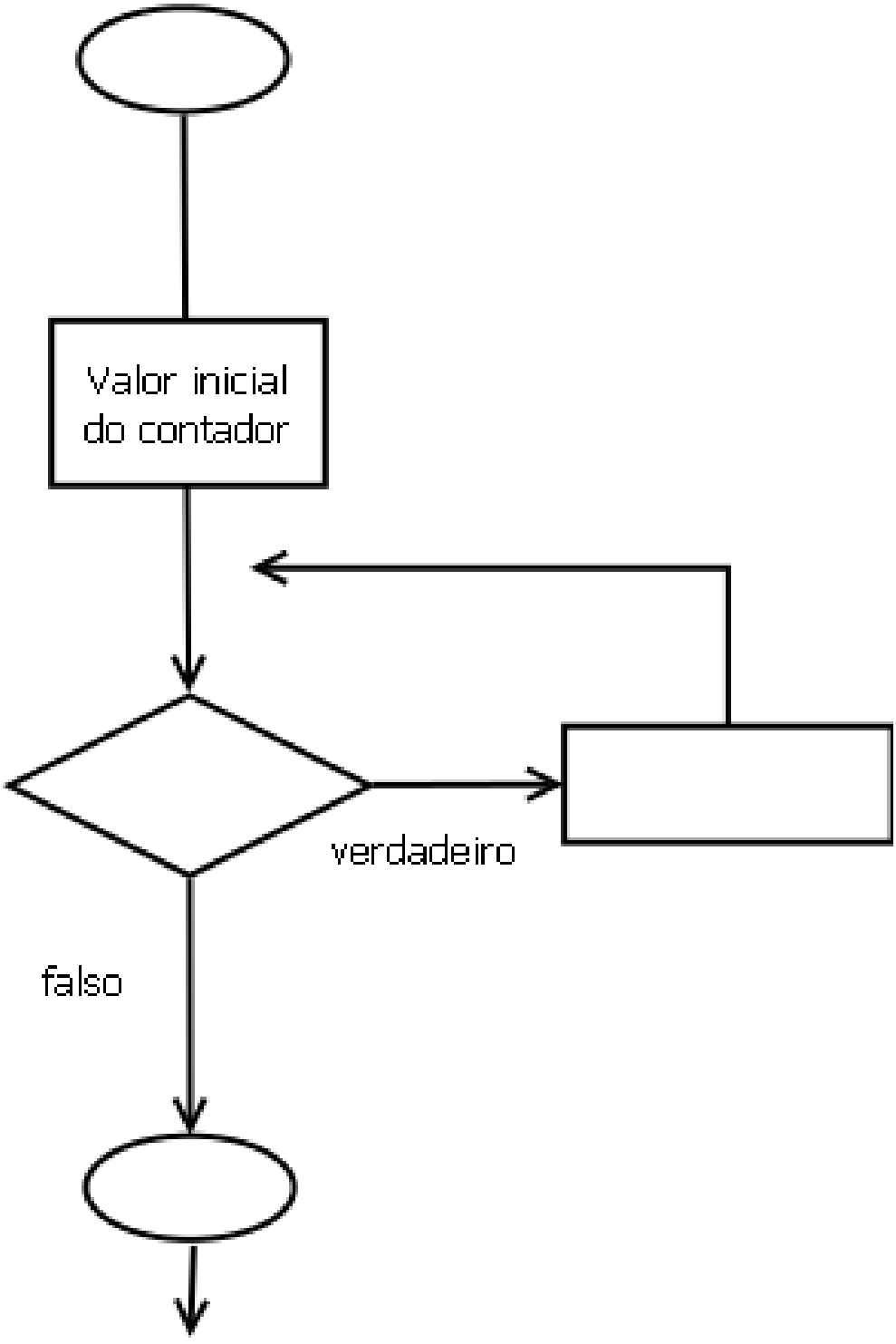


Figura D.6: Estrutura de repetição REPITA ATÉ