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.
60p.;il. color. enc.

VLADEMIRO LANDIM JUNIOR

INTRODUÇÃO A PROGRAMAÇÃO USANDO A LINGUAGEM HARBOUR.

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

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

Aos meus Pais.

•

Agradecimentos

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

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

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

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

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

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

Resumo

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

Abstract

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

Keywords: . . .

Lista de Figuras

Figura 1.1 Matriz coluna 17

Figura 1.2 Array com duas dimensões 17

Figura 1.3 Array com três dimensões 17

Figura 1.4 Array com uma dimensão (um vetor) 19

Figura 1.5 Um array pode ser usado para representar um tabuleiro de xadrez 32

Figura 1.6 Uma tabela com dados é um ente bidimensional 33

Figura 1.7 Um array unidimensional (ou vetor) 35

Figura 1.8 Um array multidimensional 36

Figura 1.9 Um array multidimensional (ou vetor) 36

Figura 1.10 Um array multidimensional do Harbour. 37

Lista de Tabelas

Sumário

1 TIPOS DERIVADOS 15

1.1 Introdução 16

1.2 O que é um array ? 16

1.3 Entendo os arrays através de algoritmos 18

1.3.1 Um algoritmo básico usando arrays 18

1.3.2 O problema da média da turma 19

1.4 Arrays em Harbour 21

1.4.1 Declarando arrays 22

1.4.2 Atribuindo dados a um array 23

1.4.3 Algumas operações realizadas com arrays 25

1.4.3.1 Percorrendo os itens de um array (Laço FOR) 26

1.4.3.2 Percorrendo os ítens de um array obtendo dinamicamente a sua quantidade de elementos (Função LEN) 27

1.4.3.3 Adicionando itens a um array dinamicamente (Função AADD) 28

1.4.4 Arrays unidimensionais e multidimensionais 32

1.4.5 Arrays são referências quando igualadas 37

1.4.5.1 A pegadinha da atribuição IN LINE 38

1.4.5.2 A pegadinha do operador de comparação 40

1.4.6 Arrays na passagem de parâmetros 42

1.4.7 Elementos de arrays na passagem de parâmetros 44

1.4.8 Retornando arrays a partir de funções 45

1.4.9 Lembra do laço FOR EACH ? . . . 46
1.4.10 Algumas funções que operam sobre arrays . . . 48
1.4.10.1AADD() . . . 48
1.4.10.2ACLONE() . . . 48
1.5 O Hash . . . 48
1.5.1 Criando um Hash . . . 48
1.5.1.1 Com a função HB_HASH() . . . 48
1.5.1.2 Sem a função HB_HASH() . . . 49
1.5.2 Percorrendo um Hash com FOR EACH . . . 49
1.5.3 As características de um hash . . . 51
1.5.3.1 AutoAdd . . . 51
1.5.3.2 Binary . . . 53
1.5.3.3 CaseMatch . . . 55
1.5.3.4 KeepOrder . . . 56
1.5.4 Funções que facilitam o manuseio de Hashs . . . 59
1.5.4.1 HB_hHashKey() . . . 59
1.5.4.2 HB_hGet() . . . 59
1.5.4.3 HB_hGetDef() . . . 59
1.5.4.4 HB_hSet() . . . 59
1.5.4.5 HB_hDel() . . . 59
1.5.4.6 HB_hPos() . . . 59
1.5.4.7 HB_hKeyAt() . . . 59
1.5.4.8 HB_hPairAt() . . . 59
1.5.4.9 HB_hDelAt() . . . 59
1.5.4.10HB_hKeys() . . . 59
1.5.4.11HB_hValues() . . . 59
1.5.4.12HB_hFill() . . . 59
1.5.4.13HB_hClone() . . . 59
1.5.4.14HB_hCopy() . . . 59
1.5.4.15HB_hMerge() . . . 59

1.5.4.16HB_hEval() 59
1.5.4.17HB_hScan() 59
1.5.4.18HB_hSort() 59
1.5.4.19HB_hAllocate() 59
1.5.4.20HB_hDefault() 59
REFERÊNCIAS BIBLIOGRÁFICAS 60

1 Tipos derivados

A contra-almirante Dra. Grace Murray Hopper (U.S. Navy) foi uma mulher notável, que progrediu grandemente nos desafios da programação dos primeiros computadores. Foi ela quem criou a linguagem de programação Flow-Matic, que serviu como base para a criação do COBOL. Ela acreditava que "sempre foi feito dessa maneira" não era necessariamente uma boa razão para continuar a fazer assim.

Anita Borg

Objetivos do capítulo

- Entender o que é um tipo derivado.
- Compreender o uso de arrays.
- Saber manipular um hash.
- Usar as principais funções manipuladoras de arrays e de hashs.

1.1 Introdução

Até agora nós vimos quatro tipos básicos de variáveis, são eles: numérico, caractere, data e lógico. Esses dados básicos possuem uma limitação: processam apenas um valor por vez, ou seja, se você declarar uma variável, ela só poderá conter um valor por vez. Existem casos, entretanto, em que você necessitará processar vários valores por variável. Para resolver esse problema, o Harbour possui dois tipos especiais que são chamados de tipos derivados de variáveis: o array[1] e o hash. O array é o tipo mais antigo, por isso mesmo nós o abordaremos primeiro. Em seguida nós estudaremos o hash, que é um tipo especial de array.

1.2 O que é um array ?

Um array é uma sequência de posições que se pode acessar diretamente através de uma variável e de um índice numérico. Aguilar define um array como sendo "um conjunto finito e ordenado de elementos" (AGUILAR, 2008, p. 228). Essa definição é simples e abrange tudo o que precisamos saber sobre arrays. Dizer que é um conjunto ordenado significa dizer que cada elemento de um array possui uma posição numérica. Por exemplo: primeiro elemento, segundo elemento, etc.

Para que as coisas fiquem mais claras, vamos usar uma analogia adaptada de (FORBELLONE; EBERSPACHER, 2005, p. 69): imagine um prédio com um número finito de andares e imagine também que cada andar possui apenas um apartamento. Contudo, não basta saber qual apartamento desejamos chegar se não soubermos o endereço do prédio. O que precisamos de antemão é o endereço do prédio e só então nos preocuparmos para qual daqueles apartamentos queremos ir. Interessa chegar no apartamento, mas antes o morador precisa do :

- endereço do prédio;
- número do apartamento.

A mesma coisa acontece com os arrays: o prédio em questão equivale a variável de memória que armazenará o array, o número do apartamento é um identificador que aponta para o elemento do array e o conteúdo do apartamento é o valor armazenado nesse elemento. Quando o prédio possui um apartamento por andar isso quer dizer que o array tem apenas uma dimensão (quando isso acontece ele também recebe o nome de **vetor**. Se você se lembra das aulas de matemática do ensino médio basta recordar o conceito de "matriz coluna", que é um tipo especial de matriz. A figura 1.1 ilustra esse conceito.

---

[1] Iremos usar o termo em inglês por ser amplamente aceito na computação. Outros termos usados são : matriz e vetor (que é um tipo especial de matriz com apenas uma dimensão).

Figura 1.1: Matriz coluna

$$M_{3 \times 1} = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{pmatrix}$$

Quando o prédio possui mais de um apartamento por andar dizemos que o array tem duas dimensões. As figura a seguir ilustra um array bidimensional.

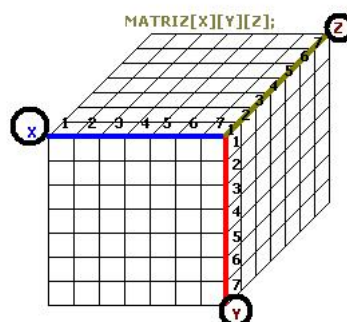
Figura 1.2: Array com duas dimensões

$$A = \begin{bmatrix} 2 & 3 & 5 & 6 \\ 4 & 2 & 1 & 1 \\ 5 & 1 & 2 & 3 \\ 6 & 1 & 3 & 2 \end{bmatrix}$$

A grande maioria dos conjuntos representados em computação possuem apenas duas dimensões. A programação voltada para sistemas de informação é, em sua maioria, composta de tais estruturas. Uma planilha eletrônica é um exemplo bem clássico.

Até agora podemos representar facilmente essas matrizes com lápis e papel, mas existem arrays que possuem três dimensões, cuja representação aproximada seria algo como o desenho a seguir :

Figura 1.3: Array com três dimensões



Existem também arrays cuja representação gráfica se torna impossível: são os arrays que possuem quatro ou mais dimensões. Tais estruturas são matematicamente possíveis, e as linguagens de programação podem representá-las, mas elas não podem “ser visualizadas” por nós em toda a sua completude, pois o nosso cérebro só consegue representar três dimensões no máximo.

Dica 1

É importante não confundir o índice com o elemento. O índice é a posição do elemento no array (o número do apartamento), enquanto o elemento é o que está contido no elemento do array (o conteúdo do apartamento) (FORBELLONE; EBERSPACHER, 2005, p. 71).

1.3 Entendo os arrays através de algoritmos

1.3.1 Um algoritmo básico usando arrays

Considere o problema a seguir:

Criar um programa que efetue a leitura dos nomes de 20 pessoas e em seguida apresente-os na mesma ordem em que foram informados.

O algoritmo exige os seguintes passos :

- Definir o array NOME com 20 elementos.
- Iniciar o programa, fazendo a leitura dos 20 nomes.
- Apresentar após a leitura, os 20 nomes.

O algoritmo, em pseudo-código, seria assim:

Algoritmo 1: Coleta e exibição de 20 nomes

```
  Entrada:
  Saída:
1 início
2 │ NOME ← array( 20 ) // Array NOME
3 │ para contador de 1 até 20 faça
4 │ │ leia NOME[ contador ]
5 │ fim
6 │ para contador de 1 até 20 faça
7 │ │ escreva NOME[ contador ]
8 │ fim
9 fim
```

Se você está buscando entender através da representação de matriz que você viu no ensino médio, talvez você esteja estranhando o exemplo acima. Isso porque, mesmo uma matriz

coluna (uma matriz unidimensional) tem os elementos da coluna representados, conforme a figura a seguir :

Figura 1.4: Array com uma dimensão (um vetor)

$$M_{3 \times 1} = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{pmatrix}$$

Já a representação de um elemento de uma matriz unidimensional (um vetor) possui apenas a representação da linha, porque representar a coluna seria desnecessário. Se nós fossemos imprimir os elementos da matriz da figura 1.4 usando um pseudo-código, a representação seria algo como :

```
? a[1]  , a[2], a[3]

e não

? a[1][1], a[2][1], a[3][1]
```

A grande vantagem do array é que nós precisamos criar apenas uma variável que represente todo um conjunto de elementos. Note que é fácil recuperar os elementos gravados, basta percorrer o array através de um laço. Arrays e laços (estruturas de repetição) são dois elementos das linguagens de programação que trabalham juntos.

1.3.2 O problema da média da turma

Considere o seguinte problema:

Um colégio deseja saber quantos alunos, de um grupo de 10 alunos, tiraram uma nota superior a média da turma.

Primeiramente nós vamos tentar resolver esse problema **sem usar arrays**. Veja que a solução não é tão fácil quanto parece.

Algoritmo 2: Cálculo da média aritmética de 10 notas

```
    Entrada:
    Saída:
 1  início
 2  |  média ← 0 // Média da turma
 3  |  nota ← 0 // Nota do aluno
 4  |  contador ← 0 // Contador
 5  |  acumulador ← 0 // Acumulador
 6  |  enquanto contador < 10 faça
 7  |  |  leia nota
 8  |  |  contador ← contador + 1
 9  |  |  acumulador ← acumulador + nota
10  |  fim
11  |  média ← acumulador / 10
12  |  escreva “A média anual da turma é” , média
13  |  .
14  |  .
15  |  // E agora ?????
16  |  // Como eu vou calcular quantos alunos tiraram a nota maior que
17  |  // a média da turma ?
18  |  // Eu não tenho mais a nota de cada aluno. Elas se perderam
19  |  // durante o processamento do laço “enquanto”.
20  fim
```

Temos um problema: eu só posso contar quantos alunos obtiveram uma nota acima da média da turma logo após o cálculo da média (linha 11). Mas como eu posso comparar, se o valor da nota do aluno é sempre sobreposto pelo aluno seguinte ? (linha 7).

Quando surge um problema dessa natureza os arrays surgem como a única solução prática. Em vez de termos apenas uma variável chamada *nota* que seria sobreposta a cada iteração, nós teríamos um array chamado *nota* com dez posições, onde cada posição equivaleria a um aluno. Em vez de usar o laço *enquanto* (WHILE) eu passaria a usar o laço *para* (FOR).

Algoritmo 3: Cálculo da média aritmética de 10 notas usando vetores

```
    Entrada:
    Saída:
 1  início
 2  |  média ← 0 // Média da turma
 3  |  nota ← array( 10 ) // Array nota do aluno
 4  |  acumulador ← 0 // Acumulador
 5  |  notaAcima ← 0 // Contador de notas acima da média
 6  |  para contador de 1 até 10 faça
 7  |  |  leia nota[ contador ]
 8  |  |  acumulador ← acumulador + nota
 9  |  fim
10  |  média ← acumulador / 10
11  |  escreva "A média anual da turma é" , média
12  |  .
13  |  // laço para verificar valores que estão acima da média
14  |  para contador de 1 até 10 faça
15  |  |  se nota[ contador ] > média então
16  |  |  |  notaAcima ← notaAcima + 1 //
17  |  fim
18  |  escreva "Número de valores acima da média" , notaAcima
19  fim
```

Agora que você já tem a definição de arrays através do algoritmo, vamos passar para a definição do array no Harbour. Ao final da próxima seção você estará apto a implementar o algoritmo acima usando Harbour.

1.4 Arrays em Harbour

As primeiras versões do array foram criadas pelo dBase, ainda na década de 1980. Quando o Clipper surgiu o conceito foi expandido para dar suporte a arrays de várias dimensões. Como muitos problemas de empresas que os computadores tratam dizem respeito a dados que podem ser agrupados para melhor compreensão, não é surpreendente que as características dos arrays no Clipper 5.0 tenham se tornado um dos mais poderosos conceitos da linguagem (HEIMENDINGER, 1994, p. 99).

Como nós já vimos no pseudo-código, os arrays são simplesmente conjuntos de variáveis de memória com um nome comum. Elementos individuais são endereçados usando índices. São números que aparecem entre colchetes após o nome do array. O fragmento a seguir ilustra o que foi dito:

```
LOCAL choices[ 10 ] // Declara um array de tamanho 10
```

Da mesma forma que as variáveis de memória, você precisa atribuir valores para os elementos antes de usá-los. O fragmento de código a seguir mostra como se faz isso:

```
choices[ 3 ] := 20
```

Quando queremos nos referir ao terceiro elemento desse array, podemos fazer conforme o fragmento a seguir:

```
? choices[ 3 ] // Imprime o terceiro elemento do array.
```

1.4.1 Declarando arrays

Portanto, um array é uma variável, igual a todas que estudamos anteriormente, por isso ela precisa ser declarada e inicializada. As regras de classe de variável são válidas para arrays tembém, e isso quer dizer que podemos ter os tipos LOCAL, STATIC, PRIVATE e PUBLIC.

Porém, diferente das outras variáveis, o array precisa obrigatoriamente ser inicializado antes de ser usado. Isso porque você precisa informar ao Harbour que a variável é um array e não uma variável comum. O exemplo a seguir (listagem 1.1) exemplifica algumas formas de se inicializar um array. Note que nós preferimos declarar os arrays como locais, mas as regras de declaração são as mesmas das variáveis comuns.

Listing 1.1: Declarando e inicializando arrays

```
1 /*
2 Arrays
3 */
4 PROCEDURE Main
5 LOCAL aClientes := ARRAY( 2 )
6 LOCAL aSalarios := {}
7
8 RETURN
```

O exemplo acima declara e inicializa dois arrays[2]. O primeiro array (linha 5) possui dois elementos, ou seja, ele poderá conter dois valores de qualquer tipo (no caso específico, dois clientes). O segundo array (linha 6) não possui elementos e não pode receber nenhuma atribuição. A maioria das pessoas (principalmente programadores de outras linguagens) acham que a primeira forma é a mais usada, mas a maioria dos códigos em Harbour, que usam arrays,

[2]De agora em diante nós chamaremos as “variáveis arrays” simplesmente de “arrays”.

preferem a segunda forma de inicialização. Isso porque, geralmente, nós não sabemos de antemão quantos elementos o array irá possuir, então nós inicializamos um array vazio para, só depois, ir adicionando elementos.

Você também pode inicializar um array com valores pré-definidos, conforme o fragmento abaixo:

```
LOCAL aNotas := {  10 , 20 , 45 , 10 }
```

Lembre-se, o fragmento a seguir também está correto. Porém, se você notar, o array só foi declarado, mas não inicializado.

```
LOCAL choices[ 10 ] // Declara um array de tamanho 10
```

Dica 2

Apesar de ser a forma preferida pelos programadores Harbour, inicializar um array com zero elementos nos trás um problema de performance: caso eu não saiba quantos elementos ele irá possuir, eu posso ir adicionando os elementos de um por um, a medida que eu vou precisando deles. É aí que está o problema: eu sempre vou ter que criar um elemento no final do array, e essa operação consome um recurso extra de processamento. Se você não souber quantos elementos o array conterá, você deve tentar descobrir o total logo de uma vez e usar o a função ARRAY, mas como isso geralmente não é possível, você terá que usar a função AADD para ir adicionando elemento por elemento. Veremos esses exemplos nas seções seguintes.

Outro fato digno de nota é a nomenclatura usada. Um array pode conter vários tipos diferentes de dados, por exemplo: o primeiro elemento de um array pode ser uma string, o segundo elemento pode ser uma data, etc. Portanto, não faz sentido prefixar um array com "c", "n", "l" ou "d", já que um mesmo array pode conter variáveis de diversos tipos. A solução encontrada foi prefixar o array com a letra "a" de array.

Dica 3

As regras de nomenclatura de um array são as mesmas regras de qualquer variável, porém nós iremos prefixar um array com a letra "a". Habitue-se a fazer isso nos seus programas, isso irá torná-los mais claros e fáceis de se entender.

1.4.2 Atribuindo dados a um array

Agora que nós já aprendemos a declarar e a inicializar um array vamos aprender a inserir dados nele. Gravar dados em um array é muito parecido com inicializar uma variável,

a única diferença é que você deve informar um dado adicional chamado de índice. O índice é sempre informado entre colchetes. Veja no exemplo a seguir (listagem 1.2).

Listing 1.2: Inicializando arrays

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aClientes := ARRAY( 2 )

    aClientes[ 1 ] := "Claudio Soto"
    aClientes[ 2 ] := "Roberto Linux"

    ? aClientes[ 1 ]
    ? aClientes[ 2 ]

RETURN
```

.:Resultado:.

```
Claudio Soto
Roberto Linux
```

O índice é aquele número que se encontra entre os colchetes logo após o nome da variável array. Ele identifica o elemento do array que esta sendo usado. O array foi declarado com o tamanho de dois elementos (linha 5), dessa forma, você precisa informar o índice do array para poder atribuir (linhas 7 e 8) ou acessar (linhas 10 e 11) os dados.

Dica 4

Os índices do array iniciam sempre com o valor 1. Nem todas as linguagens são assim. Na realidade, a maioria das linguagens iniciam o array com o valor zero. No Harbour, o índice do último elemento do array equivale a quantidade de elementos do array, mas em linguagens como C, C++, PHP, Java e C#, o tamanho do array sempre é o índice do último elemento mais um, pois os arrays nessas linguagens iniciam com o índice igual a 0.

Cuidado. Você deve ter cuidado quando for referenciar ou atribuir dados a um array, pois o índice precisa estar no intervalo aceitável. Por exemplo: na listagem 1.2 o array aClientes possui dois elementos. Assim, você só poderá trabalhar com dois índices (1 e 2). O exemplo a seguir (listagem 1.3) nos mostra o que acontece quando atribuímos o índice 3 a um array que só tem tamanho 2.

Listing 1.3: Inicializando arrays (ERRO)

```
/*
Arrays
```

```
3  */
4  PROCEDURE Main
5  LOCAL aClientes := ARRAY( 2 )
6
7      aClientes[ 1 ] := "Claudio Soto"
8      aClientes[ 2 ] := "Roberto Linux"
9      aClientes[ 3 ] := "Elemento fora do intervalo"
10
11 RETURN
```

.:Resultado:.

```
Error BASE/1133  Bound error: array assign
Called from MAIN(9)
```

Erros do tipo “array assign” ocorrem quando você tenta atribuir um valor a um elemento de array com um índice que está além dos seus limites. Tenha muito cuidado, pois esses erros acontecem sempre em tempo de execução e nem mesmo o compilador pode gerar algum aviso (warning) sobre eles.

Dica 5

O Harbour implementa um array de uma forma diferente da maioria das linguagens. Formalmente, as outras linguagens definem o array com um tamanho fixo, e também não permitem tipos de dados diferentes por array. O Harbour também define um array com um tamanho fixo, mas o seu tamanho pode ser aumentado (ou diminuído) dinamicamente. Durante a execução, o array pode crescer e encolher. E, o número de dimensões que um array pode ter é ilimitado. Além disso, o Harbour permite tipos diferentes de dados dentro de um mesmo array. Em resumo:

- Arrays em Harbour: são dinâmicos e permitem vários tipos de dados por array.
- Arrays na maioria das linguagens: são estáticos e permitem apenas um tipo de dado por array.

1.4.3 Algumas operações realizadas com arrays

Os exemplos a seguir abordarão algumas operações comuns que são realizadas com o array.

1.4.3.1 Percorrendo os itens de um array (Laço FOR)

Uma vez criado o array você pode usar o laço FOR para percorrer os seus elementos facilmente. O exemplo a seguir (listagem 1.4) calcula o quadrado dos cem primeiros números inteiros sem utilizar arrays.

Listing 1.4: Obtendo o quadrado dos cem primeiros inteiros

```
1  /*
2  Arrays
3  */
4  PROCEDURE Main
5  LOCAL x
6
7      FOR x := 1 TO 100
8          ? "Quadrado de ", x , " é " , x ^ 2
9      NEXT
10
11 RETURN
```

De acordo com Spence, "FOR loops são uma maneira excelente de processar vetores." (SPENCE, 1991, p. 153). Veja, a seguir, as últimas linhas do programa ao ser executado.

.:Resultado:.

```
Quadrado de          91  é        8281.00
Quadrado de          92  é        8464.00
Quadrado de          93  é        8649.00
Quadrado de          94  é        8836.00
Quadrado de          95  é        9025.00
Quadrado de          96  é        9216.00
Quadrado de          97  é        9409.00
Quadrado de          98  é        9604.00
Quadrado de          99  é        9801.00
Quadrado de         100  é       10000.00
```

Mas nós temos problema no nosso código (não é um erro de execução nem um erro de lógica). Imagine que você precise dos respectivos valores calculados para uma operação posterior. Note que você não tem mais esses valores após o laço finalizar. É para isso que servem os arrays: para manter na memória do computador os valores de um processamento anterior que envolva múltiplas ocorrências de uma mesma variável.

O programa a seguir (listagem 1.5) resolve isso com arrays.

Listing 1.5: Armazenando valores para uma operação posterior

```
1  /*
2  Arrays
3  */
```

```
4  #define ELEMENTOS 100
5  PROCEDURE Main
6  LOCAL aQuadrado := ARRAY( ELEMENTOS )
7  LOCAL x
8
9      FOR x := 1 TO ELEMENTOS
10         aQuadrado[ x ] := x ^ 2
11     NEXT
12
13     // Outras operações
14
15     // Agora ainda tenho os 100 primeiros quadrados
16     // graças ao array aQuadrado
17     FOR x := 1 TO ELEMENTOS
18         ? "Quadrado de ", x , " é " , aQuadrado[ x ]
19     NEXT
20
21 RETURN
```

.:Resultado:.

```
Quadrado de        91  é       8281.00
Quadrado de        92  é       8464.00
Quadrado de        93  é       8649.00
Quadrado de        94  é       8836.00
Quadrado de        95  é       9025.00
Quadrado de        96  é       9216.00
Quadrado de        97  é       9409.00
Quadrado de        98  é       9604.00
Quadrado de        99  é       9801.00
Quadrado de       100  é      10000.00
```

1.4.3.2 Percorrendo os ítens de um array obtendo dinamicamente a sua quantidade de elementos (Função LEN)

Vamos continuar com o exemplo da listagem 1.5 para aprimorarmos o nosso conhecimento sobre os arrays. O programa já acumula o processamento na forma de arrays, mas ele possui mais uma limitação: ele processa uma quantidade fixa de elementos (sempre cem elementos). Caso você queira mudar isso você precisa modificar o programa e compilar de novo para que um novo executável seja gerado. O que é impraticável em situações da vida real. Vamos, então, realizar uma pequena mudança no nosso programa. Você, com certeza, deve estar lembrado da função LEN, que é usada para nos informar a quantidade de caracteres de uma string. Pois bem, a função LEN pode ser usada com arrays também, e ela retorna a quantidade de elementos do array. O programa a seguir aprimora o programa da listagem 1.5 para que ele

processe uma quantidade informada em tempo de execução.

Listing 1.6: Armazenando N valores para uma operação posterior

```
/*
Arrays
*/
PROCEDURE Main
LOCAL aQuadrado
LOCAL x

    CLS
    INPUT "Informe a quantidade de elementos : " TO x

    aQuadrado := ARRAY( x )

    FOR x := 1 TO LEN( aQuadrado )
        aQuadrado[ x ] := x ^ 2
    NEXT

    FOR x := 1 TO LEN( aQuadrado )
        ? "Quadrado de ", x , " é " , aQuadrado[ x ]
    NEXT

RETURN
```

A seguir temos um exemplo quando o usuário digita 10.

.:Resultado:.

```
Informe a quantidade de elementos : 10
Quadrado de          1  é          1.00
Quadrado de          2  é          4.00
Quadrado de          3  é          9.00
Quadrado de          4  é         16.00
Quadrado de          5  é         25.00
Quadrado de          6  é         36.00
Quadrado de          7  é         49.00
Quadrado de          8  é         64.00
Quadrado de          9  é         81.00
Quadrado de         10  é        100.00
```

1.4.3.3 Adicionando itens a um array dinamicamente (Função AADD)

Vamos prosseguir do exemplo da seção anterior (listagem 1.6). Note que ele exige que eu saiba a quantidade de elementos do array previamente para, só depois, fazer o cálculo.

Mas, nós já vimos quando estudamos os loops, que essa não é a única forma de processamento de dados em laços. Existe um caso em que você não sabe quantos elementos irá processar e a decisão precisa ser tomada no interior do laço. Ou seja, você precisa adicionar elementos ao seu array dinamicamente, pois você não sabe (de antemão) quantos elementos ele terá. Outra limitação é que os valores sempre começam com 1 e vão até um valor pré-determinado por você, sempre em ordem crescente. E se o usuário precisasse calcular uma quantidade de valores quaisquer ?

O exemplo da listagem 1.7 ilustra essa situação usando uma função chamada AADD. Essa função adiciona um elemento ao final do array.

Listing 1.7: Armazenando N valores para uma operação posterior

```
1  /*
2  Arrays
3  */
4  PROCEDURE Main
5  LOCAL aQuadrado := {}
6  LOCAL x
7  LOCAL nValor
8
9
10     DO WHILE .T.
11        INPUT "Informe o elemento (999 FINALIZA) " TO nValor
12        IF nValor == 999
13           EXIT
14        ENDIF
15        AADD( aQuadrado , nValor )
16     ENDDO
17
18     FOR x := 1 TO LEN( aQuadrado )
19         ? "O quadrado de " , aQuadrado[x] , " é " , ( aQuadrado[x]
              ^ 2 )
20     NEXT
21
22 RETURN
```

A seguir temos um exemplo quando o usuário digita 3 valores quaisquer.

.:Resultado:.

```
Informe o elemento (999 FINALIZA) 6
Informe o elemento (999 FINALIZA) 100
Informe o elemento (999 FINALIZA) 549
Informe o elemento (999 FINALIZA) 999
O quadrado de           6  é          36.00
O quadrado de         100  é       10000.00
O quadrado de         549  é      301401.00
```

Note que o array é declarado e inicializado, na linha 5, como um array vazio. A função AADD é usada na linha 15. O primeiro parâmetro é o array e o segundo é o valor que o elemento que será inserido irá ter. A função AADD está descrita a seguir.

Descrição sintática 1

1. Nome : AADD()
2. Classificação : função.
3. Descrição : Adiciona um novo elemento ao final de um array.
4. Sintaxe

```
AADD( <aArray>, [<expValor>] ) -> Valor
```

Fonte : (NANTUCKET, 1990, p. 5-1)

Note que AADD avalia <expValor> e retorna o seu valor. Caso <expValor> não seja especificado, AADD retorna NIL. Caso <expValor> seja um outro array, o novo elemento no array destino conterá uma referência[3] ao array especificado por <expValor> (NANTUCKET, 1990, p. 5-1).

Dica 6

Lembre-se:
Você pode formar arrays simplesmente escrevendo conforme o fragmento abaixo. Essa forma chama-se *array literal*.

```
LOCAL aEscolha := { "Adiciona", "Edita", "Relatório" }
```

É o mesmo que escrever :

```
LOCAL aEscolha[ 3 ]

  aEscolha[1]  := { "Adiciona" }
  aEscolha[2] :=  { "Edita"}
  aEscolha[3] :=  { "Relatório" }
```

Que equivale a

[3]Veremos o que é referência nas seções posteriores, ainda nesse capítulo. Fique atento.

```
LOCAL aEscolha := ARRAY( 3 )

  aEscolha[1]  := { "Adiciona" }
  aEscolha[2] :=  { "Edita"}
  aEscolha[3] :=  { "Relatório" }
```

Que, também é a mesma coisa que :

```
LOCAL aEscolha := {}

  AADD( aEscolha,  "Adiciona" )
  AADD( aEscolha,  "Edita" )
  AADD( aEscolha,  "Relatório" )
```

Dica 7

Na listagem 1.7 nós usamos um laço FOR em conjunto com a função LEN para processar um array. Veja um fragmento de código semelhante:

```
    FOR x := 1 TO LEN( aArray )
       ? aArray[ x ]
    NEXT
```

Isso está correto, mas nós temos um pequeno problema de performance no laço FOR. Isso ocorre porque a função LEN() será processada várias vezes, obrigando o Harbour a reavaliar os limites do loop durante toda a iteração. O ideal seria evitar o uso de funções no cabeçalho de laços ou dentro de laços (nem sempre isso é possível, mas se for possível, lhe renderá ganhos de tempo).
A função LEN(), nesse caso, poderia ser colocada fora do laço FOR.

```
    nTamanho :=  LEN( aArray )
    FOR x := 1 TO nTamanho
       ? aArray[ x ]
    NEXT
```

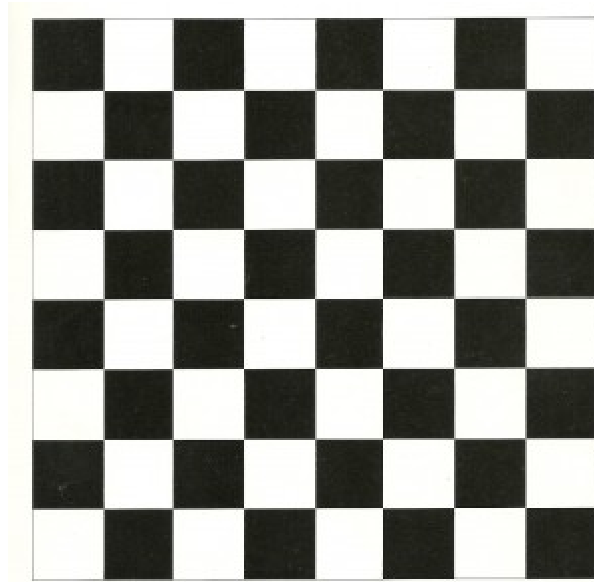
Atenção: nem sempre a função de avaliação pode ser colocada fora do laço. Você precisa analisar caso a caso.

1.4.4 Arrays unidimensionais e multidimensionais

O equivalente matemático dos arrays são as matrizes que você viu durante o ensino médio. Até agora nós trabalhamos somente com arrays unidimensionais, cujo equivalente matemático seria uma matriz coluna.

Vimos que os arrays podem ser usados para representarem objetos com mais de uma dimensão. Por exemplo, a figura 1.5 possui duas dimensões.

Figura 1.5: Um array pode ser usado para representar um tabuleiro de xadrez



A figura 1.5 poderia ser representada através do seguinte array:

```
LOCAL aChess[ 8 ][ 8 ] // Tabuleiro de xadrez
```

Dica 8

Arrays bidimensionais são os mais usados em qualquer linguagem de programação e servem para representar uma variedade de objetos do mundo real: uma planilha eletrônica, um cardápio de lanchonete, assim como qualquer tabela com dados.

Figura 1.6: Uma tabela com dados é um ente bidimensional

O Harbour implementa o array aChess de uma forma bem incomum: é como se cada elemento do array principal (de 8 posições) fosse, ele também, um array de 8 posições. Quando um elemento de um array é um array também, podemos dizer que se trata de um array multidimensional.

O array aChess poderia ser representado da seguinte forma :

```
aChess := { { "a11", "a12", "a13", "a14", "a15", "a16", "a17", "a18" } ,;
            { "a21", "a22", "a23", "a24", "a25", "a26", "a27", "a28" } ,;
            { "a31", "a32", "a33", "a34", "a35", "a36", "a37", "a38" } ,;
            { "a41", "a42", "a43", "a44", "a45", "a46", "a47", "a48" } ,;
            { "a51", "a52", "a53", "a54", "a55", "a56", "a57", "a58" } ,;
            { "a61", "a62", "a63", "a64", "a65", "a66", "a67", "a68" } ,;
            { "a71", "a72", "a73", "a74", "a75", "a76", "a77", "a78" } ,;
            { "a81", "a82", "a83", "a84", "a85", "a86", "a87", "a88" } }
```

Para inserir um valor em um elemento você também pode fazer assim :

```
aChess[ 5 ][ 2 ] := 12 // Substitui a string‘‘a52’’ por 12
```

Note que tais representações acima se assemelham as matrizes estudadas no ensino médio. Se fossemos representar o elemento do array acima conforme a notação usada no ensino de matrizes, ela seria assim : $aChess_{5,2}$.

Dica 9

A representação de arrays multidimensionais obedece ao mesmo dos arrays bidimensionais vistos até agora. O fragmento a seguir declara um array com quatro dimensões.

```
LOCAL aProperty[ 4 ][ 3 ][ 2 ][ 7 ]
```

Você também pode usar a função AADD para ir montando o seu array multimensional, conforme o fragmento abaixo:

```
LOCAL aProperty := {}

AADD( aProperty , {} ) // O primeiro elemento é um array
AADD( aProperty , 23 ) // O segundo elemento é um número

AADD( aProperty[ 1 ] , 12 )
AADD( aProperty[ 1 ] , 13 )
```

O resultado final é algo como :

```
aProperty[ 1 ][ 1 ] é igual a 12.
aProperty[ 1 ][ 2 ] é igual a 13.
aProperty[ 2 ] é igual a 23.
```

Essa forma de "ir montando" um array confere ao Harbour uma flexibilidade maior, pois nós não precisamos criar arrays rigidamente retangulares.

Dica 10

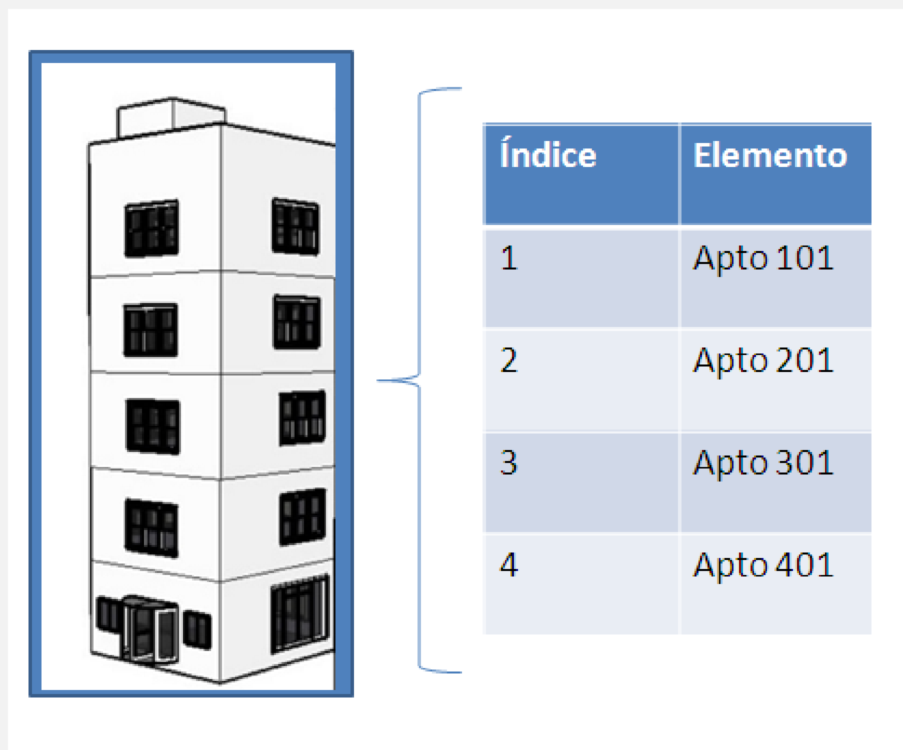
Você deveria anotar o fato de que muitas linguagens impõem uma rígida estrutura sobre os arrays. Cada elemento em uma dimensão deve ter o mesmo tamanho e tipo de todos os seus pares, e o número de elementos em qualquer dimensão é o mesmo para todo array. Em outras palavras, se você tem um array com 10 elementos na primeira dimensão, 20 na segunda e 5 na terceira, ele terá um array de 10x20x5 contendo 1000 elementos. [...] O fato mais significativo sobre arrays [...] é que qualquer elemento do array pode ser ele próprio um array. **Esse conceito é muito mais poderoso que um array multidimensional**. [...] Você pode criar *arrays irregulares*, ou arrays que não são "quadrados" no sentido m x n x ... Como um exemplo, você pode ter um array com 10 elementos, o segundo dos quais é também um array com 6 elementos, e o terceiro elemento desse

array é também um array...
Fonte : (HEIMENDINGER, 1994, p. 99)

Dica 11

Lembra do exemplo do apartamento que foi apresentado no início desse capítulo ? Pois bem, como no nosso suposto prédio existia apenas um apartamento por andar, por analogia, o array era unidimensional (um vetor).

Figura 1.7: Um array unidimensional (ou vetor)



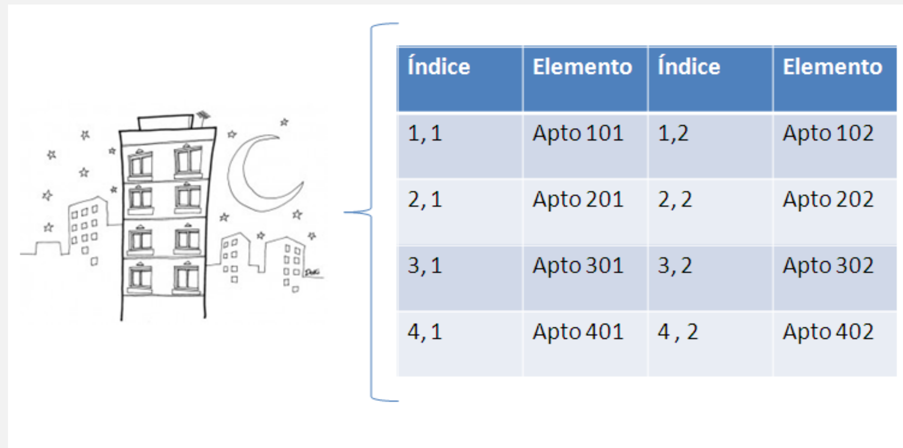
O fragmento a seguir ilustra como seria o array da figura 1.7 em Harbour:

```
aPredio := { "Apto 101" , "Apto 201" , "Apto 301" , "Apto 401" }

? aPredio[ 2 ] // Imprime "Apto 201"
```

Mas existem prédios com mais de um apartamento por andar (por exemplo dois apartamentos por andar), esse seria o caso de um array multidimensional (duas dimensões).

Figura 1.8: Um array multidimensional



Índice	Elemento	Índice	Elemento
1, 1	Apto 101	1,2	Apto 102
2, 1	Apto 201	2, 2	Apto 202
3, 1	Apto 301	3, 2	Apto 302
4, 1	Apto 401	4 , 2	Apto 402

O fragmento a seguir ilustra como seria o array da figura 1.8 em Harbour:

```
aPredio := { { "Apto 101" , "Apto 102" } ,;
             { "Apto 201" , "Apto 202" } ,;
             { "Apto 301" , "Apto 302" } ,;
             { "Apto 401" , "Apto 402" } }

? aPredio[ 2 ][ 1 ] // Imprime "Apto 201"
? aPredio[ 4 ][ 2 ] // Imprime "Apto 402"
```

A seguir a representação de um prédio com 4 apartamentos por andar.

Figura 1.9: Um array multidimensional (ou vetor)

Índice	Elemento	Índice	Elemento	Índice	Elemento	Índice	Elemento
1, 1	Apto 101	1,2	Apto 102	1,3	Apto 103	1,4	Apto 104
2, 1	Apto 201	2, 2	Apto 202	2,3	Apto 203	2,4	Apto 204
3, 1	Apto 301	3, 2	Apto 302	3,3	Apto 303	3,4	Apto 304
4, 1	Apto 401	4 , 2	Apto 402	4,3	Apto 403	4,4	Apto 404

O fragmento a seguir ilustra como seria o array da figura 1.9 em Harbour:

```
aPredio := { { "Apto 101" , "Apto 102" , "Apto 103" , "Apto 104"  } ,;
```

```
                 { "Apto 201" , "Apto 202" , "Apto 203" , "Apto 204"} ,;
                 { "Apto 301" , "Apto 302" , "Apto 303" , "Apto 304"} ,;
                 { "Apto 401" , "Apto 402" , "Apto 403" , "Apto 404"} }

? aPredio[ 2 ][ 1 ] // Imprime "Apto 201"
? aPredio[ 3 ][ 3 ] // Imprime "Apto 303"
? aPredio[ 4 ][ 3 ] // Imprime "Apto 403"
```

O Harbour vai mais além: como cada elemento do array pode conter uma quantidade variada de arrays, então isso equivaleria a um prédio com uma quantidade variada de apartamentos por andares. Por exemplo:

Figura 1.10: Um array multidimensional do Harbour.

Índice	Elemento	Índice	Elemento	Índice	Elemento	Índice	Elemento
1, 1	Apto 101	1,2	Apto 102				
2, 1	Apto 201						
3, 1	Apto 301	3, 2	Apto 302				
4, 1	Apto 401	4 , 2	Apto 402	4,3	Apto 403	4,4	Apto 404

O fragmento a seguir ilustra como seria o array da figura 1.9 em Harbour:

```
aPredio := { { "Apto 101" , "Apto 102"  } ,;
             { "Apto 201" } ,;
             { "Apto 301" , "Apto 302" } ,;
             { "Apto 401" , "Apto 402" , "Apto 403" , "Apto 404"} }

? aPredio[ 2 ][ 1 ] // Imprime "Apto 201"
? aPredio[ 4 ][ 3 ] // Imprime "Apto 403"

// O comando abaixo gera um erro de ‘‘array asign’’.
? aPredio[ 3 ][ 3 ] // Erro, pois o elemento não existe.
```

1.4.5 Arrays são referências quando igualadas

Se você usar um operador de atribuição para atribuir o conteúdo de um array em outro array você não esta criando um novo array, mas criando uma referência para o array

antigo.

```
a1 := { 10, 20, 30 }
a2 := a1

a1[ 2 ] := 7

? a2[ 2 ] // Imprime 7 em vez de 20
```

Entendeu ? Quando eu uso o operador para atribuir um valor através de outra variável as duas variáveis apontam para o mesmo endereço de memória. Se eu mudar a2, a1 também mudará, e vice-verça. Isso porque as duas variáveis, apesar de diferentes, apontam para o mesmo local na memória (para o mesmo conteúdo).

1.4.5.1 A pegadinha da atribuição IN LINE

Uma consequência direta disso é a atribuição IN LINE. Até agora, com variáveis comuns, quando você inicializa diversas variáveis através de atribuição IN LINE você está criando variáveis independentes. Por exemplo:

```
LOCAL nVal1 := nVal2 := nVal3 := 0 // Cria 3 variáveis numéricas independentes
```

Já com arrays isso não existe. Se você fizer uma atribuição IN LINE com arrays, você estará criando 3 referências para o mesmo array. Por exemplo:

```
LOCAL a1 := a2 := a3 := {}
```

Se você inserir um elemento em a2, automaticamente a1 e a3 receberão esse mesmo elemento. O Harbour cria referências para a mesma variável. O exemplo 1.8 ilustra esse comportamento.

Listing 1.8: Atribuição IN LINE de arrays: CUIDADO!

```
1 /*
2 Atribuição IN LINE
3 */
4 PROCEDURE Main
5 LOCAL a1 := a2 := a3 := {}
```

```
6  LOCAL x
7
8     AADD( a1 , 10 )
9     AADD( a1 , 20 )
10    AADD( a1 , 30 )
11
12
13    FOR x := 1 TO LEN( a3 )
14        ? a2[ x ]
15    NEXT
16
17
18 RETURN
```

.:Resultado:.

```
        10
        20
        30
```

Note que apenas o array a1 recebeu elementos, mas como eles foram inicializados IN LINE, o array a2 e a3 viraram referências para o array a1. Programadores C que estão familiarizados com ponteiros sabem porque isso ocorreu, mas como nosso curso é introdutório, basta que você evite inicializar arrays IN LINE e também evitar atribuições.

Dica 12

Evite inicializar arrays IN LINE.

Dica 13

Da mesma forma que você deve evitar atribuições IN LINE você deve ter cuidado com atribuições de igualdade entre os arrays.

Dica 14

Caso você necessite criar um array independente de uma variável existente use a função ACLONE(). Essa função cria uma cópia independente do array.

```
a1 := { 10, 20, 30 }
a2 := ACLONE( a1 )

a1[ 2 ] := 7

? a2[ 2 ] // Imprime 20
```

1.4.5.2 A pegadinha do operador de comparação

De acordo com Ramalho

> uma matriz [array] só pode ser comparada com outra usando-se o duplo operador de igualdade "==". O operador retornará .t. apenas se elas fizerem referência à mesma localização de memória (RAMALHO, 1991, p. 397).

Listing 1.9: Igualdade de arrays: CUIDADO!

```
/*
Atribuição IN LINE
*/
PROCEDURE Main
LOCAL a1 := a2 := {}  // Mesma referência
LOCAL a3 := {}

   ?
   ? "Vou preencher a1 e a3 com o mesmo conteúdo..."
   ?
   AADD( a1 , 10 )
   AADD( a1 , 20 )
   AADD( a1 , 30 )

   AADD( a3 , 10 )
   AADD( a3 , 20 )
   AADD( a3 , 30 )


   ? "a1 e a2 possuem a mesma referência, por isso säo iguais"
   ? "a1 == a2  :  ", a1 == a2

   ? "a1 e a3 säo independentes, por isso näo säo iguais"
   ? "a1 == a3  :  ", a1 == a3


RETURN
```

Note que, apesar de a1 e a3 terem os mesmos elementos eles não são considerados iguais. Para serem iguais, os arrays precisam compartilhar o mesmo endereço.

.:Resultado:.

```
Vou preencher a1 e a3 com o mesmo conteúdo...

a1 e a2 possuem a mesma referência, por isso são iguais
a1 == a2  :   .T.
a1 e a3 são independentes, por isso não são iguais
```

```
a1 == a3  :   .F.
```

Caso você deseje verificar se dois arrays tem os mesmos elementos, você pode usar a função hb_valtoexp(). Essa função converte um valor qualquer para string[4].

Listing 1.10: Igualdade entre elementos de arrays.

```
/*
Outras soluções em :
   http://www.pctoledo.com.br/forum/viewtopic.php?f=4&t=17476
*/
PROCEDURE Main
LOCAL a1 := a2 := {}  // Mesma referência
LOCAL a3 := {}

   ?
   ? "Vou preencher a1 e a3 com o mesmo conteúdo..."
   ?
   AADD( a1 , 10 )
   AADD( a1 , { 20 , 30 } )
   AADD( a1 , 30 )

   AADD( a3 , 10 )
   AADD( a3 , { 20 , 30 } )
   AADD( a3 , 30 )


   ? "a1 e a2 possuem a mesma referência, por isso säo iguais"
   ? "a1 == a2  :  ", a1 == a2

   ? "a1 e a3 säo independentes, por isso não säo iguais"
   ? "a1 == a3  :  ", a1 == a3

   ? "Agora vou comparar o conteúdo de a1 e a3 usando
      hb_valtoexp() ..."
   ? "hb_valtoexp( a1 ) == hb_valtoexp( a3 )  :  ", hb_valtoexp(
      a1 ) == hb_valtoexp( a3 )
   ?
   ? "Veja o que faz hb_valtoexp()"
   ? hb_valtoexp( a1 )


RETURN
```

[4]Outras soluções podem ser obtidas em http://www.pctoledo.com.br/forum/viewtopic.php?f=4&t=17476. Você deve ter cuidado quando for realizar essa verificação com arrays gigantescos, pois a conversão pode demandar muito tempo.

.:Resultado:.

```
Vou preencher a1 e a3 com o mesmo conteúdo...

a1 e a2 possuem a mesma referência, por isso são iguais
a1 == a2  :   .T.
a1 e a3 são independentes, por isso não são iguais
a1 == a3  :   .F.
Agora vou comparar o conteúdo de a1 e a3 usando hb_valtoexp() ...
hb_valtoexp( a1 ) == hb_valtoexp( a3 )  :   .T.

Veja o que faz hb_valtoexp()
{10, {20, 30}, 30}
```

1.4.6 Arrays na passagem de parâmetros

Quando nós estudamos as rotinas e a passagem de parâmetros, vimos que os argumentos são passados por valor. Se quisermos passar um argumento por referência devemos prefixar ele com um @. Os arrays também se comportam segundo essa regra, mas existem algumas mudanças significativas.

Vamos tentar explicar através de exemplos.

Primeiro, analise o código da listagem

Listing 1.11: Arrays passados por valor

```
1  /*
2  Exemplo retirado de : Clipper 5.2 - Rick Spence , p.159
3  */
4  PROCEDURE Main
5  LOCAL aNomes := { "Spence", "Thompson" }
6
7     Test( aNomes )
8     ? aNomes[ 1 ] // Spence
9
10
11 RETURN
12
13 PROCEDURE Test( aFormal )
14
15    ? aFormal[1] // Spence
16    aFormal := { "Rick" , "Malcolm" }
17
18 RETURN
```

.:Resultado:.

```
Spence
Spence
```

Até agora nada de novo com relação aos outros tipos de dados. O array foi passado por valor, conforme os outros tipos, e o valor não foi alterado fora da rotina.

Se quiséssemos, nós poderíamos passar o array por referência. Basta prefixar o argumento com um @. Também nada de novo aqui.

Listing 1.12: Arrays passados por referência

```
1  /*
2  Exemplo retirado de : Clipper 5.2 - Rick Spence , p.159
3  */
4  PROCEDURE Main
5  LOCAL aNomes := { "Spence", "Thompson" }
6
7     Test( @aNomes )
8     ? aNomes[ 1 ] // Rick
9
10
11 RETURN
12
13 PROCEDURE Test( aFormal )
14
15    ? aFormal[1] // Spence
16    aFormal := { "Rick" , "Malcolm" }
17
18 RETURN
```

.:Resultado:.

```
Spence
Rick
```

Agora muita atenção: Se eu passar um array por valor e for alterar **um elemento seu** dentro da rotina chamada, isso irá alterar o array na rotina chamadora. Acompanhe o código a seguir.

Listing 1.13: Arrays sempre são passados por referência quando modificamos seus elementos internos.

```
1  PROCEDURE Main
2  LOCAL a1 := { 10, 20, 30 }
3
4     ? "O primeiro elemento é ", a1[ 1 ]
5
6     Muda( a1 )
7
```

```
8     ? "O primeiro elemento foi alterado : ", a1[ 1 ]
9
10
11
12 RETURN
13
14 PROCEDURE Muda( aArray )
15
16    aArray[ 1 ] := "Mudei aqui"
17
18 RETURN
```

.:Resultado:.

```
O primeiro elemento é           10
O primeiro elemento foi alterado :  Mudei aqui
```

Dica 15

Tenha cuidado quando for passar arrays como um argumento de uma rotina. Lembre-se de que os seus elementos podem sofrer alteração dentro dessa rotina e o array será alterado também na rotina chamadora.

1.4.7 Elementos de arrays na passagem de parâmetros

De acordo com Spence, “os elementos de um array são sempre enviados por valor, o que significa que a rotina não pode alterá-los” (SPENCE, 1994, p. 162)

Listing 1.14: Elementos de arrays na passagem de parâmetros

```
1
2  PROCEDURE Main
3  LOCAL aNomes := { "Spence", "Thompson" }
4
5     Test( aNomes[ 1 ] )
6     ? aNomes[ 1 ] // Spence
7     Test( @aNomes[ 1 ] )
8     ? aNomes[ 1 ] // Outro qualquer
9
10
11 RETURN
12
13 PROCEDURE Test( cNome )
14
15    cNome := "Outro qualquer"
16
17 RETURN
```

.:Resultado:.

```
Spence
Outro qualquer
```

Ou seja: elementos de arrays se comportam como se fossem variáveis comuns. Note que na linha 7 o elemento do array foi passado por referência, por isso ele foi modificado fora da rotina também.

1.4.8 Retornando arrays a partir de funções

Arrays podem ser retornados a partir de funções, inclusive o Harbour possui algumas funções que retornam arrays, como é o caso da função Directory(), que retorna um array com dados sobre uma pasta qualquer. Veremos essa e outras funções ainda nesse capítulo.

Dica 16

Retornar arrays de funções é útil quando você deseja retornar informações complexas. **Faça suas funções de retorno de arrays ao invés de passar muitos parâmetros por referência** (SPENCE, 1991, p. 156).

Listing 1.15: Retornando arrays de funções

```
1
2  PROCEDURE Main
3  LOCAL aNomes1, aNomes2
4
5     aNomes1 := Test( "Gilberto" , "Silvério" )
6     aNomes2 := Test( "William" , "Manesco" )
7
8     ? "Nome 1 : " , aNomes1[ 1 ] , aNomes1[ 2 ]
9     ? "Nome 2 : " , aNomes2[ 1 ] , aNomes2[ 2 ]
10
11 RETURN
12
13 FUNCTION Test( cFirst, cLast )
14
15 RETURN { cFirst , cLast }
```

.:Resultado:.

```
Nome 1 :  Gilberto Silvério
Nome 2 :  William Manesco
```

Note que, a cada retorno da função vem um array com uma nova referência. Ou seja, eles não tem relação entre si e podem ser usados livremente.

1.4.9 Lembra do laço FOR EACH ?

O laço FOR EACH foi apresentado quando nós estudávamos as estruturas de repetição. Como ainda não havíamos visto arrays e hashs, os exemplos dados foram apenas com strings. A listagem a seguir ilustra como devemos fazer para percorrer todos os elementos de um array unidimensional.

Listing 1.16: Laço FOR EACH com arrays

```
1
2  PROCEDURE Main
3  LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" , "F" , "G" }
4  LOCAL cElemento
5
6     FOR EACH cElemento IN aLetras
7         ? cElemento
8     NEXT
9
10 RETURN
```

.:Resultado:.

```
A
B
C
D
E
F
G
```

O enumerador do laço FOR EACH tem propriedades especiais que podem ser acessadas usando o seguinte formato abaixo:

```
<elemento>:<propriedade>
```

Como estamos estudando os arrays, vamos demonstrar no fragmento a seguir as propriedades especiais do enumerador aplicada ao array.

```
<elemento>:__enumindex // O índice do elemento corrente
<elemento>:__enumvalue // O valor do elemento corrente
<elemento>:__enumbase // O array
```

A listagem 1.17 ilustra isso:

Listing 1.17: Laço FOR EACH com arrays

```

PROCEDURE Main
LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" , "F" , "G" }
LOCAL cElemento

   FOR EACH cElemento IN aLetras
      ? cElemento:__enumindex , " ==> "
      ?? cElemento:__enumvalue , " = " ,  cElemento
   NEXT

RETURN
```

.:Resultado:.

```

1 ==> A = A
2 ==> B = B
3 ==> C = C
4 ==> D = D
5 ==> E = E
6 ==> F = F
7 ==> G = G
```

Note que a propriedade __enumvalue é dispensável (linha 8), já que digitar cElemento é a mesma coisa que digitar cElemento:__enumvalue.

Temos também a propriedade __enumbase, que retorna o array completo que gerou a consulta. A listagem 1.18 ilustra isso:

Listing 1.18: Laço FOR EACH com arrays

```

PROCEDURE Main
LOCAL aLetras := { "A" , "B" , "C" , "D" , "E" , "F" , "G" }
LOCAL cElemento

   FOR EACH cElemento IN aLetras
      ? hb_valtoexp( cElemento:__enumbase )
   NEXT

RETURN
```

.:Resultado:.

```

{"A", "B", "C", "D", "E", "F", "G"}
```

```
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
{"A", "B", "C", "D", "E", "F", "G"}
```

1.4.10 Algumas funções que operam sobre arrays

O Harbour possui inúmeras funções que operam sobre os arrays, mas como nosso curso é introdutório nós veremos apenas as principais funções.

1.4.10.1 AADD()

1.4.10.2 ACLONE()

Adicionar mais funções nesse local

1.5 O Hash

Array associativo ou Hash é um recurso oriundo de linguagens interpretadas, como o PHP, Perl, Python e Ruby. O Clipper não tinha essa estrutura de dados. Com o surgimento do Harbor e as constantes melhorias decorrentes da cultura de software-livre, o Harbour acabou implementando essa estrutura de dados também. Grosso modo, um Hash é um tipo especial de array, cujo ponteiro pode ser uma string também. O fragmento de código abaixo ilustra essa diferença:

```
aCliente[ 10 ] := "Daniel  Crocciari’’ // Array
hCliente[ "nome" ] := "Daniel  Crocciari’’ // Hash
```

1.5.1 Criando um Hash

Um hash novo é criado com a função HB_HASH() ou sem o uso dessa função.

1.5.1.1 Com a função HB_HASH()

Primeira forma: sem parâmetros. Usada sem parâmetros, a função criará um hash vazio (sem elementos). Apenas declara uma variável e informa que é um hash.

```
LOCAL hClientes := HB_HASH()
```

Segunda forma: com parâmetros separado por vírgulas.

```
LOCAL hClientes := HB_HASH( "nome" , "Alexandre" , "sobrenome" , "Santos" )

 ? hClientes[ "nome" ] // Alexandre
 ? hClientes[ "sobrenome" ] // Santos
```

1.5.1.2 Sem a função HB_HASH()

Primeira forma: Um Hash vazio.

```
LOCAL hClientes := { => }
```

Segunda forma: Semelhante a um array mas com o par chave/valor separado por setas (“=>”). Esse formato é semelhante ao das outras linguagens que tem hash.

```
LOCAL hClientes := { "nome" => "Alexandre" , "sobrenome" => "Santos" }
```

1.5.2 Percorrendo um Hash com FOR EACH

Um Hash pode ser percorrido facilmente com o laço FOR EACH, conforme a listagem 1.19.

Listing 1.19: Laço FOR EACH com hashs

```
1
2  PROCEDURE Main
3  LOCAL hLetras := { "A" => "Letra A" , "B" => "Letra B" , "C" =>
      "Letra C" }
4  LOCAL cElemento
5
6     FOR EACH cElemento IN hLetras
7         ? cElemento
8     NEXT
```

```
9
10 RETURN
```

.:Resultado:.

```
Letra A
Letra B
Letra C
```

O laço FOR EACH também tem possui suas próprias propriedades especiais para os enumeradores.

```
<elemento>:__enumindex // A ordem numérica do elemento corrente
<elemento>:__enumvalue // O valor do elemento corrente
<elemento>:__enumbase // O hash
<elemento>:__enumkey // A chave do hash
```

A listagem 1.20 ilustra isso:

Listing 1.20: Laço FOR EACH com hashs

```
1
2  PROCEDURE Main
3  LOCAL hLetras := { "A" => "Letra A" , "B" => "Letra B" , "C" =>
      "Letra C"  }
4  LOCAL cElemento
5
6     FOR EACH cElemento IN hLetras
7         ? cElemento:__enumindex , cElemento:__enumkey , " ==> "
8         ?? cElemento:__enumvalue, " = " ,  cElemento
9     NEXT
10
11 RETURN
```

.:Resultado:.

```
         1 A  ==> Letra A  =  Letra A
         2 B  ==> Letra B  =  Letra B
         3 C  ==> Letra C  =  Letra C
```

A novidade aqui é a propriedade __enumkey, que retorna a chave do hash. As demais propriedades também estão presentes nos arrays.

Note que a propriedade __enumindex serve tanto para arrays quanto para hashs. Note também que a propriedade __enumvalue é dispensável (linha 8), já que digitar cElemento é a mesma coisa que digitar cElemento:__enumvalue.

Temos também a propriedade __enumbase, que retorna o hash completo que gerou a consulta. A listagem 1.21 ilustra isso:

Listing 1.21: Laço FOR EACH com hashs

```
1
2  PROCEDURE Main
3  LOCAL hLetras := { "A" => "Letra A" , "B" => "Letra B" , "C" =>
      "Letra C"  }
4  LOCAL cElemento
5
6     FOR EACH cElemento IN hLetras
7         ? hb_valtoexp( cElemento:__enumbase )
8     NEXT
9
10 RETURN
```

.:Resultado:.

```
{"A"=>"Letra A", "B"=>"Letra B", "C"=>"Letra C"}
{"A"=>"Letra A", "B"=>"Letra B", "C"=>"Letra C"}
{"A"=>"Letra A", "B"=>"Letra B", "C"=>"Letra C"}
```

1.5.3 As características de um hash

Sempre que você cria um hash, ele já vem configurado com algumas características básicas que definem o seu comportamento. Essas propriedades assumem dois valores: ligada (.t.) ou desligada (.f.). As propriedade estão listadas nas próximas subseções:

1.5.3.1 AutoAdd

Essa propriedade determina se um elemento de um hash será adicionado automaticamente logo após a sua definição. Essa propriedade pode ser entendida se você recordar dos arrays. Acompanhe o fragmento abaixo :

```
LOCAL aClientes := {}

      aClientes[ 1 ] := "Rick Spence" // Erro de array assign
```

Como você já deve ter percebido, o fragmento acima gerará um erro de execução pois o array foi inicializado com zero elementos, mas nós tentamos inserir um elemento que, é claro, está fora do intervalo válido.

O código a seguir (listagem 1.22) foi implementado usando um hash e faz uma atribuição semelhante.

Listing 1.22: Propriedades do hash

```
1
2  PROCEDURE Main
3  LOCAL hLetras := {  =>   }
4
5     hLetras[ "A" ] := "Letra A"
6     ? hLetras[ "A" ]
7
8  RETURN
```

Porém, ao contrário do array, o programa fez exatamente o desejado.

.:Resultado:.

```
Letra A
```

Isso só foi possível porque a propriedade *AutoAdd* do hash é setada como verdadeira (.t.) por default. Mas pode existir um caso onde você quer que o hash se comporte como um array, ou seja, que ele só aceite atribuições em chaves previamente criadas. Para que isso aconteça você deve setar a propriedade *AutoAdd* como falsa (.t.), e isso é feito através da função HB_hAutoAdd(). No código da listagem 1.23 nós tentamos fazer a mesma coisa que o codigo da listagem 1.22, mas agora nós obtemos um erro.

Listing 1.23: Propriedades do hash - AutoAdd

```
1
2  PROCEDURE Main
3  LOCAL hLetras := {  =>   }
4
5     hb_hAutoAdd( hLetras , .f. )
6
7     hLetras[ "A" ] := "Letra A"
8     ? hLetras[ "A" ]
9
10 RETURN
```

Na linha 5 nós alteramos a propriedade *AutoAdd* do hash. Note que o erro ocorreu na linha 7, onde eu tentei fazer a atribuição. Veja que a mensagem de erro é a mesma gerada para um array.

.:Resultado:.

```
Error BASE/1133  Bound error: array assign
Called from MAIN(7)
```

1.5.3.2 Binary

Determina a forma como o hash será ordenado, se no padrão binário ou se de acordo com o codepage usado. Talvez você não se sinta muito a vontade com essa propriedade, porque ela necessita do entendimento de dois conceitos ainda não vistos completamente: ordenação de hash e codepage. Mesmo assim vamos tentar explicar essa propriedade. Primeiramente vamos entender como um array é ordenado, acompanhe a listagem 1.24.

Listing 1.24: Propriedades do hash - Binary

```
1
2  PROCEDURE Main
3  LOCAL hLetras := {  =>   }
4  LOCAL hOrdenado
5  LOCAL cElemento
6
7     hLetras[ "K" ] := "Letra K"
8     hLetras[ "A" ] := "Letra A"
9     hLetras[ "á" ] := "A minúscula"
10    hLetras[ "Z" ] := "Letra Z"
11
12    hOrdenado := HB_hSort( hLetras )
13
14    FOR EACH cElemento IN hOrdenado
15       ? cElemento
16    NEXT
17
18 RETURN
```

Observe que a ordenação é feita através da função HB_hSort(), e que ela ordena pela chave do hash, e não pelo seu valor.

.:Resultado:.

```

Letra A
Letra K
Letra Z
A minúscula
```

Note que a chave de um hash pode ter caracteres acentuados também caracteres que não pertencem ao nosso alfabeto, como mandarim, árabe, etc. Tudo irá depender do codepage ativo. Pois bem, quando a propriedade binary está ativa (.t.), que é o default, o hash irá desprezar

a ordenação de caracteres especiais e irá ordenar apenas levando em consideração o código binário. Caso contrário, ela irá levar em conta os caracteres especiais de cada idioma.

Dica 17

Deixe o padrão de busca Binary com o seu valor default (.t.), pois o algoritmo de ordenação é mais eficiente.

Dica 18

Evite colocar acentos ou caracteres especiais na chave de um hash.

Para alterar a forma como a ordenação é feita use a função HB_hBinary(). Veja um exemplo logo a seguir.

Listing 1.25: Propriedades do hash - Binary

```

PROCEDURE Main
LOCAL hLetras := {  =>   }
LOCAL hOrdenado
LOCAL cElemento

   hb_hBinary( hLetras , .f. ) // Leva em consideração o codepage

   hLetras[ "K" ] := "Letra K"
   hLetras[ "A" ] := "Letra A"
   hLetras[ "á" ] := "A minúscula"
   hLetras[ "Z" ] := "Letra Z"

   hOrdenado := HB_hSort( hLetras )

   FOR EACH cElemento IN hOrdenado
      ? cElemento
   NEXT

RETURN
```

Note que a mudança no padrão de ordenamento foi mudado na linha 7. Acompanhe o resultado a seguir:

.:Resultado:.

```
Letra A
Letra K
Letra Z
A minúscula
```

Veja que o resultado é exatamente igual a busca binária, porém, de acordo com o codepage e do símbolo usados, essa ordenação poderá se alterar. Por isso nós recomendamos ordenar sempre pelo padrão binário, pois não mudará a forma com que as chaves são ordenadas, independente de qual codepage estiver ativo, além do mais, como já foi dito, a busca binária é mais rápida.

1.5.3.3 CaseMatch

CaseMatch determina se a chave do hash deverá ser sensível a maiúsculas/minúsculas ou não. Acompanhe um exemplo na listagem 1.26.

Listing 1.26: Propriedades do hash - CaseMatch

```
1
2  PROCEDURE Main
3  LOCAL hClientes := {  =>   }
4  LOCAL cElemento
5
6
7     hClientes[ "Cliente" ] := "Robert Plant"
8     hClientes[ "CLIENTE" ] := "Jimmy Page"
9     hClientes[ "cliente" ] := "Jonh Bonham"
10    hClientes[ "CLIente" ] := "Jonh Paul Jones"
11
12
13    FOR EACH cElemento IN hClientes
14       ? cElemento
15    NEXT
16
17 RETURN
```

Esse exemplo serve para ilustrar que o comportamento padrão **leva em consideração maiúsculas e minúsculas**. Acompanhe o resultado a seguir:

.:Resultado:.

```
Robert Plant
Jimmy Page
Jonh Bonham
Jonh Paul Jones
```

Esse é o comportamento padrão. Para alterar esse comportamento você deve alterar a propriedade CaseMatch através da função HB_hCaseMatch(). Veja a alteração na listagem a seguir.

Listing 1.27: Propriedades do hash - CaseMatch

```
1
2 PROCEDURE Main
3 LOCAL hClientes := {  =>   }
4 LOCAL cElemento
5
6    HB_hCaseMatch( hClientes , .f. )
7
8    hClientes[ "Cliente" ] := "Robert Plant"
9    hClientes[ "CLIENTE" ] := "Jimmy Page"
10   hClientes[ "cliente" ] := "Jonh Bonham"
11   hClientes[ "CLIente" ] := "Jonh Paul Jones"
12
13
14   FOR EACH cElemento IN hClientes
15      ? cElemento
16   NEXT
17
18 RETURN
```

Esse exemplo serve para ilustrar duas coisas:

1. Quando a propriedade CaseMatch é setada com o valor falso, o Harbour passa a desconsiderar as diferenças entre maiúsculas e minúsculas nas chaves dos hashs.
2. Já que o Hash considera "CLIENTE" = "cliente", por exemplo, ele não vai criar um novo elemento para o hash (nas linhas 10,11 e 12), mas vai sobrepor o elemento anterior.

Acompanhe o resultado a seguir:

.:Resultado:.

```
Jonh Paul Jones
```

Note que o Hash possui apenas um valor. Os valores anteriores foram sobrepostos pois a chave foi considerada a mesma.

1.5.3.4 KeepOrder

KeepOrder significa "Manter a ordem". Mas manter a ordem de que ? A resposta é : a ordem natural com que os elementos foram inseridos. Vamos explicar melhor: quando o hash é criado, a ordem com que os seus elementos são dispostos é a ordem com que foram adicionados. Essa é a propriedade padrão do hash.

Listing 1.28: Propriedades do hash - KeepOrder

```
1
2 PROCEDURE Main
3 LOCAL hLetras := { => }
4 LOCAL cElemento
5
6    hLetras[ "K" ] := "Letra K"
7    hLetras[ "A" ] := "Letra A"
8    hLetras[ "B" ] := "Letra B"
9    hLetras[ "Z" ] := "Letra Z"
10
11   FOR EACH cElemento IN hLetras
12      ? cElemento
13   NEXT
14
15 RETURN
```

.:Resultado:.

```
Letra K
Letra A
Letra B
Letra Z
```

Para desativar essa propriedade use a função HB_hKeepOrder(). Quando essa propriedade está desativada o hash é ordenado automaticamente pelos valores das chaves. Acompanhe a mudança:

Listing 1.29: Propriedades do hash - KeepOrder

```
1
2 PROCEDURE Main
3 LOCAL hLetras := { => }
4 LOCAL cElemento
5
6    HB_hKeepOrder( hLetras , .f. )
7
8    hLetras[ "K" ] := "Letra K"
9    hLetras[ "A" ] := "Letra A"
10   hLetras[ "B" ] := "Letra B"
11   hLetras[ "Z" ] := "Letra Z"
12
13   FOR EACH cElemento IN hLetras
14      ? cElemento
15   NEXT
16
17 RETURN
```

.:Resultado:.

```
Letra A
Letra B
Letra K
Letra Z
```

A função de ordenamento HB_hSort() não é mais necessária, pois o hash é ordenado automaticamente. É bom lembrar que, se essa propriedade for desativada o Harbour terá um trabalho extra de processamento (para manter o hash sempre indexado pela chave). Portanto, caso você não vá ordenar o seu Hash, evite desativar essa propriedade.

1.5.4 Funções que facilitam o manuseio de Hashs

1.5.4.1 HB_hHashKey()

1.5.4.2 HB_hGet()

1.5.4.3 HB_hGetDef()

1.5.4.4 HB_hSet()

1.5.4.5 HB_hDel()

1.5.4.6 HB_hPos()

1.5.4.7 HB_hKeyAt()

1.5.4.8 HB_hPairAt()

1.5.4.9 HB_hDelAt()

1.5.4.10 HB_hKeys()

1.5.4.11 HB_hValues()

1.5.4.12 HB_hFill()

1.5.4.13 HB_hClone()

1.5.4.14 HB_hCopy()

1.5.4.15 HB_hMerge()

1.5.4.16 HB_hEval()

1.5.4.17 HB_hScan()

1.5.4.18 HB_hSort()

Veja a explicação dessa função no tópico sobre a propriedade Binary, na seção 1.5.3.2.

1.5.4.19 HB_hAllocate()

1.5.4.20 HB_hDefault()

Referências Bibliográficas

AGUILAR, L. J. **Fundamentos de programação: algoritmos, estruturas de dados e objetos**. Editora McGraw-Hill, São Paulo, 2008.

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

HEIMENDINGER, L. **Clipper 5.0 - Básico**. Campus, São paulo, 1994.

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

RAMALHO, J. A. A. **Clipper avançado**. 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.