Variáveis de memória

Sumário

Introdução

Em toda linguagem de programação existe a necessidade de variáveis de memória. Mas este material foi especialmente direcionado ao estudo de variáveis na linguagem ADVPL.

Criação e Atribuição de Variáveis

Variáveis de memória são um dos recursos mais importantes de uma linguagem.

São áreas de memória criadas para armazenar informações utilizadas por um programa para a execução de tarefas. Por exemplo, quando o usuário digita uma informação qualquer, como o nome de um produto, em uma tela de um programa esta informação é armazenada em uma variável de memória para posteriormente ser gravada ou impressa.

A partir do momento que uma variável é criada, não é necessário mais se referenciar ao seu conteúdo, e sim ao seu nome. O nome de uma variável é um identificador único que segue duas regras básicas:

1-Máximo de 10 caracteres. O ADVPL não impede a criação de uma variável de memória cujo nome contenha mais de 10 caracteres, porém apenas os 10 primeiros serão considerados para a localização do conteúdo armazenado. Portanto se forem criadas duas variáveis cujos 10 primeiros caracteres forem iguais, como nTotalGeralAnual e nTotalGeralMensal, as referências a qualquer uma delas no programa resultarão o mesmo. Ou seja, serão a mesma variável:

```
nTotalGeralMensal := 100
nTotalGeralAnual  := 300
Alert("Valor mensal: " + cValToChar(nTotalGeralMensal))
```

Quando o conteúdo da variável nTotalGeralMensal é exibido, o seu valor será

de 300. Isso acontece porque no momento que esse valor foi atribuido à variável nTotalGeralAnual, o ADVPL considerou apenas os 10 primeiros caracteres (assim como o faz quando deve exibir o valor da variável nTotalGeralMensal), ou seja, considerou-as como a mesma variável. Assim o valor original de 100 foi substituido pelo de 300.

2-Limitação de caracteres no nome. Os nomes das variáveis devem sempre começar por uma letra ou caracter de sublinhado (_). No restante, pode conter letras, números e o caracter de sublinhado. Qualquer outro caracter, incluindo espaços em branco, não são permitidos.

O ADVPL permite a criação ilimitada de variáveis, dependendo apenas da memória disponível. A seguir estão alguns nomes válidos para variáveis:

Numero
nNumero
_nNumero

E alguns inválidos:

1CODIGO (Inicia por um número)
M CARGO (contém um espaço em branco)
LOCAL (palavra reservada do ADVPL)

O ADVPL não é uma linguagem de tipos rígidos para variáveis, ou seja, não é necessário informar o tipo de dados que determinada variável irá conter no momento de sua declaração, e o seu valor pode mudar durante a execução do programa. Também não há necessidade de declarar variáveis em uma seção específica do seu código fonte, embora seja aconselhável declarar todas as variáveis necessárias no começo, tornando a manutenção mais fácil e evitando a declaração de variáveis desnecessárias.

Para declarar uma variável deve-se utilizar um identificador de escopo, seguido de uma lista de variáveis separadas por vírgula (,). Um identificador de escopo é uma palavra chave que indica a que contexto do programa a variável declarada pertence. O contexto de variáveis pode ser local (visualizadas apenas dentro do programa atual), público (visualizadas por qualquer outro programa), entre outros. Os diferentes tipos de contexto de variáveis são explicados na documentação sobre escopo de variáveis.

Considere as linhas de código de exemplo:

```
nResultado := 250 * (1 + (nPercentual / 100))
```

Se esta linha for executada em um programa ADVPL, ocorrerá um erro de execução com a mensagem "variable does not exist: nPercentual", pois esta variável está sendo utilizada em uma expressão de cálculo sem ter sido declarada. Para solucionar este erro, deve-se declarar a variável previamente:

```
Local nPercentual, nResultado

nResultado := 250 * (1 + (nPercentual / 100))
```

Neste exemplo, as variáveis são declaradas previamente utilizando o **identificador de escopo local**. Quando a linha de cálculo for executada, o erro de “variável não existente”, não ocorrerá mais. Porém variáveis não inicializadas têm sempre o valor default nulo (Nil) e este valor não pode ser utilizado em um cálculo pois também gerará erros de execução (nulo não pode ser dividido por 100). A solução deste problema é efetuada inicializando-se a variável através de uma das formas:

```
Local nPercentual,nResultado
Store 10 To nPercentual
nResultado := 250 * (1 + (nPercentual / 100))
```

ou

```
Local nPercentual, nResultado
nPercentual := 10
nResultado  := 250 * (1 + (nPercentual / 100))
```

ou

```
Local nPercentual := 10, nResultado
nResultado := 250 * (1 + (nPercentual / 100))
```

A diferença entre o último exemplo e os dois anteriores é que a variável é inicializada no momento da declaração. Nos dois primeiros exemplos, a variável é primeiro declarada e então inicializada em uma outra linha de código. O comando store existe apenas por compatibilidade com versões anteriores e outras linguagens xBase, mas é obsoleto. Deve-se utilizar o operador de atribuição (:= ou somente =). É aconselhável optar pelo operador de atribuição composto de dois pontos e sinal de igual, pois o operador de atribuição utilizando

somente o sinal de igual pode ser facilmente confundido com o operador relacional (para comparação) durante a criação do programa.

Uma vez que um valor lhe seja atribuído, o tipo de dado de uma variável é igual ao tipo de dado do valor atribuído. Ou seja, uma variável passa a ser numérica se um número lhe é atribuído, passa a ser caracter se uma string de texto lhe for atribuída, etc. Porém mesmo que uma variável seja de determinado tipo de dado, pode-se mudar o tipo da variável atribuindo outro tipo a ela:

```
01 Local xVariavel // Declara a variável inicialmente com valor nulo
02
03 xVariavel := "Agora a variável é caracter..."
04 Alert("Valor do Texto: " + xVariavel)
05
06 xVariavel := 22 // Agora a variável é numérica
07 Alert(cValToChar(xVariavel))
08
09 xVariavel := .T. // Agora a variável é lógica
10 If xVariavel
11    Alert("A variável tem valor verdadeiro...")
12 Else
13    Alert("A variável tem valor falso...")
14 Endif
15
16 xVariavel := Date() // Agora a variável é data
17 Alert("Hoje é: " + DtoC(xVariavel))
18
19 xVariavel := nil // Nulo novamente
20 Alert("Valor nulo: " + xVariavel)
21
22 Return
```

Figura 1 - Programa de exemplo

No programa de exemplo anterior, a variável xVariavel é utilizada para armazenar diversos tipos de dados. A letra "x" em minúsculo no começo do nome é utilizada para indicar uma variável que pode conter diversos tipos de dados, segundo a Notação Húngara (consulte documentação específica para detalhes). Este programa troca os valores da variável e exibe seu conteúdo para o usuário através da função alert. Essa função recebe um parâmetro que deve ser do tipo string de caracter, por isso dependendo do tipo de dado da variável xVariavel é necessário fazer uma conversão antes.

Apesar dessa flexibilidade de utilização de variáveis, deve-se tomar cuidados na passagem de parâmetros para funções ou comandos, e na concatenação (ou soma) de valores. Note a linha 20 do programa de exemplo. Quando esta linha é executada, a variável xVariavel contém o valor nulo. A tentativa de soma de

tipos de dados diferentes gera erro de execução do programa. Nesta linha do exemplo, ocorrerá um erro com a mensagem "type mismatch on +". Exetuando-se o caso do valor nulo, para os demais deve-se sempre utilizar funções de conversão quando necessita-se concatenar tipos de dados diferentes (por exemplo, nas linhas 07 e 17.

Note também que quando uma variável é do tipo de dado lógico, ela pode ser utilizada diretamente para checagem (linha 10):

```
If xVariavel
```

é o mesmo que

```
If xVariavel = .T.
```

A declaração de variáveis para os demais tipos de dados, matrizes e blocos de código, é exatamente igual ao descrito até agora. Apenas existem algumas diferenças quanto a inicialização, que podem ser consultadas na documentação de inicialização de matrizes e blocos de código.

Diferenciação entre variáveis e nomes de campos

Muitas vezes uma variável pode ter o mesmo nome que um campo de um arquivo ou tabela aberto no momento. Neste caso, o ADVPL privilegiará o campo. Assim uma referência a um nome que identifique tanto uma variável como um campo, resultará no conteúdo do campo.

Para especificar qual deve ser o elemento referenciado, deve-se utilizar o operador de identificação de apelido (->) e um dos dois identificadores de referência, variável de Memória **(MEMVAR**) ou campo de uma tabela (**FIELD**).

```
cRes := M->NOME
```

Esta linha de comando identifica que o valor atribuído à variável cRes deve ser o valor da variável de memória chamada NOME.

cRes := SA1->A1_NOME

Neste caso, o valor atribuído à variável cRes será o valor do campo NOME existente no arquivo ou tabela aberto na área atual.

O identificador FIELD pode ser substituído pelo apelido de um arquivo ou tabela aberto, para evitar a necessidade de selecionar a área antes de acessar o conteúdo de terminado campo.

cRes := CLIENTES->NOME

Para maiores detalhes sobre abertura de arquivos com atribuição de apelidos, consulte a documentação sobre acesso a banco de dados ou a documentação da função dbUseArea.

Escopo de variáveis

O Contexto de Variáveis dentro de um Programa

As variáveis declaradas em um programa ou função, são visíveis de acordo com o escopo onde são definidas. Como também do escopo depende o tempo de existência das variáveis. A definição do escopo de uma variável é efetuada no momento de sua declaração.

Local nNumero := 10

Esta linha de código declara uma variável chamada **nNumero** indicando que pertence ao escopo local.

Os identifadores de escopo são:

- LOCAL
- PRIVATE
- PUBLIC
- STATIC

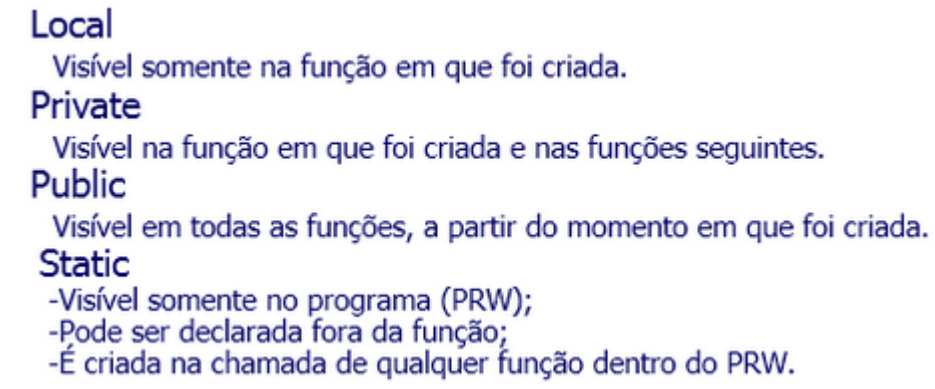

Figura 2 - Escopos

O ADVPL não é rígido em relação à declaração de variáveis no começo do programa. A inclusão de um identificador de escopo não é necessária para a declaração de uma variável, contanto que um valor lhe seja atribuído.

nNumero2 := 15

Quando um valor é atribuído à uma variável em um programa ou função, o ADVPL criará a variável caso ela não tenha sido declarada anteriormente. A variável então é criada como se tivesse sido declarada como PRIVATE.

Devido a essa característica, quando pretende-se fazer uma atribuição a uma variável declarada previamente mas escreve-se o nome da variável de forma incorreta, o ADVPL não gerará nenhum erro de compilação ou de execução. Pois

compreenderá o nome da variável escrito de forma incorreta como se fosse a criação de uma nova variável. Isto alterará a lógica do programa, e é um erro muitas vezes difícil de identificar.

Variáveis estáticas

Variáveis estáticas funcionam basicamente como as variáveis locais, mas mantêm seu valor através da execução. Variáveis estáticas devem ser declaradas explicitamente no código com o identificador STATIC.

O escopo das variáveis estáticas é limitado ao programa a qual foi declarada, independente do ponto onde a variável foi declarada.

Exemplo

Os dois trechos de código abaixo tem o mesmo resultado:

Declaração no corpo da função

Declaração no corpo da função

```
user function vStatic()
  Pai()
return

static function Pai()
static nVar := 10 // Declaração no corpo da função
  conOut("Pai")
  conOut(nVar)
  Filha()
return(.T.)

static function Filha()
  conOut("Filha")
  conOut(nVar)
return
```

Declaração fora do escopo de função

Declaração fora do escopo de função

```
static nVar := 10 // Declaração fora do escopo de função

user function vStatic()
  Pai()
return

static function Pai()
  conOut("Pai")
  conOut(nVar)
  Filha()
return(.T.)

static function Filha()
  conOut("Filha")
  conOut(nVar)
return
```

Variáveis Locais

Variáveis locais são pertencentes apenas ao escopo da função onde foram declaradas. Devem ser explicitamente declaradas com o identificador LOCAL, como no exemplo:

Function Pai()
Local nVar := 10, aMatriz := {0,1,2,3}
.
<comandos>
.
Filha()
.
<mais comandos>
.
Return(.T.)

Neste exemplo, a variável nVar foi declarada como local e atribuída com o valor 10. Quando a função Filha é executada, nVar ainda existe mas não pode ser acessada. Quando a execução da função Pai terminar, a variável nVar é destruída. Qualquer variável com o mesmo nome no programa que chamou a função Pai não é afetada.

Variáveis locais são criadas automaticamente cada vez que a função onde forem declaradas for ativada. Elas continuam a existir e mantêm seu valor até o fim da ativação da função (ou seja, até que a função retorne o controle para o código que a executou). Se uma função é chamada recursivamente (por exemplo,

chama a si mesma), cada chamada em recursão cria um novo conjunto de variáveis locais.

A visibilidade de variáveis locais é idêntica ao escopo de sua declaração. Ou seja, a variável é visível em qualquer lugar do código fonte em que foi declarada. Se uma função é chamada recursivamente, apenas as variáveis locais criadas na mais recente ativação são visíveis.

A declaração de variáveis locais dentro de uma função deve preceder qualquer comando interno ou declaração de outros tipos de variáveis (Private ou Public) da função caso contrário será gerado um erro de compilação.

Exemplo:

```
Function A( )
Private x:= 0
Local b:=0 <<<<< ERRADO, ERRO DE COMPILAÇÃO
...
Return
```

Versão correta:

```
Function A( )
Local b:=0 // correto
Private x:=0
....
Return
```

Variáveis Privadas

A declaração é opcional para variáveis privadas. Mas podem ser declaradas explicitamente com o identificador PRIVATE.

Adicionalmente, a atribuição de valor a uma variável não criada anteriormente automaticamente cria a variável como privada. Uma vez criada, uma variável privada continua a existir e mantem seu valor até que o programa ou função onde

foi criada termine (ou seja, até que a função onde foi criada retorne para o código que a executou). Neste momento, é automaticamente destruída.

É possível criar uma nova variável privada com o mesmo nome de uma variável já existente. Entretanto, a nova (duplicada) variável pode apenas ser criada em um nível de ativação inferior ao nível onde a variável foi declarada pela primeira vez (ou seja, apenas em uma função chamada pela função onde a variável já havia sido criada). A nova variável privada irá esconder qualquer outra variável privada ou pública (veja a documentação sobre variáveis públicas) com o mesmo nome enquanto existir.

Uma vez criada, uma variável privada é visível em todo o programa enquanto não for destruída automaticamente quando a rotina que a criou terminar ou uma outra variável privada com o mesmo nome for criada em uma sub-função chamada (neste caso, a variável existente torna-se inacessível até que a nova variável privada seja destruída).

Em termos mais simples, uma variável privada é visível dentro da função de criação e todas as funções chamadas por esta, a menos que uma função chamada crie sua própria variável privada com o mesmo nome.

Por exemplo:

```
Function Pai()
Private nVar := 10
.
<comandos>
.
Filha()
.
<mais comandos>
.
Return(.T.)
```

Neste exemplo, a variável nVar é criada como privada e inicializada com o valor 10. Quando a função Filha é executada, nVar ainda existe e, diferente de uma variável local, pode ser acessada pela função Filha. Quando a função Pai terminar, nVar será destruída e qualquer declaração de nVar anterior se tornará acessível novamente.

Variáveis Públicas

Pode-se criar variáveis públicas dinamicamente no código com o identificador PUBLIC. As variáveis públicas continuam a existir e mantêm seu valor até o fim da execução.

É possível criar uma variável privada com o mesmo nome de uma variável pública existente. Entretanto, não é permitido criar uma variável pública com o mesmo nome de uma variável privada existente.

Uma vez criada, uma variável pública é visível em todo o programa onde foi declarada até que seja escondida por uma variável privada criada com o mesmo nome. A nova variável privada criada esconde a variável pública existente, e esta se tornará inacessível até que a nova variável privada seja destruída. Por exemplo:

```
Function Pai()
Public nVar := 10
.
<comandos>
.
Filha()
.
<mais comandos>
.
Return(.T.)
```

Neste exemplo, nVar é criada como pública e inicializada com o valor 10. Quando a função Filha é executada, nVar ainda existe e pode ser acessada. Diferente de variáveis locais ou privadas, nVar ainda existe após o término da a execução da função Pai.

Diferentemente dos outros identificadores de escopo, quando uma variável é declarada como pública sem ser inicializada, o valor assumido é falso (.F.) e não nulo (nil).

Tipagem de Dados

Os tipos de dados disponíveis são: **numeric, character, date, codeblock, logical, array e object.**

Em ADVPL é utilizado a notação húngara para tipagem, isso quer dizer que uma letra é utilizada para indicar cada tipo de dado, essa letra será mostrada nas mensagens de erro.

Tipo	Descrição	Indicador
numeric	Utilizado para valores numéricos inteiros ou decimais, positivos ou negativos.	N
char / character	Utilizado para valores do tipo caracter	C
date	Utilizado para armazenar datas	D
block / codeblock	Armazena um bloco de código para macro execução	B
logical	Armazena valores lógicos, verdadeiro (.T.) ou falso (.F.)	L
array	Utilizado para armazenar uma matriz de valores	A
object	Armazena objetos de interface ou classes	O

Figura 3 – Tipos de variáveis

Funções

A criação de funções com o uso de tipagem de dados deve seguir a seguinte sintaxe:

```
Function <Nome da Funcao>([<parametro1> as <Tipo1>, <paramentroN as <TipoN>]) [as <Tipo>]
    ...
Return
```

O <Nome da Funcao> deve seguir os mesmos critérios usados em uma função sem tipagem de dados.

Cada parâmetro deve ser especificado com o tipo de dado desejado. O retorno da função também pode ter um tipo de dado indicado.

Classe

A uso de tipo de dados em classes deve usar a seguinte sintaxe:

```
Class <Nome da Classe>

    // Declaração de propriedades

    Data <nPropriedad1> [as <tipo>]

    Data <nPropriedad2> [as <tipo>]



    // Declaração de métodos

  Method New()      // Construtor, retorna Self

   Method <Nome do Método>([<Param1>, <Param2>,...,<ParamN>]) [as <tipo>]

EndClass

// Construtor da Classe

Method New() Class <Nome da Classe>

Return Self

// Definição de métodos

Method <Nome do Método>(nParam1) Class <Nome da Classe> [as <tipo>]

Local n1 as numeric



Parameter <Param1> [as <tipo>]



ConOut("Classe:MethodA")



Return n1
```

Em classes, podemos utilizar a tipagem de dados em propriedades e métodos.

Na construção de métodos os parâmetros e seus respectivos tipos devem ser especificados após a declaração das variáveis locais.

Compilação

No processo de compilação de código fonte, que utilize os recursos de tipagem de dados, serão avaliados conforme as definições, as variáveis, parâmetros e retorno de dados.

Caso alguma inconsistência na definição dos tipos seja encontrada será emitida uma mensagem de warning mostrando em qual linha está à divergência na declaração dos dados.

Por exemplo, na compilação do código fonte abaixo:

```
#include  "protheus.ch"


Function xTipagem()
Local c1 as character
Local n1 as numeric


n1 := "ABCDE"
c1 := 10


Return
```

Temos o seguinte resultado na compilação:

TIPAGEM.PRW(7) warning W0015 Incompatible Types : cannot convert from 'C' to 'N'

TIPAGEM.PRW(8) warning W0015 Incompatible Types : cannot convert from 'N' to 'C'

Execução

Durante a execução de uma função e/ou classe que utilize o recurso de tipagem de dados existe uma verificação do tipo da variável que é utilizada na passagem de parâmetro com o que foi definido no protótipo da função.

Caso os tipos de dados sejam diferentes será exibido uma mensagem no console do TOTVS AppServer e o valor NIL será atribuído a variável que foi passada como parâmetro, causando um comportamento diferente do esperado ou erro de tipo de dado.

Para exemplificar este comportamento temos o seguinte exemplo:

```
User Function xTMain()
Local c1 as character
Local l1 as logical

c1 := "ABCDE"
l1 := .T.

ConOut("Chamada 1 - xTLog")
ConOut(U_xTLog(l1))

ConOut("Chamada 2 - xTLog")
ConOut(U_xTLog(c1)) //Invalid parameter
Return

User Function xTLog(l1 as logical)
 If l1 = .T.
      ConOut("Logical .T.")
Else
      ConOut("Logical .F.")
EndIf

Return l1
```

Neste exemplo temos duas funções sendo que a primeira (xTMain) faz duas chamadas da função xTLog, sendo que a primeira chamada utiliza um parâmetro lógico, como

esperado no protótipo da função e a segunda utiliza um parâmetro do tipo character, incorreto para a função.

Após a execução temos os seguintes resultados:

```
Chamada 1 - xTLog
Logical .T.
.T.

Chamada 2 - xTLog
*** Warning - Mismatched parameters type calling U_XTLOG at parameter 1 - expected L -> C
Logical .F.
NIL
```

Na chamada 1 a execução ocorre conforme o esperado pois o parâmetro utilizado é do mesmo tipo indicado no protótipo da função.

Na chamada 2 é passado um parâmetro do tipo character, diferente do logical indicado no protótipo da função, sendo assim quando a função xTLog é executada o valor da variável l1 é NIL, diferente de true (.T.) atribuído a ela antes da chamada da função, sendo assim o resultado é diferente da primeira chamada.

Tipos de Dados

O ADVPL não é uma linguagem de tipos rígidos (strongly typed), o que significa que variáveis de memória podem diferentes tipos de dados durante a execução do programa.

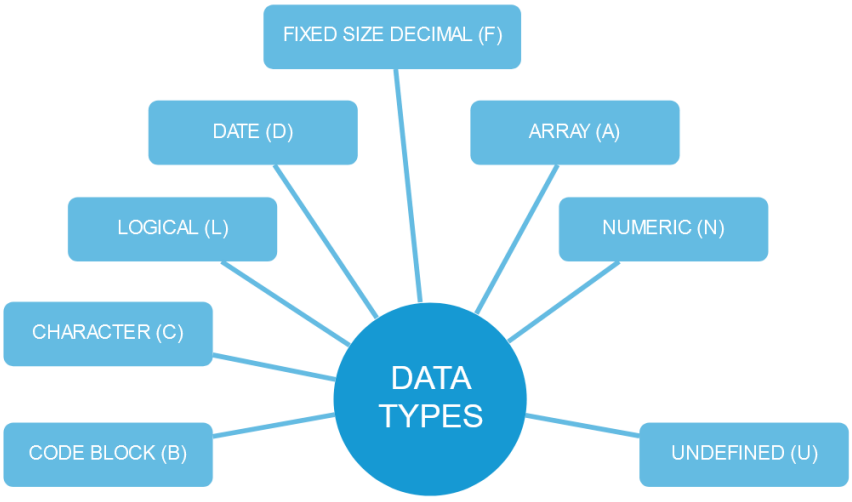

Figura 4 - Tipos de Dados

Veja também

- Arredondamento
- Decimais de Ponto Fixo
- Limite do tipo de dado Numérico

Array (A)

Matrizes são um tipo de dado especial. É a disposição de outros elementos em colunas e linhas. O ADVPL suporta matrizes uni ou multidimensionais. Os elementos de uma matriz são acessados através de índices numéricos iniciados em 1, identificando a linha e coluna para quantas dimenões existirem.

Uma matriz pode conter no máximo 100000 elementos, independentemente do número de dimensões.

Matrizes devem ser utilizadas com cautela, pois se forem muito grandes podem exaurir a memória do servidor.

Code Block (B)

O bloco de código é um tipo de dado especial. É utilizado para armazenar instruções escritas em ADVPL que poderão ser executadas posteriormente.

Character (C)

Strings ou cadeias de caracteres são identificadas em ADVPL por blocos de texto entre aspas duplas (") ou aspas simples ('):

- "Olá mundo!"
- 'Esta é uma string'
- "Esta é 'outra' string"

Uma variável do tipo caractere pode conter strings com no máximo 1 Mb, ou seja, 1048576 caracteres.

A declaração de conteúdos string em ADVPL não possui caracteres de "escape" para a declaração de strings. Uma string iniciada com aspas simples deve terminar com aspas simples. Caso seja necessário inserir uma aspa simples em uma string, você pode delimitá-la com aspas duplas; e vice-versa. Caso seja necessário declarar uma string que, ao mesmo tempo contenha aspas simples ou duplas, você deve realizar uma concatenação. Por exemplo:

"Esta é "+""+"uma"+""+" string 'diferente'"

Date (D)

O ADVPL tem um tipo de dados específico para datas. Internamente as variáveis deste tipo de dado são armazenadas como um número correspondente a data *juliana*.

Variáveis do tipo de dados Data não podem ser declaradas diretamente, e sim através da utilização de funções específicas como por exemplo ctod que converte uma string para data.

Object (O)

Trata-se de um tipo que permite vários tipos de dados: caracter, numérico, logico, data, bloco de códigos e inclusive a aplicação de funções internas.

Logical (L)

Valores lógicos em ADVPL são identificados através de **.T.** para verdadeiro e **.F.** para falso (independentemente se os caracteres estiverem em maiúsculo ou minúsculo).

Numeric (N)

O tipo numérico do ADVPL trabalha com aritmética de ponto flutuante, capaz de armazenar números inteiros e números fracionários.

- 2
- 43.53
- 0.5
- 0.00001
- 1000000

Uma variável do tipo de dado numérico pode conter números de mais de 15 dígitos, incluindo os dígitos decimais, porém a precisão garantida é de 15 dígitos, da esquerda para a direita – contemplando a parte inteira e dígitos decimais.

Blocos de Código

Blocos de código são um conceito existente há muito tempo em linguagens xBase. Não como algo que apareceu da noite para o dia, e sim uma evolução progressiva utilizando a combinação de muitos conceitos da linguagem para a sua implementação.

Primeiro Lembrete

O ADVPL é uma linguagem baseada em funções. Funções têm um valor de retorno. Assim como o operador de atribuição **:=**. Assim, ao invés de escrever:

```
x := 10 // Atribui o valor 10 à variável chamada X
Alert("Valor de x: " + cValToChar(x))
```

Pode-se escrever:

```
// Atribui e então exibe o valor da variável X
Alert("Valor de x: " + cValtoChar(X := 10))
```

A expressão x:=10 é avaliada primeiro, e então seu resultado (o valor de X, que agora é 10) é passada para a função cValToChar() para a conversão para caracter e, em seguida, para a função Alert() para a exibição. Por causa desta regra de precedência é possível atribuir um valor a mais de uma varíavel ao mesmo tempo:

```
Z := Y := X := 0
```

Por causa dessa regra, essa expressão é avaliada como se fosse escrita assim:

```
Z := ( Y := (X := 0) )
```

Apesar do ADVPL avaliar expressões da esquerda para a direita, no caso de atribuições isso acontece ao contrário, da direita para a esquerda. O valor é atribuído à variável X, que retorna o valor para ser atribuído à variável Y e assim sucessivamente. Pode-se dizer que o zero foi "propagado através da expressão".

Outro Lembrete

Em ADVPL pode-se juntar diversas linhas de código em uma única linha física do arquivo. Por exemplo, o código:

```
If lAchou   Alert("Cliente encontrado!")
 Endif
```

pode ser escrito assim:

```
If lAchou ; Alert("Cliente encontrado!") ; Endif
```

O ponto-e-vírgula indica ao ADVPL que a nova linha de código está para começar. Pode-se então colocar diversas linhas lógicas de código na mesma linha física através do editor de texto utilizado. Apesar da possibilidade de se escrever todo o programa assim, em uma única linha física, isto não é recomendado pois dificulta a legibilidade do programa e, consequentemente, a manutenção.

Lista de Expressões

A evolução dos blocos de código começa com as listas de expressões. Nos exemplos a seguir, o símbolo **==>** indicará o retorno da expressão após sua avaliação (seja para atribuir em uma variável, exibir para o usuário ou imprimir em um relatório), que será impresso em um relatório por exemplo.

Duas Linhas de Código

```
@00,00 PSAY x := 10     ==>    10
@00,00 PSAY y := 20     ==>    20
```

Cada uma das linhas terá a expressão avaliada, e o valor da variável será então impresso.

Duas Linha de Código em uma , Utilizando Ponto-e-Vírgula

Este é o mesmo código que o anterior, apenas escrito em uma única linha:

```
Alert( cValToChar( x := 10 ; y := 20 ) )    ==>    10
```

Apesar desse código se encontrar em uma única linha física, existem duas linhas lógicas separadas pelo ponto-e-vírgula. Ou seja, esse código é equivalente a:

```
Alert( cValToChar( x := 10 ) )
 y := 20
```

Portanto, apenas o valor 10 da variável x será passado para as funções cValToChar() e Alert() para ser exibido. E o valor 20 apenas será atribuído à variável y.

Convertendo para uma Lista de Expressões

Quando parênteses são colocados ao redor do código e o sinal de ponto-e-vírgula substituído por uma vírgula apenas, o código torna-se uma lista de expressões:

Alert(cValToChar ((X := 10 , Y := 20))) ==> 20

O valor de retorno resultante de uma lista de expressões é o valor resultante da última expressão ou elemento da lista. Funciona como se fosse um pequeno programa ou função, que retorna o resultado de sua última avaliação (efetuadas da esquerda para a direita).Neste exemplo, a expressão x := 10 é avaliada, e então a expressão y := 20, cujo valor resultante é passado para a função Alert() e cValToChar(), e então exibido. Depois que essa linha de código é executada, o valor de X é igual a 10 e o de y igual a 20, e 20 será exibido.

Teoricamente, não há limitação para o número de expressões que podem ser combinadas em uma lista de expressões. Na prática, o número máximo é por volta de 500 símbolos.Debugar listas de expressões é difícil porque as expressões não estão divididas em linhas de código-fonte, o que torna todas as expressões associadas a uma mesma linha de código. Isto pode tornar muito difícil determinar onde um erro ocorreu.

Onde pode-se utilizar uma Lista de Expressões ?

O propósito principal de uma lista de expressões é agrupá-las em uma única unidade. Em qualquer lugar do código ADVPL que uma expressão simples pode ser utilizada, pode-se utilizar uma lista de expressões. E ainda, pode-se fazer com que várias coisas aconteçam onde normalmente apenas uma aconteceria.

```
X := 10 ; Y := 20
 If X > Y
      Alert("X")
      Z := 1
 Else
      Alert("Y")
      Z := -1
 Endif
```

Aqui temos o mesmo conceito, escrito utilizando listas de expressões na função iif:

```
X := 10 ; Y := 20
 iif(  X > Y ,  ;
      ( Alert("X"), Z :=  1 ) , ;
      ( Alert("Y"), Z := -1 )   )
```

De Listas de Expressões para Blocos de Código

Considere a seguinte lista de expressões:

```
Alert( cValToChar( ( x := 10, y := 20 ) ) )  ==>  20
```

O ADVPL permite criar funções, que são pequenos pedaços de código, como se fosse um pequeno programa, utilizados para diminuir partes de tarefas mais complexas e reaproveitar código em mais de um lugar num programa. Para mais detalhes, consulte a documentação sobre a criação de funções em ADVPL. Porém, a idéia neste momento é que a lista de expressões utilizada na linha anterior pode ser criada como uma função:

```
Function Lista()
 X := 10
 Y := 20
 Return Y
```

E a linha de exemplo com a lista de expressões pode ser substituída, tendo o mesmo resultado, por:

```
Alert( cValToChar( Lista() ) )  ==>  20
```

Como mencionado anteriormente, uma lista de expressões é como um pequeno programa ou função. Com poucas mudanças, uma lista de expressões pode se tornar um bloco de código:

```
( X := 10 , Y := 20 )   // Lista de Expressões{|| X := 10 , Y := 20 } // Bloco de Código
```

Observe as chaves {} utilizadas no bloco de código. Ou seja, um bloco de código é uma matriz. Porém na verdade, não é uma lista de dados, e sim uma lista de comandos, uma lista de código.

```
// Isto é uma matriz de dadosA := {10, 20, 30}
 // Isto é um bloco de código, porém funciona como
 // se fosse uma matriz de comandos
 B := {|| x := 10, y := 20}
```

Executando um Bloco de Código

Diferentemente de uma matriz, não se pode acessar elementos de um bloco de código através de um índice numérico. Porém blocos de código são semelhantes a uma lista de expressões, e a uma pequena função. Ou seja, podem ser executados. Para a execução, ou avaliação, de um bloco de código, deve-se utilizar a função EVal():

```
B := {|| x := 10, y := 20}
nRes := Eval(B)  ==>  20
```

Essa função recebe como parâmero um bloco de código e avalia todas as expressões contidas neste bloco de código, retornando o resultado da última expressão avaliada. (O resultado de uma atribuição é o próprio conteúdo atribuído).

Passando Parâmetros

Já que blocos de código são como pequenas funções, também é possível a passagem de parâmetros para um bloco de código. Os parâmetros devem ser informados entre as barras verticais (||) separados por vírgulas, assim como em uma função.

```
B := {| N | X := 10, Y := 20 + N}
```

Porém, deve-se notar que já que o bloco de código recebe um parâmetro, um valor deve ser passado quando o bloco de código for avaliado.

```
C := Eval(B, 1)  ==>  21
```

Retorno de um bloco de código

O retorno de um bloco de código será o resultado da última expressão avaliada dentro do bloco de código. Veja o exemplo abaixo:

```
A = 0
B = {| x | IIF( x = 0 , time(), date()  ) }
C = Eval( B , 0 )
D = Eval( B , 1 )
```

Se a chamada do bloco de código informar o parâmetro 0 (zero), será retornado pelo bloco de código o retorno da função Time() do ADVPL. Ao ser chamado com um valor diferente de zero, será devolvido o retorno da função Date(), que retorna a data atual no servidor.

Utilizando Blocos de Código - Exemplo aSort()

Blocos de código podem ser utilizados em diversas situações. Geralmente são utilizados para executar tarefas quando eventos de objetos são acionados. Por exemplo, considere a matriz abaixo:

```
A := {"GARY HALL", "FRED SMITH", "TIM JONES"}
```

Esta matriz pode ser ordenada pelo primeiro nome, utilizando-se a chamada da função ASort(A), resultando na matriz com os elementos ordenados dessa forma:

```
{"FRED SMITH", "GARY HALL", "TIM JONES"}
```

A ordem padrão para a função ASort é ascendente. Este comportamento pode ser modificado através da informação de um bloco de código que ordena a matriz de forma descendente:

```
B := { |X, Y| X > Y }
aSort(A ,,, B)
```

O bloco de código utilizado neste exemplo (de acordo com a documentação da função ASort()) deve ser escrito para aceitar dois parâmetros, que são os dois

elementos da matriz para comparação. Observe que o bloco de código não conhece que elemento está comparando - a função aSort() seleciona os elementos e passa-os para o bloco de código. O bloco de código compara-os e retorna verdadeiro (.T.) se encontram na ordem correta, caso contrário retorna falso (.F.). A função ASort() executará quantas chamadas forem necessárias para ordenar o *array* até que a sequência de elementos esteja devidamente ordenada.

Para ordenar a mesma matriz pelo último nome, também em ordem decrescente, pode-se utilizar o seguinte bloco de código:

```
B := { |X, Y| Substr(X,At(" ",X)+1) > Substr(Y,At(" ",Y)+1) }
```

Observe que este bloco de código procura e compara as partes dos caracteres imediatamente seguinte a um espaço em branco. Depois de utilizar esse bloco de código para a função ASort, a matriz estará na seguinte ordenação:

{"GARY HALL", "TIM JONES", "FRED SMITH"}

Utilização Avançada de Blocos de Código

É possivel criar um bloco de código dinamicamente no ADVPL, utilizando-se do operador de macro-execução, bem como chamar de dentro de um bloco de código qualquer função da linguagem ADVPL e funções compiladas no repositório. Estes recursos possuem algumas particularidades de uso, exploradas nos exemplos abaixo.

Exemplo 01 : Bloco de código referenciando variável local

```
USER FUNCTION BL001()
Local nContador := 0
Local bIncr := {  | i | nContador := nContador + i }
conout("Antes : " + str( nContador , 5 ))
BLCONTA( bIncr )
conout("Depois : " + str( nContador , 5 ))
Return
```

STATIC FUNCTION BLCONTA(bBloco)
Eval(bBloco , 5)
Return

Após executar a função U_BL001(), o resultado obtido no log de console do Application Server será :

Antes: 0
Depois: 5

Repare que a variável nContador foi declarada no escopo LOCAL da função BL001. A princípio, por ser uma variável local, apenas instruções dentro da função BL001 poderiam acessar ou alterar este valor, ou o conteúdo desta variável poderia ser passado por referência para uma outra função. No exemplo, esta variável foi usada dentro de um bloco de código, onde o bloco atualiza o conteúdo da variável com o conteúdo atual somado ao parâmetro recebido pelo bloco de código. O bloco de código foi passado como parâmetro para outra função, onde dentro dela foi feita a chamada deste bloco através da função Eval(), informando o parâmetro numérico 5 . Ao ser executado o bloco de código, a variável nContador referenciada dentro do bloco de código será atualizada. Quando a função retorna, e consultamos o conteúdo de nContador, vemos que ele foi atualizado para o número 5.

Exemplo 02 : Bloco de código com parâmetros por referência

```
USER FUNCTION BL002()
 Local bTeste := { |x| x := x + 1 , u_mostra(x)   }

 Local nContador := 0
 // retorna NIL , mostra valor 1, chamada passando apenas nContador

Eval( bTeste , nContador )
// nContador continua com 0 ( zero )

conout( " Contador = " + str( nContador,10 ))
// retorna NIL , mostra valor 1, chama passando nContador por referência

Eval( bTeste , @nContador )
// Mas agora o nContador está com 1 ( um )

conout( " Contador = " + str( nContador,10 ))


ReturnUser Function Mostra(nValor)
```

```
conout( "Valor = " + str( nValor , 10) )
Return
```

Neste exemplo, o bloco de código não referência explicitamente a variável nContador. Na verdade, ele recebe um parâmetro em x, usa o próprio parâmetro x para receber o resultado da soma dele mesmo com o numero 1, e chama uma função de usuário (U_MOSTRA), para mostrar o conteúdo de x. Os parâmetros de um bloco de código são como os parâmetros de uma função ADVPL normal: São todos considerados locais dentro do bloco de código. Da mesma forma que uma função ADVPL, ao passar uma variável com um conteúdo *string*, data, numérico ou booleano para uma função, a função recebe uma cópia do conteúdo desta variável, de modo que uma alteração na variável que recebe o parâmetro não reflete na variável usada na chamada. Porém, é possível passar um parâmetro por referência a uma função, e se a variável que recebe este parâmetro é alterada dentro da função, esta alteração é refletida na variável original usada na chamada da função. Esta mesma regra é válida para um bloco de código, onde podemos passar por referência um ou mais parâmetros na chamada da função Eval(), reproduzindo o mesmo comportamento.

Dicas Importantes/Informações Adicionais

- Uma chamada de função dentro de um bloco de código, pode conter uma chamada para uma STATIC FUNCTION. Porém, como podemos passar um bloco de código para uma função de outro fonte e executar o Eval() em outra pilha de execução, e fontes diferentes podem conter uma função STATIC com o mesmo nome, o comportamento será indeterminado, isto é, não é possível afirmar qual das funções será chamada, e este comportamento pode mudar dependendo da ordem de execução do bloco de código. É fortemente não recomendado que um bloco de código seja criado com chamada de função de escopo estático.

- É boa prática de uso de blocos de código evitar a troca de contexto de um bloco de código para níveis superiores da pilha de chamadas ADVPL. Um bloco de código trafegado na pilha de chamadas leva consigo uma área de memória correspondente ao ambiente de escopo local da pilha de chamadas no momento da declaração do bloco de código (para bloco de código estático), ou da criação do bloco de código na memória (para bloco de código dinâmico criado com macro-execução). Caso o bloco de código seja trafegado para um nível superior da pilha de execução de funções ADVPL e armazenado neste nível (por exemplo, guardado em uma variável de escopo estático - STATIC), a memória alocada para levar o bloco de código e o contexto de criação somente será liberada automaticamente pelo servidor de aplicação quando o programa for finalizado, ou manualmente caso a variável usada para armazenar o bloco de código receba o conteúdo NIL, e o bloco de código em si não seja mais referenciado em nenhum outro ponto da aplicação em tempo de execução.

- Um bloco de código deve obedecer aos requisitos para o qual o mesmo foi criado. Uma função projetada para receber um bloco de código como argumento provavelmente vai, em algum momento, chamar a execução deste bloco. A passagem de parâmetros e retorno esperados do bloco são informações que devem ser providas na documentação da função que está realizando a chamada, para que o desenvolvedor saiba o que ele receberá quando o bloco for executado, sob que contexto originalmente a aplicação fará a execução do bloco de codigo, e qual é o retorno esperado para o bloco de código.

- Seguindo as boas práticas de programação ADVPL, a utilização de um bloco de código pode facilitar muito o desenvolvimento de uma aplicação, mas não é uma solução que serve para todos os problemas. Um bloco de código muito grande ou muito amarrado pode tornar mais difícil a manutenção e a depuração de um código. Se o bloco de código ficar muito extenso, e existe a real necessidade de utilizá-lo, ao invés de colocar 4 KB de *statements* dentro de um bloco de código, escreva uma função que atenda a necessidade da aplicação, e use o bloco de código para realizar a chamada da função.

Matrizes

Matrizes ou arrays, são coleções de valores. Ou, de uma maneira mais fácil de entender, uma lista. Uma matriz pode ser criada através de diferentes maneiras. Consulte a documentação sobre Inicialização de Matrizes para maiores detalhes.

Cada item em uma matriz é referenciado pela indicação de sua posição numérica na lista, iniciando pelo número 1. O exemplo a seguir declara uma variável, atribui uma matriz de três elementos a ela, e então exibe um dos elementos e o tamanho da matriz:

```
Local aLetras                       // Declaração da variável
aLetras := {"A", "B", "C"}          // Atribuição da matriz à variável
Alert(aLetras[2])                   // Exibe o segundo elemento da matriz
Alert(cValToChar(Len(aLetras)))     // Exibe o tamanho da matriz
```

O ADVPL permite a manipulação de matrizes facilmente. Enquanto que em outras linguagens como C ou Pascal é necessário alocar memória para cada elemento de uma matriz (o que tornaria a utilização de "pointeiros" necessária), o ADVPL se encarrega de gerenciar a memória e torna simples adicionar elementos a uma matriz, utilizando a função aAdd:

```
aAdd(aLetras,"D")       // Adiciona o quarto elemento ao final da matriz
Alert(aLetras[4])       // Exibe o quarto elemento
Alert(aLetras[5])       // Erro! Não há um quinto elemento na matriz
```

Matrizes como Estruturas

Uma característica interessante do ADVPL é que uma matriz pode conter qualquer coisa: números, datas, lógicos, caracteres, objetos, etc. E ao mesmo tempo. Em outras palavras, os elementos de uma matriz não precisam ser necessariamente do mesmo tipo de dado, em contraste com outras linguagens como C e Pascal.

aFunct1 := {"Pedro",32,.T.}

Esta matriz contem uma string, um número e um valor lógico. Em outras linguagens como C ou Pascal, este "pacote" de informações pode ser chamado como um "struct" (estrutura em C, por exemplo) ou um "record" (registro em Pascal, por exemplo). Como se fosse na verdade um registro de um banco de dados, um pacote de informações construído com diversos campos. Cada campo tendo um pedaço diferente de dado.

Suponha que no exemplo anterior, o array aFunct1 contenha informações sobre o nome de uma pessoa, sua idade e sua situação matrimonial. Os seguintes #defines podem ser criados para indicar cada posição dos valores dentro da matriz:

```
#define FUNCT_NOME   1
#define FUNCT_IDADE  2
#define FUNCT_CASADO 3
```

E considere mais algumas matrizes para representar mais pessoas:

```
aFunct2 := {"Maria"  , 22, .T.}
aFunct3 := {"Antônio", 42, .F.}
```

Os nomes podem ser impressos assim:

```
Alert(aFunct1[FUNCT_NOME])
Alert(aFunct2[FUNCT_NOME])
Alert(aFunct3[FUNCT_NOME])
```

Agora, ao invés de trabalhar com variáveis individuais, pode-se agrupá-las em uma outra matriz, do mesmo modo que muitos registros são agrupados em uma tabela de banco de dados:

```
aFuncts := {aFunct1, aFunct2, aFunct3}
```

Que é equivalente a isso:

```
aFuncts := { {"Pedro"  , 32, .T.}, ;
             {"Maria"  , 22, .T.}, ;
             {"Antônio", 42, .F.} }
```

aFuncts é uma matriz com 3 linhas por 3 colunas. Uma vez que as variáveis separadas foram combinadas em uma matriz, os nomes podem ser exibidos assim:

```
Local nCount
For nCount := 1 To Len(aFuncts)
Alert(aFuncts[nCount,FUNCT_NOME])
// O acesso a elementos de uma matriz
multidimensional
// pode ser realizado também desta forma:
// aFuncts[nCount][FUNCT_NOME]
Next nCount
```

A variável nCount seleciona que funcionário (ou que linha) é de interesse. Então a constante FUNCT_NOME seleciona a primeira coluna daquela linha.

Cuidados com Matrizes

Matrizes são listas de elementos, portanto memória é necessária para armazenar estas informações. Como as matrizes podem ser multidimensionais, a memória necessária será a multiplicação do número de itens em cada dimensão da matriz, considerando-se o tamanho do conteúdo de cada elemento contido nesta. Portanto o tamanho de uma matriz pode variar muito.

A facilidade da utilização de matrizes, mesmo que para armazenar informações em pacotes como descrito anteriormente, não é compensada pela utilização em memória quando o número de itens em um array for muito grande. Quando o número de elementos for muito grande deve-se procurar outras soluções, como a utilização de um arquivo de banco de dados temporário.

Não há limitação para o número de dimensões que uma matriz pode ter, mas o número de elementos máximo (independentes das dimensões onde se encontram) é de 100000.

Inicializando Matrizes

Algumas vezes o tamanho da matriz é conhecido previamente. Outras vezes o tamanho da matriz só será conhecido em tempo de execução.

Se o tamanho da matriz é conhecido.

Se o tamanho da matriz é conhecido no momento que o programa é escrito, há diversas maneiras de implementar o código.

```
01 Local nCnt
02 Local aX[10]
03 Local aY := Array(10)
04 Local aZ := {0,0,0,0,0,0,0,0,0,0}
05
06 For nCnt := 1 To 10
07     aX[nCnt] := nCnt * nCnt
08 Next nCnt
```

Este código preenche a matriz com uma tabela de quadrados. Os valores serão 1, 4, 9, 16 ... 81, 100. Note que a linha 07 se refere à variável aX, mas poderia também trabalhar com aY ou aZ. O objetivo deste exemplo é demonstrar trÊs

modos de criar uma matriz de tamanho conhecido no momento da criação do código.

Na linha 02 a matriz é criada usando aX[10]. Isto indica ao ADVPL para alocar espaço para 10 elementos na matriz. Os colchetes [e] são utilizados para indicar o tamanho necessário.

Na linha 03 é utilizada a função array com o parâmetro 10 para criar a matriz, e o retorno desta função é atribuído à variável aY.

Na linha 03 é efetuado o que se chama "desenhar a imagen da matriz". Como pode-se notar, existem dez 0´s na lista encerrada entre chaves ({}). Claramente, este método não é o utilizado para criar uma matriz de 1000 elementos. O terceiro método difere dos anteriores porque inicializa a matriz com os valores definitivos. Nos dois primeiros métodos, cada posição da matriz contém um valor nulo (Nil) e deve ser inicializado posteriormente.

A linha 07 demonstra como um valor pode ser atribuído para uma posição existente em uma matriz especificando o índice entre colchetes.

Se o tamanho da matriz não é conhecido.

Se o tamanho da matriz não é conhecido até o momento da execução do programa, há algumas maneiras de criar uma matriz e adicionar elementos a ela. O exemplo a seguir ilustra a idéia de criação de uma matriz vazia (sem nenhum elemento) e adição de elementos dinamicamente.

```
01 Local nCnt
02 Local aX[0]
03 Local aY := Array(0)
04 Local aZ := {}
05
06 For nCnt := 1 To nSize
07     aAdd(aX,nCnt*nCnt)
08 Next nCnt
```

A linha 02 utiliza os colchetes para criar uma matriz vazia. Apesar de não ter nenhum elemento, seu tipo de dado é matriz.

Na linha 03 a chamada da função array cria uma matriz sem nenhum elemento.

Na linha 04 está declarada a representação de uma matriz vazia em ADVPL. Mais uma vez, estão sendo utilizadas as chaves para indicar que o tipo de dados da variável é matriz. Note que {} é uma matriz vazia (tem o tamanho 0), enquanto {Nil} é uma matriz com um único elemento nulo (tem tamanho 1).

Porque cada uma destas matrizes não contém elementos, a linha 07 utiliza a função aadd para adicionar elementos sucessivamente até o tamanho necessário (especificado por exemplo na variável nSize).

Material de referência:

Variáveis de memória

- Criação e Atribuição de Variáveis
- Diferenciação entre variáveis e nomes de campos
- Escopo de variáveis
  - O Contexto de Variáveis dentro de um Programa
  - Variáveis Estáticas
  - Variáveis Locais
  - Variáveis Privadas
  - Variáveis Públicas
- Tipagem de Dados
  - Introdução
  - Requisitos
  - Tipos de Dados
  - Funções
  - Classe
  - Compilação
  - Execução
- Tipos de Dados
  - Blocos de Código
  - Matrizes
    - Inicializando Matrizes

.FIM