A linguagem Object Pascal

A linguagem de programação PASCAL foi criada para ser uma ferramenta educacional, isto no início da década de 70 pelo Prof. Niklaus Wirth da Universidade de Zurique. Foi batizada pelo seu idealizador em homenagem ao grande matemático Blaise Pascal, inventor de uma das primeiras máquinas lógicas conhecidas. Foi baseada em algumas linguagens estruturadas existentes na época, ALGOL e PLI.

Apesar de seu propósito inicial, o PASCAL começou a ser utilizado por programadores de outras linguagens, tornando-se, para surpresa do próprio Niklaus, um produto comercial. Contudo, somente ao final de 1983 foi que a empresa americana Borland International lançou o TURBO PASCAL.

A linguagem Object Pascal é a linguagem por trás de quase todas as partes de um aplicativo no Delphi. Os arquivos de projeto e as Units, como já vimos, são escritos em Object Pascal. O código usado para criar os componentes predefinidos do Delphi é também praticamente todo nessa linguagem.

Nos capítulos anteriores usamos várias vezes pequenas partes da linguagem Object Pascal (dentro do código para os eventos, por exemplo). Neste capítulo veremos como usar recursos mais avançados da linguagem e como criar programas que não dependam somente da interface com o usuário.

Elementos Básicos

Entendendo identificadores

Identificadores (*Identifiers*) são os nomes que identificam os elementos de um programa em Object Pascal. Exemplos de identificadores são nomes de variáveis, constantes, procedures, functions e componentes. Em Object Pascal todos os identificadores usados devem ser **declarados**. Quando você declara um identificador, você está definindo um *tipo* para ele (como inteiro, String, etc.).

Um identificador deve seguir as seguintes regras básicas:

- Um identificador pode ter até 63 caracteres. Qualquer caractere que passe desse limite é ignorado pelo compilador do Delphi.

- Identificadores devem começar sempre com letras, ou com o caractere de sublinhado (_). Um identificador não pode começar com um número. Os outros caracteres de um identificador (do segundo em diante) podem ser letras, números ou sublinhados. Nenhum outro caractere é permitido.

- Um identificador não pode ser uma palavra reservada da linguagem Object Pascal (como **begin**, **if** ou **end**, por exemplo).

Palavras reservadas

São palavras que têm um sentido predeterminado na linguagem e não podem ser usadas como identificadores.

ABSOLUTE	END	INLINE	PROCEDURE	TYPE
AND	EXTERNAL	INTERFACE	PROGRAM	UNIT
ARRAY	FILE	INTERRUPT	RECORD	UNTIL
BEGIN	FOR	LABEL	REPEAT	USES
CASE	FORWARD	MOD	SET	VAR
CONST	FUNCTION	NIL	SHL	WHILE
DIV	GOTO	NOT	SHR	WITH
DO	IF	OF	STRING	XOR
DOWNTO	IMPLEMENTATION	OR	THEN	
ELSE	IN	PACKED	TO	

Expressões Aritméticas

São expressões onde utilizamos os números inteiros ou reais como operandos e os operadores aritméticos, dando sempre como resultado valores numéricos.

Operadores Aritméticos

Os operadores aritméticos representam as operações mais comuns da matemática. São eles:

Operador	Operação	Operandos	Resultado
+	Adição	Inteiro, Real	Inteiro, Real
-	Subtração	Inteiro, Real	Inteiro, Real
*	Multiplicação	Inteiro, Real	Inteiro, Real
/	Divisão Real	Inteiro, Real	Real
DIV	Divisão Inteira	Inteiro	Inteiro
MOD	Resto da Divisão	Inteiro	Inteiro

Exemplos:

Expressão	Resultado
1 + 2	3
5.0 - 1	4.0
2 * 1.5	3.0
5 / 2	2.5
5 DIV 2	2
5 MOD 2	1

Prioridades

Em uma expressão aritmética, a ordem de avaliação dos operadores obedece a tabela abaixo:

Prioridade	Operadores
1ª	* / DIV MOD
2ª	+ -

OBSERVAÇÕES:

a) Quando existe em uma expressão operadores com a mesma prioridade, a execução é da esquerda para direita.
b) Caso seja necessário alterar a ordem de prioridade, deve-se utilizar parênteses. A expressão entre parênteses terá prioridade máxima. Caso haja parênteses aninhados, a ordem de execução será do mais interno para o mais externo.

EXEMPLOS:

2 + 3 / 2 = 2 + 1.5 = 3.5
(2 + 3) / 2 = 5 / 2 = 2.5

Funções e Procedimentos Predefinidos

São subprogramas já prontos à disposição dos usuários, para o cálculo das funções matemáticas mais comuns.

Função	Finalidade	Tipo do argumento	Tipo do resultado
ABS(X)	Valor Absoluto	Inteiro, Real	o mesmo do argumento
FRAC(X)	Parte Fracionária	Real	Real
TRUNC(X)	Parte Inteira	Real	Inteiro
ROUND(X)	Valor Arredondado	Real	Inteiro
SQR(X)	Eleva ao quadrado	Inteiro, Real	o mesmo do argumento
SQRT(X)	Raiz quadrada	Inteiro, Real	Real
LN(X)	Logaritmo Natural	Real	Real
EXP(X)	Exponencial	Real	Real

Como não existe em Pascal um operador nem uma função específica para a operação de Potenciação, podemos conseguí-la utilizando as funções LN(X) e EXP(X). Para calcular o valor de X^N é suficiente usar:

EXP(LN(X)*N)

Exemplos:

Expressão	Resultado
ABS(-2.5)	2.5
ABS(8)	8
FRAC(5.234)	0.234
TRUNC(2.78)	2
ROUND(2.78)	3
SQR(2)	4
SQR(1.5)	2.25
SQRT(4)	2.0
SQRT(2.25)	1.5
EXP(LN(2)*3)	8

Expressões Lógicas

As operações lógicas podem ser consideradas afirmações que serão testadas pelo computador, tendo como resultado, um valor ***verdadeiro*** ou ***falso***. São utilizadas com os operadores ***relacionais*** e ***lógicos***.

Operadores Relacionais

São usados na comparação de duas expressões de qualquer tipo, retornando um valor lógico (**TRUE** ou **FALSE**) como resultado da operação.

Operador	Operação
=	igual
>	maior
<	menor
>=	maior ou igual
<=	menor ou igual
<>	diferente

Obs: as operações lógicas só podem ser efetuadas com relação a valores do mesmo tipo.

Exemplos:

Expressão	Resultado
1 = 2	FALSE
'A' = 'a'	FALSE
5 > 2	TRUE
3 <= 3	TRUE
TRUE < FALSE	FALSE
'JOAO' > 'JOSE'	FALSE
2 + 3 <> 5	FALSE
'comp' <> 'COMP'	TRUE
'11' < '4'	TRUE

Operadores Lógicos

São usados para combinar expressões lógicas.

Operador	Operação
not	não (negação)
and	e (conjunção)
or	ou (disjunção)

A tabela verdade (abaixo) apresenta o resultado de cada operador lógico, com os valores dados para as expressões lógicas A e B:

A	B	A and B	A or B	not A	not B
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE	TRUE

Prioridade

Em uma expressão lógica, a ordem de avaliação dos operadores segue a tabela abaixo:

Prioridade	Operadores
1ª	NOT
2ª	AND
3ª	OR
4ª	= > < >= <= <>

Como a ordem de precedência dos operadores lógicos é maior que a dos operadores relacionais, devem sempre ser usados parênteses quando se escrever uma expressão lógica complexa. Por exemplo:

(A > B) OR (B = C)

Declarando variáveis

A declaração de uma variável define o **nome** e o **tipo** da variável. Declarações de variáveis devem ser precedidas pela palavra-chave **var**. As declarações, junto com a palavra-chave **var** constituem o chamado **bloco var**.

Declarações devem ser agrupadas no início do programa, ou no início de uma *procedure* ou *function* (antes da palavra-chave **begin**).

O programa (trivial) abaixo mostra um exemplo de declaração de variáveis. São declaradas três variáveis inteiras (**X**, **Y** e **Soma**). Assume-se que há dois componentes *Edit* no formulário, onde são digitados os dois valores a serem somados. As funções **StrToInt** e **IntToStr** convertem o tipo de *String* para o *Integer* (Inteiro) e de *Integer* para *String*, respectivamente. Um caixa de mensagem é exibida no final, com o resultado da soma.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X,Y : Integer;
  Soma: Integer;
begin
  X := StrToInt(Edit1.Text);
  Y := StrToInt(Edit2.Text);
  Soma := X + Y;
  ShowMessage('Soma = ' + IntToStr(Soma));
end;
```

Declarando constantes

Constantes são usadas em programas para armazenar valores que não podem (ou não devem) ser alterados em um programa. Declarações de constantes devem ser precedidas pela palavra-chave **const**. As declarações das constantes, junto com a palavra-chave **const** constituem o chamado **bloco const**.

Ao contrário das variáveis, as constantes devem ter seus valores definidos logo na declaração. Além disso, os tipos das constantes não são definidos explicitamente. Eles são deduzidos pelo Delphi, de acordo com o valor definido para cada constante. Veja alguns exemplos:

```
const
 Altura  = 100;
 Largura = 200;
 VelocidadeMaxima = 225.17;
 Titulo           = 'Ecossistemas';
```

Aqui, as constantes "Altura" e "Largura" são armazenadas com tipo *Integer*, "VelocidadeMaxima" com tipo *Real* (ou *Double*) e "Titulo" com tipo *String*.

Note como é usado o símbolo "**=**" para definir os valores para as constantes e não o símbolo de atribuição: "**:=**".

Comentários

Um comentário é usado para aumentar a clareza de um programa, embora não seja analisado pelo computador.

Comentário Simples para uma única linha: //
Comentário de Múltiplas linhas: { }
Ex:

```
{
************** PROGRAMA REAJUSTA SALÁRIO *****************
  Finalidade.......: Calc. o reajuste de um salário em 20%
  Versão...........: 1.0.1
  Última Alteração.: 12/02/2008
*********************************************************
}

Program REAJUSTE_SALARIO;
  {
Uses
  Crt;
Const
  IND = 0.20; //indice do reajuste}
Var
  SAL_ATUAL, //salario atual
  SAL_NOVO, //novo salario
  AUMENTO: Real; //valor do aumento
Begin
  {leitura do salario atual}
  write('Digite o salario atual: ');
  readln(SAL_ATUAL);
  {calculo do reajuste}
  AUMENTO := SAL_ATUAL * IND;
  SAL_NOVO := SAL_ATUAL + AUMENTO;
  {exibicao do resultado}
  writeln('Novo Salario = ',SAL_NOVO:10:2);
End.
```

Formato de Programa Pascal

Pascal é uma linguagem altamente estruturada que possui uma rigidez definida, embora sua estrutura de programa seja flexível. Cada seção ou parte de um programa em Pascal deve aparecer numa seqüência apropriada e ser sistematicamente correta, senão ocorrerá um erro.

Por outro lado, no Pascal não há regras específicas para o uso de espaço, linhas quebradas, requisições e assim os comandos podem ser escritos no formato livre em quase todos os estilos em que o programador deseja utilizar.

Um programa escrito em Pascal tem o seguinte formato:

```
PROGRAM <identificador>;
<bloco>.
```

O <bloco>, por sua vez, está dividido em seis áreas, onde somente a última é obrigatória e devem obedecer a seqüência abaixo. São elas:

- Área de declaração de uso de unidades
- Área de declaração de tipos
- Área de declaração de constantes
- Área de declaração de variáveis
- Área de declaração de procedimentos e funções
- Área de comandos

Declaração de uso de unidades

Um programa Object Pascal pode fazer uso de algumas unidades padrão que estão disponíveis no Sistema, tais como: SysUtils, Windows, DOS, PRINTER, GRAPH, etc.

A área de declaração de uso de unidades possui o seguinte formato:

```
USES  <unidade> , ... , <unidade> ;
```

EXEMPLO:

```
USES SysUtils;
```

Declaração de Tipos

Serve para definirmos novos tipos e estruturas de dados. Não detalharemos esse tópico agora por ser assunto dos próximos capítulos.

Ex:

```
type
  TDiaUtil    = (Segunda, Terca, Quarta, Quinta, Sexta);
  TFabricante = (Chevrolet, Ford, Volkswagen);
  TSexo       = (Masculino, Feminino);
```

Declaração de Procedimentos e Funções

Nesta área são definidos os procedimentos e funções utilizados pelo programa. Também é um assunto que será detalhado mais adiante.

Ex:

```
function Dobro1(Numero: Integer): Integer;
begin
  Dobro := Numero*2;
end;

function Dobro2(Numero: Integer): Integer;
begin
  Result := Numero*2;
end;
```

Área de Comandos

É nesta área onde é inserido o algoritmo do programa. Os comandos são separados entre si pelo delimitador ponto-e-vírgula. A forma geral é:

```
Begin
  <comando> ;
   ... ;
  <comando>
End
```

Tipos de Dados

Tipos Simples

O tipo declarado para uma variável define o conjunto de valores permitidos para aquela variável. Há uma grande quantidade de tipos disponíveis em Object Pascal. Há vários tipos muito parecidos entre si, que são usados apenas para manter compatibilidade com versões anteriores da linguagem Pascal. Abaixo listamos os tipos mais comumente usados.

Númericos

Inteiros

São caracterizados como tipos inteiros, os dados numéricos positivos ou negativos. Excluindo-se destes qualquer número fracionário.

Ex: 0, -156, 10, 234, -80

Tipos	Mínimo	Máximo	Tamanho byte
Integer	-2.147.483.647	2.147.483.647	4
Cardinal	0	4294967295	4
Shortint	-128	127	1
Smallint	-32768	32767	2
Longint	-2.147.483.647	2.147.483.647	4
Int64	-2^63	2^63-1	8
Byte	0	255	1
Word	0	65535	2
Longword	0	4294967295	4

Reais

São caracterizados como tipos reais, os dados numéricos positivos, negativos e números fracionários.

Ex: 35.00, 0, -56, 1.2, -45.659

Tipos	Mínimo	Máximo	Tamanho byte	Precisão
Real	5.0 x 10^-324	1.7 x 10^308	8	15-16
Real48	2.9 x 10^-39	1.7 x 10^38	6	11-12
Single	1.5 x 10^-45	3.4 x 10^38	4	7-8
Double	5.0 x 10^-324	1.7 x 10^308	8	15-16
Extended	3.6 x 10^-4951	1.1 x 10^4932	10	19-20
Comp	-2^63+1	2^63 -1	8	19-20
Currency			8	19-20

Char

Representa um único caracter, escrito entre apóstrofos (‘). A maioria dos computadores utilizam a tabela de códigos ASCII para representar todos os caracteres disponíveis.

Ex:

'A', 'B', 'a', '1', '@', ' '

Boolean

São caracterizados como tipos boleanos, os dados com valores TRUE (verdadeiro) e FALSE (falso), sendo que este tipo de dado poderá representar apenas um dos dois valores. Ele é chamado de boleano, devido à contribuição do filósofo e matemático inglês George Boole na área da lógica matemática.

Tipos Estruturados

À medida que evoluímos nos tipos de programas que projetamos surge a necessidade de armazenar certos tipos de informações que os tipos simples de dados, sozinhos, não são capazes de guardar, devido à pequena quantidade de informação que uma variável desse tipo armazena. É para isso que existem os tipos de dados compostos, ou estruturados, ou, simplesmente, as estruturas de dados.

Estruturas de dados são construções de uma linguagem de programação que agregam um ou mais elementos de dados para formar um tipo de dado que armazena uma quantidade maior de informações.

Os elementos de dados que uma estrutura de dados armazena podem ou não ser do mesmo tipo. Existem estruturas de dados que trabalham só com um tipo de elemento, como o vetor, e outras que combinam elementos de dados diferentes, no caso do registro, por exemplo. Uma estrutura de dados também pode conter outra estrutura de dados.

As estruturas de dados são subdivididas em dois grupos, estáticas e dinâmicas. Uma estrutura de dados estática é aquela que não pode ter a sua quantidade de elementos de dados alterada durante a execução de um programa. As mais comuns são os vetores, as strings e os registros. Uma estrutura de dados dinâmica frequentemente tem elementos inseridos ou retirados do seu interior. Arquivos e listas, filas, pilhas e árvares são as estruturas de dados dinâmicas mais comuns.

Strings

Strings em Object Pascal são armazenados internamente como **arrays de caracteres** (arrays do tipo Char). Há dois tipos de Strings em Object Pascal: **Strings longos** e **Strings curtos**.

Ex:

```
'BONITO', 'Informática', '123’
```

Strings longos têm o comprimento praticamente ilimitado (2 GigaBytes). Eles são declarados usando somente a palavra-chave **string**. **String curtos** têm o comprimento limitado em 255 caracteres. Para declarar um String curto, use a palavra-chave **string** seguida pelo tamanho desejado, entre colchetes.

Uma string é declarada da seguinte maneira:

```
NomeDaString: String[TamanhodaString];          onde:
```

NomeDaString: contém o identificador a ser utilizado no programa;
String: é a palavra reservada da linguagem, utilizada para declarar variáveis desse tipo;
TamanhodaString: fornece a quantidade de caracteres máxima a ser alocada para a variável declarada.

Veja a seguir exemplos de declarações dos dois tipos de Strings:

```
var
  Nome: String[10];{um String curto com no máximo 10 caracteres}
  Texto: String;   {um String longo}
```

Como uma String é um array de caracteres, pode-se acessar cada caractere do String individualmente, usando índices. Por exemplo, Nome[5] retorna o quinto caractere do String "Nome" (os caracteres são indexados a partir de 1).

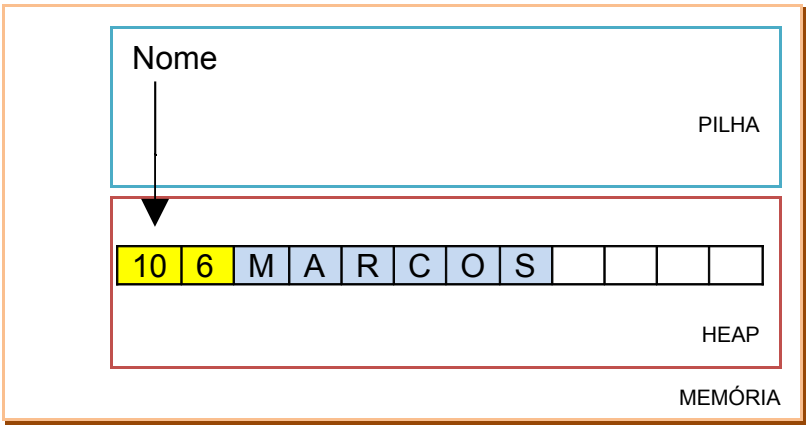
Tipo	Tamanho Máximo em caracter	Memória Requerida	Usado para
ShortString	255	2 até 256 bytes	Compatibilidade com versões anteriores
AnsiString = String = String Longa	~2^31	4 bytes até 2GB	Caracteres 8-bit (ANSI), DBCS ANSI, MBCS ANSI,etc
WideString	~2^30	4 bytes até 2GB	Caracteres Unicode, servidores multi-usários e aplicações multi-linguagens

AnsiString, também chamada string longa, representa uma alocação dinâmica de uma string que o tamanho máximo é limitado apenas pela quantidade de memória disponível.

Uma variável string é um ponteiro ocupando quatro bytes de memória. Quando a variável está vazia, ou seja, quando ela contém uma string de tamanho zero - o ponteiro é nulo e não usa a string de armazenamento adicional. Quando a variável não estiver vazia, ela apontará para um bloco de memória dinamicamente alocado que contém os valores da string. Os oito primeiros bytes da localização da string contém 4 bytes para indicar o tamanho da string e 4 bytes para indicar o tamanho atual da variável. Esta memória é alocado no heap, mas seu gerenciamento é totalmente automático e não exigindo qualquer código do usuário.

Ex:

```
Var
  Nome: String[10];
Begin
  Nome := 'MARCOS';
End;
```



Como as variáveis strings longas são ponteiros, duas ou mais delas podem referenciar o mesmo valor, sem consumir memória adicional. O compilador explora isto para conservar os recursos e executar tarefas de forma mais rápida. Se o contador de referência de uma string chegar a zero, a sua memória é

automaticamente liberada do heap.

Registros

Muitas vezes precisa-se criar algum tipo de variável que armazene informações de tipos de dados diferentes numa única estrutura, e o vetor não supre essas necessidades. É para isso que existe a estrutura de dados estática registro.

Um registro é composto por elementos de dados ou por outras estruturas que não sejam necessariamente do mesmo tipo. Cada elemento de um registro é conhecido como campo.

Para se declarar uma variável do tipo registro em Pascal, usa-se a sintaxe:

```
Var
  NomeDoRegistro: Record
                    Campo1: TipoDeDado;
                    Campo2: TipoDeDado;
                    CampoN: TipoDeDado;
                   End;
```

Ex:

```
Var
  Funcionario: Record
                  Nome    : String[60];
                  Endereco: String[50];
                  Salario : Real;
                  Idade   : Byte;
                 End;
```

O tamanho total de uma variável registro na memória será o somatório da quantidade de bytes que cada campo (que pode ser de elementos ou de estruturas de dados) ocupa na memória. A variável Funcionario, por exemplo, ocupará 119 bytes consecutivos na memória.

Para se atribuir valores nos campos referentes aos dados de um funcionário, devemos utilizar as seguintes notações:

```
Funcionario.Nome     := 'Manoel da Silva';
Funcionario.Endereco := 'Avenida Maracanã, 524';
Funcionario.Salario  := 1597.00;
Funcionario.Idade    := 29;

ou

with Funcionario do
Begin
  Nome     := 'Manoel da Silva';
  Endereco := 'Avenida Maracanã, 524';
  Salario  := 1597.00;
  Idade    := 29;
End;
```

Vetores – Arrays

O vetor é uma estrutura de dados **estática** e **ordenada**, com **elementos do mesmo tipo**. Não pode armazenar numa posição um valor booleano e noutra um caractere, e nem aumentar de tamanho no momento da execução de um programa. A definição de um vetor em Pascal é feita da seguinte maneira:

```
NomeDoVetor: array[ÍndiceMínimo..ÍndiceMáximo] of TipoDeDado;
```

Ex:

```
var
  Nomes   : array[1..100] of String;
  Salarios: array[1..50]  of Real;
       Meses   : array[1..12]  of integer
  Vetor   : array[-10..10] of array[1..5] of real;
```

Aqui, a variável "Nomes" é uma seqüência de 100 *Strings* e "Salarios" é uma seqüência de 50 valores do tipo *Real*.

Os elementos de um array são **indexados**: para acessar um valor de um array, digite o nome do array, seguido do **índice** entre colchetes. Por exemplo: Nomes[32]. O índice pode ser qualquer expressão que tenha como resultado um valor inteiro.

O exemplo a seguir declara um array "quadrados" com 100 inteiros e depois preenche esse array com os quadrados dos números de -50 a 250:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Quadrados: array[1..100] of Integer;
begin
 for I := -50 to 250 do
  Quadrados[I] := I*I;
end;
```

Tipos de Dados definidos pelo usuário

Tipos enumerados

A declaração de um tipo enumerado lista diretamente todos os valores que uma variável pode conter. Você define novos tipos usando a palavra-chave **type**, que deve ser colocada antes das definições dos tipos (tal como **var** e **const**). O **bloco type** deve ser posicionado antes dos blocos **var** e **const**. Veja alguns exemplos:

```
type
  TDiaUtil    = (Segunda, Terca, Quarta, Quinta, Sexta);
  TFabricante = (Chevrolet, Ford, Volkswagen);
  TSexo       = (Masculino, Feminino);
```

Depois de definir os novos tipos, você pode usá-los normalmente nas declarações das variáveis, como se fossem tipos predefinidos. Veja exemplos de declarações que usam os tipos definidos acima:

```
var
  DiaInicial, DiaFinal: TDiaUtil;
  Fab : TFabricante;
  Sexo: TSexo;
```

Nesse caso, por exemplo, a variável Fab só pode ter três valores diferentes. Qualquer valor diferente de "Chevrolet", "Ford", ou "Volkswagen" não é permitido.

Você atribui valores a uma variável de tipo enumerado normalmente. Veja alguns exemplos:

```
Fab         := Ford;
Sexo        := Masculino;
DiaDaSemana := Terca;
```

Note como não são usadas aspas para os valores. Isso acontece porque os valores de um tipo enumerado são armazenados como *valores inteiros*, na seqüência em que são definidos (começando a partir de zero). Para o tipo *TFabricante*, por exemplo, temos Ford = 0, Chevrolet = 1 e Volkswagen = 2. O uso de tipos enumerados em vez de inteiros pode tornar um programa muito mais legível.

Tipos de intervalo (*Subrange*)

Os tipos de intervalo são *seqüências de valores* dos tipos *Boolean*, *Char*, *Integer*, ou de tipos enumerados. Tipos de intervalo são úteis quando é necessário limitar uma variável a um intervalo contínuo de valores, como de 1 a 100, ou de "A" a "Z". Veja a seguir exemplos de definições de tipos de intervalo:

```
type
  TGraus      = 0..360;
  THoraValida = 0..23;
  TMaiuscula  = 'A'..'Z';
var
  Hora  : THoraValida;
  Angulo: TGraus;
```

No exemplo, a variável "hora" só pode estar entre 0 e 23 e a variável "angulo" deve estar entre 0 e 360.

Quando uma variável e definida com um tipo de intervalo, o Delphi verifica, durante a compilação, as alterações nesta variável e gera erros de compilação quando os limites do intervalo são violados. O seguinte exemplo gera um erro de compilação, na linha destacada.

```
procedure TFormTesteRange.Button1Click(Sender: TObject);
 type
   TGrau = 0..360;
 var
   Temp: TGrau;
 begin
   for Temp := 0 to 720 do {Erro de compilação!}
     begin
       ... {código omitido}
     end;
 end;
```

A mensagem de erro gerada é "*Constant expression violates subrange bounds*", indicando que os limites do intervalo foram violados (a variável *Temp* passaria do limite, 360, no loop **for**).

No caso anterior, o Delphi pôde detectar o erro, porque foram usados valores constantes para a variável. Quando são usadas expressões mais complexas (cálculos por exemplo), o Delphi só pode detectar o erro em *tempo de execução*. O exemplo a seguir passaria na compilação, mas a linha destacada geraria uma exceção do tipo **ERangeError** (*Temp* receberia o valor 400, maior que o máximo permitido pelo tipo *Tgrau*: 360).

```
procedure TFormTesteRange.Button2Click(Sender: TObject);
type
  TGrau = 0..360;
var
  Temp: TGrau;
begin
  Temp := 200;
  Temp := Temp*2; {Erro de execução!}
  ShowMessage(IntToStr(Temp));
end;
```

NOTA**: O comportamento padrão do Delphi é não verificar erros de intervalo em tempo de execução (como o erro do exemplo anterior). Para que o Delphi realmente verifique se as variáveis estão dentro dos limites, escolha o comando **Project | Options**, mude para a página **Compiler** e ative a opção **Range Checking

Comandos Básicos da Linguagem Pascal

Usando o comando de atribuição

O comando de atribuição é um dos mais usados em muitas linguagem de programação. O comando de atribuição é usado para a alterar valores de variáveis e propriedades no Delphi. Já mostramos vários exemplos que usam atribuição. Aqui está mais um:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Color    := clRed;
  Edit1.Text     := 'Novo texto';
  Memo1.ReadOnly := True;
end;
```

O comando de atribuição, como se vê é formado por duas partes, a parte esquerda é um nome de uma variável ou de uma propriedade; a parte direita é o novo valor que será *atribuído* à variável ou propriedade. Esse valor pode ser outra variável ou propriedade, uma constante, ou outra expressão. Os tipos dos dois lados devem ser os mesmos, ou pelo menos compatíveis; caso contrário, o Delphi gera erros na compilação.

Entrada

Um comando de entrada serve para que o programa solicite dados no momento em que o mesmo está sendo executado. Esses dados fornecidos serão armazenados em variáveis na memória. Em geral a unidade de entrada é o teclado, podendo também ser uma memória auxiliar como o winchester.

Considerando a unidade de entrada padrão, o teclado, o comando seria:

```
READ (<identificador-1>,...<identificador-n>)
```

ou

```
READLN (<identificador-1>,...,<identificador-n>)
```

Com *READ* o cursor permanece na mesma linha após a execução do comando; com o *READLN* o cursor muda para a próxima linha.

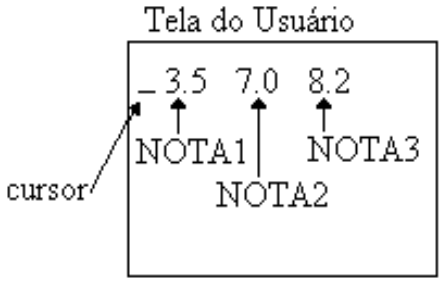
Observação: No Turbo Pascal, o comando READ só deve ser utilizado para a leitura de arquivos. Portanto, para a leitura de variáveis, devemos sempre utilizar o comando READLN.

EXEMPLOS:

1) Se o programa deve solicitar as três notas de um aluno, teríamos:

```
readln (NOTA1,NOTA2,NOTA3); ...
```

No momento da execução do comando acima, o programa mostra a tela do usuário e o cursor aparece esperando a digitação dos três valores que devem ser separada por, pelo menos, um espaço em branco.



3.5 será armazenado na variável NOTA1, o 7.0 em NOTA2 e o 8.2 em NOTA3.

2) Se o programa deve solicitar o nome e o salário de um funcionário teríamos:

```
readln (NOME);
readln (SALÁRIO);
...
```

Saída

Um comando de saída serve para que o programa mostre ao usuário os resultados desejados. A unidade de saída padrão é o monitor de vídeo, podendo ser também a impressora ou uma memória auxiliar como o disco.

Considerando a unidade de saída padrão, o monitor de vídeo, o comando seria:

```
WRITE (<expressão-1>,...,<expressão-n>)
```

ou

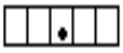
```
WRITELN (<expressão-1>,...,<expressão-n>)
```

Com *WRITE* o cursor permanece na mesma linha após a execução do comando; com *WRITELN* o cursor muda para a próxima linha.

EXEMPLO:
```
A:=1; B:=2;
writeln ('Soma de ',A,' e ',B,' = ',A+B);
```

No caso de variáveis do tipo REAL os valores são mostrados na notação exponencial, num campo de 16 posições, a menos que uma formatação seja especificada.

EXEMPLO:
```
MEDIA := (8.0 + 2.0) / 2
writeln (MEDIA); {saída ⟶ 5.0000000000E+00}
writeln (MEDIA:5:2); {saída ⟶ 5.00}
```

Na formatação, se a variável for *real* especificamos o total de posições ocupadas e a quantidade de casas decimais  . Se *inteira*, só o total de posições.

Se desejarmos que a saída seja através da impressora e não do monitor de vídeo, devemos especificar no começo da lista da saída o parâmetro *LST* e a *unidade PRINTER* com o *USES*.

EXEMPLO:
```
Uses PRINTER;
:::
writeln (LST, 'Media = ', MEDIA:5:2);
```

No instante da solicitação de dados, podemos usar junto com o *READ* ou *READLN* um comando de saída com a finalidade de emitir mensagens que orientem o usuário na digitação dos dados.

EXEMPLOS:
```
writeln('Digite o Nome:');
readln(NOME);

writeln('Digite as 3 Notas:');
readln(NOTA1,NOTA2,NOTA3);

writeln('Digite o salário do funcionário:');
readln(SALARIO);
```

Controle de fluxo

Os programas mostrados até aqui não fazem nada além de definir novos tipos, declarar variáveis e alterá-las. Para que programas realmente "façam alguma coisa", eles devem ser capazes de tomar decisões de acordo com as entradas do usuário, ou repetir comandos, até que uma certa condição seja satisfeita. São essas operações que constituem o **controle de fluxo** de um programa.

Usando blocos

Um bloco é um conjunto de comandos delimitado pelas palavras chave **begin** e **end**. Já usamos blocos várias vezes nos exemplos mostrados até aqui . No arquivo de projeto, por exemplo, um bloco contém o código principal do projeto (onde, entre outras coisas, são criados os formulários do aplicativo). Todo programa em Object Pascal deve ter pelo menos um bloco: o bloco principal.

Quando você precisa executar ou não um *conjunto* de comandos dependendo de uma condição, você deve usar blocos para delimitar esses conjuntos. Outro uso importante de blocos é para a repetição de um conjunto de comandos várias vezes, como nos *loops* **for**, **while** e **repeat**, que veremos a seguir.

Blocos podem ser *aninhados*, isto é pode haver blocos dentro de blocos. Isso é útil em estruturas complexas, como **if**s dentro de **if**s, ou loops dentro de loops.

A estruturação em blocos da linguagem Object Pascal é uma de suas características mais elegantes e poderosas.

If-then-else

O comando mais usado para fazer decisões simples é o comando **if**. O comando **if** verifica uma condição e executa um comando ou bloco de comandos somente se a condição for verdadeira. O comando **if** é sempre usado com a palavra-chave **then**. Há várias maneiras de usar o comando **if**. Veja a mais simples a seguir:

```
if condição then
   comando;
```

Aqui a *condição* é qualquer expressão que tenha valor booleano (*True* ou *False*). Condições são geralmente comparações, como **a > b**, **x = 1**, **total <= 1000**, etc. Pode-se também usar o valor de uma *propriedade* como condição. O exemplo abaixo, ativa um botão (chamado **"**Button2") somente se ele estiver desativado (com *Enabled=False*). Caso contrário, nada acontece.

```
if Button2.Enabled = False then
   Button2.Enabled = True;
```

Há também várias funções predefinidas que retornam valores booleanos e que também podem ser usadas como condições.

Usando *else*

Um comando **if** pode também apresentar uma segunda parte, delimitada pela palavra-chave **else**. O comando da segunda parte (depois de **else**) é executado quando a condição é falsa. O exemplo a seguir mostra uma mensagem diferente dependendo do valor da variável "x":

```
if x > limite then
  ShowMessage('Limite ultrapassado!');
else
  ShowMessage('Sem problemas.');
```

Usando blocos com o comando *if*

A versão simples do comando **if** usada nos exemplos acima tem uso muito limitado. Na maioria das vezes, é necessário executar *mais de um comando* se uma certa condição for verdadeira. Para isso você deve usar *blocos* de comandos. Veja um exemplo completo:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Dividendo, Divisor, Resultado: Real;
begin
  Dividendo := StrToFloat(Edit1.Text);
  Divisor := StrToFloat(Edit2.Text);
  if Divisor = 0 then
  begin
    Color := clRed;
    ShowMessage('Divisor inválido');
  end
  else
  begin
    Color := clSilver;
    Resultado := Dividendo/Divisor;
    ShowMessage('Resultado = ' + FloatToStr(Resultado));
  end;
end;
```

Note os dois blocos de comandos: um para o **if-then** e outro para o **else**. O programa calcula a divisão de dois valores do tipo *Real* (lidos a partir de dois componente *Edit*). Se o divisor for zero, o primeiro bloco do **if** é executado: o formulário muda para vermelho e uma mensagem é exibida. Se o divisor não for zero, o segundo bloco (depois do **else**) é executado, mudando a cor do formulário para cinza.

As funções **StrToFloat** e **FloatToStr** convertem de Strings para números reais e de números reais para Strings, respectivamente.

Note como o **end** depois do primeiro bloco não é seguido de ponto-e-vírgula (;). Essa é uma regra obrigatória da linguagem Object Pascal: um **end** antes de um **else** nunca deve ser seguido por ";". Somente o último **end** do comando **if** deve terminar com ponto-e-vírgula.

Aninhando comandos *if*

Muitas vezes, um programa deve decidir entre mais de duas opções. Para isso, pode-se usar comandos **if** aninhados (um **if** dentro de outro). Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Nota: Real;
begin
  Nota := StrToFloat(Edit1.Text);
  if nota < 5.0 then
    ShowMessage('Reprovado por média')
  else
    if (Nota >= 5.0) and (Nota < 7.0) then
      ShowMessage('Aprovado na final')
    else
      if Nota > 8.5 then
        ShowMessage('Aprovado com excelência')
      else
        ShowMessage('Aprovado simplesmente')
end;
```

O exemplo lê uma nota digitada pelo usuário e mostra uma mensagem que varia de acordo com o valor da nota. Acompanhe o fluxo do programa: se a nota for menor que 5, a primeira mensagem é exibida; caso contrário (qualquer valor maior ou igual a 5), o programa checa a segunda condição: se a nota estiver entre 5 e 7, a segunda mensagem é exibida. Se não, o programa verifica se a nota é maior que 8.5 e, se for, mostra a terceira mensagem. A mensagem depois do último **else** só é exibida se *nenhuma* das condições anteriores for verdadeira (isso acontece se a nota estiver entre 7 e 8.5).

***NOTA**: no programa acima foi usada a palavra chave **and** para criar uma condição mais complexa. Há mais duas palavras-chaves semelhantes: **or** e **not**. Pode-se usar qualquer combinação de **not**, **and** e **or** para criar novas condições. Lembre-se, entretanto de usar parênteses para agrupar as condições. Os parênteses usados no programa anterior, por exemplo, são todos obrigatórios.*

A estrutura *case*

Quando é necessário decidir entre muitas opções em um programa, a estrutura **case** da linguagem Object Pascal é em geral mais clara e mais rápida que uma seqüência de **if**s aninhados. Uma estrutura **case** só pode ser usada com valores **ordinais** (inteiros, caracteres, tipos de intervalo e tipos enumerados). Na estrutura **case** uma variável é comparada com vários valores (ou grupos de valores) e o comando (ou bloco de comandos) correspondente é executado.

Veja a seguir um exemplo simples, que mostra mensagens diferentes, de acordo com o número entrado em um *Edit*.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Numero: Integer;
begin
  Numero := StrToInt(Edit1.Text);
  case Numero of
   1, 2, 3, 5, 7 :
     begin
       ShowMessage('Número primo menor que 10');
       Color := clBlue;
```

```
      end;
    4, 6, 8   : ShowMessage('Número par menor que dez');
    9         : ShowMessage('Número ímpar menor que dez');
    10..1000  : ShowMessage('Número entre 10 e 1000');
  else
   ShowMessage('Outro número');
  end; // final do case
end; // final da procedure
```

Uma estrutura case deve ser terminada com **end** e pode ter uma parte opcional **else**. Os comandos depois do **else** só são executados se a variável não for igual a nenhum dos valores especificados (se não houver a parte **else**, a estrutura **case** é pulada inteiramente). No exemplo acima, a mensagem "Outro número" só é mostrada se o valor da variável "numero" for maior que 1000 ou menor que 1.

Usando loops

Os loops ("laços") são estruturas usadas para repetir várias vezes sequências de comandos. Há três tipos de loops em Object Pascal. O loop **for** é usado para realizar um número fixo de repetições. Os loops **while** e **repeat** repetem um bloco de comandos até que uma condição se torne falsa ou verdadeira, respectivamente.

O loop *for*

O loop **for** é o mais rápido e mais compacto dos três tipos de loops. Esse loop usa um *contador,* uma variável inteira que é aumentada (*incrementada*) automaticamente cada vez que o loop é executado. O número de repetições do loop **for** é fixo. Ele depende somente do valor inicial e do valor final do contador, que são definidos no início do loop. Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
  Numero: Integer;
begin
  Numero := StrToInt(Edit1.Text);
  for I := 1 to 50 do
    ListBox1.Items.Add(IntToStr(Numero*I));
end;
```

Este programa mostra os primeiros 50 múltiplos do número digitado em um *Edit*. Os múltiplos são adicionados a um *ListBox*. O loop **for** usado aqui contém um único comando. Por isso não é necessário usar **begin** e **end**.

O loop *while*

O loop **while** é um loop mais versátil que o **for**, embora um pouco mais complexo. O loop **while** repete um comando, ou um bloco de comandos, até que a condição especificada se torne falsa. O número de repetições não é preestabelecido como no loop **for**.

No exemplo a seguir, o valor digitado em um *Edit* é lido, convertido e colocado em uma variável "x". Em seguida, é subtraído 10 do valor de "x" e adicionado o resultado (convertido em um string) a um *ListBox*, repetindo o processo até que "x" seja menor ou igual a 0.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  x: Integer;
begin
  x := StrToInt(Edit1.Text); {Ler valor inicial para x}
  ListBox1.Items.Clear; {Limpar a lista de valores}
  while x > 0 do
   begin
     x := x - 10;
     ListBox1.Items.Add(IntToStr(x));
   end;
end;
```

No loop **while**, a condição é testada no início de cada repetição. Quando a condição é falsa, o programa sai do loop e executa o comando imediatamente depois. Se a condição testada for verdadeira, o loop é

executado novamente.

Note que é possível que os comandos de um loop **while** não sejam executados **nenhuma** vez. Isso acontece quando a condição é *falsa* já na entrada do loop. No programa acima, por exemplo, um valor inicial negativo para a variável "x" faria com que o **while** fosse pulado inteiramente na primeira vez.

Outra característica importante do loop **while**, é que a condição usada no loop deve se tornar falsa em algum momento. Caso contrário, o loop é executado para sempre. O programa entra em um "loop infinito". Veja uma exemplo de um loop infinito com **while**:

```
x:= 1;
while x > 0 do
x := x + 1;
```

Aqui, a condição **x > 0** nunca se tornará falsa (x será sempre maior que zero), e o programa ficará preso para sempre dentro do while.

Esse é um erro cometido com freqüência, principalmente em loops **while** com condições mais complexas. Tenha cuidado, portanto, de alterar pelo menos uma das variáveis envolvidas na condição, *dentro* do **while**.

O loop repeat

O loop **repeat** é uma versão "invertida" do loop **while**. Nele, a condição é verificada somente no **final** do loop. Isso faz com que o loop seja sempre executado, **pelo menos uma vez**. A condição do loop **repeat** aparece no final, ao lado da palavra-chave **until** ("até"). Além disso, devido à estrutura simples do loop **repeat**, os comandos do loop não precisam ser delimitados por **begin** e **end**. Veja um exemplo:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Entrada: String;
  Valor: Integer;
begin
 repeat
   Entrada:= (InputBox('', 'Entre com um número de 1 a
   100',''));
    Valor:= StrToInt(entrada);
 until (Valor >= 1) and (Valor <= 100);
   ShowMessage('Entrada aceita.');
end;
```

O código mostra uma *InputBox* várias vezes, até que a entrada seja válida (entre 1 e 100). Note que a *InputBox* é exibida pelo menos uma vez, pois é usado um loop **repeat**.

Se **while** tivesse sido usado em vez de **repeat**, o código seria mais longo e menos elegante:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Valor: Integer;
  Entrada: string;
begin
  Entrada:=InputBox('','Entre com um número de 1 a 100','');
  Valor := StrToInt(entrada);
  while (Valor < 1) or (Valor > 100) do
   begin
     Entrada := InputBox('','Entre com um número de 1 a
     100','');
```

```
      Valor := StrToInt(Entrada);
    end;
   ShowMessage('Entrada aceita.');
end;
```

Este código tem exatamente o mesmo efeito que o anterior. Mas note que a condição usada no loop **while** é o oposto da condição do loop **repeat**. Além disso, há uma repetição das linhas de código que lêem a entrada. Isso é necessário porque a variável "valor" é testada no início do loop e precisa ser inicializada primeiro.

Na verdade, qualquer programa que use **repeat** pode ser convertido em um programa usando **while**, fazendo-se mudanças semelhantes às mostradas aqui. O loop **repeat** é apenas uma conveniência, que pode algumas vezes tornar o código mais simples.

O comando *break*

O comando **break** é usado para sair imediatamente de um loop (**while**, **for** ou **repeat**). Veja um exemplo simples:

```
var
  A,B: array[1..1000] of Integer;
  i: Integer;
...
for i:= 1 to 1000 do
 if A[i] <> B[i] then break;
 if i = 1000 then
   ShowMessage('Todos os elementos são iguais')
 else
      ShowMessage('O elemento com índice ' + IntToStr(i) + ' é
diferente');
```

...

O exemplo compara os elementos de dois arrays **A** e **B**, de 1000 elementos cada um (o cálculo para os valores dos arrays não é mostrado). Quando é encontrada a primeira diferença nos arrays, o comando **break** interrompe o loop **for** imediatamente e os outros elementos dos arrays não são checados. Se todos os elementos forem iguais, o valor da variável **i** no final do loop será 1000 e a primeira mensagem é mostrada. Se o loop for quebrado antes de terminar, com o comando **break**, o valor de **i** será menor que 1000 e a segunda mensagem será mostrada.

Procedures e functions

A linguagem pascal permite que possamos declarar trechos de código fora do programa principal e associados a um identificador que podem ser executadas sempre que invocados. Chamaremos estes trechos de código por *módulos* ou *rotinas*.

Uma das técnicas mais utilizadas e tida como vantajosa na confecção de programas grandes é a modularização. Consiste em dividir o programa em diversos módulos ou subprogramas, de certa forma independentes uns dos outros. Existe um módulo que é o principal, a partir do qual são chamados os outros módulos, esse módulo recebe o nome de programa principal, enquanto que os outros são chamados de subprogramas.

Os principais motivos para se usar rotinas são:

1. Evitar codificação: trocar certos trechos de programas que se repetem por chamadas de apenas uma rotina que será codificada apenas uma vez.
2. Modularizar o programa, dividindo-o em módulos (rotinas) logicamente coerentes, cada uma com função bem definida. Isto facilita a organização do programa, bem como o entendimento dele.

No sistema Turbo Pascal, existem dois tipos de subprogramas, a saber:

- Procedures (procedimentos)
- Functions (funções)

Procedures

A procedure é como se fosse um programa. Ela tem a estrutura praticamente igual a de um programa, como veremos mais adiante. A procedure deve ser ativada(chamada) pelo programa principal ou por uma outra procedure, ou até por ela mesma.

Uma procedure tem praticamente a mesma estrutura de um programa, ou seja, ela contém um cabeçalho, área de declarações e o corpo da procedure. Na área de declarações, podemos ter as seguintes subáreas:

```
Const - Type - Var - Procedures - Functions
```

Devemos salientar que tudo que for declarado dentro das subáreas só será reconhecido dentro da procedure. Mais para frente, voltaremos a falar sobre isso.

Sintaxe das procedures

A sintaxe para uma procedure é a seguinte:

```
procedure NomeDaProcedure(param1, param2, ...:Tipo1; param1,...:tipo2;
...);
type
  {definições de tipos}
var
  {declarações de variáveis }
const
  {definições de constantes}
begin
  { corpo }
end;
```

Ex:

```
Program Exemplo_1; (* cabeçalho do programa *)

Procedure linha; (* cabeçalho da procedure linha *)
Begin (* corpo da procedure linha *)
  for i:=1 to 80 do
    write('-');
  End;
End;

Var
  i : integer; (* subárea Var da procedure linha *)
Begin (* corpo do programa principal *)
  linha; (* ativaçäo da procedure linha *)
  writeln('teste');
  linha; (* ativaçäo da procedure linha,novamente *)
End.
```

Funções

As funções são muito parecidas com as procedures. A principal diferença é que o identificador de uma função assume o valor de retorno da função. Uma função deve sempre retornar um valor e em Turbo Pascal, este valor é retornado no nome da função.

Para functions, a sintaxe é praticamente idêntica. A única diferença é que o tipo do valor retornado deve ser especificado no final da primeira linha.

```
function nome da function (param1, param2, ...:tipo1; param1,...:tipo2; ...): tipo de retorno;
type
  {definições de tipos}
var
  {declarações de variáveis }
const
  {definições de constantes}
begin
  {corpo}
end;
```

A primeira linha, chamada de **cabeçalho**, deve sempre terminar com ";". Os parâmetros, se existirem, devem ter seus tipos especificados e devem ser separados por vírgulas. Parâmetros do mesmo tipo podem ser agrupados; esses grupos devem ser separados por ";".

Depois do cabeçalho, vêm os blocos de declaração **type**, **var** e **const** (não necessariamente nessa ordem). Todos estes são *opcionais*, pois a procedure ou functions pode não precisar de nenhuma variável além das especificadas nos seus parâmetros. Em seguida, vem o bloco principal (ou **corpo**) da procedure, delimitado por **begin** e **end**. É neste bloco que fica o código executável da procedure.

Definindo o valor de retorno de uma function

O valor retornado por uma function pode ser definido de duas maneiras: atribuindo um valor para o nome da function, ou alterando a variável especial **Result**. As duas funções a seguir são exatamente equivalentes:

```
function Dobro1(Numero: Integer): Integer;
begin
  Dobro := Numero*2;
end;

function Dobro2(Numero: Integer): Integer;
begin
  Result := Numero*2;
end;
```

Entendendo parâmetros

Os parâmetros são variáveis passadas para as procedures e functions (rotinas). Os parâmetros de uma rotina são opcionais. São comuns rotinas (principalmente procedures) sem nenhum parâmetro.
Quando são usados parâmetros, estes podem ser passados **por valor** ou **por referência**. A forma como os parâmetros são passados é definida no cabeçalho da rotina.

Quando uma variável é passada **por valor**, a rotina recebe apenas um *cópia* da variável. A variável passada se comporta como uma variável local. Alterações realizadas na variável não têm efeito depois da rotina terminar (ou retornar). A passagem por valor é o tipo padrão de passagem de parâmetros na linguagem Object Pascal.

Quando uma variável é passada **por referência**, a rotina recebe uma *referência* à variável passada ou, em outras palavras, *a própria variável*. Com esse tipo de passagem de parâmetros, a rotina pode alterar *diretamente* a variável passada. Para especificar a passagem por referência, use a palavra-chave **var** antes do nome do parâmetro, no cabeçalho da rotina.Veja um exemplo que usa os dois tipos de passagem de parâmetros:

```
procedure Descontar(Taxa: Real; var Preco: Currency);
begin
  Preco := Preco - Preco*(Taxa/100);
end;
```

Esta procedure recebe dois parâmetros: *Taxa* e *Preco*. O parâmetro *Taxa* é passado *por valor*. O valor de *Taxa* pode ser alterado dentro da procedure, mas as alterações não terão efeito depois de a procedure retornar.

O parâmetro *Preco* é passado por referência (usando-se a palavra **var** antes do parâmetro). *Preco* é alterado dentro da procedure e essa alteração é permanente.

Chamando procedures e functions

Depois de definir uma procedure ou function, você pode chamá-la diretamente no seu código, em qualquer lugar da Unit em que foi definida.

Para chamar uma procedure ou function, basta especificar o seu nome e valores para seus parâmetros (se houver). O seguinte trecho de código mostra como chamar a procedure *Descontar* (definida acima) para realizar um desconto de 10% sobre um preço *p*. (O valor de *p* no final será 450.0).

```
...
p := 500.0;
Descontar(10, p);
...
```

Onde criar procedures e functions

Há regras para o posicionamento das procedures no código: as procedures devem ser colocadas somente depois da parte *implementation* da Unit (veja a seguir), depois das declarações das variáveis globais.

O exemplo a seguir usa a procedure *Descontar* e uma variação da function *Dobrar*, dos exemplos anteriores. Os valores de cem preços, armazenados no array *Precos*, são alterados no evento *OnClick* do componente *Button1*. (O programa, apesar de completo, foi mantido deliberadamente simples, para ilustrar melhor os conceitos envolvidos).

```
{Código inicial da Unit}

...

implementation

var
  Precos: array[1..100] of Currency;

procedure Descontar(Taxa: Double; var Preco: Currency);
begin
  Valor := Preco - Preco*(taxa/100);
end;

function Dobrar(Preco: Currency): Currency
begin
  Result := Preco*2;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  // Descontar em 25% os primeiros 20 produtos...
  for i := 1 to 20 do
    Descontar(25, Precos[i]);
  //...descontar em 50% os próximos 30...
  for i := 21 to 50 do
    Descontar(50, Precos[i]);
  //...e dobrar o preço dos 50 produtos restantes
  for i := 51 to 100 do
    Precos[i]:= Dobrar(Precos[i]);
end;

end.
```

Entendendo o código das Units

As Units contêm praticamente todo o código de um aplicativo no Delphi. As Units geralmente são associadas a formulários, mas podem ser criadas de forma totalmente independente. Nesta seção veremos detalhes sobre cada parte de uma Unit.

A estrutura básica de uma Unit

Todas as Units têm a mesma estrutura básica:

```
program

unit <nome da unit>

interface

uses <lista de Units>

type

var

procedure

function

implementation

uses <lista de Units>

{código para os procedures e functions}

initialization {opcional}

{código para inicialização}

finalization {opcional}

{codigo para finalização}

end.
```

A primeira linha da Unit identifica o **nome** da Unit (que deve ser um identificador válido na linguagem Object Pascal). Veja a seguir descrições sobre cada uma das partes.

Parte Interface

A parte **interface** começa com a palavra **interface** e termina imediatamente antes da palavra-chave **implementation**. Esta parte contém a definição de todos os tipos, constantes, variáveis, procedures e funnctions que devem ser "visíveis" (ou acessíveis) para outras Units que se referenciem a esta. Somente o que estiver definido na parte **interface** pode ser acessado por outras Units.

A primeira cláusula **uses** fica dentro da parte **interface**. As Units listadas nessa cláusula são geralmente adicionadas pelo próprio Delphi, para fazer referência a Units do sistema (predefinidas). É rara a necessidade de adicionar manualmente nomes de Units a essa cláusula **uses**. Para se referenciar a outras

Units, altera-se a claúsula **uses** na parte **implementation**.

Parte Implementation

A parte **implementation** contém todo o código das procedures e functions da Unit. Esta parte pode conter também declarações de variáveis e constantes, mas estas só serão "visíveis" pelo código desta Unit. Nenhuma outra Unit pode ter acesso às variáveis e constantes declaradas aqui.

A parte **implementation** contém a segunda cláusula **uses**. As Units listadas nessa cláusula são geralmente adicionadas pelo programador, manualmente, ou usando o comando **File | Use unit**.

Parte Initialization

A parte **initialization** é opcional. Você pode usar a parte **initialization** para declarar e inicializar variáveis, por exemplo. O código nesta parte é executado antes de qualquer outro código na Unit. Se um aplicativo tiver várias Units, a parte **initialization** (se houver) de *cada* Unit é executada antes de qualquer outro código na Unit.

Parte finalization

A parte **finalization** também é opcional. O código nessa parte é executado logo antes do término do aplicativo. Essa parte é geralmente usada para realizar "operações de limpeza", como recuperar memória e outros recursos, ao final da execução do aplicativo.

Trabalhando com Exceções

As exceções são um mecanismo poderoso para lidar com erros nos seus programas. O uso de exceções permite uma separação entre o código normal de um programa e o código usado para lidar com erros que podem surgir.

Quando ocorre um erro em um programa, o Delphi *levanta* uma exceção. Se uma exceção que foi levantada não for *tratada*, o programa é interrompido e é mostrada uma caixa de diálogo com uma descrição da exceção. Uma exceção não-tratada pode causar danos aos seus dados, deixar o programa em uma situação instável, ou até levar o programa a "travar".

Para evitar que isso aconteça, você deve tratar as exceções, **protegendo** blocos de código que contenham comandos que possam causar erros. Para proteger um bloco de código cerque-o com as palavras-chave **try** e **end**. Dentro de um **bloco protegido**, pode-se usar as comandos **except** ou **finally**, para tratar as exceções. Há dois de tipos blocos protegidos:

```
try
  {comandos que podem levantar exceções }
except
  {comandos executados quando uma exceção é levantada}
end;

try
  {comandos que podem levantar exceções }
finally
  {comandos sempre executados, havendo exceções ou não}
end;
```

O primeiro tipo de bloco protegido é chamado de bloco **try-except**. O segundo, de bloco **try-finally**.

No bloco **try-except**, os "comandos protegidos" ficam entre as palavras-chave **try** e **except**. Quando uma exceção ocorre em um desses comandos, o programa pula imediatamente para o primeiro comando depois de **except**.

No caso do bloco **try-finally**, os comandos depois de **finally** são executados sempre (mesmo quando exceções não são levantadas). Pode-se usar esses comandos para realizar operações de "limpeza", como destruir componentes que não são mais necessários, por exemplo.

Há vários tipos de exceções. Veja a seguir um exemplo que trata a exceção **EDivByZero**, gerada quando é feita uma divisão por zero (o divisor aqui é o valor contido no componente *Edit1*). Assume-se que os valores dos arrays *resultado* e *valores* foram declarados e inicializados antes.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Divisor: Real;
begin
  Divisor:= StrToFloat(Edit1.Text);
  for i:= 1 to 50 do
  try
    Resultado[i] := Valores[i]/Divisor;
  except
    on EDivByZero do Resultado[i] := 0;
  end;
end;
```

Note o uso da palavra-chave **on** dentro do bloco **except**. **On** é usada para especificar o *tipo* de exceção a ser tratado. Um bloco **except** pode lidar com vários tipos de exceção, definindo-se um bloco.

on [tipo de exceção] **do** [comandos]para cada tipo.

Na tabela abaixo, listamos alguns tipos comuns de exceções:

Tipo de exceção	**Quando ocorre**
EDivByZero	Quando se tenta dividir um inteiro por zero.
EZeroDivide	Quando se tenta dividir um número real (de ponto flutuante) por zero.
EConvertError	Quando é feita uma conversão ilegal de tipos.
EInOutError	Quando há um erro de entrada ou saída.
EDatabaseError	Quando ocorre um erro geral de banco de dados.
EDBEngineError	Quando há um erro no BDE (*Borland Database Engine*).

Rotinas úteis

Nas seções a seguir, apresentaremos algumas rotinas (functions e procedures) úteis na programação em Object Pascal.

Rotinas para manipulação de strings

A manipulação de strings é uma tarefa muito comum em programação. A seguir, listamos as principais rotinas de manipulação de strings oferecidas pelo Delphi. (As rotinas cujos nomes começam com "Ansi" são capazes de lidar com caracteres acentuados e, portanto, são muito úteis para strings em português).

Função	**Descrição**
Copy**(S:**string**; Indice,Comp: Integer): **string	*A função **Copy** retorna um string contendo Comp caracteres, começando com S[Indice].*
***Delete**(**var** S: **string**; Indice, Comp: Integer)*	*A procedure **Delete** remove o substring com Comp caracteres do string S, começando com S[Index]. O string resultante é retornado na variável S.*
***Length**(S: **string**): Integer*	*A função **Length** retorna o número de caracteres no string S.*
Trim**(S: **string**): **string	*A função **Trim** remove espaços à esquerda e à direita do string S.*

Funções de conversão de tipo

A linguagem Object Pascal é especialmente exigente com relação aos tipos usados em expressões, e nas chamadas de procedures e functions. Se o tipo usado não for compatível com o tipo esperado, é gerado um erro de compilação. **Funções de conversão de tipo** devem ser usadas para converter valores para os tipos adequados.

As principais funções de conversão de tipo são listadas na tabela a seguir. Se uma conversão for ilegal, essas funções levantam a exceção **EconvertError**.

Função	**Tipo de origem**	**Tipo de destino**
StrToCurr	String	Currency
StrToDate	String	TDate
StrToDateTime	String	TDateTime
StrToFloat	String	Real
StrToInt	String	Integer
StrToTime	String	TTime
IntToStr	Integer	String
CurrToStr	Currency	String
DateToStr	TDate	String
TimeToStr	TTime	String

Bibliografia

Boratti, Isaias Camilo. **Programação Orientada a Objetos usando Delphi**, 3 º Edição, Visual Books, 2004.
Cantu, Marco. **Dominando o Delphi 6: a Bíblia**, São Paulo, Editora Makron Books, 2003.
SENAC. DN. **Estruturas de Dados**./Ricardo de Souza Oliveira; Gilda Aché Taveira; Joana Botini. Rio de Janeiro: Ed. Senac Nacional, 1999.
Manzano, José Augusto N. G., **Algoritmos: lógica para desenvolvimento de programação**, 2 º Edição, Érica, 1996.
Barnes, J. David; Kolling Michael. **Programação Orientada a Objetos com Java**, Pearson, 2004
Facunte, Emerson. **Delphi 7 Internet e Banco de Dados**, Brasport,2003.
Melo, Ana Cristina. **Desenvolvendo Aplicações com UML**, 1 º Edição, Brasport, 2002.
Todd, Bill; Kellen, Vince. **Delphi 2 – Guia do Desenvolvedor**, Makron Books, 1997.