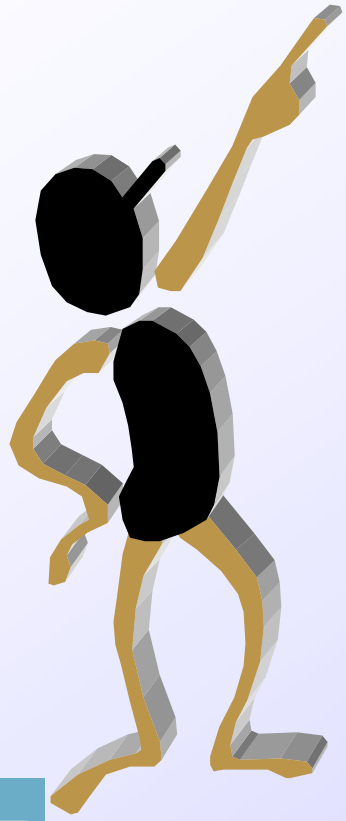
Programação em PASCAL

Agradecimentos:
Francisco Morgado, Dr.

Ernesto Afonso, Eng.º
Manuel A. E. Baptista, Eng.º

Programação em PASCAL

Turbo Pascal



3. Origens da linguagem Pascal

Em 1970, o professor suíço *Niklaus Wirth* criou uma linguagem de programação inspirada no *ALGOL-60* à qual atribuiu o nome de *Pascal*, por homenagem ao matemático *Blaise Pascal*. Os principais objectivos que estiveram subjacentes à sua criação foram:

- ☑ **Elaborar uma linguagem estruturada, de fácil leitura e manipulação e bastante disciplinada, de forma a promover bons hábitos de programação sendo, por isso, adequada ao ensino da programação;**
- ☑ **Reunir e unificar os conceitos comuns às outras linguagens existentes definindo uma linguagem normalizada de forma a facilitar a aprendizagem de outras linguagens partindo do *Pascal*;**

Blaise Pascal foi um matemático francês que em *1645* construiu e patenteou uma máquina baseada em engrenagens que realizava as quatro operações matemáticas elementares e cuja aplicação se destinava à contabilidade. Actualmente é reconhecida como a primeira calculadora e antecessora dos computadores actuais.

3.1. Ambiente de programação

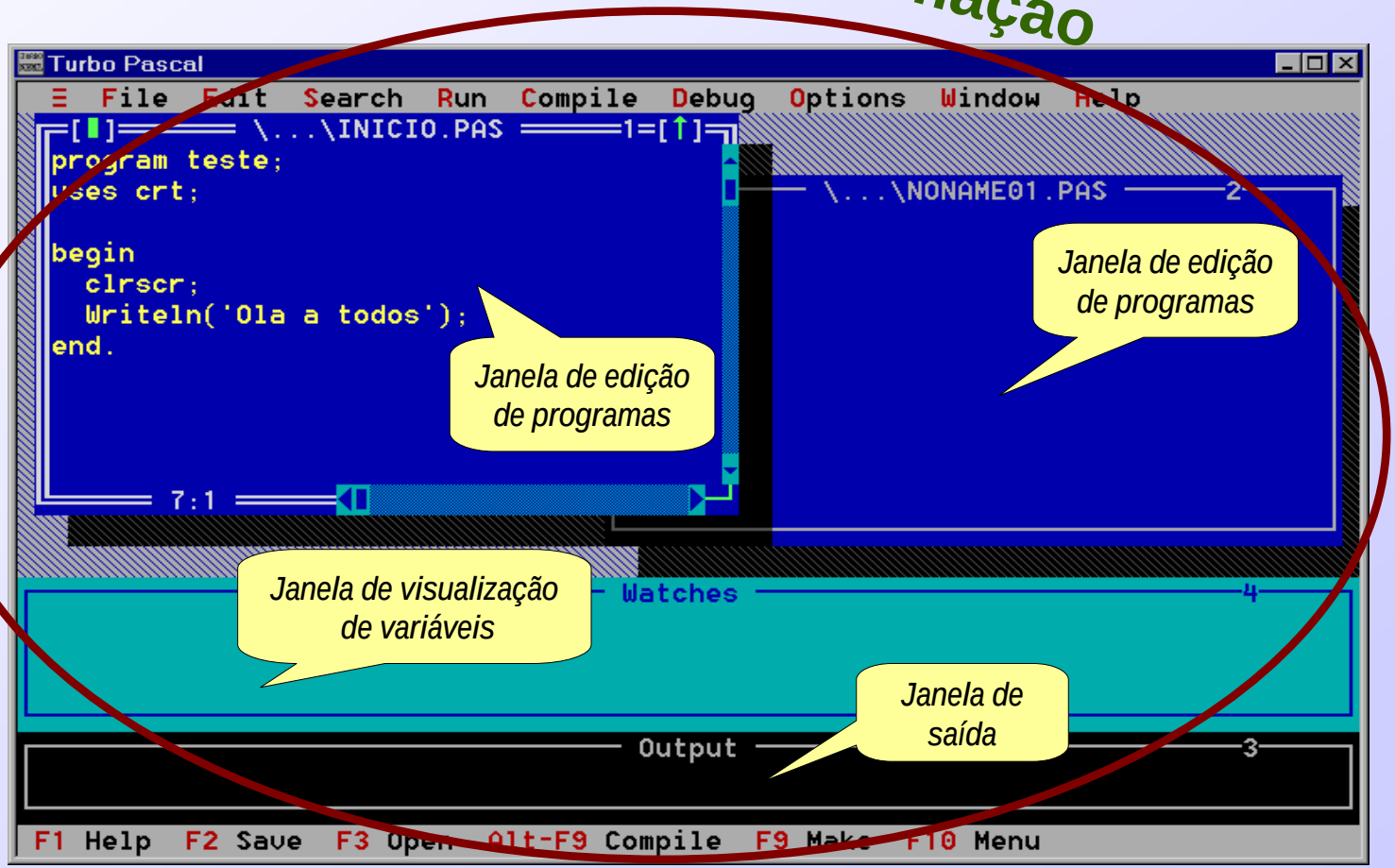
3.1.1. Ambiente do Turbo Pascal

Ambiente de programação

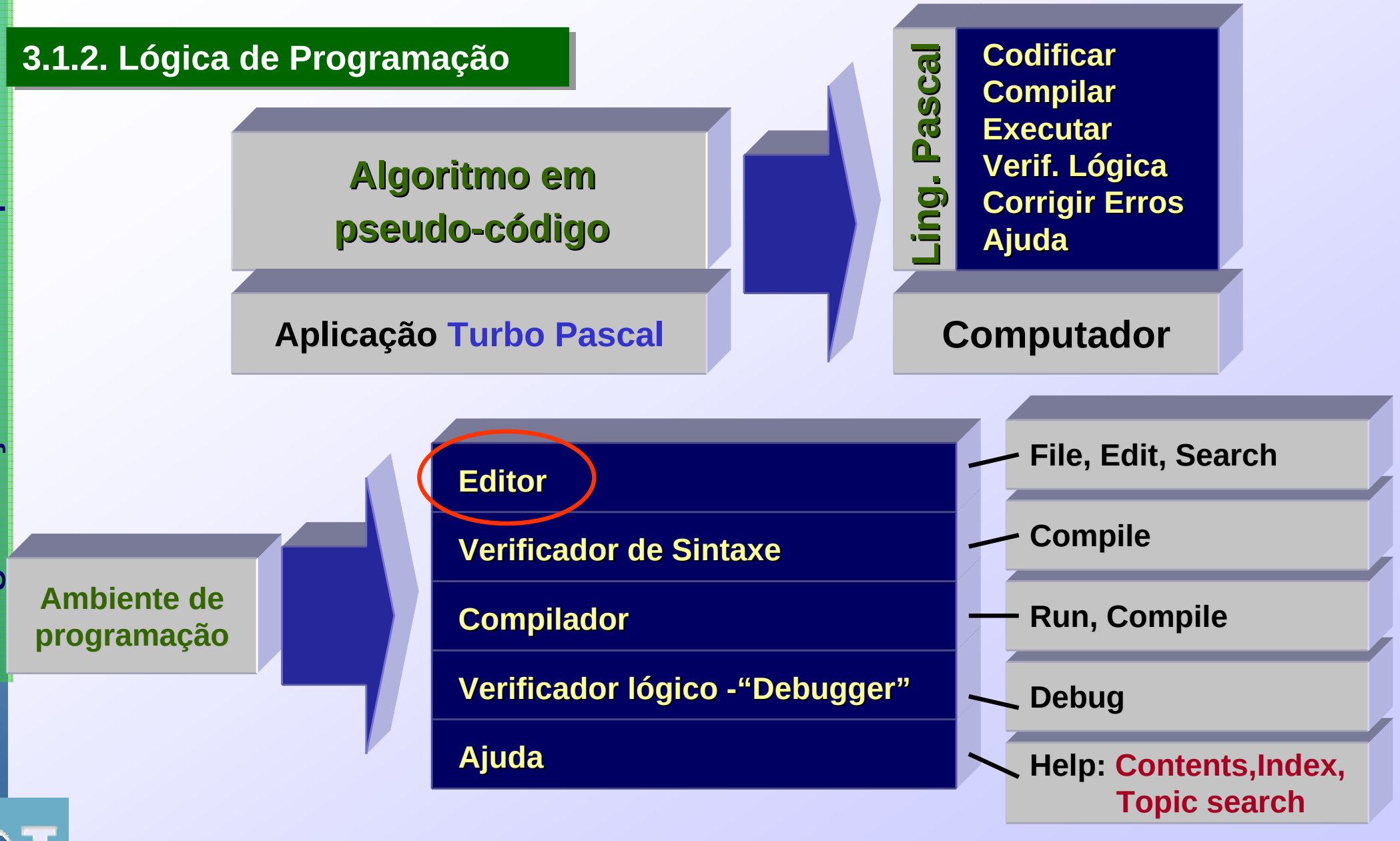
Em ambiente Windows

Em ambiente DOS

```
BGI             <DIR>          19-04-99
UTILS           <DIR>          19-04-99
TURBO3          <DIR>          19-04-99
TP114B7A $$$                0  26-04-99
        25 ficheiro(s)          1.813.657
        10 dir(s)               3.818,81

C:\Programas\Turbo Pascal>turbo_
```



3.1.2. Lógica de Programação



3.2. Programação em PASCAL

3.2.1. Estrutura dum programa

Cabeçalho:
Nome e parâmetros de entrada
(Opcional em Turbo Pascal)

Parte Declarativa:
variáveis, constantes e tipos
Funções e Procedimentos, etc.

Parte Operativa:
Expressões, Instruções,
Estruturas de controle

```
INICIO
   ECREVER(" Valor 1  e  Valor 2 ? ");
   LER(valor1, valor2);
   resultado ← 0;
   SE valor2 <> 0 ENTÃO
      resultado ← valor1/valor2;
   FIM SE
   ESCREVER(" O resultado é ", resultado);
FIM
```

Algoritmo

```
PROGRAM Exemplo(Input, Output);

VAR
  valor1, valor2 : INTEGER;
  resultado      : REAL;

BEGIN
   WRITE(' Valor 1  e  Valor 2 ? ');
   READ(valor1, valor2);
   resultado := 0;
   IF valor2 <> 0 THEN
     resultado := valor1 / valor2;
   WRITE(' O resultado é ', resultado);
END.
```

Componentes dum programa

Cabeçalho: Contem a palavra reservada **PROGRAM** que identifica o inicio do programa, seguido do nome que serve de identificador. Pode ainda conter os parâmetros **Input**, **Output** e outros. No Turbo Pascal este cabeçalho é meramente decorativo, o que não se verifica noutras versões.

Parte declarativa: Está dividida em secções identificadas pelas palavras reservadas **USES**, **LABEL**, **CONST**, **TYPE**, **VAR**, **PROCEDURE** e **FUNCTION**. Cada secção pode aparecer várias vezes (excepto Uses) e por qualquer ordem, embora seja aconselhável seguir a ordem indicada. Uma secção começa sempre por uma destas palavras reservadas e termina quando começa outra secção.

- **Uses:** onde se declaram unidades de código externas ao módulo (programação modular).
- **Label:** zona de declaração de rótulos de instruções, utilizados pela instrução de salto (*goto*).
- **Const:** onde se faz a identificação das constantes e atribuição do seu valor.
- **Type:** zona onde se pode definir novos identificadores para tipo de dados.
- **Var:** zona de declaração das variáveis a utilizar pelo programa. Todas as variáveis utilizadas na parte operativa têm que ser aqui identificadas.
- **Procedure:** parte do programa com funcionalidade específica, com parâmetros de entrada mas que não devolve valores. Também tem o seu próprio cabeçalho, parte declarativa e parte operativa.
- **Function:** Idêntico à Procedure mas devolvendo um valor.

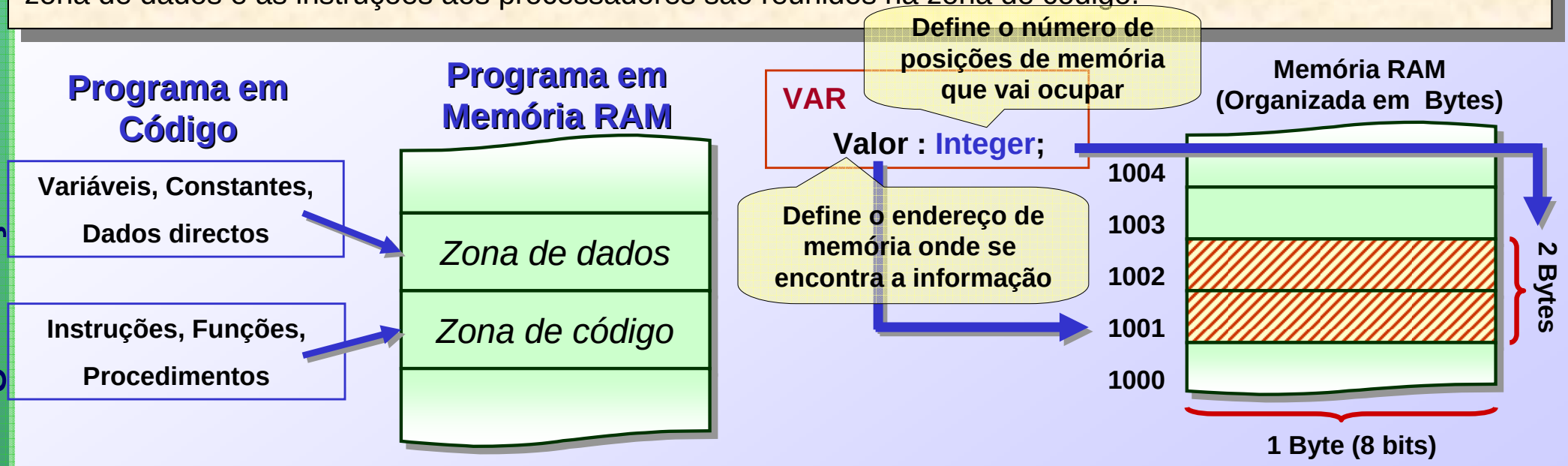
Parte Operativa: Começa com a palavra reservada **BEGIN** e termina com a palavra **END** seguida do ponto final. Entre essas duas palavras pode existir qualquer número e tipo de instruções, expressões e estruturas de controle, que serão resultantes da conversão do algoritmo.

3.2.2. Variáveis e constantes

3.2.2.1. Tipos de dados

3.2.2.2. Relação entre dados, código e memória RAM

Organização de um programa em RAM - O compilador ao analisar um programa faz já a separação entre dados e instruções, de forma a que todos os dados (variáveis, constantes, valores directos) são reunidos na zona de dados e as instruções aos processadores são reunidos na zona de código.



Relação entre tipo de dados, identificador de variável e RAM - A utilização num programa de qualquer tipo de variável ou constante implica a reserva de espaço na zona de dados da memória RAM, cuja dimensão está directamente ligada ao tipo de dados.

3.2.2.3. Sumário dos vários tipos de dados

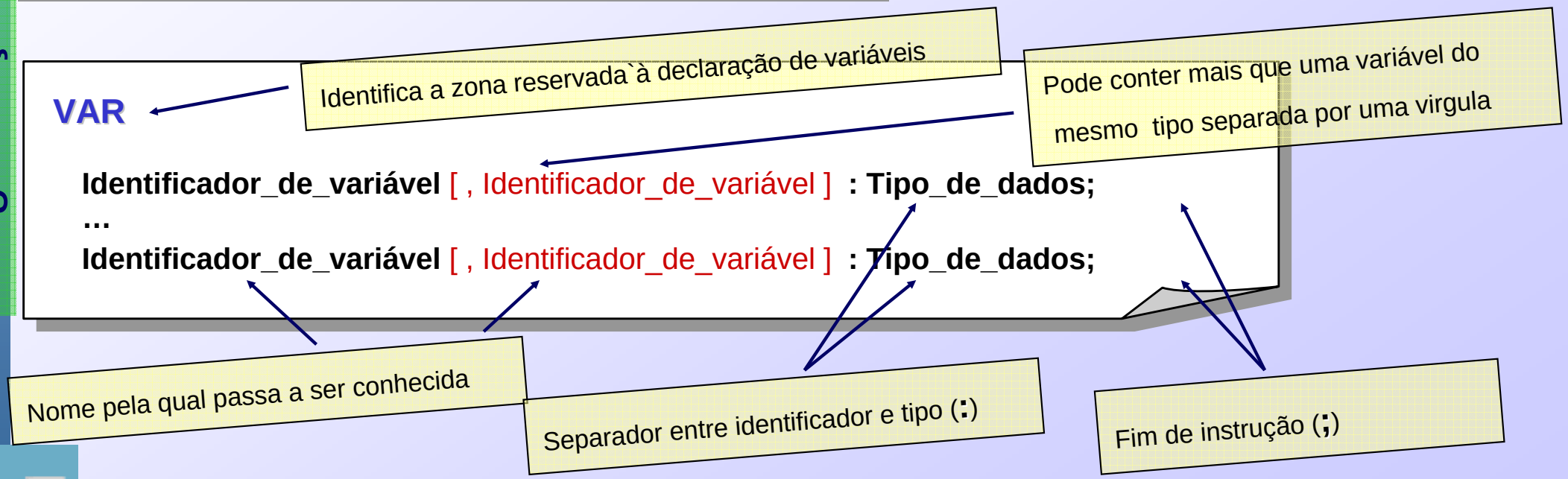
Tipos de Dados	Ocupação (em bytes)	Gama de valores
Boolean	1 *byte*	***True*** ou ***False***
Shortint (inteiro pequeno)	1 *byte*	-128 a 127
Integer	2 *bytes*	-32768 a 32767
Longint (inteiro grande)	4 *bytes*	-2 147 483 648 a 2 147 483 647
Byte (inteiro pequeno sem sinal)	1 *byte*	0 a 255
Word (inteiro sem sinal)	2 *bytes*	0 a 65535
Real (virgula flutuante)	6 *bytes*	-2,9e-39 a 1,7e38 → 11 a 12 dígitos de precisão
Single (virgula flutuante de precisão simples)	4 *bytes*	-1,5e-45 a 3,4e38 → 7 a 8 dígitos de precisão
Double (virgula flutuante de precisão dupla)	8 *bytes*	-5,0e-324 a 1,7e308 → 15 a 16 dígitos de precisão
Extended (virgula flutuante de precisão extra)	10 *bytes*	-3,4e-4932 a 1,1e34932 → 19 a 20 dígitos de precisão
Comp (virgula flutuante de tamanho reduzido)	8 *bytes*	9,2e-18 a 9,2e18 → 19 a 20 dígitos de precisão
Char	1 *byte*	Tipo utilizado para guardar um caracter ASCII, pode tomar valores de 0 a 255.
String	(1 *byte*/caracter)+1	1 a 255 caracteres

3.2.2.4. Declaração de variáveis

Uma variável representa a área de memória, que contém um valor dum determinado tipo de dados. Para associar um determinado tipo de dados a uma dada variável, é necessário proceder-se à **declaração da variável**. A declaração da variável condiciona o tipo de dados que esta pode conter, durante a execução dum programa.

A zona de declaração de variáveis é identificada pela palavra chave **VAR**.

3.2.2.5. Sintaxe: declaração duma variável



Algumas considerações: declaração de variáveis

- O identificador (nome) de uma variável **não pode**:
  - conter espaços e operadores;
  - começar por um número;
  - ser igual a uma palavra reservada.
- O identificador pode conter qualquer número de caracteres, embora só sejam válidos os primeiros 63 (em Turbo Pascal).
- Não existe distinção entre identificadores escritos em letra maiúscula ou minúscula, ou seja, são exactamente o mesmo identificador.
- O nome escolhido para a variável deve indicar qual é a informação que esta contem.
- Para melhor clareza de leitura, deve-se identificar palavras reservadas com letra maiúscula e identificadores com letra minúscula.

{ Exemplos }

```
var
   x, y, z :  REAL;
   i, j, k :  INTEGER;
   Vector:  ARRAY[1..10] OF REAL;
   Nome:  STRING[15];
   Letras:  SET OF 'A'..'Z';
   Pronto, Erro : BOOLEAN;
```

{ Exemplos }

```
var
   Ponto1, Ponto2 : REAL;
   Contador : INTEGER;
   Teste : BOOLEAN;
   Notas_Aluno : ARRAY[1..10] OF INTEGER;
   Nome_Cliente :  STRING[15];
```

3.2.2.6. Declaração de constantes

```
CONST
   identificador_de_variável [ : tipo_de_constante ] = valor ou expressão;
   ...
   identificador_de_variável [ : tipo_de_constante ] = valor ou expressão;
```

A diferença em relação à declaração de variáveis reside na palavra reservada utilizada, **CONST** em vez de **VAR**, e ainda na necessidade de indicar antecipadamente o valor atribuído à constante (valor que nunca poderá ser alterado).

```
PROGRAM Exemplo;
CONST
  ValorPi : REAL = 3.14; {ou ValorPI = 3.14}
VAR
  Area, Raio : REAL;
BEGIN
  WRITE('Indique o valor do raio');
  READ(Raio);
  Area := ValorPi * Raio * Raio;
  WRITE('A área do círculo é ', Area);
END.
```

Nota: Em toda a codificação um programa utiliza-se sempre o nome da constante (e não o seu valor), pelo que se quisermos alterar essa configuração, basta alterar o valor na linha da declaração da constante.

3.2.3. Instruções

A parte operativa dum procedimento, função ou programa é composto por um conjunto de **instruções**. Numa linguagem de programação as **instruções** correspondem à codificação dum algoritmo (pseudo-linguagem ou fluxograma), de acordo com a sintaxe específica da linguagem.

3.2.3.1. Tipos de Instruções

- **Leitura e Escrita**
- **Atribuição**
- **Expressões**
- **Comentários**
- **Estruturas de controlo**

3.2.4. Leitura e Escrita

As instruções de Leitura e Escrita, correspondem na linguagem Pascal, às funções de entrada e saída de dados : ***WRITE*** e ***WRITELN***, ***READ*** e ***READLN***.

A diferença entre WRITE e WRITELN reside nesta última, que acrescenta uma mudança de linha à informação de saída. Entre o READ e READLN diferem nesta última, ao eliminar toda a informação que tenha sido introduzida e seja excedente (mais informação que número de variáveis).

Exemplo

Descrição Algorítmica → ***Codificação em Pascal***

```
...
Inst.1: Escrever ("Cálculo da área do rectângulo")
Inst.2: Escrever ("Indique o valor da base")
Inst.3: Ler (base)
Inst.4: Escrever ("Indique o valor da altura")
Inst.5: Ler (altura)
Inst.6: area ← base * altura
Inst.7: Escrever ( "A área é:" area)
...
```

```
PROGRAM AreaRectangular;
VAR
  Area, Base, Altura : INTEGER;

BEGIN
  WRITELN('Cálculo da área do rectângulo');
  WRITE('Indique o valor da base - ');
  READ(Base);
  WRITE('Indique o valor da altura - ');
  READLN(Altura);
  Area := Base * Altura;
  WRITELN('A área é: ', Area);
END.
```

Sintaxe:

Quando se pretende escrever num local diferente do ecrã, a primeira variável tem que ser do tipo ***FILE***.

Identificador de variável a ser escrita

Write ([Var_de_ Ficheiro,] variável1[:n:m] [, variável2, variável3, … , variáveln]);

WriteLn ([Var_de_ Ficheiro,] variável1[:n:m] [, variável2, variável3, … , variáveln]);

Acrescenta uma mudança de linha

Formatação da variável. "n" indica o número mínimo de espaços a reservar. "m" indica o número de casas decimais para valores Reais.

Outros identificadores de variáveis separados por vírgulas (opcionais)

Read ([Var_de_ Ficheiro,] variável1 [, variável2, variável3, … , variáveln]);

ReadLn ([Var_de_ Ficheiro,] variável1 [, variável2, variável3, … , variáveln]);

Elimina o resto da informação até ao fim da linha

Identificador de variável a ser lida

3.2.5. Atribuição

Em Pascal, o operador de atribuição é constituído pelos sinais dois pontos e igual "**:=**". Tal como o símbolo utilizado na descrição algorítmica ("←"), também este tem o significado: " *toma o valor de* ".

A instrução em pseudo-linguagem **i←3**, seria codificado em Pascal através da instrução **i:=3**

Exemplo

Descrição Algorítmica → ***Codificação em Pascal***

...
Inst.1: **Escrever** ("Cálculo da área do rectângulo")
Inst.2: **Escrever** ("Indique o valor da base")
Inst.3: **Ler** (**base**)
Inst.4: **Escrever** ("Indique o valor da altura")
Inst.5: **Ler** (**altura**)
Inst.6: **area ← base * altura**
Inst.7: **Escrever** ("A área é:" **area**)
...

```
PROGRAM AreaRectangular;
VAR
  Area, Base, Altura : INTEGER;

BEGIN
  WRITELN('Cálculo da área do rectângulo');
  WRITE('Indique o valor da base - ');
  READ(Base);
  WRITE('Indique o valor da altura - ');
  READLN(Altura);
  Area := Base * Altura;
  WRITELN('A área é: ', Area);
END.
```

3.2.6. Expressões

Tal como num algoritmo uma expressão (ou fórmula) é composta por um conjunto de operandos (dados directos ou indirectos), relacionados entre si por operadores e/ou por funções, permitindo o cálculo de valores, a partir dum outro conjunto de valores.

3.2.6.1. Tipos de expressões

Numéricas - são expressões que utilizam apenas operadores aritméticos, funções matemáticas e trigonométricas e operandos numéricos (inteiros ou reais).

```
{ Exemplos }
Var_A := Sin(x) + 2* Cos(x);
Var_B := 100* (1+0.15)* Var_A;
Area  := base * altura;
```

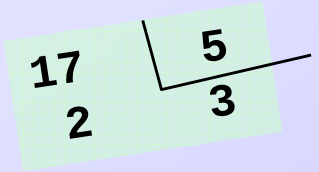
Booleanas ou Lógicas - são expressões que utilizam apenas operadores lógicos e operadores relacionais, e têm como resultado valores do tipo booleano (*True*, *False*).

```
{ Exemplos }
Var_A >= Var_B
Var_F = NOT Var_C
Var_G = (Var_C AND Var_D)
```

3.2.6.2. Tipos de operadores e respectiva precedência

Operadores aritméticos

Operador	Significado	Exemplo
+	Adição	
-	Subtracção ou sinal negativo	
*	Produto	
/	Divisão	17/5 vale 3.4
Div	Parte inteira da divisão	17 Div 5 vale 3
Mod	Resto da divisão inteira	17 Mod 5 vale 2



Operadores relacionais

Operador	Significado
<	Menor que
<=	Menor ou igual a
>	Maior que
>=	Maior ou igual a
=	Igual a
<>	diferente

Tome atenção: a interpretação da relação entre dois valores depende do seu tipo; enquanto a relação entre valores numéricos se baseia na ordem que estes ocupam em termos da recta real (onde se incluem os inteiros), a relação entre *strings* (cadeias de caracteres) baseia-se na sua ordem de acordo com a tabela ASCII:

ex.: -5 < 2 < 4 < 10

'1' < '10' < '1006' < '184' < '2' < '30' < 'ola'

Operadores lógicos (booleanos)

Operador	Significado	Símbolo matemático
Not	Negação	~
And	"e"	$\wedge$
Or	"ou"	$\vee$
Xor	"ou" exclusivo	$\veebar$
Shl	Deslocamento à esquerda de n bits	
Shr	Deslocamento à direita de n bits	

Tabelas de Verdade

Negação

A	~A
V	F
F	V

Operador "E"

A	B	A $\wedge$ B
V	V	V
F	V	F
V	F	F
F	F	F

Operador "OU"

A	B	A $\vee$ B
V	V	V
F	V	V
V	F	V
F	F	F

Operador "OU" exclusivo

A	B	A $\veebar$ B
V	V	F
F	V	V
V	F	V
F	F	F

Tabela de precedências dos operadores:

Operadores Unários	`-`
	`+`
	`Not`
		1ª	Sinal de negativo
			Sinal de positivo
			Negação

Operadores multiplicação	`*, /`
	`Div`
	`Mod`
	`And`
	`Shl`
	`Shr`
		2ª	Multiplicação e divisão
			Divisão inteira
			Resto da divisão inteira
			"E" lógico
			Deslocar n bits à esquerda
			Deslocar n bits à direita

Operadores Adição	`+`
	`-`
	`Or`
	`Xor`
		3ª	Soma
			Subtracção
			"Ou" lógico
			Ou exclusivo

Operadores Relacionais	`=, <>, <, >, <=, >=` `In`	4ª	Igual a; diferente de; menor que; maior que; menor ou igual a; maior ou igual a; Pertencente a

Quando queremos construir uma expressão que envolva diferentes operadores teremos de ter em atenção a respectiva tabela de precedências.

Tome atenção:

a) quando estamos perante operadores com o mesmo nível de precedência estes deverão ser considerados de acordo com a ordem em que se apresentam: da esquerda para a direita.

b) quando queremos forçar uma deter-minada ordem na execução das operações duma expressão usam-se parêntesis

(ex.: `-3 -(4+6/(3-1))*5`).

3.2.6.3. Funções disponíveis no Pascal

Uma expressão (ou fórmula) pode conter além de operandos relacionados entre si através de operadores, também funções. O Pascal fornece já algumas funções próprias (*standard*) que ajudam a realizar diversas tarefas:

Funções	Designação	Tipo do Argumento	Tipo do resultado
Abs(x)	Valor absoluto de x	Inteiro ou real	Inteiro ou real
Round(x)	Valor arredondado de x	Real	Inteiro
Trunc(x)	Valor truncado de x	Real	Inteiro
Sqr(x)	Quadrado de x (x^2)	Inteiro ou Real	Inteiro ou Real
Sqrt(x)	Raiz quadrada de x	Real	Real
Exp(x)	Exponencial de e (e^x)	Real	Real
Ln(x)	Logaritmo natural de x	Real	Real
Sin(x)	Seno de x	Real (Radianos)	Real
Cos(x)	Coseno de x	Real (Radianos)	Real
ArcTan(x)	Arco tangente de x	Real	Real
Pi	Valor de Pi (3,14159…)	---	Real
Odd(x)	Avalia se x é ímpar	Inteiro	Booleano
Ord(x)	Determina o ordinal de x	Ordinal	Inteiro
Chr(x)	Caracter ASCII	Inteiro	Caracter
Pred(x)	Predecessor de x	Ordinal	Ordinal
Succ(x)	Sucessor de x	Ordinal	Ordinal
Eoln(f)	Avalia se é fim de linha	Ficheiro de texto	Booleano
Eof(f)	Avalia se é fim de ficheiro	Ficheiro	Booleano
Random(x)	Valor aleatório entre 0 e x	Inteiro	Inteiro

3.2.7. Estruturas de controlo

Existem vários tipos de estruturas de controlo, cada uma das quais com comportamentos distintos em termos da sequência de execução das diversas instruções dentro dum programa. Estas condicionam a ordem pela qual as várias instruções serão executadas no programa, tal como vimos na descrição algorítmica.

3.2.7.1. Tipos estruturas de controlo

- **Estruturas Sequenciais**
- **Estruturas de Decisão Condicional**
- **Estruturas de Repetição com n.º de iterações pré-definidas**
- **Estruturas de Repetição Condicional**

3.2.8. Estrutura Sequencial

De acordo com esta estrutura, as instruções são executadas pela ordem em que estas se encontram definidas num programa. Por defeito, esta é estrutura usada em qualquer programa.

Exemplo

Descrição Algorítmica → ***Codificação em Pascal***

```
...
Inst.1: Escrever ("Cálculo da área do rectângulo")
Inst.2: Escrever ("Indique o valor da base")
Inst.3: Ler (base)
Inst.4: Escrever ("Indique o valor da altura")
Inst.5: Ler (altura)
Inst.6: area ← base * altura
Inst.7: Escrever ( "A área é:" area)
...
```

Sequencial

```
PROGRAM AreaRectangular;
VAR
  Area, Base, Altura : INTEGER;

BEGIN
  WRITELN('Cálculo da área do rectângulo');
  WRITE('Indique o valor da base - ');
  READ(Base);
  WRITE('Indique o valor da altura - ');
  READLN(Altura);
  Area := Base * Altura;
  WRITELN('A área é ', Area);
END.
```

Sequencial

3.2.9. Estruturas de Decisão Condicional

3.2.9.1. Decisão Simples

Nesta estrutura de controlo, a instrução ou bloco de instruções serão executadas apenas se a ***condição*** testada for verdadeira (***True***).

Sintaxes:

Sintaxe 1

```
IF condição THEN
   Instrução;
```

ou

```
IF condição THEN Instrução;
```

A sintaxe acima utiliza-se quando é necessário executar uma única instrução quando a ***condição*** é verdadeira. Pode-se utilizar as duas versões indiferentemente, sendo obrigatório utilizar as palavras reservadas **IF**, **THEN** e o ponto e vírgula no final da instrução.

Sintaxe 2

```
IF condição THEN
  BEGIN
    Instrução 1;
    ...
    Instrução n;
  END;
```

No caso em que é necessário executar várias instruções (bloco de instruções) quando a ***condição*** é verdadeira, utilizam-se as palavras reservadas **BEGIN** e **END** para limitar o bloco de instruções, seguido de ponto e virgula no final de cada instrução e a seguir ao END.

Exemplo - Análise da nota dum aluno: (Nota >= 9,5 então ficou ***Aprovado***).

Descrição Algorítmica → ***Codificação em Pascal***

```
...
Inst. 1:  Situação ← ""
Inst. 2:  Escrever ("Indique a nota do aluno")
Inst. 3:  Ler (Nota)
Inst. 4:  SE Nota >= 9.5 ENTÃO
            Situação ← "Aprovado"
          FIM SE
Inst. 5   SE Situação = "Aprovado" ENTÃO
Inst. 5.1:   Escrever ("Indique o nome do aluno ")
Inst. 5.2    Ler (nome)
Inst. 5.3    Escrever ( "O aluno", nome, " foi ",Situação)
          FIM SE
...
```

```
PROGRAM AnaliseNota;
VAR
  Nota            : INTEGER;
  Situação, Nome  : STRING;

BEGIN
  Situação := ''; {Inicialização da var.}
  WRITE('Indique a nota do aluno: ');
  READ(Nota);
  IF Nota >= 9.5 THEN
    Situação := 'Aprovado';
  IF Situação = 'Aprovado' THEN
    BEGIN
      WRITE('Indique o nome do aluno: ');
      READLN(Nome);
      WRITELN('O aluno ',Nome,'foi ',Situação);
    END;
END.
```

3.2.9.2. Decisão Dupla (ou alternativa)

Na estrutura de controlo de decisão dupla, se a ***condição*** testada for verdadeira ("***True***"), serão executadas a *instrução 1* ou *bloco de instruções 1*, caso contrário ("***False***") serão executadas a *instrução 2* ou o *bloco de instruções 2*.

Sintaxe 1

```
IF condição THEN
   Instrução1
ELSE
   Instrução2;
```

ou

```
If condição Then Instrução1 Else Instrução2;
```

Não leva ponto e vírgula no final da instrução antes do *ELSE*

Sintaxe 2

```
IF condição THEN
  BEGIN
   Instrução 1.1;
   ...
   Instrução 1.n;
  END
ELSE
  BEGIN
   Instrução 2.1;
   ...
   Instrução 2.n;
  END;
```

Neste tipo de estrutura, a instrução ou bloco de instruções a executar quando a avaliação da ***condição*** resulta no valor falso, é identificada no seu inicio pela palavra reservada **ELSE**.

O ponto e vírgula nunca deve existir na instrução imediatamente antes do *ELSE*, porque indicaria o fim da estrutura de decisão o que não é correcto.

Exemplo - Análise da nota dum aluno: (Se Nota >= 9,5 então ***Aprovado*** Senão ***Reprovado***).

Descrição Algorítmica → ***Codificação em Pascal***

```
...
Inst.1:  Escrever ("Indique o nome do aluno")
Inst.2:  Ler (Nome)
Inst.3:  Escrever ("Indique a nota do aluno")
Inst.4:  Ler (Nota)
Inst.5:  SE Nota >= 9.5 ENTÃO
           Situação ← "Aprovado"
         SENÃO
           Situação ← "Reprovado "
         FIM SE
Inst.6:  Escrever ("O aluno ", Nome, "foi ", Situação)
...
```

```
PROGRAM AnaliseNota;
VAR
  Nota            : INTEGER;
  Situação, Nome  : STRING;

BEGIN
  WRITE('Indique o nome do aluno: ');
  READ(Nome);
  WRITE('Indique a nota do aluno: ');
  READ(Nota);
  IF Nota >= 9.5 THEN
    Situação := 'Aprovado'
  ELSE
    Situação := 'Reprovado';
  WRITELN('O aluno ',Nome,'foi ',Situação);
END.
```

3.2.9.3. Decisão Múltipla (ou selectiva)

As estruturas anteriores, encontram-se limitadas ao resultado lógico da sua condição ("***True***" ou "***False***"), pelo que podemos ter no máximo duas alternativas. Se quisermos utilizar mais do que uma condição, correspondendo a igual número de alternativas, teremos de utilizar uma **estrutura de decisão múltipla ou selectiva**.

Sintaxe 1

```
IF condição_1 THEN
    Instrução_1 ou bloco de instruções_1
ELSE
  IF condição_2 THEN
    Instrução_2 ou bloco de instruções_2
      ..... ..... ..... ..... ..... .....
  ELSE
    IF condição_n THEN
      Instrução_n ou bloco de instruções_n
    ELSE
      Instrução_n+1 ou bloco de instruções_n+1;
```

Exemplo

Descrição Algorítmica → *Codificação em Pascal*

```
...

Inst.1:  Escrever ("Indique a nota do aluno")
Inst.2:  Ler (Nota)
Inst.3:  SE Nota < 10 ENTÃO
           Nota_Qual ← "Insuficiente"
         SENÃO
           SE Nota < 14 ENTÃO
             Nota_Qual ← "Suficiente"
           SENÃO
             SE Nota < 17 ENTÃO
               Nota_Qual ← "Bom"
             SENÃO
               SE Nota <= 20 ENTÃO
                 Nota_Qual ← "Muito Bom"
               SENÃO
                 Nota_Qual ← "Nota Inválida"
               FIM SE
             FIM SE
           FIM SE
         FIM SE
Inst.4:  Escrever ("A nota", Nota, "corresponde a ", Nota_Qual)
...
```

```
PROGRAM AvaliacaoDisciplina;
VAR
  Nota      : INTEGER;
  Nota_Qual : STRING;

BEGIN
  WRITE('Indique a nota do aluno: ');
  READ(Nota);
  IF Nota < 10 THEN
    Nota_Qual := 'Insuficiente'
  ELSE
    IF Nota < 14 THEN
      Nota_Qual := 'Suficiente'
    ELSE
      IF Nota < 17 THEN
        Nota_Qual := 'Bom'
      ELSE
        IF Nota <= 20 THEN
          Nota_Qual := 'Muito Bom'
        ELSE
          Nota_Qual := 'Nota inválida';
  WRITELN('A nota ', Nota,
          'corresponde a ', Nota_Qual);
END.
```

Sintaxe 2

```
CASE expressão OF
  lista_de_constantes_1 :
        Instrução_1 ou bloco de instruções_1;
        …… …… …… …… …… ……
  lista_de_constantes_n :
        Instrução_n ou bloco de instruções_n;
  ELSE
    Instrução_n+1 ou bloco de instruções_n+1;
END;
```

Exemplo

Descrição Algorítmica

```
...
Inst.1:  Escrever ("Indique a nota do aluno")
Inst.2:  Ler (Nota)
Inst.3:  CASO Nota SEJA
           0 a 9 FAZ
              Nota_Qual ← "Insuficiente"
           10 a 13 FAZ
              Nota_Qual ← "Suficiente"
           14 a 16 FAZ
              Nota_Qual ← "Bom"
           17 a 20 FAZ
              Nota_Qual ← "Muito Bom"
         SENÃO FAZ
           Nota_Qual ← "Inválida"
         FIM CASO
Inst.4:  Escrever ("A nota ", Nota, "corresponde a ", Nota_Qual)
...
```

Codificação em Pascal

```
PROGRAM AvaliacaoDisciplina;
VAR
  Nota      : INTEGER;
  Nota_Qual : STRING;
BEGIN
  WRITE('Indique a nota do aluno: ');
  READ(Nota);
  CASE Nota OF
    0 .. 9 :
       Nota_Qual := 'Insuficiente';
    10 .. 13 :
       Nota_Qual := 'Suficiente';
    14, 15, 16 :
       Nota_Qual := 'Bom';
    17, 18, 19, 20 :
       Nota_Qual := 'Muito Bom';
    ELSE
       Nota_Qual := 'Nota inválida';
  END;
  WRITELN('A nota ', Nota,
          'corresponde a ', Nota_Qual);
END.
```

Algumas considerações:

- A **lista_de_constantes** indicada em cada linha que antecede os separador "**:**" pode conter constantes ou gamas de valores, separados por vírgulas. Exemplos:

- **constante** : qualquer número de constantes, todas separadas por vírgulas.

Ex.: `2, 3, 4`

- **Constante .. Constante** : permite a especificação duma gama de valores, através da utilização de "**..**".

Ex.: `'a' .. 'z', 'A' .. 'Z', '0' .. '9'`

- **ELSE** : destina-se a indicar a instrução ou bloco de instruções a executar no caso da expressão indicada na linha **CASE ... OF**, tomar um valor diferente de todos os referidos listas de constantes. A sua utilização é opcional.

Ex.:
```
ELSE WRITELN('ERRO: Valor não válido');
```

Nota: se existir mais do que uma linha cuja gama de valores seja coincidente, será apenas executada a instrução ou o bloco de instruções correspondente à primeira linha que coincida com esses valores.

3.2.10. Estrutura de Repetição com n.º de iterações pré-definidas

3.2.10.1. For ... To ... Do

Esta estrutura de controlo permite a repetição duma determinada instrução ou bloco de instruções, o número de vezes que tiverem sido pré-definidas, e duma forma controlada. Cada repetição corresponde a um ciclo.

Sintaxe:

```
FOR variável_de_iteração := valor_inicial TO valor_final DO
    Instrução;
```

ou

```
FOR var_de_iteração := val_inicial TO val_final DO
   BEGIN
     Instrução 1;
     ...
     Instrução n;
   END;
```

Algumas considerações:

- Esta estrutura implica a utilização duma **variável_de_iteração** ou contador, que em cada nova iteração (ciclo) é somado 1 valor. Começa com o **valor_inicial** e vai até ao **valor_final**. Se o **valor_final** for menor que **valor_inicial** então não faz nada.

Exemplo 1

$$S = \sum_{i=1}^{N} (2 - i) \times i$$

Cálculo do Somatório dos primeiros N elementos

Descrição Algorítmica

```
...
Inst.1: Soma ← 0
Inst.2: DE i ← 1 ATÉ N FAZ
            Soma ← Soma + (2-i)*i
        FIM DE
Inst.3: Escrever ( "O resultado do somatório é", Soma)
...
```

Codificação em Pascal

```
PROGRAM Somatorio;
VAR
  i, N, Soma : INTEGER;

BEGIN
  WRITE('Indique o nº de elementos do somatório:');
  READ(N);
  Soma := 0;
  FOR i := 1 TO N DO
    Soma := Soma + (2-i)*i;
  WRITELN('O resultado do somatório é:',soma);
END.
```

Exemplo 2

- Ler valores da matriz A
- Escrever valores da matriz A

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{22} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Descrição Algorítmica

```
...
' Ler matriz A
Inst.1: DE i ← 1 ATÉ N_L FAZ
          DE j ← 1 ATÉ N_C FAZ
            Escrever("Indique o valor de A(i,j)")
            Ler(A[i,j])
          FIM DE (j)
        FIM DE (i)
'Escrever matriz A
Inst.2: DE i ← 1 ATÉ N_L FAZ
          DE j ← 1 ATÉ N_C FAZ
            Escrever("A(i,j) =", A[i,j])
          FIM DE (j)
        FIM DE (i)
...
```

Codificação em Pascal

```
PROGRAM LerVisualizarMatriz2D;
CONST
  N_L : INTEGER = 3;
  N_C : INTEGER = 3;
VAR
  Matriz : ARRAY[1..3, 1..3] OF INTEGER;
  i, j   : INTEGER;

BEGIN
  {Ler a Matriz}
  FOR i := 1 TO N_L DO
    FOR j := 1 TO N_C DO
     BEGIN
       WRITE('Indique o valor de A(',i,',',j,')');
       READ(Matriz[i,j]);
     END;
  {Escrever a Matriz}
  FOR i := 1 TO N_L DO
    FOR j := 1 TO N_C DO
      WRITELN('A(',i,',',j,') =', Matriz[i,j]);
END.
```

3.2.10.2. For ... Downto ... Do

Igualmente à anterior, esta estrutura de controlo permite a repetição um número de vezes pré-definido, mas neste caso á variável de iteração deve ser atribuído o valor máximo, que vai decrescendo a cada repetição ou ciclo até atingir o valor final.

Sintaxe:

```
FOR variável_de_iteração := valor_inicial DOWNTO valor_final DO
    Instrução;
```

ou

```
FOR var_de_iteração := val_inicial DOWNTO val_final DO
   BEGIN
     Instrução 1;
     ...
     Instrução n;
   END;
```

Algumas considerações:

- Esta estrutura também implica a utilização duma **variável_de_iteração** ou contador, que em cada nova iteração (ciclo) é subtraído 1 valor. Começa com o **valor_inicial** e decresce até ao **valor_final**. Se o **valor_final** for maior que **valor_inicial** então não faz nada.

Exemplo

$$S = \sum_{i=1}^{N} (2 - i) \times i$$

Cálculo do Somatório dos primeiros N elementos

Descrição Algorítmica

```
...
Inst.1: Soma ← 0
Inst.2: DE i ← N ATÉ 1 FAZ (SALTO -1)
            Soma ← Soma + (2-i)*i
        FIM DE
Inst.3: Escrever ( "O resultado do somatório é", Soma)
...
```

Codificação em Pascal

```
PROGRAM SomatorioInvertido;
VAR
  i, N, Soma : INTEGER;

BEGIN
  WRITE('Indique o nº de elementos do somatório:');
  READ(N);
  Soma := 0;
  FOR i := N DOWNTO 1 DO
    Soma := Soma + (2-i)* i;
  WRITELN('O resultado do somatório é:', soma);
END.
```

3.2.11. Repetição Condicional

Esta estrutura de controlo, de acordo com as suas variantes, permite a repetição duma instrução ou bloco de instruções em ciclo, em função do resultado do teste a uma dada condição em cada ciclo, podendo ser feito no início ou no fim de cada iteração.

Teste no início - o teste à condição é realizado antes da execução de qualquer instrução contida numa estrutura de controlo destas. Estrutura ***WHILE … DO***.

Teste no fim - o teste à condição é realizado depois de cada instrução contida numa estrutura de controlo destas ter sido executada, de forma a verificar se se deve avançar para outra iteração. Esta estrutura de controlo permite pelo menos uma repetição da instrução ou bloco de instruções. Estrutura ***REPEAT … UNTIL***.

3.2.11.1. While Condição Do ...

Esta estrutura de repetição faz o teste duma condição antes de executar qualquer instrução ou bloco de instruções presentes no ciclo e, enquanto ***condição*** for verdadeira, volta a executar a instrução ou bloco de instruções. Se a condição for "*False*" logo no início, a instrução ou o bloco de instruções do ciclo nunca serão executadas.

Sintaxe:

```
WHILE Condição DO
   Instrução;
```

ou

```
WHILE Condição DO
   BEGIN
     Bloco de instruções;
   END;
```

Exemplo:

$$S = \sum_{i=1}^{N} \frac{1}{(i+1)^2}$$

Somar até que o termo seja inferior a 0.001.

Descrição Algorítmica

```
...
Inst.1: i ← 1
Inst.2: soma ← 0
Inst.3: termo ← 1/((i+1)² )
Inst.4: ENQUANTO termo > 0.001
           s ← soma+termo
           i← i+1
           termo ← 1/((i+1)² )
        FIM ENQUANTO
...
```

Codificação em Pascal

```
PROGRAM Somatorio;
CONST
  erro = 0.001;
VAR
  i            : INTEGER;
  Soma, Termo  : REAL;
BEGIN
  i := 1; Soma := 0; Termo := 1/sqr(i+1);
  WHILE Termo > 0.001 DO
    BEGIN
      Soma  := Soma + Termo;
      i     := i+1;
      Termo := 1/sqr(i+1);
    END;
  WRITELN('O resultado é:', soma:0:3);
END.
```

3.2.11.2. Repeat ... Until Condição

Esta variante faz o teste da condição no fim de executar pelo menos uma vez a instrução ou bloco de instruções presentes no ciclo e, repete a instrução ou bloco de instruções até ***condição*** se tornar verdadeira. Se a condição for "*Verdadeira*" logo no início, a instrução ou bloco de instruções serão executadas pelo menos uma vez.

Sintaxe:

```
REPEAT
   Instrução 1;
   ...
   Instrução n;
UNTIL Condição
```

Exemplo: $S = \sum_{i=1}^{N} \frac{1}{(i+1)^2}$ **Somar até que o termo seja inferior a 0.001.**

Descrição Algorítmica

```
...
Inst.1: i ← 1
Inst.2: soma ← 0
Inst.3: REPETIR
          termo ← 1/((i+1)² )
          soma ← soma+termo
          i ← i+1
        ATÉ termo <= 0.001
...
```

Codificação em Pascal

```
PROGRAM Somatorio;
CONST
  erro = 0.001;
VAR
  i            : INTEGER;
  Soma, Termo  : REAL;
BEGIN
  i := 1; Soma := 0;
  REPEAT
    Termo := 1/sqr(i+1);
    Soma  := Soma + Termo;
    i     := i+1;
  UNTIL Termo <= 0.001;
  WRITELN('O resultado é:', soma:0:3);
END.
```

3.2.12. Comentários

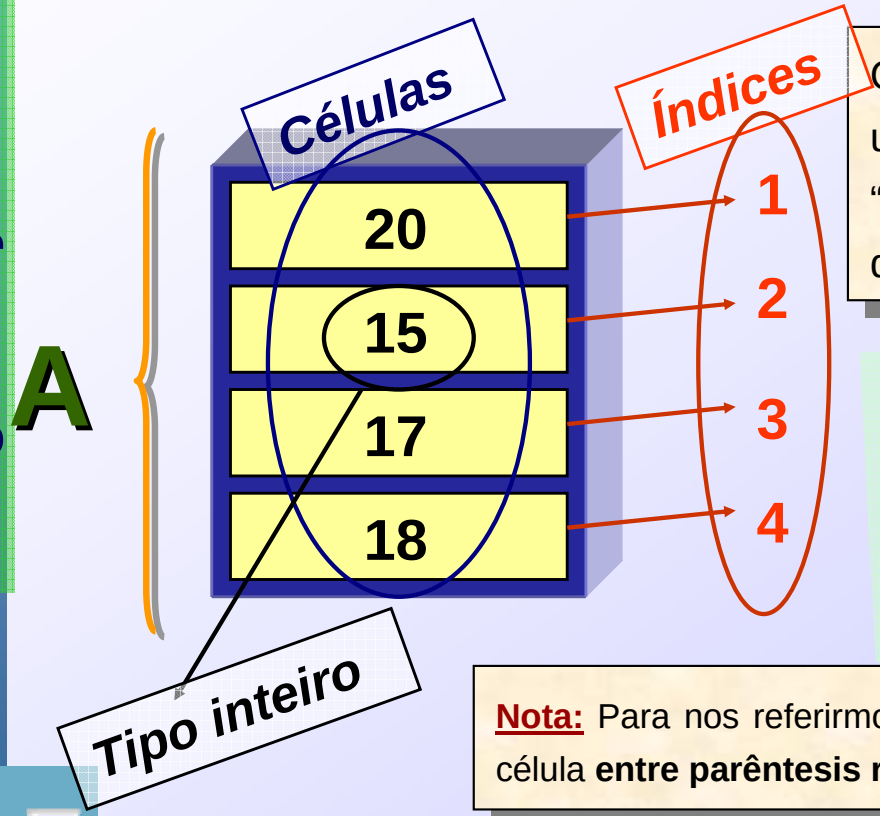
Os comentários são frases que não são interpretadas pelo compilador mas que têm como missão facilitar a compreensão do conteúdo dos módulos (declaração das variáveis ou instruções). Estão sempre entre chavetas **{ ... }** ou parêntesis curvo e asterisco **(* ...*)**.

Exemplos de comentários

```
PROGRAM AreaRectangular; {Calcula a área do rectângulo}
VAR     {Declarações de variáveis}
  Area   : INTEGER;    {Área do rectângulo}
  Base   : INTEGER;    {Base do rectângulo}
  Altura : INTEGER;    {Altura do rectângulo}

BEGIN    {Bloco de instruções codificadas}
  WRITELN('Cálculo da área do rectângulo');
  WRITE('Indique o valor da base - ');
  READ(Base);
  WRITE('Indique o valor da altura - ');
  READLN(Altura);
  Area := Base * Altura; {Expressão de cálculo da área}
  WRITELN('A área é ', Area);
END.  {Fim do programa}
```

3.3. Manipulação de variáveis estruturadas: arrays

Por vezes torna-se necessário/conveniente referenciar uma zona de memória onde estão guardados vários valores dum mesmo tipo de dados, utilizando um mesmo nome. Para o efeito, utilizam-se variáveis do tipo estruturado: ***Arrays***.



O ***Array*** trata-se dum tipo de variável estruturada, constituída por um conjunto de "células", identificadas univocamente por um "índice" (endereço), cujo o conteúdo é constituído por um valor dum determinado tipo de dados.

```
{ Exemplo }
VAR
   A : ARRAY[1..4] OF INTEGER;
   var_1 : INTEGER;
   ...
   var_1 := A[2]; {fica com o valor 15 do array A}
```

Nota: Para nos referirmos a cada um dos seus valores, utilizamos o nome do **Array** e o índice da célula **entre parêntesis rectos**, na qual o valor se encontra.

3.3.1. Sintaxe: declaração de variáveis de tipo estruturado ou Arrays

```
Nome_do_array : ARRAY [ gama_de_índices ] OF tipo_das_células;
```

A gama_de_índices, apresenta-se do seguinte modo:

menor_índice .. maior_índice,

o que é equivalente a dizer que vai "*desde o valor mais baixo até* (**..**) *ao valor mais alto*".

Exemplos de declaração de Arrays:

```
Array_A : ARRAY [ -2 .. 3 ] OF INTEGER;
```

O *array* **Array_A**, teria 6 células (desde a célula -2 à célula 3), com valores do tipo **Integer**.

```
Array_B : ARRAY [ 6 .. 8 ] OF CHAR;
```

O *array* **Array_B**, teria 3 células (desde a célula 6 à célula 8), com valores do tipo **Char**.

Algumas considerações:

```
Array_1 : ARRAY[3..7] OF INTEGER;
```

⟺

```
Array_1 : ARRAY[5..9] OF INTEGER;
```

- As duas formas de declarar a gama de índices contêm o mesmo número de células (5 células em ambos os casos), mas diferem na gama de índices, ou seja, na forma de acesso às células. A primeira célula tem o índice 3 no primeiro caso e o índice 5 no segundo caso.

```
Array_1 : ARRAY[1..5] OF INTEGER;
```

- Apesar de ser possível qualquer gama de índices (desde que o valor do índice inferior seja menor que o valor do índice superior), em geral é mais prático que o valor do índice inferior comece em 1.

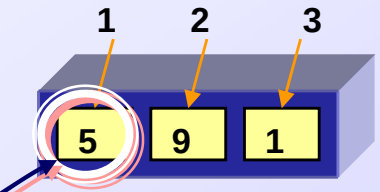
```
Array_N : ARRAY[1..6] OF INTEGER;

...

Array_N[2] := 4;

Array_N[20] := 5;
```

- Quando quisermos referir uma dada célula, indicamos o nome do *array* seguido do índice da célula ***entre parêntesis rectos***. Contudo, teremos de escolher sempre um valor do índice, que se encontre dentro da gama de índices.

Arrays multi-dimensionais:

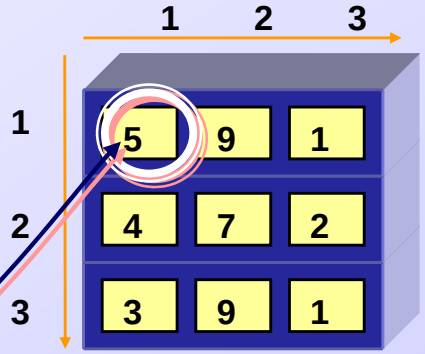
Array uni-dimensional - (1-D)

```
UniArray : ARRAY[1..3] OF INTEGER;

UniArray[1] := 5;
```



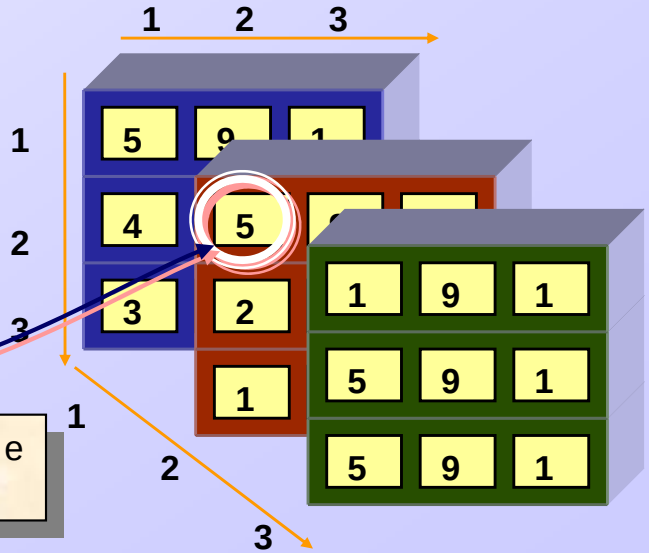
Array bi-dimensional - (2-D)

```
BiArray : ARRAY[1..3, 1..3] OF INTEGER;

BiArray[1, 1] := 5;
```



Array tri-dimensional - (3-D)

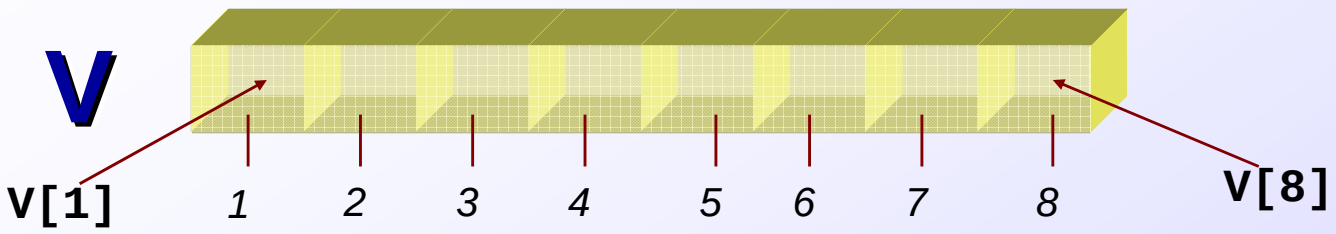
```
TriArray : ARRAY[1..3, 1..3, 1..3] OF INTEGER;

TriArray[1, 1, 2] := 5;
```



Array k-dimensional - (K-D)

Embora não se tenha representado aqui um **array** K-D, para K>3, a sua declaração e utilização é idêntica em Pascal.

3.3.2. Vectores: *Array* Unidimensional

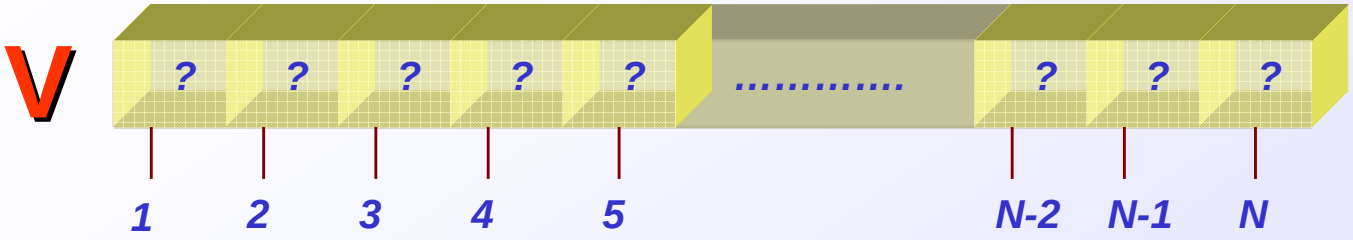



```
V : ARRAY [1 .. 8] OF INTEGER;
{ Array unidimensional c/ 8 células, com valores inteiros }
```

3.3.3. Operações básicas com vectores

- Inserir valores num vector
- Ler valores dum vector
- Máximo e mínimo e respectiva posição
- Soma de todos os elementos
- Média aritmética dos elementos
- Troca de dois elementos
- Inversão
- Permutação circular
- Remoção dum elemento
- Inserção dum novo elemento
- Pesquisa dum elemento
- Número de elementos repetidos
- Ordenação : *Bubble Sort*

3.3.3.1. Inserir valores num vector

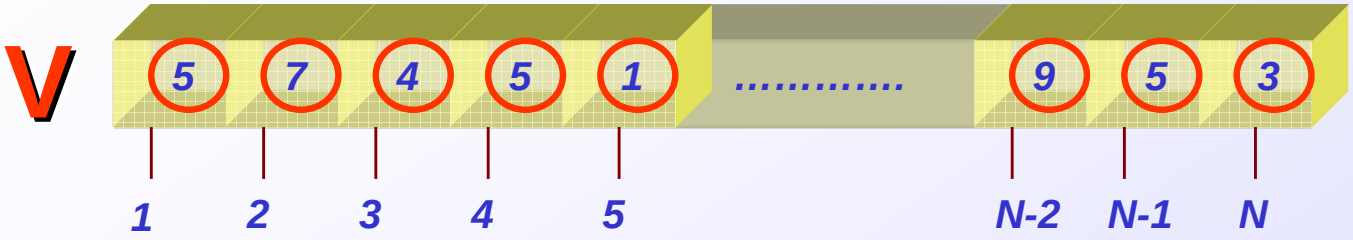


Descrição Algorítmica → **Codificação em Pascal**

```
…
De índice ← 1 Até N Faz
   Escrever (“Insira o elemento”, índice, ”: ”)
   Ler ( v( índice ) )
Fim De
…
```

```
PROGRAM Ler_Vector;
VAR
  i, N : INTEGER;
  v    : ARRAY [1 .. 100] OF INTEGER;
BEGIN
  WRITE(‘Quantos elementos? ‘);
  READLN(N);
  FOR i := 1 TO N DO
    BEGIN
      WRITE(‘Insira o elemento ‘, i, ‘: ‘);
      READLN(v[i]);
    END;
END.
```

3.3.3.2. Ler valores dum vector

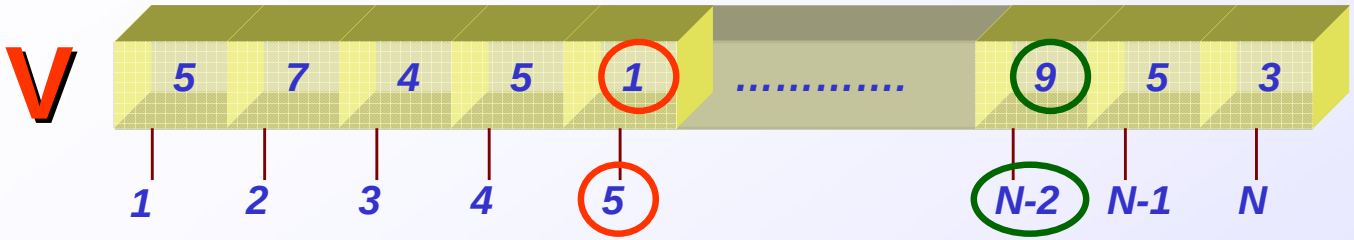


Descrição Algorítmica → **Codificação em Pascal**

```
…
De índice ← 1 Até N Faz
   Escrever (“v(”, índice,”)= “,V( índice ))
Fim De
…
```

```
PROGRAM Escrever_Vector;
VAR
  i, N : INTEGER;
  v    : ARRAY [1 .. 100] OF INTEGER;
BEGIN
  WRITE(‘Quantos elementos? ‘);
  READLN(N);
  ...
  FOR i := 1 TO N DO
    WRITELN(‘v[‘, i,‘] = ‘,v[i]);
END.
```

3.3.3.3. Máximo e mínimo dum vector e respectiva posição



Descrição Algorítmica

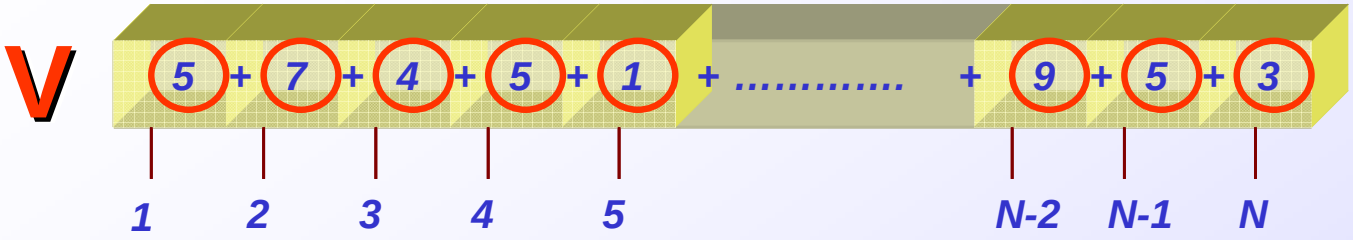
```
...
Máximo ← v(1)
Pos_Max ← 1
De índice ← 2 Até N Faz
   Se v(índice) > Máximo Então
      Máximo ← v(índice)
      Pos_Max ←índice
   Fim Se
Fim De
Escrever("Valor máximo:", Máximo," posição:", Pos_Max);
...
```

```
...
Mínimo ← v(1)
Pos_Min ← 1
De índice ← 2 Até N Faz
   Se v(índice) < Mínimo Então
      Mínimo ← v(índice)
      Pos_Min ←índice
   Fim Se
Fim De
Escrever("Valor mínimo:", Mínimo," posição:", Pos_Min);
...
```

Codificação em Pascal

```
PROGRAM Max_Min_Pos;
VAR
  Maximo, Minimo, Pos_Max, Pos_Min, N, i : INTEGER;
  v : ARRAY [1 .. 100] OF INTEGER;
BEGIN
  ...
  { Escolher o valor máximo do vector }
  Maximo  := v[1];
  Pos_Max := 1;
  FOR i := 2 TO N DO
    IF v[i] > Maximo Then
      BEGIN
        Maximo  := v[i];
        Pos_Max := i;
      END;
  WRITELN('Valor máximo:',Maximo,' posição:' , Pos_Max);
  { Escolher o valor mínimo do vector }
  Minimo  := v[1];
  Pos_Min := 1;
  FOR i := 2 TO N DO
    IF v[i] < Minimo THEN
      BEGIN
         Minimo  := v[i];
         Pos_Min := i;
      END;
  WRITELN('Valor mínimo:',Minimo,' posição:' , Pos_Min);
END.
```

3.3.3.4. Soma de todos os elementos dum vector

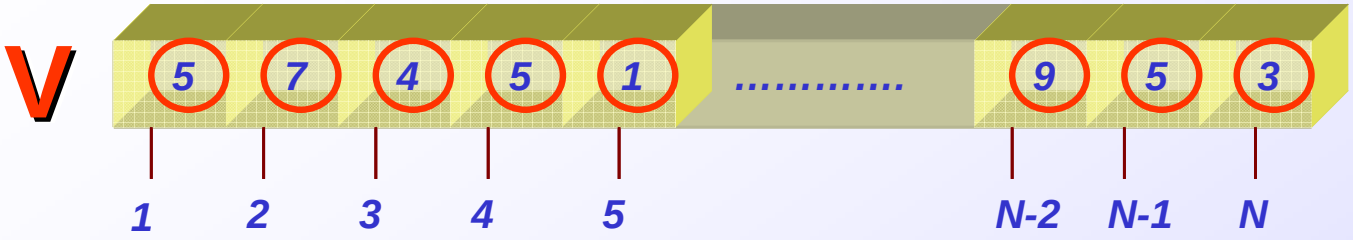


Descrição Algorítmica → **Codificação em Pascal**

```
...
soma ← 0
De índice ← 1 Até N Faz
   soma ← soma + v( índice )
Fim De
...
```

```
PROGRAM Soma_elementos_vector;
VAR
  v    : ARRAY [1 .. 100] OF INTEGER;
  i, N : INTEGER;
  Soma : LONGINT;

BEGIN
  ...
  Soma := 0;
  FOR i := 1 TO N DO
    Soma := Soma + v[i];
  WRITELN('A soma de todos os elementos do vector é ', soma);
END.
```

3.3.3.5. Média aritmética de todos os elementos dum vector



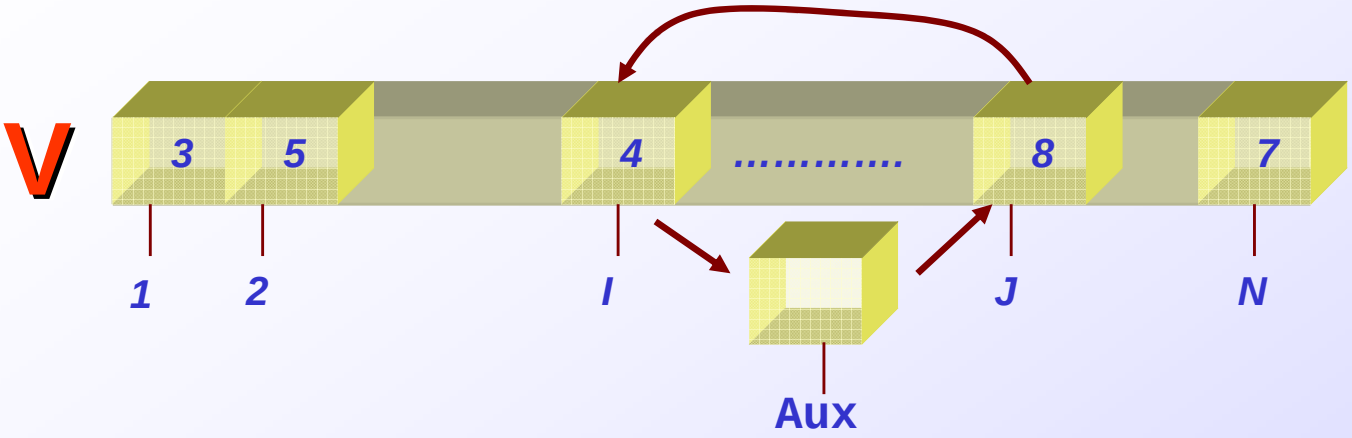
$$\overline{V} = \frac{\sum_{i=1}^{N} V(i)}{N}$$

Descrição Algorítmica → Codificação em Pascal

```
...
Média ← 0
De índice ← 1 Até N Faz
   Média ← Média + v( índice )
Fim De
Se N <> 0 Então
   Média ← Média / N
Fim Se
...
```

```
PROGRAM Media_Aritmetica_vector;
VAR
  v      : ARRAY [1 .. 100] OF INTEGER;
  i, N   : INTEGER;
  Media  : REAL;

BEGIN
  ...
  Media := 0;
  FOR i := 1 TO N DO
    Media := Media + v[i];
  IF N <> 0 THEN
    BEGIN
      Media := Media/N;
      WRITELN('A Média dos elementos do vector é ', Media);
    END;
END.
```

3.3.3.6. Troca de dois elementos dum vector



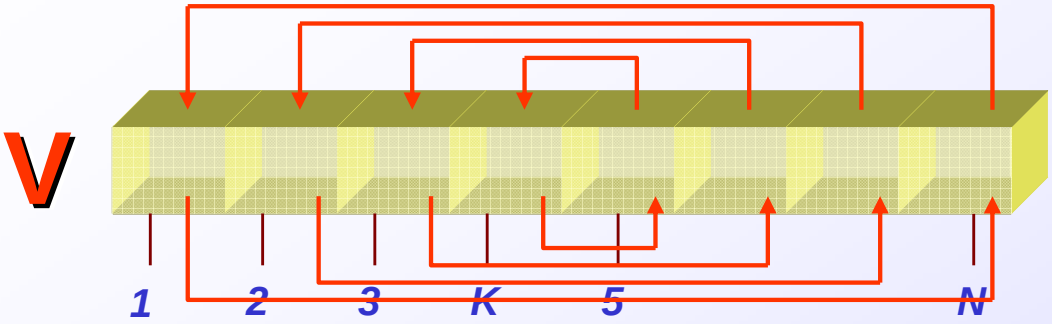
Descrição Algorítmica

```
...
Escrever("Qual o índice do 1.º elemento a trocar?")
Ler ( i )
Escrever("Qual o índice do 2.º elemento a trocar?")
Ler ( j )
Aux ← v( i )
v( i ) ← v( j )
v( j ) ← Aux
...
```

Codificação em Pascal

```
PROGRAM Troca_2_Elementos_vector;
VAR
  v              : ARRAY [1 .. 100] OF INTEGER;
  i, j, Aux, N : INTEGER;

BEGIN
  ...
  WRITELN('Qual o índice do 1º elemento a trocar? ');
  READLN(i);
  WRITELN('Qual o índice do 2º elemento a trocar? ');
  READLN(j);
  Aux := v[i];
  v[i]:= v[j];
  v[j]:= Aux;
  ...
END.
```

3.3.3.7. Inversão dum vector



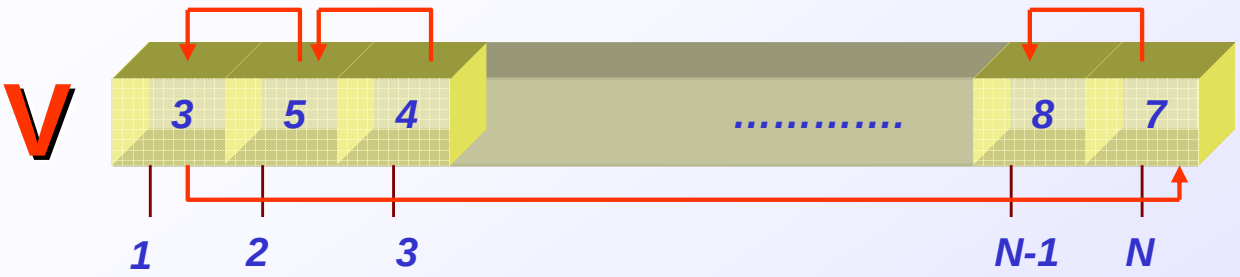
Descrição Algorítmica

```
...
k ← N \ 2   { Divisão inteira }
De índice ← 1 Até k Faz
   Aux ← v( índice )
   v( índice ) ← v( N – índice + 1 )
   v( N – índice + 1 ) ← Aux
Fim De
...
```

Codificação em Pascal

```
PROGRAM Inverte_vector;
VAR
  v             : ARRAY [1 .. 100] OF INTEGER;
  i, k, Aux, N : INTEGER;

BEGIN
  ...
  k := N DIV 2;
  FOR i := 1 TO k DO
    BEGIN
      Aux      := v[i];
      v[i]     := v[N-i+1];
      v[N-i+1]:= Aux;
    END;
...
END.
```

3.3.3.8. Permutação circular dum vector

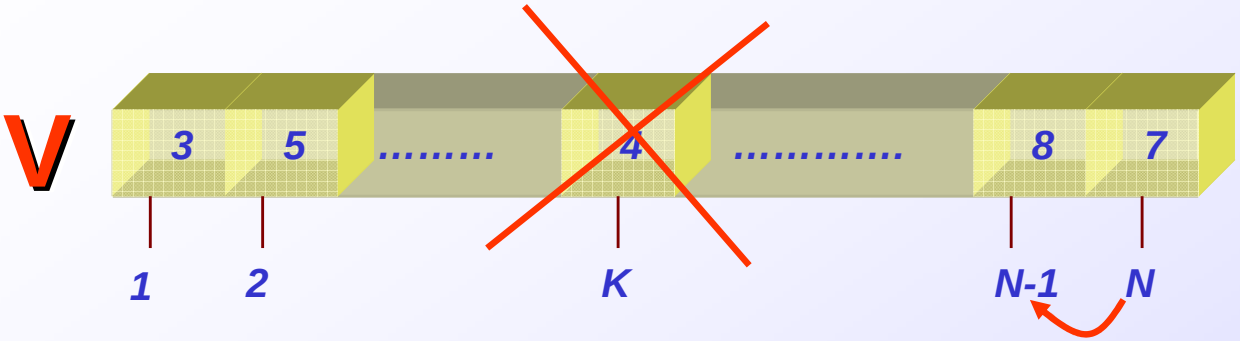


Descrição Algorítmica → **Codificação em Pascal**

```
...
Aux ← v( 1 )
De  índice ←1  Até  N -1  Faz
      v( índice ) ← v( índice + 1 )
Fim De
v( N ) ← Aux
...
```

```
PROGRAM Permuta_Circular_vector;
VAR
  v          : ARRAY [1 .. 100] OF INTEGER;
  i, Aux, N  : INTEGER;

BEGIN
  ...
  Aux := v[1];
  FOR i := 1 TO N-1 DO
    v[i] := v[i+1];
  v[N]:= Aux;
  ...
END.
```

3.3.3.9. Remoção dos elementos dum vector

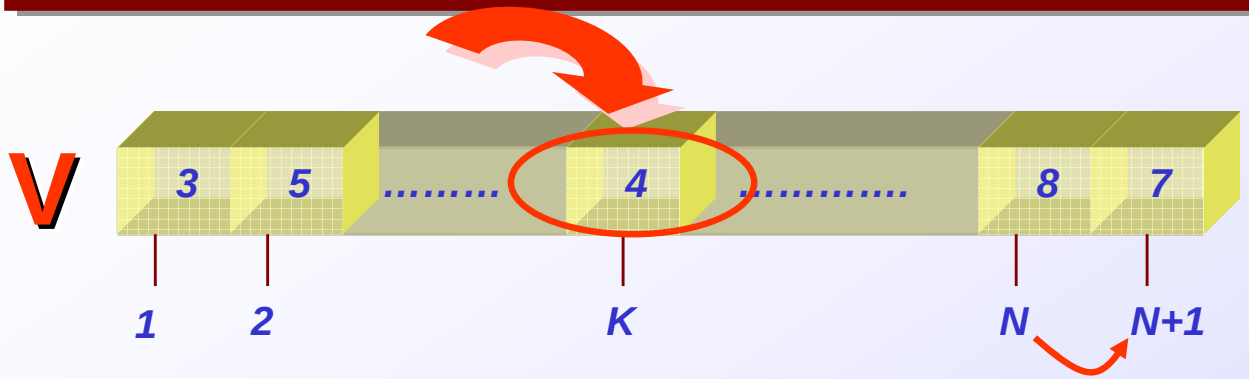


Descrição Algorítmica → **Codificação em Pascal**

```
…
Escrever("Qual o índice do elemento a remover?")
Ler ( k )
De índice ← k Até N - 1 Faz
    v( índice ) ← v( índice + 1 )
Fim De
N ← N - 1
…
```

```
PROGRAM Remover_Elemento_vector;
VAR
  v       : ARRAY [1 .. 100] OF INTEGER;
  i, k, N : INTEGER;

BEGIN
  ...
  WRITELN('Qual o índice do elemento a remover? ');
  READLN(k);
  FOR i := k TO N-1 DO
    v[i] := v[i+1];
  N := N-1;
  ...
END.
```

3.3.3.10. Inserção dum novo elemento no vector

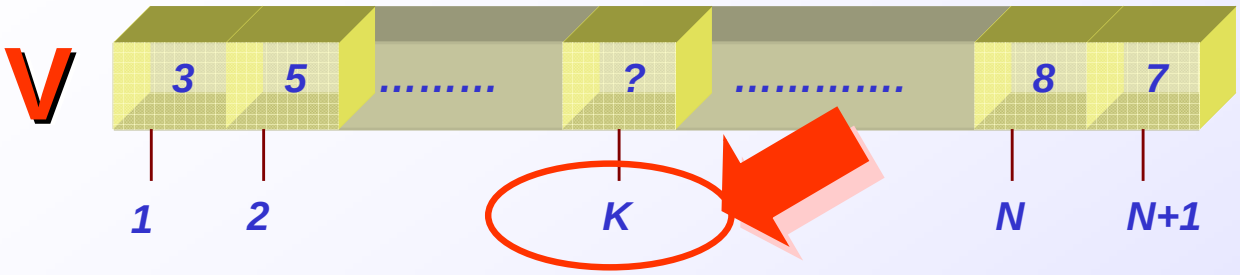


Descrição Algorítmica → **Codificação em Pascal**

```
...
Escrever("Qual o índice do elemento a inserir?")
Ler ( k )
De índice ← N + 1 Até k Faz (Salto –1)
    v( índice ) ← v( índice – 1 )
Fim De
N ← N+1
...
```

```
PROGRAM Inserir_Elemento_vector;
VAR
  v       : ARRAY [1 .. 100] OF INTEGER;
  i, k, N : INTEGER;

BEGIN
  ...
  WRITELN('Qual o índice do elemento a inserir? ');
  READLN(k);
  N := N+1;
  FOR i := N DOWNTO k DO
    v[i] := v[i-1];
  v[k] := 0; { Elemento livre }
  ...
END.
```

3.3.3.11. Pesquisa dum elemento dum vector



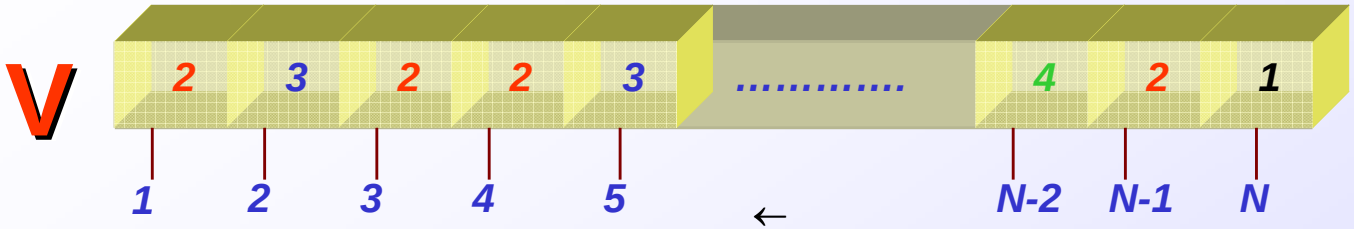
Descrição Algorítmica

```
...
índice ← 1
Achou ← Falso
Enquanto (Achou = Falso) E (índice <= N) Faz
    Se  v(índice) = valor  Então
        Achou ← Verdadeiro
    Fim Se
    índice ← índice +1
Fim Enquanto
Se  (Achou = Verdadeiro)  Então
    Escrever("Está na posição ", índice -1)
Senão
    Escrever("Não está no vector")
Fim Se
...
```

Codificação em Pascal

```
PROGRAM Procurar_Elemento_vector;
VAR
  v            : ARRAY [1 .. 100] OF INTEGER;
  i, N, Valor  : INTEGER;
  Achou        : BOOLEAN;

BEGIN
  ...
  WRITELN('Qual o valor a procurar? ');
  READLN(valor);
  i := 1;
  Achou := FALSE;
  WHILE (Achou = FALSE) AND (i <= N) DO
    BEGIN
      IF v[i] = valor THEN
         Achou := TRUE;
      i := i + 1;
    END;
  IF  Achou = TRUE THEN
    WRITELN('Está na posição ', i-1)
  ELSE
    WRITELN('Não está no vector');
END.
```

3.3.3.12. Número de elementos repetidos num vector



Ex.: 2 e 3 repetido,
2 elementos repetidos.

Descrição Algorítmica

```
...
NRepetidos ← 0
De índice ← 1 Até N -1 Faz
   j ← índice
   Repetir
      j ← j + 1
      Se v( índice ) = v( j ) Então
         Repetição ← Falso
         De k ← 1 Até i - 1 Faz
            Se v( i ) = v( k ) Então
               Repetição ← Verdadeiro
            Fim Se
         Fim De
         Se Repetição = Falso Então
            NRepetidos ← NRepetidos + 1
         Fim Se
      Fim Se
   Até (v( índice ) = v( j )) OU ( j = N )
Fim De
...
```

Codificação em Pascal

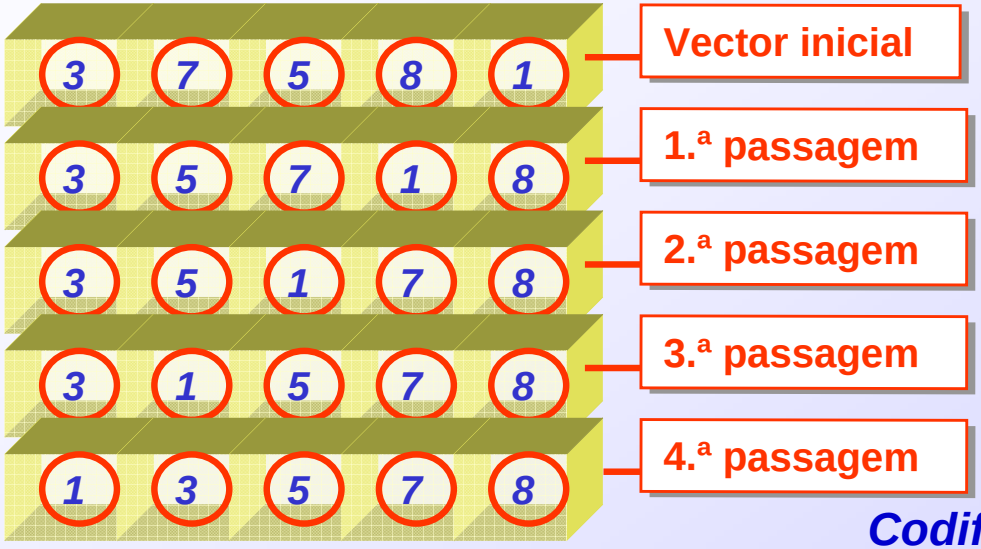
```
PROGRAM Procurar_Elementos_Repetidos_vector;
VAR
  v : ARRAY [1 .. 100] OF INTEGER;
  i, j, k, N, NRepetidos : INTEGER;
  Repeticao : BOOLEAN;
BEGIN
  ...
  NRepetidos := 0;
  FOR i := 1 TO N-1 DO
    BEGIN
      j := i;
      REPEAT
        j := j + 1;
        IF v[i] = v[j] THEN
          BEGIN
            Repetição := FALSE;
            FOR k := 1 TO i-1 DO
              IF v[i] = v[k] THEN
                Repeticao := TRUE;
            IF Repeticao := FALSE THEN
              NRepetidos := NRepetidos + 1;
          END;
      UNTIL (v[i] = v[j]) OR (j = N)
    END;
  WRITELN('Número de elem. repetidos:', NRepetidos);
END.
```

Procura elementos iguais para índices superiores a i

Procura nos índices inferiores a i se esse elemento já foi contado

3.3.3.13. Ordenação dum vector: Bubble Sort

V



Para a ordenação deste vector de 5 elementos foram necessárias 4 passagens. Por maioria de razão, **se o vector tivesse N elementos seriam necessárias N-1 passagens**.

Nota: a troca de valores entre 2 variáveis a e b:

```
Aux = a
a = b
b = Aux
```

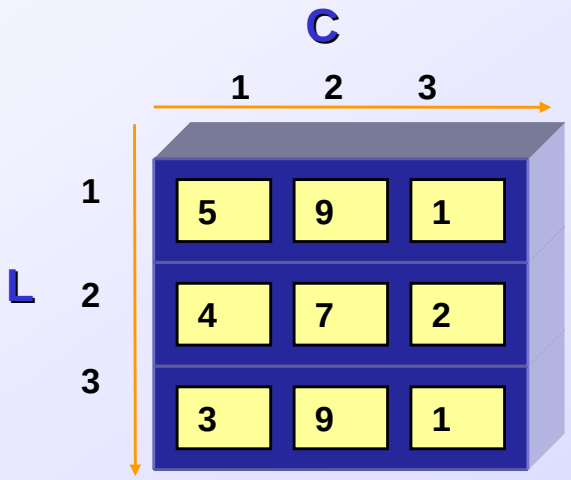
Codificação em Pascal

```
PROGRAM Ordenacao_Bubble_vector;
VAR
  v : ARRAY [1 .. 100] OF INTEGER;
  N, i, limite, Aux : INTEGER;

BEGIN
  ...
  FOR limite := N-1 DOWNTO 1 DO
    FOR i := 1 TO limite DO
      IF v[i] > v[i+1] THEN
        BEGIN
          Aux  := v[i];
          v[i] := v[i+1];
          v[i+1] := Aux;
        END;
  ...
END.
```

Ordenação por ordem crescente

3.3.4. Matrizes: *Array* Bidimensional

Matriz

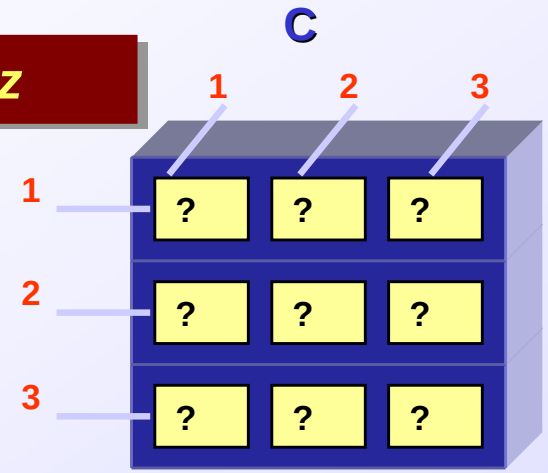



```
M : ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
{ Array bidimensional c/ 3 x 3 células, com valores inteiros }
```

3.3.5. Operações básicas com matrizes

- Inserir valores numa matriz
- Ler valores duma matriz
- Traço duma matriz
- Transposta duma matriz
- Adição/Subtracção de matrizes
- Multiplicação de matrizes

3.3.5.1. Inserir valores numa matriz



Matriz

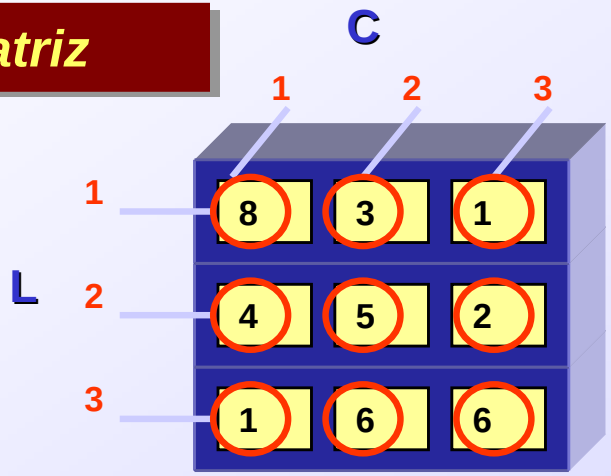
Descrição Algorítmica → *Codificação em Pascal*

```
…
De L ← 1 Até Tot_L Faz
  De C ← 1 Até Tot_C Faz
    Escrever(“Insira o elemento”, L, ”,”, C, ”da matriz”)
    Ler ( Matriz( L , C ) )
  Fim De
Fim De
…
```

```
PROGRAM Inserir_valores_Matriz;
CONST
  Tot_l = 3;
  Tot_c = 3;
VAR
  Matriz : ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
  l, c   : INTEGER;

BEGIN
  FOR l := 1 TO Tot_l DO
    FOR c := 1 TO Tot_c DO
      BEGIN
        WRITE('Insira o elemento ',l,',',c,': ');
        READLN(Matriz[l, c]);
      END;
END.
```

3.3.5.2. Escrever valores duma matriz

Matriz



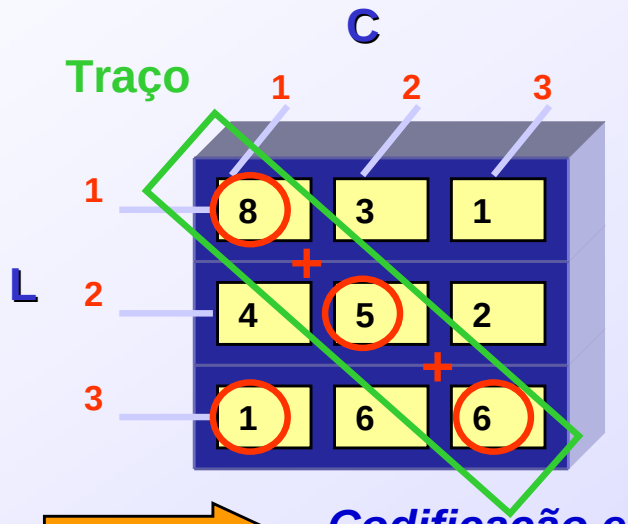
Descrição Algorítmica → *Codificação em Pascal*

```
...
De L ← 1 Até Tot_L Faz
 De C ← 1 Até Tot_C Faz
   Escrever ("Matriz(", L , "," , C, ")=", Matriz(L, C))
 Fim De
Fim De
...
```

```
PROGRAM Escrever_valores_Matriz;
CONST
  Tot_l = 3;
  Tot_c = 3;
VAR
  Matriz : ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
  l, c   : INTEGER;

BEGIN
  ...
  FOR l := 1 TO Tot_l DO
    FOR c := 1 TO Tot_c DO
      WRITELN('Matriz[',l,',',c,']= ',Matriz[l, c]);
  ...
END.
```

3.3.5.3. Traço duma matriz

Matriz



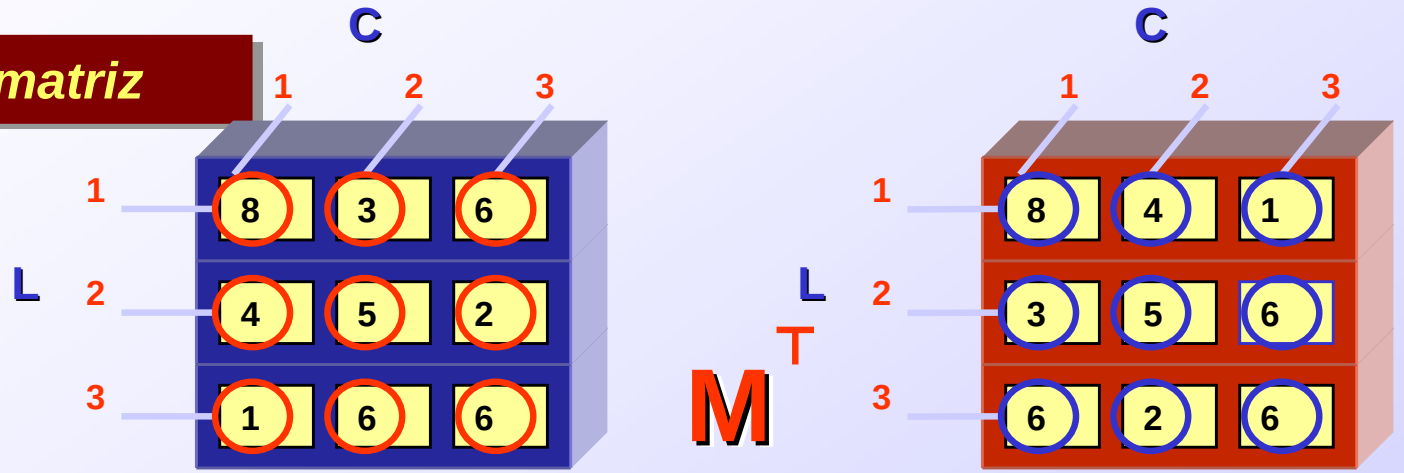
Descrição Algorítmica → *Codificação em Pascal*

```
...
Se Tot_L = Tot_C Então
    Traco ← 0
    De índice ← 1 Até Tot_L Faz
        Traco ← Traco + Matriz(índice, índice)
    Fim De
    Escrever("O traço da matriz é: ", Traco)
Senão
    Escrever("Erro: A matriz não é válida")
Fim Se
...
```

```
PROGRAM Traco_Matriz;
CONST
  Tot_l = 3;
  Tot_c = 3;
VAR
  Matriz   : ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
  i, Traco : INTEGER;
BEGIN
  ...
  IF Tot_l = Tot_c THEN
    BEGIN
      Traco := 0;
      FOR i := 1 TO Tot_l DO
        Traco := Traco + Matriz[i, i];
      WRITELN('O traço da matriz é: ', Traco);
    END
  ELSE
    WRITELN('Erro: A matriz não é válida');
END.
```

3.3.5.4. Transposta duma matriz

$$M(l,c) = M^{T}(c,l)$$



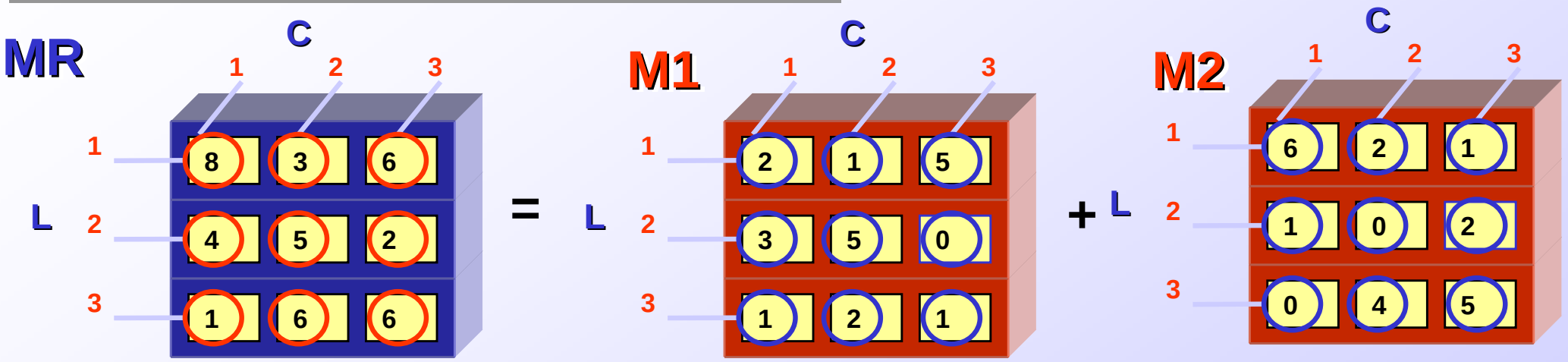
Descrição Algorítmica → *Codificação em Pascal*

```
...
De L ← 1 Até Tot_L Faz
   De C ← 1 Até Tot_C Faz
      M_T( L, C ) ← M( C, L )
   Fim De
Fim De
...
```

```
PROGRAM Transpor_Matriz;
CONST
  Tot_l = 3;
  Tot_c = 3;
VAR
  Mat, Mat_T : ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
  l, c       : INTEGER;

BEGIN
  ...
  FOR l := 1 TO Tot_l DO
    FOR c := 1 TO Tot_c DO
      Mat_T[l, c] := Mat[c, l];
  ...
END.
```

3.3.5.5. Adição/Subtracção de Matrizes

$$MR(l,c) = M1(l,c) \pm M2(l,c)$$



Descrição Algorítmica

```
...
De L ← 1 Até Tot_L Faz
    De C ← 1 Até Tot_C Faz
        MR(L, C) ← M1(L, C) + M2(L, C)
    Fim De
Fim De
...
```

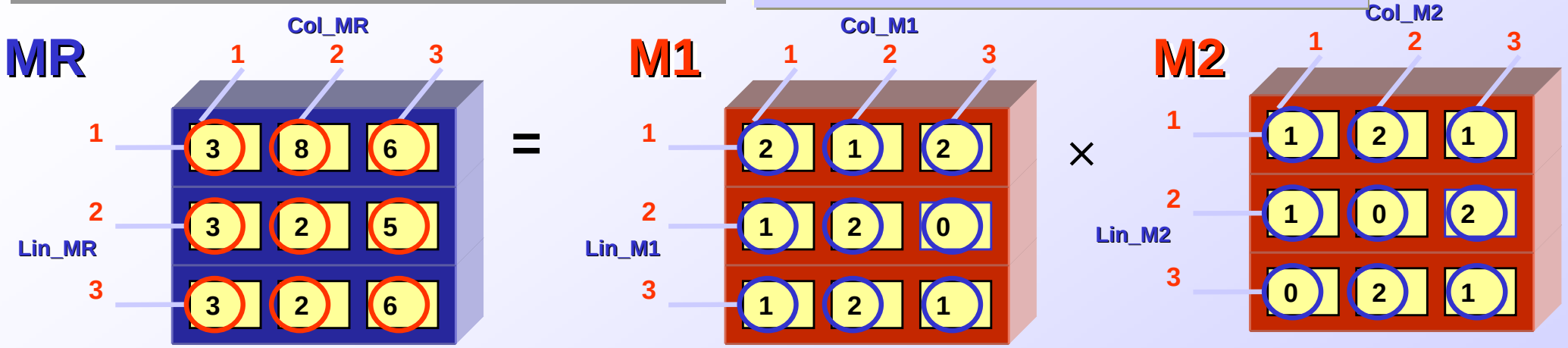
Codificação em Pascal

```
PROGRAM Soma_Matriz;
CONST
  Tot_l = 3;
  Tot_c = 3;
VAR
  MR, M1, M2: ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
  l, c      : INTEGER;
BEGIN
  ...
  FOR l := 1 TO Tot_l DO
    FOR c := 1 TO Tot_c DO
      MR[l, c] := M1[l, c] + M2[l, c];
  ...
END.
```

No caso da subtracção basta trocar o sinal para "–".

3.3.5.6. Multiplicação de Matrizes

$$MR_{i,j} = \sum_{k=1}^{Col\_MR} \left(M1_{i,k} \times M2_{k,j} \right)$$



Descrição Algorítmica

```
...
Se Col_M1 = Lin_M2 Então
 Lin_MR ← Lin_M1
 Col_MR ← Col_M2
 De i ← 1 Até Lin_MR Faz
  De j ← 1 Até Col_MR Faz
   MR( i, j ) ← 0
   De k ← 1 Até Col_M1 Faz
      MR( i, j ) ← MR( i, j ) + M1( i, k) x M2( k , j )
   Fim De
  Fim De
 Fim De
Fim Se
...
```

Codificação em Pascal

```
PROGRAM Multiplica_Matriz;
CONST
  Lin_M1 = 3;  Col_M1 = 3;  Lin_M2 = 3;  Col_M2 = 3;
VAR
  MR, M1, M2: ARRAY [1 .. 3, 1 .. 3] OF INTEGER;
  i, j, k, Lin_MR, Col_MR : INTEGER;
BEGIN
  ...
  IF Col_M1 = Lin_M2 THEN
   BEGIN
     Lin_MR := Lin_M1;  Col_MR := Col_M2;
     FOR i := 1 TO Lin_MR DO
      FOR j := 1 TO Col_MR DO
       BEGIN
         MR[i, j] := 0;
         FOR k := 1 TO Col_M1 DO
           MR[i, j] := MR[i, j] + M1[i, k] * M2[k, j];
       END;
   END;
END.
```

3.3.6. Utilização de Strings: *Array* unidimensional

O tipo de dados ***String*** é um caso especial de *Array* unidimensional do tipo *Char*, como tal, pode ser utilizado com as funções e operadores específicos para Strings ou, como array, utilizando as técnicas descritas anteriormente.

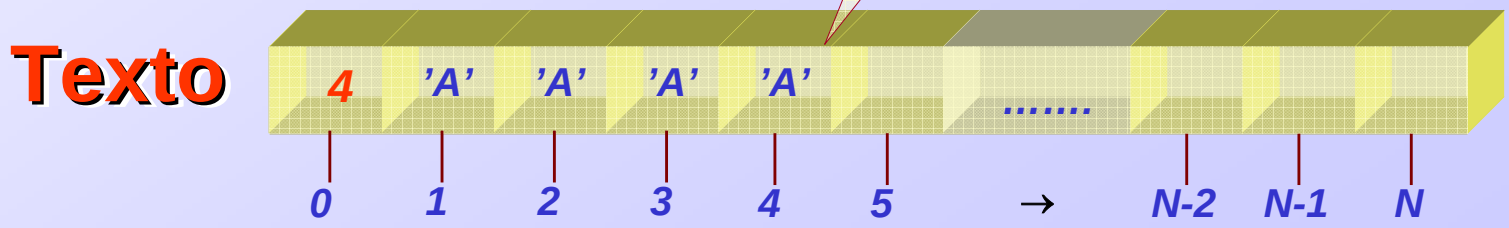
3.3.6.1. Definição de um variável tipo String

```
VAR
   Texto1: STRING;          {String de 255 caracteres}
   Texto2: STRING[n];       {String de n caracteres (Máximo:255)}
   ...
   Texto3: ARRAY [0..n] OF CHAR; {Array de n caracteres}
```

Embora Texto2 e Texto3 sejam formalmente iguais em termos de declaração, o tipo String permite funcionalidades suplementares relativamente ao tipo Array, ou seja: atribuições, operadores e funções específicas.
Ex.: Texto2 : = 'ABCD';

O tipo String reserva a posição zero do Array para guardar o tamanho do Texto. Esse valor é colocado automaticamente quan-do utilizado como string. Ex:

```
Texto : = 'AAAA';  { como String }

FOR i : = 1 TO 4 DO  {como Array}
    Texto[ i ] : = 'A';
Texto[ 0 ] : = #4;  { como Array é necessário definir o nº de carac-teres do texto utilizando o # }
```



Operadores para cadeias de caracteres (*Strings*)

Operador	Significado	Exemplo
+	Concatenação	'bom' + 'dia' vale 'bom dia'
<, <=, >, >=, =, <>	Comparação	'bom' <> 'dia' vale **True**

Tome atenção: Cadeias de caracteres ou *strings* são identificadas pelo caracter " ' " no inicio e final dum conjunto de caracteres. O operador soma junta duas cadeias de caracteres mas não faz a soma no caso de esses caracteres representarem valores numéricos, conforme ilustra o exemplo:

'20' + '30' vale **'2030'** e não **'50'**

Funções e procedimentos para cadeias de caracteres (*Strings*)

Funções	Designação	Exemplo de utilização
Copy (Texto, Índice, Número)	Copia uma *sub-string* de **Texto** começando em **Índice** e com **Número** caracteres.	Txt:=Copy ('ABCDEF', 2, 3); { Txt ← 'BCD' }
Concat (S1 [, S2, ..., Sn])	Junta duas ou mais *strings* numa só (eq. +).	Txt:=Concat('est','-ipv');{Txt←'est-ipv'}
Delete (Texto, Índice, Número)	Apaga uma *sub-string* de **Texto** começando em **Índice** e com **Número** caracteres.	Txt:='estv-ipv'; Delete(Txt, 6, 2); { Txt ← 'estv-v' }
Insert (SubTexto, Texto, Índice)	Insere a *string* **SubTexto** na *string* **Texto** começando em **Índice**.	Txt:='ipv'; Txt1:='-estv-'; Insert (Txt1, Txt, 2); {Txt ← 'i-estv-pv'}
Length (Texto)	Devolve o número de caracteres de **Texto**.	Txt:='ESTV'; a:=Length(Txt); { a ← 4 }
Pos (SubTexto, Texto)	Procura a posição de **SubTexto** em **Texto**.	a := Pos ('T', 'ESTV'); { a ← 3 }

3.4. Estruturas de dados definidos pelo utilizador

3.4.1. Definição de novos tipos de dados

No PASCAL, além dos tipos de dados apresentados numa das secções anteriores (Integer, Real, etc.), é possível ao utilizador definir novos tipos de dados com base nos existentes, desde que a definição seja realizada na secção ***TYPE*** do programa.

Sintaxe de definição

```
TYPE
  Nome_do_novo_tipo = Tipo_já_existente;
  ...
```

Sintaxe de utilização

```
VAR
  Nome_var1[,...] : Nome_do_novo_tipo;
  ...
```

Depois da definição na secção Type, é possivel utilizar o novo tipo de dados da mesma forma como se utiliza dado do tipo inteiro, real, etc.

Exemplos

```
TYPE
  Matriz = ARRAY[1 .. 3,1 .. 3] OF INTEGER;
  Vector_Real = ARRAY[1 .. 100] OF REAL;
  Mat100x100 = ARRAY[1..100] OF Vector_Real;
  Inteiro = INTEGER;
  Texto   = STRING[50];

VAR
  Mat1, Mat2 : Matriz;
  Var_Vector : Vector_Real;
  Armazem    : Mat100x100;
  var1, var2 : Inteiro; { ou INTEGER }
  i, j       : INTEGER; { ou Inteiro }
  nome, Morada, Localidade : Texto;

BEGIN
  ...
END.
```

3.4.2. Conjuntos enumerados e subconjuntos de ordinais

O PASCAL permite definir os seus próprios conjuntos de dados do tipo ***ordinal***, por enumeração dos seus elementos, bem como subconjuntos, com base nos conjuntos de ordinais já existentes.

- ***Conjunto Enumerado***: é o conjunto em que se indicam todos os elementos que o constituem.

- ***SubConjunto***: consiste em definir um intervalo, com base num conjunto de ordinais, no qual as variáveis assim definidas só podem assumir esses valores.

Sintaxe de definição

```
TYPE
  Nome_Tipo_Conjunto =(nome1,..., nomeN);
VAR
  Nome_Var_Conjunto : (nome1,..., nomeN);
```

Exemplos:

```
TYPE
  Cores  = (Vermelho, Verde, Azul, Amarelo);
  Cartas = (Copas, Ouros, Paus, Espadas);
  Meses  = (Jan, Fev, Mar, Abr, Mai, Jun,
            Jul, Ago, Set, Out, Nov, Dez);
VAR
  Cor   : Cores;
  Semana:(Seg, Ter, Qua, Qui, Sex, Sab, Dom);
  Meses_Ano : Meses;
BEGIN
  Cor := Verde;
  Cor := Preto;
  Cor := Copas;
END.
```

Erro: Identificador Desconhecido

Erro: Tipo Incompatível

Sintaxe de definição

Const_1 < Const_2

```
TYPE
  Nome_Tipo_SubConjunto = Const_1 .. Const_2;
VAR
  Nome_Var_SubConjunto : Const_1 .. Const_2;
```

Exemplos:

```
TYPE
  Intervalo = 1..100;
  Letras    = 'Z'..'A';
  Numeros   = '0'..'9';
VAR
  Vector : ARRAY [Intervalo] OF Numeros;
  Nota   : 0..20;
BEGIN
  Vector[10] := '5';
  Vector[0]  := '5';
  Nota := 21;
END.
```

Erro: Limite inferior maior que limite superior

Erro: Constante fora do intervalo

3.4.3 Definição de estruturas de dados (*Records*)

Os **registos** ou ***Records*** são um outro tipo de dados estruturados que permitem agrupar dados de **vários tipos diferentes**, sob a forma de **campos**. A definição de um *Record* pode ser feita como variável ou como tipo.

Sintaxe de definição

```
TYPE
  Nome_do_Tipo = RECORD {como tipo}
    Nome_do_Campo_1[,...] : Tipo_1;
    Nome_do_Campo_2[,...] : Tipo_2;
    ...
    Nome_do_Campo_N[,...] : Tipo_N;
  END;
VAR
  Nome_Variável_1 : Nome_do_Tipo;
  {ou em alternativa: como variável}
  Nome_Variável_2 : RECORD
    Nome_do_Campo[,...] : Tipo;
    ...
  END;
```

Sintaxe de utilização

```
BEGIN
  ...
  Nome_Variável_1 . Nome_do_Campo_1 ...;
  Nome_Variável_1 . Nome_do_Campo_2 ...;
  ...
  Nome_Variável_2 . Nome_do_Campo ...;
  ...
END;
```

*Para aceder aos campos duma variável do tipo RECORD, indica-se o **nome dessa variável** seguido por um **ponto** e pelo **nome do campo**.*

Algumas considerações:

- Na sintaxe de definição, os **Tipo_1**, **Tipo_2**, … e **Tipo_N** podem ser um tipo de dados do PASCAL, ou um outro tipo anteriormente também definido pelo utilizador.

Alguns exemplos

Exemplo 1 Criação dum Tipo de Dados, responsável por guardar um ponto no espaço 3D (X,Y,Z)

```
PROGRAM Coordenadas;
TYPE
  Ponto3D = RECORD
      x, y, z : REAL;
  END;

VAR
  Ponto : Ponto3D;

BEGIN
  ...
  WRITE('Indique o valor da coordenada x: ');
  READLN(Ponto.x);
  WRITE('Indique o valor da coordenada y: ');
  READLN(Ponto.y);
  WRITE('Indique o valor da coordenada z: ');
  READLN(Ponto.z);
  WRITELN('Coordenadas do ponto: A=(',Ponto.x,',',Ponto.y,',',Ponto.z,')');
  ...
END.
```

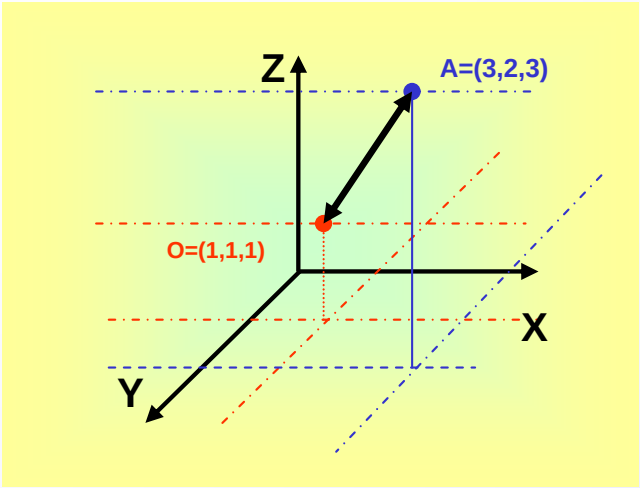
Definição de um novo tipo de dados chamado Ponto3D que associa num registo as variáveis x, y e z.

Definição de uma variável do tipo Ponto3D. A variável Ponto terá associada a si os campos x, y e z.

A utilização de variáveis do tipo Record requer sempre: identificador da variável - ponto - identificador do campo.

Exemplo 2:

Criação dum novo Tipo para guardar os dados dum segmento de recta: ponto de origem, ponto extremo, cor e tipo de linha.



Segmento OA

O = (1,1,1)
A = (3,2,3)
cor = verde
traço = 2

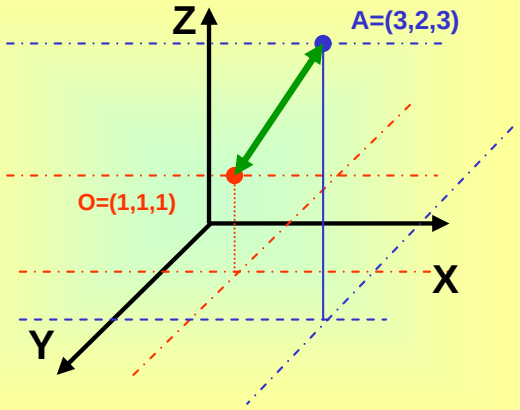
```
PROGRAM Segmento_Recta;
TYPE
  Ponto3D = RECORD
     x, y, z : REAL;
  END;
  Segmento = RECORD
     Origem, Extremidade : Ponto3D;
     cor : STRING;
     tipo_linha : INTEGER;
  END;

VAR
  SegOA : Segmento;

BEGIN
  ...
  WRITE('Ponto O, coordenada x: '); READLN(SegOA.Origem.x);
  WRITE('Ponto O, coordenada x: '); READLN(SegOA.Origem.y);
  WRITE('Ponto O, coordenada x: '); READLN(SegOA.Origem.z);
  WRITE('Ponto A, coordenada x: '); READLN(SegOA.Extremidade.x);
  WRITE('Ponto A, coordenada x: '); READLN(SegOA.Extremidade.y);
  WRITE('Ponto A, coordenada x: '); READLN(SegOA.Extremidade.z);
  WRITE('Indique a cor de OA:');    READLN(SegOA.Cor);
  WRITE('Indique o tipo de linha (1, 2, 3, 4):');
  READLN(SegOA.tipo_linha);
...
END.
```

Ponto O

Ponto A

Exemplo 3:

Exemplo idêntico ao anterior mas utilizando um *array* do tipo Ponto3D para guardar as coordenadas dos pontos.



Segmento OA

O = (1,1,1)
A = (3,2,3)
cor = verde
traço = 2

```
PROGRAM Segmento_Recta;
TYPE
  Ponto3D = RECORD
     x, y, z : REAL;
  END;
  Segmento = RECORD
     Extremos   : ARRAY [1 .. 2] OF Ponto3D;
     cor        : STRING;
     tipo_linha : INTEGER;
  END;

VAR
  SegOA : Segmento;

BEGIN
  ...
  WRITE('Ponto O, coordenada x: '); READLN(SegOA.Extremos[1].x);
  WRITE('Ponto O, coordenada x: '); READLN(SegOA.Extremos[1].y);
  WRITE('Ponto O, coordenada x: '); READLN(SegOA.Extremos[1].z);
  WRITE('Ponto A, coordenada x: '); READLN(SegOA.Extremos[2].x);
  WRITE('Ponto A, coordenada x: '); READLN(SegOA.Extremos[2].y);
  WRITE('Ponto A, coordenada x: '); READLN(SegOA.Extremos[2].z);
  WRITE('Indique a cor de OA:');    READLN(SegOA.Cor);
  WRITE('Indique o tipo de linha (1, 2, 3, 4):');
  READLN(SegOA.tipo_linha);
...
END.
```

Ponto O

Ponto A

Algumas considerações:

- Podemos também declarar *arrays* de tipos definidos pelo utilizador.

```
TYPE
  Segmento_A : RECORD ... END; {Ver exemplos}
  Segmento_B : RECORD ... END;
VAR
  Alguns_Segmentos : ARRAY[1 .. 5] OF Segmento_A;
  Mais_Segmentos   : ARRAY[1 .. 10] OF Segmento_B;
  ...
       Alguns_Segmentos[3].cor := 'vermelho';
       Mais_Segmentos[6].extremo[1].x := 2;
  ...
```

- Podem-se efectuar atribuições entre variáveis (ou campos) de tipos de dados definidos pelo utilizador desde que sejam do **mesmo tipo,** encarregando-se o PASCAL de fazer a cópia dos valores campo a campo.

```
VAR
   ponto_O, ponto_A : Ponto3D;
   SegOA : Segmento_A;
...
   ponto_O := ponto_A;
...
   SegOA.origem := ponto_A;
```

Esta instrução é equivalente às instruções:

```
ponto_O.x := ponto_A.x;
ponto_O.y := ponto_A.y;
ponto_O.z := ponto_A.z;
```

Não é possível fazer atribuições de variáveis definidas em sítios diferentes (equivale a tipos diferentes) mesmo que sejam iguais.

```
VAR
  ponto1 : RECORD
     x, y, z : INTEGER;
  END;
  ponto2 : RECORD
     x, y, z : INTEGER;
  END;
  ...
     ponto1 := ponto2;
  ...
```

3.4.4. WITH ... DO ... ;

```
VAR
    Variável : RECORD
       Campo_1 : Tipo;
       Campo_2 : Tipo;
       ...
       Campo_N : Tipo;
    END;
```

Sintaxe

```
WITH Variável DO {Tipo Record}
   Campo_1 ... ;
```

Ou

```
WITH Variável DO {Tipo Record}
  BEGIN
     Campo_1 ...;
     Campo_2 ...;
     ...
     Campo_N ...;
  END;
```

Exemplo de utilização

Exemplo

Visualização dos elementos (Nome, Morada e Número) duma lista telefónica.

```
PROGRAM Gestão_Lista_Telefonica;
TYPE
  Registo_Telefone = RECORD
     Nome, Morada : STRING;
     Numero       : LONGINT;
   END;
VAR
   Lista_Telefone : ARRAY [1 .. 100] OF Registo_Telefone;
   i, N           : INTEGER;

BEGIN
   ... { Escrever a lista telefónica no ecrã }
   FOR i := 1 TO N DO
      WRITELN(Lista_Telefone[i].Nome,' ',
              Lista_Telefone[i].Morada,' ',
              Lista_Telefone[i].Numero);
   { ou em alternativa }
   FOR i := 1 TO N DO
      WITH Lista_Telefone[i] DO
         WRITELN(Nome,' ', Morada,' ', Numero);
   ...
END.
```

3.5. Conversões entre tipos de dados

Tal como vimos na secção *3.2.2.2*, o tipo de dados que uma variável pode conter está condicionado pelo tipo com que esta foi declarada. Porém, existe a possibilidade de converter valores dum determinado tipo de dados para um outro tipo. As conversões podem ser de dois tipos:

- **Conversões Implícitas (automáticas)**
- **Conversões Explícitas**

3.5.1. Conversões Implícitas (automáticas)

As ***conversões Implícitas*** são as que o PASCAL pode levar a cabo ***automaticamente*** quando atribuímos a uma variável dum tipo de dados, um valor dum outro tipo de dados. Entre as várias conversões, salientam-se as conversões entre inteiros e inteiros, reais e reais, inteiros para reais e de caracter para string.

3.5.1.1. Conversões Inteiro → Inteiro

Integer, LongInt, ShortInt, Word, Byte → **Integer, LongInt, ShortInt, Word, Byte**

```
VAR
   it : Integer; lg : LongInt;
BEGIN
   lg:= 4;
   it:= lg;     {it passa a valer 4}
   lg:= 65600;
   it:= lg;     {it passa a valer 64!}
END.
```

Nota: a conversão implícita entre **inteiros** de tipos diferentes é sempre possível, ou seja, não ocorre nenhum erro de compilação.

No entanto, nem sempre é válida. Quando o valor a atribuir é maior que o valor máximo permitido pelo tipo da variável, ocorre um erro de ***Overflow*** que não é detectado.

3.5.1.2. Conversões Real → Real

Real, Single, Double, Extend, Comp → **Real, Single, Double, Extend, Comp**

```
VAR
   re : Real; db : Double;
BEGIN
   db:= 4.0;
   re:= db;     {re passa a valer 4.0}
   db:= 1e300;
   re:= db;     {Runtime Error 205}
END.
```

Nota: a conversão implícita entre reais de tipo diferentes só é possível quando o valor a atribuir está dentro dos limites aceite pelo tipo de variável.

Quando é atribuído um valor que exceda os limites da variável, ocorre um erro de execução com o número 205 e que corresponde a ***Floating Point Overflow***. A execução do programa é imediatamente terminada.

3.5.1.3. Conversões Inteiro → Real

Integer, LongInt, ShortInt, Word, Byte → **Real, Single, Double, Extend, Comp**

```
VAR
  it : Integer; re : Real;
BEGIN
  it:= 4;
  re:= it;     {re passa a valer 4.0}
  re:= 1.0;
  it:= re;     { Compiler Error 26 }
END.
```

Nota: a conversão implícita entre **inteiros** e **reais** é sempre possível e válida porque o conjunto dos números reais engloba todos os números inteiros, qualquer que seja o tipo desta variável.

O inverso já não se verifica, ou seja, uma conversão implícita de um número **real** para **inteiro** nunca é possível, gerando o *erro 26* de compilação que significa *variáveis incompatíveis* (*Error 26: Type Mismatch*).

3.5.1.4. Conversões Caracter → String

Char, Array [..] Of Char → **String**

```
VAR
  ch  : Char; st : String;
  arr : Array [1..n] of Char;
BEGIN
  ch:= 'a';
  st:= ch; {st vale 'a' com tamanho 1}
  arr[1]:= 'a'; arr[2]:= 'b';
  st:= arr;{st vale 'ab' com tamanho n}
END.
```

Nota: a conversão implícita de ***Char*** para ***String*** é sempre possível e válida. Já a conversão entre um ***Array*** **de Caracteres** e uma ***String*** só é possível quando o *array* tem tamanho inferior a 256 caracteres.

A conversão de *String* para *Char*, de *String* para um *Array* de Caracteres e de um *Array* de Caracteres e com tamanho superior a 255 para uma *String* não é permitida e dá origem ao erro de compilação número 26.

3.5.2. Conversões Explícitas (funções de conversão)

No PASCAL existem funções e procedimentos de conversão criadas explicitamente para permitirem a conversão de valores entre os diferentes tipos de dados (Inteiros, Reais, Caracteres e *String*).

Tabela de funções de conversão

Nome da Função	Conversão De	Para
Chr(...)	Inteiro	Caracter
Ord(...)	Caracter	Inteiro
Round(...)	Real	Inteiro Longo
Trunc(...)	Real	Inteiro Longo
Str(...)	Real ou Inteiro	String numérica
Val(...)	String numérica	Real ou Inteiro

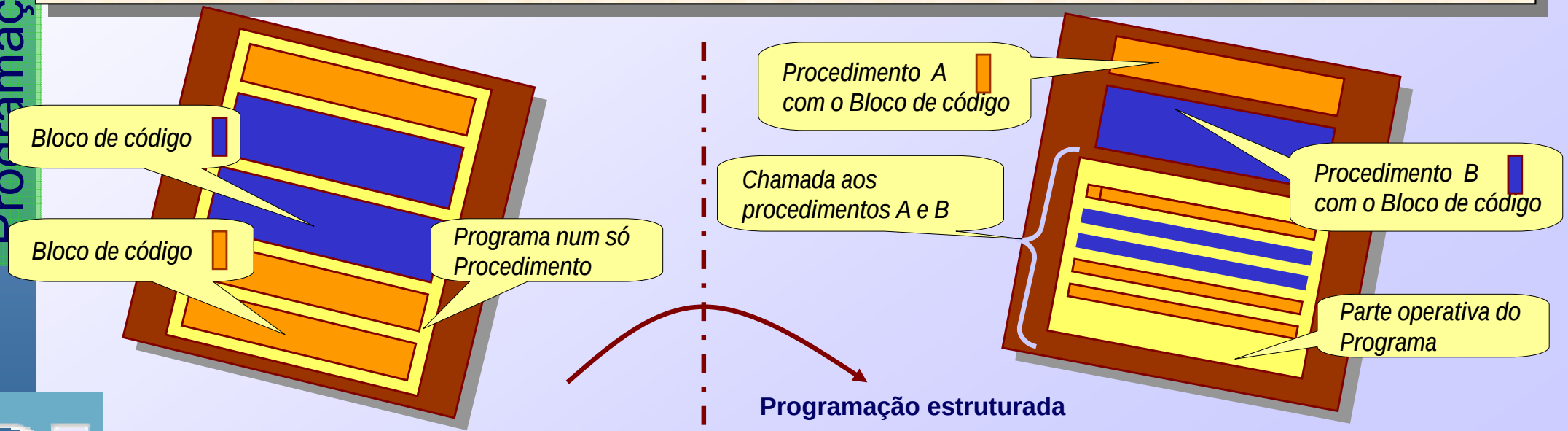
Exemplos

```
VAR
  it, c : Integer;
  st    : String;
  Re    : Real;
  ch    : Char;
BEGIN
  ch:=Chr(97);   { ch ← 'a' }
  it:=Ord('a'); { it  ← 97 }

  it:=Round(1.8); { it ← 2 }
  it:=Trunc(1.8); { it ← 1 }

  Str(1.8:5:2,st);{ ' 1.80' }
  Val('10',it,c); { it ← 10 }
  Val('1.8',re,c);{ re ← 1.8}
END.
```

3.6. Programação estruturada

3.6.1. Estrutura dum programa

Num programa em PASCAL, um conjunto de instruções codificadas podem estar "*concentradas*" apenas num procedimento ou ser distribuídas por vários procedimentos relacionados entre si. A esta distribuição das instruções codificadas por vários procedimentos (**`Procedure`** … **`End;`** e **`Function`** … **`End;`**, correspondendo à divisão sucessiva da tarefa principal em tarefas mais elementares e à qual se dá o nome de **programação estruturada**.



Algumas considerações em relação à programação estruturada:

- Existe a parte operativa de um programa a que podemos chamar como "programa principal", e os restantes procedimentos definidos na parte declarativa. Os procedimentos podem também estar fora da parte declarativa, em ficheiros separados a que no Pascal se chamam ***UNIT***;

- O "programa principal", além de executar as instruções próprias, chama também os outros procedimento, os quais executam outras instruções;

- Cada um dos outros procedimentos, pode chamar outros procedimentos;

- Para que haja comunicação entre os procedimentos é necessário que recebam valores ou variáveis, na forma de argumentos (ou parâmetros de entrada) e/ou devolvam valores quando terminam a respectiva execução.

Exemplo:

Programação não estruturada

```
PROGRAM Adicao_Subtraccao;
VAR
  x1, x2, res_a, res_b : INTEGER;

BEGIN
  WRITE('Indique x1: ');
  READLN(x1);
  WRITE('Indique x2: ');
  READLN(x2);

  { Adição }
  res_a := x1 + x2;
  WRITELN('A adição é ', res_a);

  { Subtracção }
  res_b := x1 - x2;
  WRITELN('A subtracção é ', res_b);
END.
```

Passagem de parâmetros

Declaração dos argumentos

Programação estruturada

```
PROGRAM Adicao_Subtraccao;
VAR
  x1, x2 : INTEGER;

  PROCEDURE Adicao(x, y : INTEGER)
  VAR
    res : INTEGER;
  BEGIN
    res := x + y;
    WRITELN('A adição é ', res);
  END;

  PROCEDURE Subtraccao(x, y : INTEGER)
  VAR
    res : INTEGER;
  BEGIN
    res := x - y;
    WRITELN('A subtracção é ', res);
  END;

BEGIN
  WRITE('Indique x1: ');
  READLN(x1);
  WRITE('Indique x2: ');
  READLN(x2);
  { Adição }
  Adicao(x1, x2);
  { Subtracção }
  Subtraccao(x1, x2);
END.
```

3.6.2. Definição e chamada de Procedimentos

Definição - trata-se da codificação das instruções a executar quando o procedimento for executado (i.e.: quando for chamado).

Chamada - trata-se do pedido para que um dado procedimento seja executado a pedido de outro, e para que o controlo de execução passe para as instruções codificadas, que constituem a definição do procedimento. A chamada pode envolver a passagem de argumentos e eventualmente a devolução de um valor (no caso de uma **`Function`**).

Algumas considerações:

- Um procedimento tem apenas uma definição, mas pode ser chamado (por outros) tantas vezes quantas as necessárias.

- Sempre que se chamar um determinado procedimento, tem que se lhe passar todos os argumentos (ou parâmetros de entrada) de que este estiver à espera (i.e.: os argumentos declarados na definição). Estes argumentos são os dados de entrada do procedimento.

3.6.3. Procedimentos

Procedimentos ou funções são conjuntos de instruções que desempenham uma tarefa específica, ao qual é atribuído um determinado nome. A sua execução acontece quando é referenciada em qualquer ponto do programa. Um procedimento ou função tem as mesmas secções que um programa.

3.6.3.1. Estrutura dum procedimento

Cabeçalho do Procedimento
Procedure ou Function, Nome e Argumentos

Declarações do Procedimento
Constantes e Variáveis, etc.

Parte operativa de Procedimento
Bloco de instruções codificadas

Fim do Procedimento
End *seguido de ponto e virgula*

```
{ Procedimentos: PROCEDURE e FUNCTION }
{ Cabeçalho da função }
FUNCTION CalculoArea(Raio : REAL) : REAL;
{ Declaração de variáveis e constantes }
CONST
  ValorPi = 3.14;

VAR
  Area : REAL;
{ Parte Operativa do procedimento }
BEGIN
  Area := ValorPi * Raio * Raio;
  CalculoArea := Area;
{ Fim do procedimento CalculoArea }
END;
```

3.6.3.2. Tipos de procedimentos

Existem dois tipos de procedimentos: **PROCEDURE** e **FUNCTION**. A diferença entre estes dois tipos de procedimentos reside no facto de **FUNCTION** devolver um valor a um outro procedimento que o tiver invocado, como uma atribuição a uma variável, o que não acontece com **PROCEDURE**.

Procedure ... Begin ... End;

```
PROCEDURE nome_do_procedimento [(lista_de_argumentos)];
        Dimensionamento das constantes e variáveis do procedimento;
        Definição de procedimentos;
BEGIN
        Bloco de instruções codificadas;
END;
```

Function ... Begin ... End;

```
FUNCTION nome_da_função [(lista_de_argumentos)]: tipo_do_valor_a_devolver
        Dimensionamento das constantes e variáveis do procedimento;
        Definição de procedimentos;
BEGIN
        Bloco de instruções codificadas;
        nome_da_função := expressão_do_valor_a_devolver;
END;
```

Exemplo

```
PROGRAM CalculoFuncao;

  FUNCTION F_Partes(x : REAL) : DOUBLE;
  BEGIN
    IF x < 20 THEN
      F_Partes := 6 * sqr(sin(x))
    ELSE
      IF (x >= 20) AND (x < 50) THEN
        F_Partes := 3 * cos(x) + 2 * sqr(sin(x))
      ELSE
        F_Partes := sqr(cos(x) + 2);
  END;

  PROCEDURE Rep_Valores;
  VAR
     i          : INTEGER;
     f_valores : ARRAY [1..10] OF DOUBLE;
  BEGIN
    FOR i := 1 TO 10 DO
      BEGIN
        f_valores[i] := F_Partes(i*3);
        WRITELN('f[', i, '] = ', f_valores[i]:0:3);
      END;
  END;

BEGIN
  Rep_Valores;
END.
```

Definição da função F_Partes

Atribuição duma expressão a F_Partes

Definição do procedimento Rep_Valores

Chamada de F_Partes dentro de Rep_Valores

Chamada de Rep_Valores no programa

Nota: Para que uma função possa devolver um determinado valor, é necessário que uma das suas instruções faça a atribuição desse valor ao identificador da função.

3.6.4. Argumentos

No que diz respeito aos **argumentos** temos que considerar a forma como estes são declarados na definição dum dado procedimento, e como estes serão fornecidos em cada uma das chamadas ao respectivo procedimento.

Sintaxe de declaração dos argumentos:

```
Arg1 [,Arg2, … ,ArgN] : Tipo_Arg_1aN;...
```

Exemplo:

Procedimento que recebe dois valores do tipo inteiro e calcula a soma e a subtracção

Os argumentos declarados passam a funcionar dentro do procedimento como variáveis.

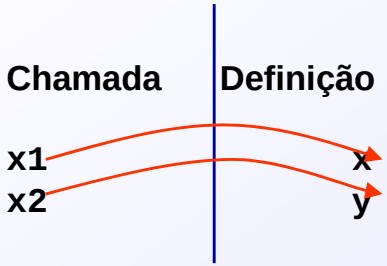
*Este procedimento (**Adi_Sub**) não pode ser executado só por si, pois funciona apenas quando lhe forem atribuídos dois argumentos, tendo por isso, ao ser chamado, que lhe sejam atribuídos os dois valores.*

```
PROGRAM Adicao_Subtraccao;
VAR
  x1, x2 : INTEGER;

  { Definição do procedimento Adi_Sub }
  PROCEDURE Adi_sub(x, y : INTEGER);
  VAR
    res_adi, res_sub : INTEGER;
  BEGIN
    res_adi := x + y;
    res_sub := x - y;
    WRITELN('A adição é ', res_adi);
    WRITELN('A subtracção é ', res_sub);
  END;

{ Parte operativa do programa }
BEGIN
  WRITE('Indique x1: ');
  READLN(x1);
  WRITE('Indique x2: ');
  READLN(x2);
  { Chamando Adi_Sub }
  Adi_Sub(x1, x2);
END.
```

Chamada

Na chamada a um procedimento, indicamos o seu nome seguido da lista de valores correspondentes a cada um dos argumentos, separados por vírgulas.

Exemplo



Não existe, nenhuma relação directa entre o nome dos argumentos do procedimento chamado e o nome dos argumentos usados na sua chamada.

```
PROGRAM Adicao_Subtraccao;
VAR
  x1, x2 : INTEGER;
  { Definição do procedimento Adi_Sub }
  PROCEDURE Adi_sub(x, y : INTEGER);
  VAR
    res_adi, res_sub : INTEGER;
  BEGIN
    res_adi := x + y;
    res_sub := x - y;
    WRITELN('A adição é ', res_adi);
    WRITELN('A subtracção é ', res_sub);
  END;
{ Parte operativa do programa }
BEGIN
  WRITE('Indique x1: ');
  READLN(x1);
  WRITE('Indique x2: ');
  READLN(x2);
  { Chamando Adi_Sub }
  Adi_Sub(x1, x2);
  Adi_Sub(x1+x2, x1-x2);
  Adi_Sub(5, 2*(x1+x2));
  Adi_Sub(5, 6);
  Adi_Sub(6, 5);
END.
```

Podemos também apresentar, além de variáveis, expressões (desde que o seu tipo seja compatível com o tipo de argumento do procedimento chamado).

Deve-se ter em atenção a ordem pela qual os argumentos são passados. Neste caso os resultados serão naturalmente diferentes.

3.6.5. Passagem de argumentos

Quando se passam argumentos a um procedimento, o PASCAL vai adequar cada um desses aos que se encontram descritos na definição do procedimento, respeitando a ordem pela qual os argumentos são apresentados: o primeiro argumento declarado na definição do procedimento toma o valor do primeiro argumento apresentado na chamada; o segundo vai tomar o valor do segundo; e assim sucessivamente.

Exemplo

Definição do procedimento

```
PROCEDURE Proc_Exemplo(v : REAL; Msg_1 : STRING; Msg_2 : STRING)
BEGIN
      {Instrução ou bloco de instruções}
END;
```

Chamada dentro de outro procedimento

```
PROGRAM Prog_Principal;
VAR
  str : STRING;
BEGIN
  str := 'Boa Tarde';
  Proc_Exemplo(2.7, str, 'Boa Noite');
  {Outra Instrução ou bloco de instruções}
END
```

3.6.5.1. Argumentos passados por valor ou por referência

No que diz respeito aos argumentos passados na chamada dum procedimento existem duas possibilidades: os argumentos da chamada são alteráveis de dentro dum procedimento ou os argumentos de chamada não são alteráveis dentro do procedimento.

- Argumentos alterados dentro dum procedimento (por referência)

Nesta situação os argumentos têm de ser passados por referência, o que implica a passagem duma referência (posição na memória) para o argumento original. As alterações realizadas nessa referência irão ter reflexo no argumento referido, ou seja, na variável exterior ao procedimento que serviu de entrada de dados na altura em que este foi chamado.

NOTA: Argumentos do tipo *String*, *Array* e *Record* são obrigatoriamente passados por referência.

- Argumentos não alterados dentro dum procedimento (por valor)

Neste caso os argumentos são passados por valor, correspondendo à uma cópia do valor do argumento apresentado na chamada do respectivo procedimento, para o argumento declarado na definição do procedimento. Qualquer alteração estará confinada à cópia efectuada, e nunca ao argumento original.

Sintaxes

Referência

- Quando o argumento deve ser passado por referência, usa-se a palavra reservada **VAR** a anteceder a declaração do argumento.

Valor

- Quando o argumento deve ser passado por valor, não se usa nenhuma palavra a anteceder a declaração do argumento.

Exemplo

```
VAR
  A, B : INTEGER;
PROCEDURE Proc_Exemplo_Val_Ref(VAR x:INTEGER;  y:INTEGER)
  BEGIN
    x:=10;
    y:=4;
  END;
BEGIN
   A:=0; B:=2;
   Proc_Exemplo_Val_Ref(A,B);
END.
```

Após a execução de ***Proc_Exemplo_Val_Ref****:*

- ***A*** *tem o valor* ***10***
- ***B*** *tem o valor* ***2***

Nota: Por defeito o PASCAL efectua a passagem de argumentos por ***Valor*** (quando não se indica **VAR**), logo, as variáveis passadas na chamada a um procedimento nunca serão alteradas por ele.

3.6.6. Programação modular: as UNITS

Para os programas mais complexos, a utilização de Funções e Procedimentos como estratégia para reduzir a complexidade do problema em pequenas partes não é suficiente porque, estes tendem a crescer muito em termos de linhas de código o que os torna difíceis de gerir. Surge assim a necessidade de dividir o programa por diversos ficheiros distintos, a que se chamam **módulos**.

Ao processo de divisão de um programa por diversos módulos chama-se ***Programação Modular*** e a cada um desses ficheiros ou módulos têm, no TURBO PASCAL, o nome de ***UNITS***.

Vantagens:

- Acrescenta mais um nível de estruturação à programação permitindo concentrar numa *Unit* todas as funções e procedimentos que estão logicamente relacionadas.
- Reduz a complexidade de um programa dividindo-o por várias *Units*.
- Facilita a gestão e desenvolvimento do programa porque se utilizam ficheiros mais pequenos.
- Permite criar livrarias de funções e procedimentos genéricas de forma a que possam ser reutilizados em diversos programas distintos.
- Permite a utilização de equipas de programadores no desenvolvimento de um programa.

3.6.6.1. Utilização de Units no Turbo Pascal

Para utilizar as Funções e procedimentos de uma *Unit* é necessário primeiro informar o compilador através da palavra reservada ***USES*** que deve estar imediatamente a seguir ao cabeçalho do programa.

Sintaxe: ***Uses*** **<Nome da Unit 1>, ... , <Nome da Unit N>;**

O TURBO PASCAL inclui seis *Units* que têm como objectivo fornecer à linguagem as funcionalidades necessárias para interagir com o hardware e o sistema operativo. As *Units* fornecidas são:

- **System**: *Unit* fundamental para o funcionamento do Turbo Pascal já que comporta as funções e procedimentos básicos da linguagem, como tal é incluída automaticamente em qualquer programa. Exemplos: ***Read***, ***Write***, ***Chr***, ***Ord***, ***Abs***, ***Exp***, ***Sqr***, ***Sqrt***, ***Val***, ***Assign***, ***Append***, etc.
- **Crt**: possibilita o controlo do teclado, som, cores e ecrã em modo texto. Exemplos: ***ClrScr***, ***ReadKey***, ***KeyPressed***, ***GotoXY***, ***WhereX***, ***WhereY***, ***Sound***, ***TextColor***, ***TextMode***, etc.
- **Printer**: declara a variável de ficheiro *LST* e associa-a com a impressora pelo que, não é necessário realizar as operações usuais com ficheiros. Exemplo: ***WriteLn(LST, 'Teste');***
- **Dos**: permite aceder às funcionalidades do DOS, tais como: manuseamento de ficheiros; data e hora; disco; etc. Exemplos: ***GetDate***, ***GetTime***, ***SetTime***, ***FSerach***, ***DiskFree***, ***DiskSize***, etc.
- **Graph**: acrescenta funcionalidades para trabalhar em modo gráfico. Exemplos: ***InitGraph***, ***CloseGraph***, ***SetGraphMode***, ***Arc***, ***Bar***, ***Circle***, ***Line***, ***Rectangle***, ***PutImage***, ***GetImage***, etc.
- **Overlay**: permite separar um programa executável em partes de modo a não o carregar todo em memória de uma só vez, deixando assim mais espaço para os dados.

3.6.6.2. Criar uma nova Unit

Para criar um novo módulo (*Unit*) é necessário primeiro criar um novo ficheiro com o mesmo nome que está no cabeçalho da *Unit*. O Turbo Pascal automaticamente procura o ficheiro e o integra no programa.

Cabeçalho: *Distinge a Unit de um programa, contem o nome pela qual será conhecida.*

Interface: *Zona onde se definem todas as variáveis, constantes, tipos de dados, procedimentos e funções que são visíveis pelos outros programas ou Units.*

Implementação: *As constantes, Variáveis, tipos de dados, funções e procedimentos aqui definidos só são acessíveis dentro da própria UNIT. As funções e procedimentos declarados na zona de Interface devem ser aqui definidas.*

Inicializações: *Instruções que serão executadas antes de qualquer instrução do programa principal.*

```
UNIT <Nome da Unit>;

INTERFACE
   USES <lista de Units importadas>;
   <definição de variáveis, constantes e tipos exportados>;
   <cabeçalho das funções e procedimentos exportados>;

IMPLEMENTATION
   USES <lista de Units importadas privadas ao módulo>;
   <definição de variáveis, constantes e tipos internos>;
   <funções e procedimentos internos à Unit>;
   <corpo das funções e procedimentos exportados>;

[ BEGIN ]
   <Comandos a serem executados na inicialização da Unit>;
END.
```

Se não existirem inicializações, a palavra reservada BEGIN pode ser omitida.

Exemplo duma Unit

Construção duma UNIT que contem:

- *procedimento para colocar uma String numa determinada linha e coluna do ecrã do computador;*
- *um Array que possa ser utilizado dentro da Unit e fora dela.*

UNIT

```
UNIT Teste;
INTERFACE   { Parte visível do exterior da Unit }
    VAR   ArrayDaUnit : ARRAY [1 .. 10] OF BYTE;
    PROCEDURE  EscreveString ( linha, coluna : BYTE;  texto :  STRING );
IMPLEMENTATION
    USES CRT;            { Unit de utilização interna }
    VAR  i : INTEGER;    { Variável interna à Unit e utilizada na inicialização}
    PROCEDURE EscreveString( linha, coluna :  BYTE;  texto :  STRING );
    BEGIN                { definição do corpo do procedimento EscreveString }
        GOTOXY( coluna, linha );                    { Necessita da Unit CRT }
        WRITE( texto  );
    END;
BEGIN
    FOR  i := 1  TO  10  DO
        ArrayDaUnit[ i ] := i;   { inicialização do array com valores de 1 a 10 }
END.
```

Exemplo do programa que utiliza o procedimento EscreveString e a variável ArrayDaUnit definidos na Unit acima.

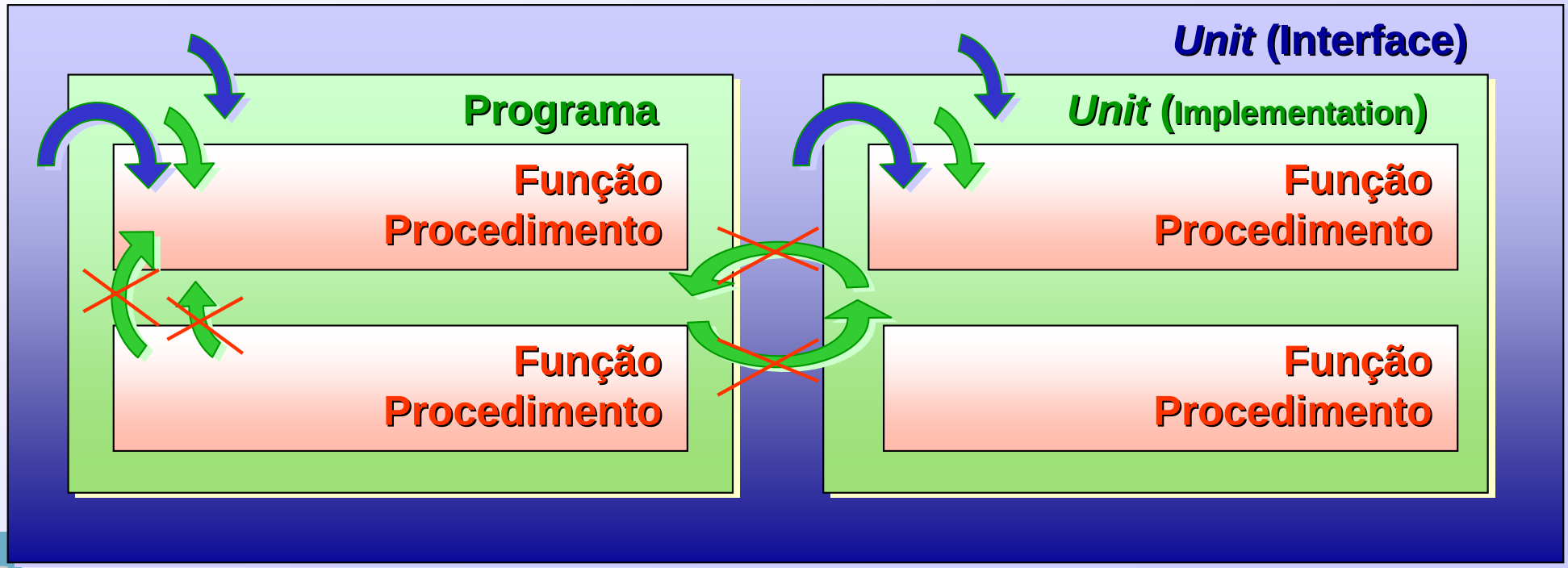
Programa Principal

```
PROGRAM TesteDaUnit;
USES  CRT, Teste;
VAR     i : BYTE;
BEGIN
    CLRSCR;                                          { Necessita da Unit CRT }
    EscreveString( 10, 10, 'Teste da Unit' );   { Escreve a String em (10,10) }
    FOR  i := 1  TO  10  DO
        WRITE( ArrayDaUnit[ i ], ' ' );          { Escreve os valores de 1 a 10 }
END.
```

3.7. Tópicos avançados

3.7.1. Âmbito das variáveis

Entende-se por âmbito (ou visibilidade) de uma variável, o local onde esta é reconhecida. Em PASCAL podem-se identificar três âmbitos distintos:



3.7.1.1. Nível do Procedimento ou Função

As variáveis declaradas dentro dum procedimento ou função têm como âmbito a parte declarativa e operativa do próprio procedimento ou função, sendo por isso apenas reconhecidas dentro dele. A estas variáveis é normal atribuir-se a designação de **variáveis locais** (do procedimento ou função).

3.7.1.2. Nível do Programa

Para que as variáveis tenham um âmbito ao nível do módulo do programa principal, a sua declaração deve realizar-se na parte declarativa deste. As variáveis com este âmbito são visíveis em todo o módulo do programa e em todos os procedimentos e funções definidos dentro nele. A estas variáveis atribui-se normalmente a designação de **variáveis globais** (ao módulo do programa).

3.7.1.3. Nível da Unit (Interface)

Numa Unit, as variáveis declaradas na parte de *Interface* são visíveis em todo o módulo tal como as variáveis globais, mas além disso, como podem ser utilizadas fora da *Unit* são designadas por **variáveis exportadas**. Quando se inclui uma *Unit* num programa através do *Uses*, essas variáveis passam a designar-se de **variáveis externas**.

3.7.1.4. Sobreposição de Variáveis com o mesmo nome

- O PASCAL permite que coexistam variáveis com o mesmo nome desde que não se encontrem dentro do mesmo nível, ou seja: dentro do mesmo procedimento, do mesmo programa ou da mesma *Unit*. Embora estejam ao mesmo nível, entre duas ou mais *Units* também podem existir variáveis com o mesmo nome.

- Sempre que existam sobreposições de nome de variáveis declaradas em níveis diferentes, o PASCAL utiliza a que encontrar primeiro, segundo a seguinte ordem de procura:

1º - Parte declarativa ou cabeçalho do Procedimento ou Função.

2º - Parte declarativa do módulo a que pertence (Programa principal ou *UNIT*).

3º - Parte de Interface das *Units* importadas para o módulo, contando a ultima variável encontrada segundo a ordem das *Units* estabelecida a seguir à palavra *Uses*.

Para evitar a dificuldade em determinar qual a variável que está a ser utilizada em cada momento e onde é que ela já foi utilizada anteriormente, o chamado ***efeito colateral***, devem-se seguir as seguintes ***Regras de Boa Programação:***

- ✓ Reduzir ao mínimo necessário o número de variáveis globais e externas.
- ✓ Quando é necessário passar a informação para dentro do procedimento ou função, devem-se utilizar sempre os parâmetros de entrada.
- ✓ Quando é necessário retirar a informação do interior do procedimento, devem-se utilizar os parâmetros passados por referência. Se for só um, utiliza-se a função.
- ✓ Se o programa se estender por várias *Units*, é preferível criar uma *Unit* especificamente para a declaração de constantes, tipos de dados e varáveis comuns a todas elas.
- ✓ Para se transportar informação entre módulos (programa principal e *Units*), devem-se utilizar os parâmetros de entrada e saída das funções e procedimentos em vez de variáveis externas.

3.7.1.5. Exemplo

Módulo

```
Program AmbitoDasVariaveis;
Uses Externas;{ Var. externa y }
Var x : Integer;
...
Procedure Proc1;
Var u : Integer;
    ...
End;
Procedure Proc2(z:Integer);
Var v : Integer;
    ...
End;

Procedure Proc3;
Var x : Integer;
  ...
End;

Procedure Proc4(x:Integer);
Begin
  y := 1;
  ...
End;
...
```

*A variável **y**, como variável externa, é visível em todo o módulo, sendo por isso também visível em **Proc1**, **Proc2**, **Proc3**, **Proc4** e parte operativa do programa, podendo ser lá utilizada.*

*As variável **x** é visível em todo o módulo, ou seja, em todos os procedimento e funções e também em toda a parte operativa do programa principal.*

*A variável **u** é visível apenas dentro do procedimento **Proc1**, não podendo por exemplo ser invocada no **Proc2**.*

*A variáveis **z** e **v** são visíveis apenas dentro do procedimento **Proc2**.*

*Dentro do procedimento **Proc3**, quando se refere à variável **x** estamos a falar desta variável e não da variável global com o mesmo nome.*

*Dentro do procedimento **Proc4**, quando falarmos de **x**, estamos a referir-nos ao argumento **x**. Neste caso, não se pode criar uma variável **x** dentro deste procedimento porque entraria em conflito com o argumento **x** (ambos com o mesmo âmbito).*

*Dentro do procedimento **Proc4**, quando falarmos de **y**, estamos a referir-nos à variável externa **y** porque não existem nem variáveis locais nem globais com o mesmo nome.*

3.7.2. Tempo de vida das variáveis

O tempo de vida das variáveis está relacionado com o âmbito a que estas se referem.

3.7.2.1. Variável declarada ao nível dum dado Programa ou Unit

Nesta situação, a variável está sempre activa, pelo que, se o seu valor sofrer alterações (devido à execução das instruções de algum procedimento desse programa) essas serão sempre preservadas. As alterações só são perdidas quanto é terminada a execução do programa.

3.7.2.2. Variável declarada ao nível dum dado Procedimento ou Função

Qualquer variável nesta situação é criada quando começa a execução do procedimento e é eliminada quando este finaliza. Se o procedimento for chamado várias vezes, uma variável é criada e eliminada de cada vez que o procedimento é executado.

Por esta razão não é possível guardar um valor numa variável declarada dentro dum procedimento para ser utilizado numa futura execução desse procedimento. Nesse caso terá que se utilizar uma variável global.

3.7.2.3. Exemplo

Módulo

```
Program Exemplo;
USES externa; { Declara a variável K }
VAR
   j : Integer;
...
Procedure TempoDeDuraçãoDaVariavel;
VAR
   i : Integer;
BEGIN
   i:=0;
   i:=i+1;
   j:=j+1;
   k:=k+1;
   WriteLn('O valor de i é ', i);
   WriteLn('O valor de j é ', j);
   WriteLn('O valor de k é ', k);
END;
...
```

Âmbito da Unit (Interface)

Âmbito do Programa

Âmbito do Procedimento

Uma vez que tem o âmbito do procedimento, sempre que se executar este procedimento, o valor de **i** *será sempre* **1**.

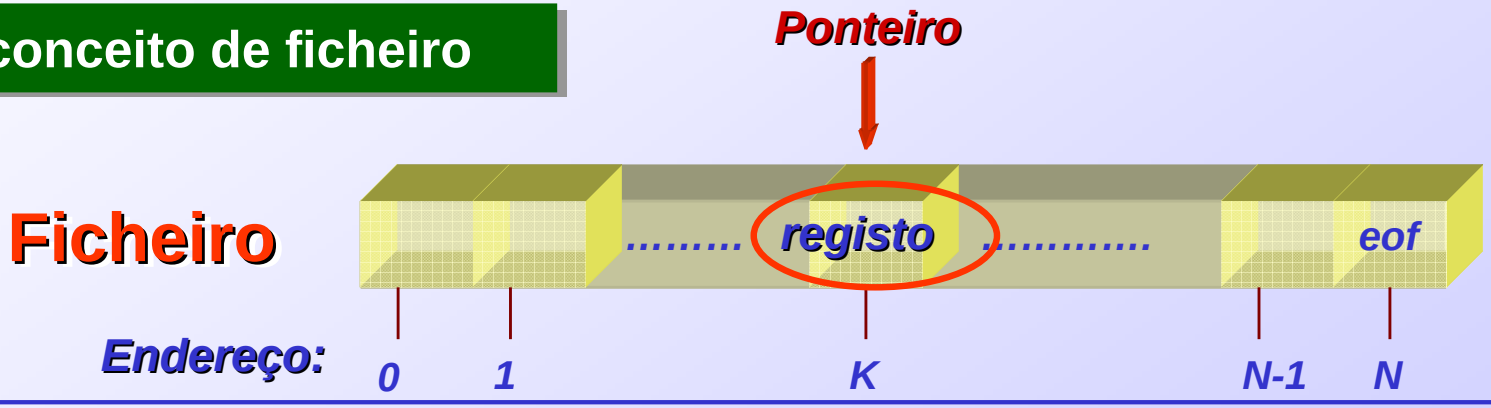
Uma vez que tem o âmbito do programa, sempre que se executar este procedimento, o valor de **j** *será incrementado em* **1** *ao valor que contém.*

Uma vez que **k** *é uma variável externa declarada na Unit , sempre que executarmos o procedimento, o valor de* **k** *será incrementado em* **1** *ao valor que contém.*

3.8. Trabalhar com ficheiros no PASCAL

A necessidade de utilizar ficheiros surge porque ao desligar o computador, os dados contidos na memória *RAM* são perdidos (memória volátil) e por isso é necessário utilizar o armazenamento em disco (memória não volátil) para guardar a informação de uma forma permanente. Como os discos estão organizados em ficheiros e pastas, é necessário utilizar as funcionalidades de criar, ler, escrever e apagar ficheiros dentro de um programa em Pascal.

3.8.1. O conceito de ficheiro



Um Ficheiro é muito semelhante a um *array* unidimensional embora com algumas particularidades:

- Cada elemento do ficheiro é um registo. Um ficheiro é formado por um conjunto de registos.
- O primeiro elemento do ficheiro é o ***0*** (zero) e o último é ***N-1***, de um ficheiro com *N* registos.
- O elemento *N* é a marca de fim de ficheiro (***eof*** – End Of File).
- Um ficheiro tem sempre associado um ***ponteiro*** que indica qual o registo a ser lido ou escrito.

3.8.2. Ficheiros de acesso aleatório ou binário

Um Ficheiro do tipo acesso aleatório ou binário ***(FILE OF)*** é caracterizado conter dados simples ou estruturados em formato binário, ou seja, não legíveis nem manipuláveis fora do programa que o criou. O acesso aos dados pode ser feito de forma **aleatória**, ou seja, é possível aceder aos dados em qualquer posição do ficheiro sem ter que o percorrer desde o início.

3.8.2.1. Definição de variáveis

```
VAR
    <variável de ficheiro>  : FILE OF <Tipo do registo>;
    <variável de memória>   : <Tipo do registo>;
```

Considerações:

- A ***<variável de ficheiro>*** é utilizada nas funções e procedimentos que trabalham com ficheiros para identificar qual o ficheiro a utilizar.
- O ***<Tipo do registo>*** identifica que dados é que estão contidos em cada um dos registos do ficheiro e deve conter apenas um tipo de dados: ***Integer***, ***Real***, ***String***, ***Byte***, etc. Quando é necessário guardar num ficheiro mais que um tipo de dados, deve-se definir primeiro um tipo ***Record***.
- A ***<variável de memória>*** é uma variável auxiliar de ficheiro, deve ser do mesmo tipo dos registos do ficheiro e é utilizada para nela colocar a informação em operações de leitura e escrita de ficheiros.

3.8.2.2. Funções e Procedimentos para trabalhar com ficheiros binários

Funções e Procedimentos	Descrição sumária	Exemplo de utilização
Assign (VarFicheiro, Nome_em_disco)	Associa uma ***<variável de ficheiro>*** ao **nome do ficheiro** em disco.	**ASSIGN** (Ficheiro, 'Dados.dat'); { Ficheiro ⇔ 'Dados.dat' }
Rewrite (VarFicheiro)	Cria e **abre**, caso não exista, um ficheiro vazio para Leitura/Escrita. Se o ficheiro já existir, todos os dados nele existentes **serão apagados**.	**REWRITE** (Ficheiro); { Ficheiro aberto e vazio para leitura/escrita }
Reset (VarFicheiro)	**Abre** para Leitura e/ou Escrita um ficheiro já existente. Se não existir, ocorrerá um erro de execução. O **ponteiro** do ficheiro fica no inicio.	**RESET** (Ficheiro); { Ficheiro aberto para leitura/escrita }
Close (VarFicheiro)	**Fecha** um ficheiro que tenha sido aberto com *Reset* ou *Rewrite*.	**CLOSE** (Ficheiro); { Fecha o ficheiro }
Write (VarFicheiro, VarMemória)	**Escreve** o conteúdo da variável de memória para um registo do ficheiro.	**WRITE** (Ficheiro, Buffer); { 'Dados.dat' ← Buffer }
Read (VarFicheiro, VarMemória)	**Lê** o conteúdo do registo do ficheiro para a variável de memória.	**READ** (Ficheiro, Buffer); { 'Dados.dat' → Buffer }
VarBooleana := **EOF** (VarFicheiro)	Devolve o valor ***True*** se atingir o fim do ficheiro ou ***False***, caso contrário.	WHILE NOT **EOF**(Ficheiro) DO **...** { Repete até encontrar o final }

Continuação:

Funções e Procedimentos	Descrição sumária	Exemplo de utilização
VarLongInt := **FilePos** (VarFicheiro)	Devolve o valor actual da **posição do ponteiro** dentro do ficheiro.	Posicao := **FILEPOS** (Ficheiro); { Posicao ← Posição do ponteiro }
VarLongInt := **FileSize** (VarFicheiro)	Devolve o **número total** de registos existentes no ficheiro.	NRegistos := **FILESIZE** (Ficheiro); { NRegistos ←Nº registos do Ficheiro }
Seek (VarFicheiro, Nº_Registo)	**Posiciona** o ponteiro do ficheiro no registro indicado.	**SEEK** (Ficheiro, 0); { Coloca o ponteiro na posição 0 }
Truncate (VarFicheiro)	**Elimina** todos os registos posteriores à posição corrente do ponteiro.	**TRUNCATE** (Ficheiro); { Corta o fich. a partir da pos. actual }
Erase (VarFicheiro)	**Elimina** do disco o ficheiro indicado (o ficheiro não pode estar aberto).	**ERASE** (Ficheiro); { Apaga *Ficheiro* do disco }
Rename (VarFicheiro, NovoNome)	Muda o nome ao ficheiro no disco (o ficheiro não pode estar aberto).	**RENAME** (Ficheiro, 'NovoNome.txt'); { Muda o nome do *Ficheiro* no disco }
VarString := **FSearch** (NomeFicheiro, DirectórioAprocurar)	Procura um ficheiro no directório indicado. Se existir, *VarString* contem o Caminho + NomeFicheiro, caso contrário, devolve uma *string* vazia.	IF **Fsearch**('Dados.dat', '.') < >'' THEN **...** { Ficheiro Dados.dat encontrado no directório actual }
{$I+}** , **{$I-} e **IOResult**	A **directiva I** activa (+) ou desactiva (-) a ocorrência de *RunTime Error* nas operações com ficheiros. A função **IOResult** indica qual o código de erro resultante das operações realizadas com ficheiros.	**{$I-}** { Confirma se o ficheiro já existe } Reset (Ficheiro); Close (Ficheiro); **{$I+}** IF **IOResult** = 0 THEN WRITELN ('Ficheiro existente);

3.8.3. Ficheiro de texto

Um Ficheiro do tipo Texto ***(TEXT)*** é caracterizado por conter informação que é totalmente armazenada no formato de caracteres (ASCII), podendo ser criados ou alterados em qualquer editor de texto. O acesso aos dados é do tipo **sequencial**, ou seja, para aceder a determinado ponto do ficheiro é necessário percorrê-lo desde o início.

3.8.3.1. Definição de variáveis

```
VAR
    <variável de ficheiro>  : TEXT;
    <variável de memória>   : Char, String, Integer, Real, etc.;
```

Considerações:

- A ***<variável de ficheiro>*** é utilizada nas funções e procedimentos que trabalham com ficheiros para distinguir qual a utilizar.
- A palavra reservada ***Text*** define que o ficheiro só pode conter sequências de caracteres legíveis separados por marcas de fim de linha (#13, #10) e terminando com a marca de fim de ficheiro.
- A ***<variável de memória>*** é uma variável auxiliar de ficheiro utilizada para nela colocar a informação em operações de leitura e escrita. Esta variável pode ser do tipo *char* ou *string* (leitura directa) ou *Integer*, *Real*, etc. (necessitando de conversão).

3.8.3.2. Funções e Procedimentos para trabalhar com ficheiros de texto

Funções e Procedimentos	Descrição sumária	Exemplo de utilização
Assign (VarFicheiro, Nome_em_disco)	Associa uma ***<variável de ficheiro>*** ao nome do **ficheiro de texto**.	**ASSIGN** (Ficheiro, 'Texto.txt'); { Ficheiro ⇔ 'Texto.txt' }
Rewrite (VarFicheiro)	Cria e **abre**, caso não exista, um ficheiro vazio **só** para Escrita. Se o ficheiro já existir, todos os dados nele existentes **serão apagados**.	**REWRITE** (Ficheiro); { Ficheiro aberto e vazio para escrita}
Reset (VarFicheiro)	**Abre** um ficheiro já existente **só** para Leitura. Se não existir, ocorrerá um erro de execução. O **ponteiro** do ficheiro fica no inicio.	**RESET** (Ficheiro); { Ficheiro aberto só para leitura }
Append (VarFicheiro)	**Abre** um ficheiro já existente **só** para Escrita. O **ponteiro** do ficheiro fica no final, logo, só se pode acrescentar dados.	**APPEND** (Ficheiro); { Ficheiro aberto só para adicionar dados no final }
Close (VarFicheiro)	**Fecha** um ficheiro previamente aberto com *Reset*, *Rewrite* ou *Append*.	**CLOSE** (Ficheiro); { Fecha o ficheiro }
Write (VarFicheiro, VarMemória)	**Escreve** o conteúdo da variável de memória para um ficheiro de texto.	**WRITE** (Ficheiro, Buffer); { 'Dados.dat' ← Buffer }
Writeln (VarFicheiro, VarMemória)	**Escreve** o conteúdo da variável de memória para um ficheiro de texto e acrescenta a marca de fim de linha.	**WRITELN** (Ficheiro, Buffer); { 'Dados.dat' ← Buffer + Nova linha (Nova linha = CHR(13) + CHR(10)) }

Continuação:

Funções e Procedimentos	Descrição sumária	Exemplo de utilização
Read (VarFicheiro, VarMemória)	**Lê** o conteúdo do ficheiro para a variável e avança para a posição seguinte.	**READ** (Ficheiro, Buffer); { 'texto.txt' → Buffer }
Readln (VarFicheiro, VarMemória)	**Lê** o conteúdo do ficheiro para a variável, ignora o resto da linha e avança para a linha seguinte.	**READLN** (Ficheiro, Buffer); { 'texto.txt' → Buffer }
VarBooleana := **EOF** (VarFicheiro)	Devolve o valor ***True*** se atingir o fim do ficheiro ou ***False***, caso contrário.	WHILE NOT **EOF** (Ficheiro) DO **...** { Repete até encontrar o final }
VarBooleana := **EOLN** (VarFicheiro)	Devolve o valor ***True*** se atingir o final da linha ou ***False***, caso contrário.	IF **EOLN** (Ficheiro) THEN READLN (Ficheiro); { Avança para a nova linha }
Erase (VarFicheiro)	Elimina do disco o ficheiro indicado (o ficheiro não pode estar aberto).	**ERASE** (Ficheiro); { Apaga *Ficheiro* do disco }
Rename (VarFicheiro, NovoNome)	Muda o nome ao ficheiro no disco (o ficheiro não pode estar aberto).	**RENAME** (Ficheiro, 'NovoNome.txt'); { Muda o nome do *Ficheiro* no disco }
VarString := **FSearch** (NomeFicheiro, DirectórioAprocurar)	Procura um ficheiro no directório indicado. Se existir, *VarString* contem o Caminho + NomeFicheiro, caso contrário, devolve uma *string* vazia.	IF **Fsearch**('Texto.txt', '.') < >'' THEN **...** { Ficheiro Texto.txt encontrado no directório actual }
{$I+}**, **{$I-} e **IOResult**	Ver ficheiros em binário.	Ver ficheiros em binário.

3.8.3.3. Caso particular de ficheiros de texto: teclado e monitor

Embora até ao momento ainda não tenha sido explicitamente referido, é agora importante salientar que a entrada e saída de dados via teclado e monitor do computador tem um tratamento idêntico ao de um ficheiro do tipo ***TEXT***. Para o efeito são criadas automaticamente as variáveis de ficheiro ***INPUT*** e ***OUTPUT*** e atribuídas ao teclado e monitor, pelo facto de serem seleccionadas por defeito, não é necessário referi-las nas operações de *Read(...)* e *Write(...)*.

READ (VarMemória); ⇔ **READ (*INPUT*, VarMemória);**
WRITE (VarMemória); ⇔ **WRITE (*OUTPUT*, VarMemória);**

3.8.3.4. Leitura de ficheiros de texto para variáveis do tipo String

A utilização de variáveis do tipo ***string*** em ficheiros de texto tem a vantagem de permitir ler um conjunto de caracteres de uma só vez mas só até à marca de fim de linha (Eoln). Esta marca não pode ser lida para a *string* e, consequentemente, não avança o ponteiro de ficheiro, resultando normalmente no bloqueio do computador. Para ultrapassar a situação é necessário aplicar a seguinte solução:

```
WHILE  NOT EOF( Ficheiro ) DO
    BEGIN
        READ ( Ficheiro, VarString );
        IF EOLN ( Ficheiro ) THEN
            READLN ( Ficheiro );
        ...
    END;
```

Todos os caracteres são lidos para a string até encontrar a marca de fim de linha, detectada pela função EOLN( ). Nessa altura utiliza-se o READLN( ) para ultrapassar a marca de fim de linha.

3.8.4. Programa típico utilizando ficheiros

Para utilizar ficheiros em PASCAL é necessário realizar um conjunto de passos que são semelhantes em todos os programas:

Definição de tipo: *Sempre que o ficheiro necessita de guardar informação de vários tipos, esta deve ser reunida num RECORD que servirá para definir o* ***registo*** *do ficheiro.*

```
TYPE
  Registo = RECORD
     <definição de variáveis do registo>;
  END;
```

Declaração de variáveis: *É necessário definir uma variável de um tipo especial (Text ou File Of) que ficará associada ao ficheiro. Para ler e escrever a informação contida num ficheiro de e para a memória é necessário uma variável do mesmo tipo que a informação contida no ficheiro.*

```
VAR
  Ficheiro : TEXT ou FILE OF  Registo;
  Buffer   : Registo;
```

1º Passo: *Associar a variável de ficheiro com o nome do ficheiro em disco.*

2º Passo: *Abrir o ficheiro para leitura e ou escrita.*

3º Passo: *Ciclo de repetição para percorrer todo o ficheiro e terminar quando encontrar a marca eof.*

4º Passo: *Operações de leitura e/ou escrita sobre o ficheiro.*

5º Passo: *Fechar o ficheiro.*

```
BEGIN
  ASSIGN(Ficheiro, 'NomeFich.Dat');
  RESET(...), REWRITE(...) ou APPEND(...);
  WHILE  NOT EOF(Ficheiro) DO
     BEGIN
        READ(Ficheiro, buffer);
        ...
        WRITE(Ficheiro, buffer);
     END;
  CLOSE(Ficheiro);
END.
```

3.8.5. Determinar se um ficheiro já existe

Versão 1: Utilização da função FSearch.

```
PROGRAM Exemplo;
USES DOS;
...
FUNCTION FicheiroExiste(NomeFich:string)
                                  :BOOLEAN;
BEGIN
  IF FSEARCH(NomeFich, '.') < > '' THEN
    FicheiroExiste := TRUE
  ELSE
    FicheiroExiste := FALSE;
END;
```

FSearch necessita da Unit DOS.

Procura o ficheiro na directoria actual.

Versão 2: Utilização da directiva I e de IOResult.

```
FUNCTION FicheiroExiste(NomeFich:string)
                                  :BOOLEAN;
VAR
  Ficheiro : FILE;
BEGIN
  ASSIGN(Ficheiro, NomeFich);
  {$I-}
  RESET(Ficheiro);
  CLOSE(Ficheiro);
  {$I+}
  IF IORESULT = 0 THEN
    FicheiroExiste := TRUE
  ELSE
    FicheiroExiste := FALSE;
END;
```

Tenta abrir o ficheiro: Se este não existir, ocorre um erro.

Exemplo de aplicação da função FicheiroExiste.

```
PROGRAM Exemplo_De_Aplicacao;
...
FUNCTION FicheiroExiste ...;
 ...
BEGIN
 WRITE('Nome do Ficheiro a Ler:');
 READLN(Nome);
 IF FicheiroExiste(Nome) = FALSE THEN
   WRITELN('O Ficheiro ',Nome,' não Existe!')
 ELSE
   BEGIN
     ASSIGN(Ficheiro, Nome);
     RESET(Ficheiro);
     WHILE NOT EOF(Ficheiro) DO
       ...
     CLOSE(Ficheiro);
   END;
END.
```

Versão 1 ou Versão 2.

Abre o ficheiro existente.

3.8.6. Exemplos de aplicação com ficheiros de texto

3.8.6.1. Ler e Escrever num ficheiro de texto

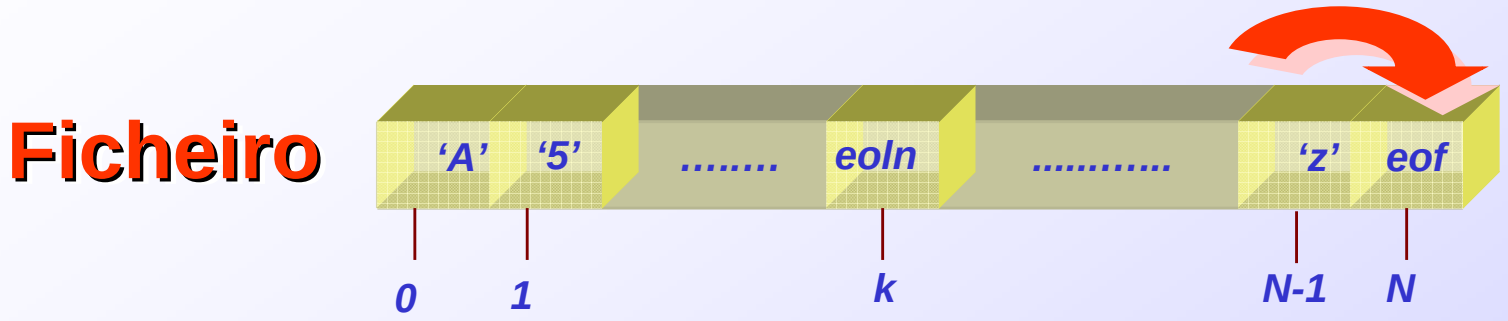
- Criar um novo ficheiro e nele colocar N linhas de texto inseridas através do teclado.

```
PROGRAM Escrever_Ficheiro_Texto;
VAR
  Ficheiro : TEXT;
  Linha : STRING;
  n, i : INTEGER;
BEGIN
  WRITE('Quantas linhas?');
  READLN(n);
  ASSIGN(Ficheiro, 'Texto.txt');
  REWRITE(Ficheiro);
  FOR i := 1 TO n DO
    BEGIN
      WRITE('Qual a ',i,'º linha?');
      READLN(Linha);
      WRITELN(Ficheiro, Linha);
    END;
  CLOSE(Ficheiro);
END.
```

- Abrir um ficheiro existente para leitura e mostrar o seu conteúdo no ecrã do computador.

```
PROGRAM Ler_Ficheiro_Texto;
VAR
  Ficheiro : TEXT;
  Buffer   : CHAR;

BEGIN
  ASSIGN(Ficheiro, 'Texto.txt');
  RESET(Ficheiro);
  WHILE NOT EOF(Ficheiro) DO
    BEGIN
      READ(Ficheiro, Buffer);
      WRITE(Buffer);
    END;
  CLOSE(Ficheiro);
END.
```

3.8.6.2. Acrescentar elementos a um ficheiro de texto




```
PROGRAM Acrescentar_Elementos;
VAR
  Ficheiro : TEXT;
  Buffer   : STRING;
  N, i     : LONGINT;

BEGIN
  WRITE('Quantas linhas a acrescentar?');
  READLN(N);
  ASSIGN(Ficheiro, 'Texto.txt');
  APPEND(Ficheiro);
  FOR i:=1 TO N DO
    BEGIN
      WRITE('Qual a ',i,'º linha?');
      READLN(Buffer);
      WRITELN(Ficheiro, Buffer);
    END;
  CLOSE(Ficheiro);
END.
```

Abre o ficheiro para escrita e posiciona o Ponteiro no final do ficheiro.

Acrescentar N linhas de texto ao ficheiro. Adicionar mudança de linha.

3.8.6.3. Modificar ficheiros de texto

Como nos ficheiros do tipo TEXT não é possível abrir um ficheiro simultaneamente para leitura e escrita além de que, também não é possível deslocar o ponteiro do ficheiro para uma posição qualquer, é então necessário recorrer a um **ficheiro auxiliar** que contem o ficheiro original e as modificações.

```
PROGRAM Modificar_Elementos;
VAR
  Ficheiro, Auxiliar : TEXT;
  Buffer   : CHAR;

BEGIN
  ASSIGN(Ficheiro, 'Texto.txt');
  RESET(Ficheiro);
  ASSIGN(Auxiliar, 'Aux.tmp');
  REWRITE(Auxiliar);
```

*Ficheiro aberto para **leitura**.*
*Auxiliar criado e aberto para **escrita**.*

```
  WHILE NOT EOF(Ficheiro) DO
    BEGIN
      READ(Ficheiro, Buffer);
      ...
      WRITE(Auxiliar, Buffer);
    END;
  CLOSE(Ficheiro);
  ERASE(Ficheiro);
  CLOSE(Auxiliar);
  RENAME(Auxiliar,'Texto.txt');
END.
```

Leitura no Ficheiro.
Executar modificações.
Escrita no Auxiliar.

Apagar Ficheiro.
Alterar nome a Auxiliar.

3.8.7. Exemplos de aplicação com ficheiros binários

3.8.7.1. Ler e Escrever num ficheiro binário

- Criar um novo ficheiro e nele colocar N elementos inseridos a partir do teclado.

```
PROGRAM Escrever_Ficheiro;
VAR
  Ficheiro     : FILE OF INTEGER;
  Buffer, n, i : INTEGER;

BEGIN
  WRITE('Quantos elementos?');
  READLN(n);
  ASSIGN(Ficheiro, 'Dados.dat');
  REWRITE(Ficheiro);
  FOR i := 1 TO n DO
    BEGIN
      WRITE('Qual o ',i,'º valor?');
      READLN(Buffer);
      WRITE(Ficheiro, Buffer);
    END;
  CLOSE(Ficheiro);
END.
```

- Abrir um ficheiro existente para leitura e mostrar todos os seus elementos no ecrã do computador.

```
PROGRAM Ler_Ficheiro;
VAR
  Ficheiro : FILE OF INTEGER;
  Buffer   : INTEGER;

BEGIN
  ASSIGN(Ficheiro, 'Dados.dat');
  RESET(Ficheiro);
  WHILE NOT EOF(Ficheiro) DO
    BEGIN
      READ(Ficheiro, Buffer);
      WRITELN(Buffer);
    END;
  CLOSE(Ficheiro);
END.
```

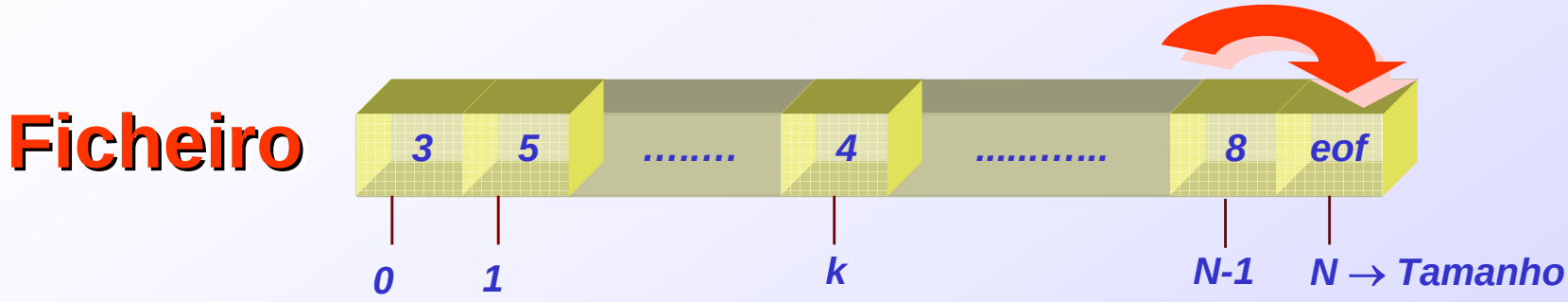
3.8.7.2. Duplicar um ficheiro binário

```
PROGRAM Duplicar_Ficheiro;
VAR
  Fich1, Fich2 : FILE OF INTEGER;
  Buffer       : INTEGER;

BEGIN
  ASSIGN(Fich1, 'Origem.dat');
  RESET(Fich1);
  ASSIGN(Fich2, 'Destino.dat');
  REWRITE(Fich2);
  WHILE NOT EOF(Fich1) DO
    BEGIN
      READ(Fich1, Buffer);
      ...
      WRITE(Fich2, Buffer);
    END;
  CLOSE(Fich1);
  CLOSE(Fich2);
END.
```

Abrir o Ficheiro de origem (já existente) para leitura.

Criar o Ficheiro de destino e abrir para escrita.

3.8.7.3. Acrescentar elementos a um ficheiro binário



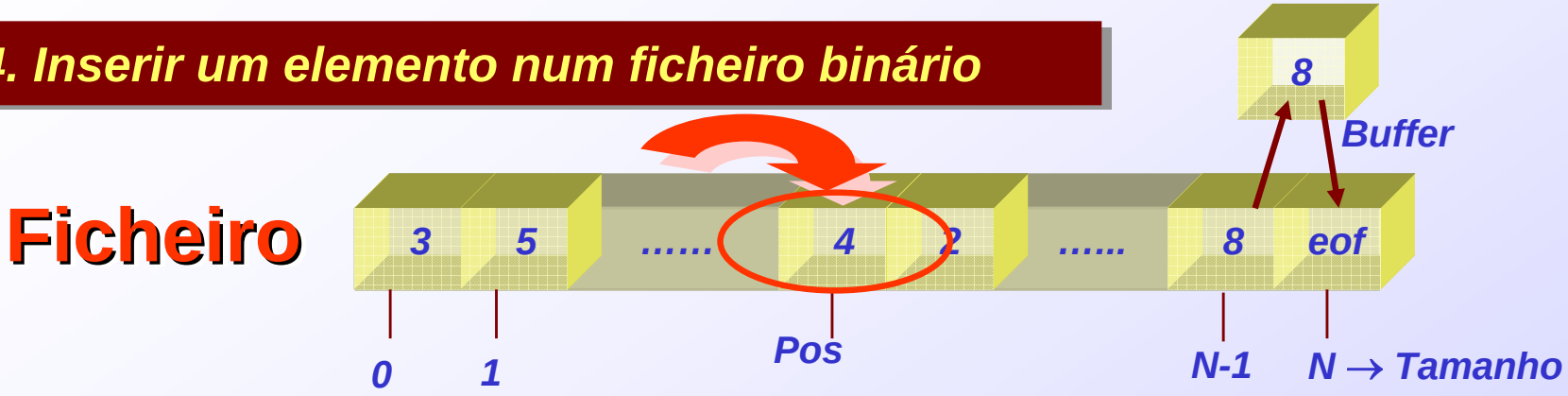

```
PROGRAM Acrescentar_Elementos;
VAR
  Ficheiro : FILE OF INTEGER;
  Buffer   : INTEGER;
  Tamanho, N, i : LONGINT;
BEGIN
  WRITE('Quantos elementos a acrescentar:');
  READLN(N);
  ASSIGN(Ficheiro, 'Dados.dat');
  RESET(Ficheiro);

  Tamanho := FILESIZE(Ficheiro);
  SEEK(Ficheiro, Tamanho);
```

Posicionar o Ponteiro no final do ficheiro.

```
  FOR i:=1 TO N DO
    BEGIN
      WRITE('Qual o ',i,'º valor?');
      READLN(Buffer);
      WRITE(Ficheiro, Buffer);
    END;
  CLOSE(Ficheiro);
END.
```

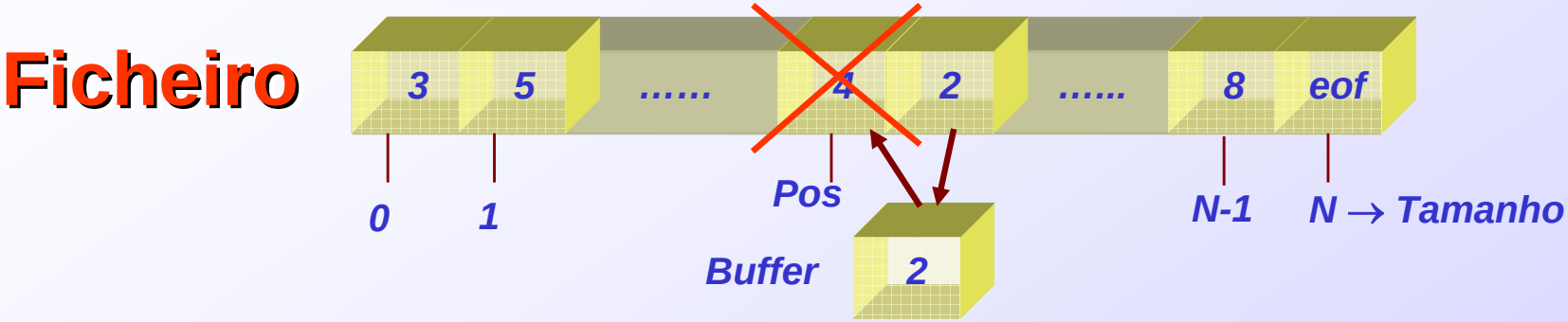
Acrescentar N elementos ao ficheiro.

3.8.7.4. Inserir um elemento num ficheiro binário



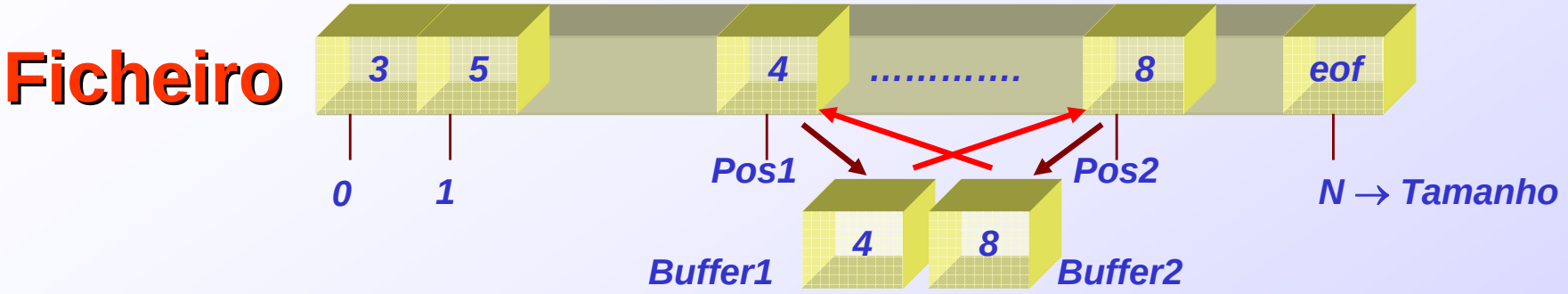

```
PROGRAM Inserir_Elemento;
VAR
  Ficheiro : FILE OF INTEGER;
  Buffer   : INTEGER;
  Tamanho, Pos, i : LONGINT;

BEGIN
  WRITE('Posição do elemento a inserir:');
  READLN(Pos);
  ASSIGN(Ficheiro, 'Dados.dat');
  RESET(Ficheiro);
  Tamanho := FILESIZE(Ficheiro);

  IF (Pos >= Tamanho) THEN
    WRITE('Erro: Fora dos limites do ficheiro')
  ELSE
    BEGIN
      FOR i:=Tamanho-1 DOWNTO Pos DO
        BEGIN
          SEEK(Ficheiro, i);
          READ(Ficheiro, Buffer);
          WRITE(Ficheiro, Buffer);
        END;
      SEEK(Ficheiro, Pos);
      WRITE('Qual o novo valor?');
      READLN(Buffer);
      WRITE(Ficheiro, Buffer);
    END;
  CLOSE(Ficheiro);
END.
```

3.8.7.5. Remover um elemento dum ficheiro binário




```
PROGRAM Eliminar_Elemento;
VAR
  Ficheiro : FILE OF INTEGER;
  Buffer   : INTEGER;
  Tamanho, Pos, i : LONGINT;
BEGIN
  WRITE('Posição do elemento a eliminar:');
  READLN(Pos);
  ASSIGN(Ficheiro, 'Dados.dat');
  RESET(Ficheiro);
  Tamanho := FILESIZE(Ficheiro);
  IF (Pos >= Tamanho) THEN
    WRITE('Erro:Fora dos limites do ficheiro')
  ELSE
    BEGIN
      FOR i := POS+1 TO Tamanho-1 DO
        BEGIN
          SEEK(Ficheiro, i);
          READ(Ficheiro, Buffer);
          SEEK(Ficheiro, i-1);
          WRITE(Ficheiro, Buffer);
        END;
      SEEK(Ficheiro, Tamanho-1);
      TRUNCATE(Ficheiro);
    END;
  CLOSE(Ficheiro);
END.
```

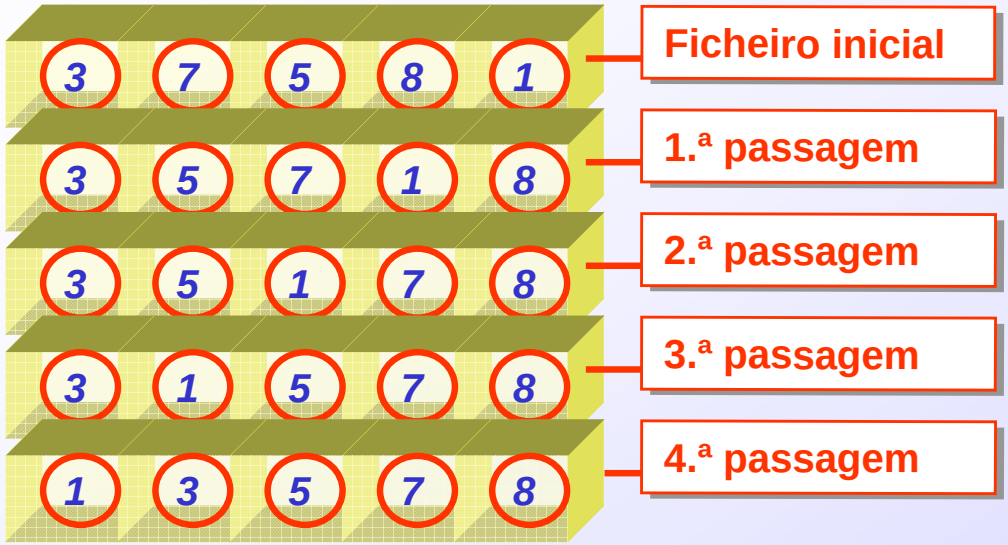
3.8.7.6. Trocar dois elementos dum ficheiro binário




```
PROGRAM Trocar_Elementos;
VAR
  Ficheiro : FILE OF INTEGER;
  Buffer1, Buffer2 : INTEGER;
  Tamanho, Pos1, Pos2 : LONGINT;
BEGIN
  WRITE('Posição dos elementos a trocar:');
  READLN(Pos1, Pos2);
  ASSIGN(Ficheiro, 'Dados.dat');
  RESET(Ficheiro);
  Tamanho := FILESIZE(Ficheiro);
  IF (Pos1 >= Tamanho) OR (Pos2 >= Tamanho) THEN
    WRITE('Erro:Fora dos limites do ficheiro')
  ELSE
    BEGIN
      SEEK(Ficheiro, Pos1);
      READ(Ficheiro, Buffer1);
      SEEK(Ficheiro, Pos2);
      READ(Ficheiro, Buffer2);
      SEEK(Ficheiro, Pos1);
      WRITE(Ficheiro, Buffer2);
      SEEK(Ficheiro, Pos2);
      WRITE(Ficheiro, Buffer1);
    END;
  CLOSE(Ficheiro);
END.
```

3.8.7.7. Ordenação dum ficheiro binário

Ficheiro

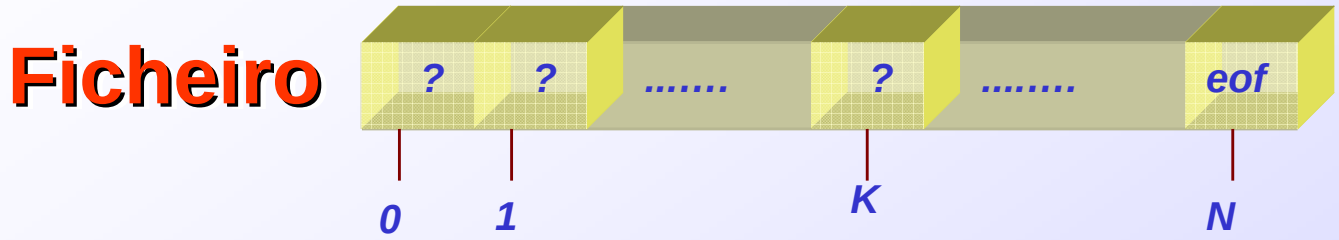


Utilizando vectores

```
FOR i := N-1 DOWNTO 1 DO
    FOR j := 1 TO i DO
      IF v[j] > v[j+1] THEN
        BEGIN
          Aux   := v[j];
          v[j] := v[j+1];
          v[j+1] := Aux;
        END;
```

```
PROGRAM Ordenacao_Ficheiro;
VAR
  Ficheiro : FILE OF INTEGER;
  i, j, Valor1, Valor2 : INTEGER;
BEGIN
  ASSIGN(Ficheiro, 'Dados.dat');
  RESET(Ficheiro);
  N := FILESIZE(Ficheiro);
  FOR i := N-2 DOWNTO 0 DO
    FOR j := 0 TO i DO
      BEGIN
        SEEK(Ficheiro, j);
        READ(Ficheiro, Valor1);
        READ(Ficheiro, Valor2);
        IF Valor1 > Valor2 THEN
          BEGIN
            SEEK(Ficheiro, j);
            WRITE(Ficheiro, Valor2);
            WRITE(Ficheiro, Valor1);
          END;
      END;
END.
```

3.8.7.8. Pesquisar um ficheiro binário




```
PROGRAM Pesquisar_Ficheiro;
TYPE
  Agenda = RECORD
    Nome, Morada : STRING[50];
    Telef : LONGINT;
  END;
VAR
  Ficheiro : FILE OF Agenda;
  Buffer   : Agenda;
  Nome     : STRING;

BEGIN
  WRITE('Nome a procurar na agenda?');
  READLN(Nome);
  ASSIGN(Ficheiro, 'Agenda.dat');
  RESET(Ficheiro);
  WHILE NOT EOF(Ficheiro) DO
    BEGIN
      READ(Fich1, Buffer);
      IF Buffer.Nome = Nome THEN
        BEGIN
         WRITELN('Nome: ',Buffer.Nome);
         WRITELN('Morada:',Buffer.Morada);
         WRITELN('Telefone:',Buffer.Telef);
        END;
    END;
  CLOSE(Ficheiro);
END.
```

3.8.7.9. Efectuar cálculos com dados em ficheiro binário

Calcular a **soma**, **média**, **máximo** e **mínimo** de um ficheiro de inteiros já existente.

```
PROGRAM CalculosComFicheiros;
VAR
 Ficheiro : FILE OF INTEGER;
 Media, Soma : REAL;
 Buffer, Min, Max, Cont : INTEGER;

FUNCTION FicheiroExiste ...;

BEGIN
 IF FicheiroExiste('Dados.dat') = FALSE THEN
   WRITELN('O Ficheiro Dados.dat não Existe!')
 ELSE
   BEGIN
     Soma:=0; Cont:=0;
     Min:=0; Max:=0; Media:=0;
     ASSIGN(Ficheiro, 'Dados.dat');
     RESET(Ficheiro);
     IF NOT EOF(Ficheiro) THEN
       BEGIN
         READ(Ficheiro, Buffer);
         Soma := Soma+Buffer;
         Min:=Buffer; Max:=Buffer; Cont:=1;
       END;
     WHILE NOT EOF(Ficheiro) DO
       BEGIN
         READ(Ficheiro, Buffer);
         IF Min > Buffer THEN
           Min:=Buffer;
         IF Max < Buffer THEN
           Max:=Buffer;
         Soma := Soma+Buffer;
         Cont := Cont+1;
       END;
     CLOSE(Ficheiro);
     IF Cont < > 0 THEN
       Media:=Soma/Cont;
     WRITELN('Mínimo:',Min,' Máximo:',Max);
     WRITELN('Soma:',Soma,' Média:',Media);
   END;
END.
```

Verifica se o 1º elemento existe e inicializa as variáveis.

Lê os elementos seguintes do ficheiro, efectuando todos os cálculos.

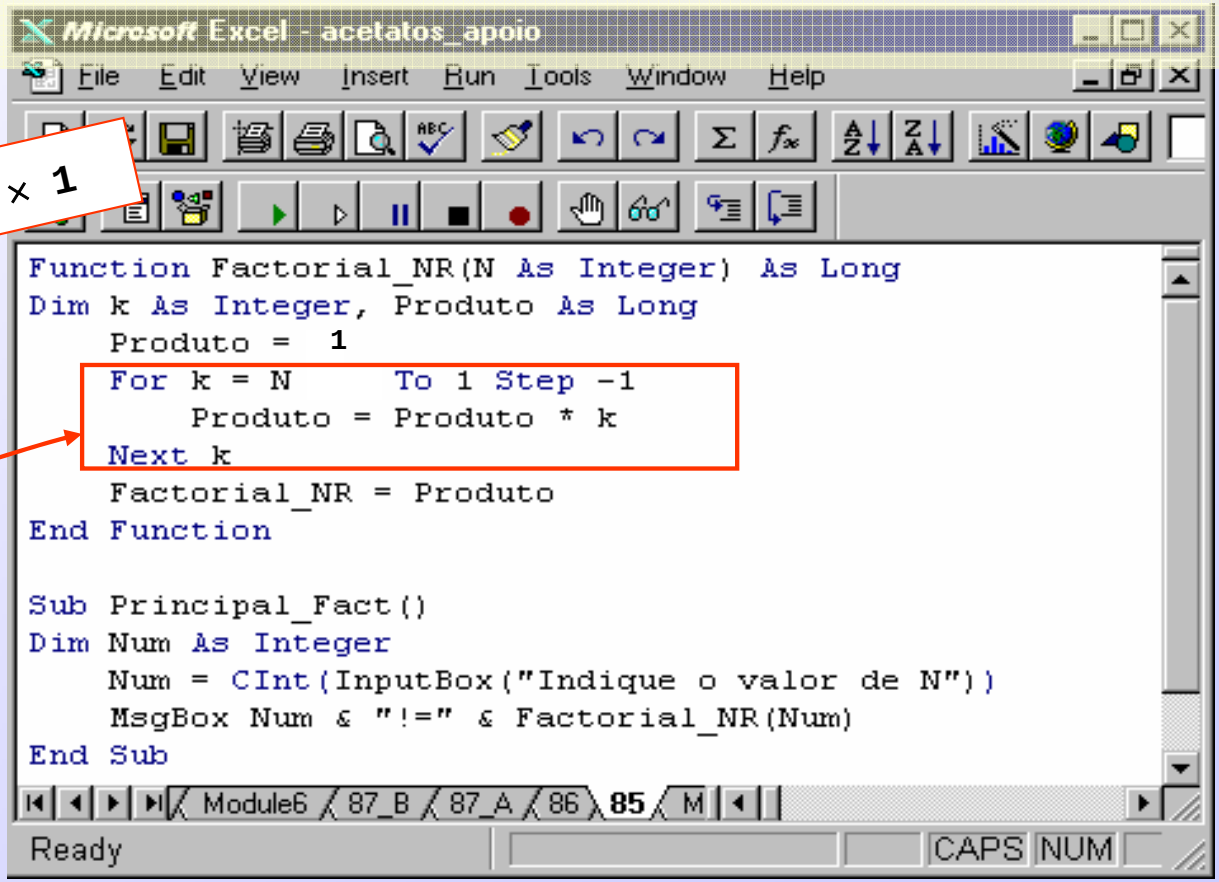
3.7.3. Recursividade

A possibilidade de um procedimento poder chamar-se a si próprio em PASCAL, designa-se por recursividade. A sua explicação, será mais explícita se tomarmos como exemplo:

CÁLCULO DO FACTORIAL DE N : N!

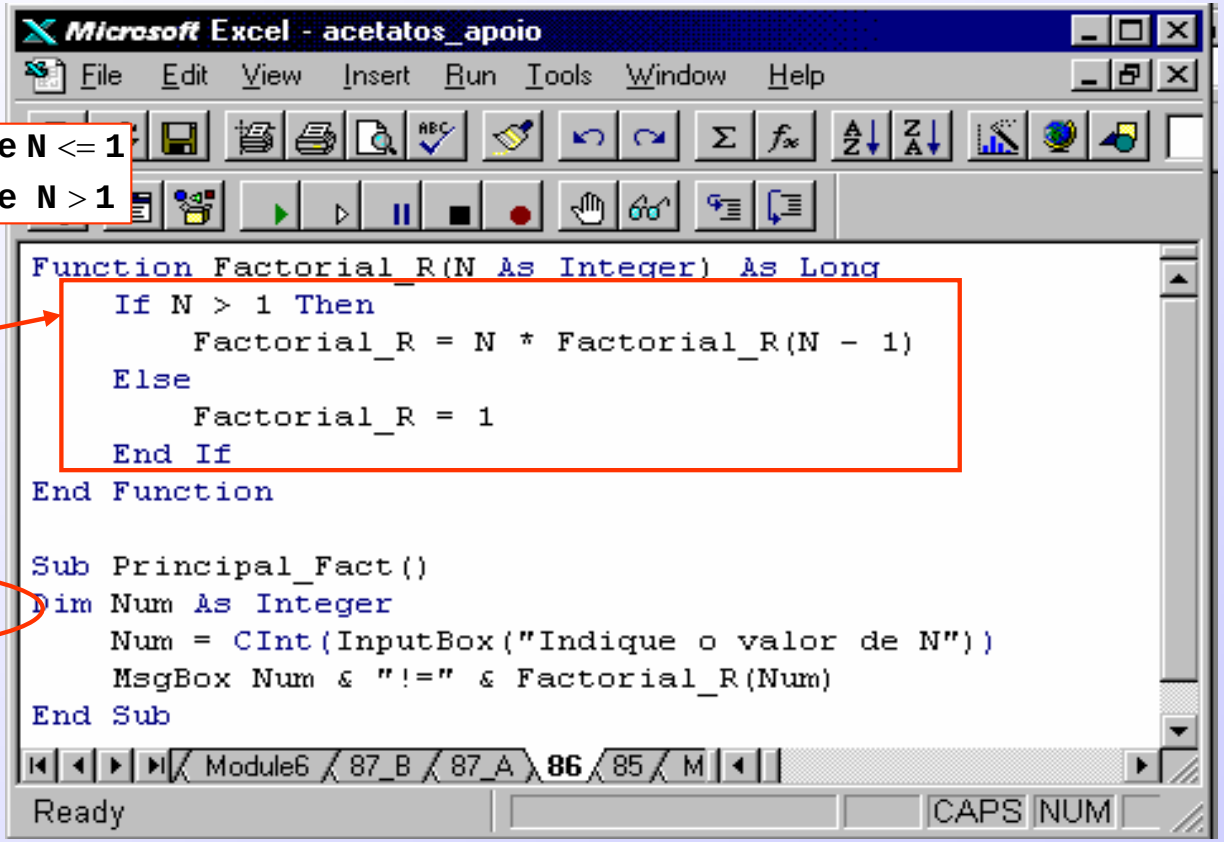
Solução não recursiva

$$Factorial(N) = N \times (N-1) \times (N-2) \times \ldots \times 1$$

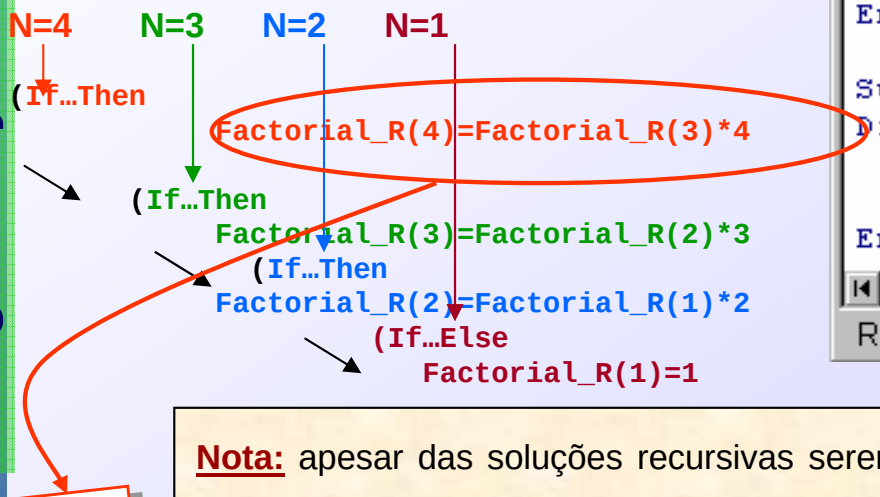


Solução recursiva

$$Factorial(N) = \begin{cases} 1 & \text{se } N <= 1 \\ N \times Factorial(N-1) & \text{se } N > 1 \end{cases}$$



Supondo que N =4, testemos a solução recursiva:



Nota: apesar das soluções recursivas serem mais elegantes, os programadores tendem a evitá-las, porque um sub-programa recursivo necessita de mais espaço (um conjunto completo de endereços de memória deve ser reservado de cada vez que um sub-programa se chama a si próprio) e é mais lento (devido às operações auxiliares para a entrada e saída dum programa) do que um sub-programa não recursivo.

Exemplo

Cálculo de A^k, onde A e k são números inteiros, com N > 0.

Solução não recursiva

$$\text{Potência}(A,k) = \overbrace{A \times A \times A \times \ldots \times A}^{\text{N-1 multiplicações}}$$

Solução recursiva

$$\text{Potência}(A,k) = \begin{cases} A & \text{se } N = 1 \\ A \times \text{Potência}(A,k) & \text{se } N > 1 \end{cases}$$

Microsoft Excel - acetatos_apoio

File Edit View Insert Run Tools Window Help

```
Function Potencia_NR(A As Integer, k As Integer) As Long
Dim i As Integer, Produto As Long
    Produto = A
    For i = 1 To k - 1
        Produto = Produto * A

    Next i
    Potencia_NR = Produto
End Function

Sub Principal_Pot()
Dim k As Integer, A As Integer
    A = CInt(InputBox("Indique o base A"))
    k = CInt(InputBox("Indique o expoente k"))
    MsgBox A & " elevado a " & k & " = " & Potencia_NR(A, k)
End Sub
```

Module6 | 87_B | 87_A | 86 | 85 | Modul

Ready CAPS NUM

Microsoft Excel - acetatos_apoio

File Edit View Insert Run Tools Window Help

```
Function Potencia_R(A As Integer, k As Integer) As Long
    If k = 1 Then
        Potencia_R = A
    Else
        Potencia_R = A * Potencia(A, k)
    End If
End Function

Sub Principal_Pot()
Dim k As Integer, A As Integer
    A = CInt(InputBox("Indique o base A"))
    k = CInt(InputBox("Indique o expoente k"))
    MsgBox A & " elevado a " & k & " = " & Potencia_R(A, k)
End Sub
```

Module6 | 87_B | 87_A | 86 | 85 | Modul

Ready CAPS NUM