Universidade Federal de Sergipe

Departamento de Computação

Turbo Pascal 7.0

Autor: Prof. Alberto Costa Neto

Maio / 2010

Sumário

1. Introdução ao Turbo Pascal

1.1. A linguagem Pascal

A linguagem Pascal foi desenvolvida pelo Professor Niklaus Wirth durante o período de 1960 a meados de 1970, com o objetivo principal da mesma ser uma linguagem fácil de aprender além de permitir a utilização de estilos de programação considerados como sendo de boa prática.

O Pascal, assim como outras linguagens, possui mais de um dialeto. O Pascal padrão foi definido pela Organização Internacional de Padrões (ISO). O Turbo Pascal é uma variante do Pascal desenvolvida pela Borland que traz características adicionais.

1.2. O Turbo Pascal

Os programas em linguagens de alto nível devem ser traduzidos antes de serem executados pelo computador. Quem faz essa traduções são os programas tradutores.

Existem basicamente 2 tipos de programa tradutor: o interpretador; e o compilador. Os dois aceitam como entrada um programa em linguagem de alto nível (fonte) e produzem como saída um programa em linguagem de máquina (objeto).

A diferença entre eles está na forma de executar a tradução. O interpretador traduz para a linguagem de máquina e roda uma linha por vez, até que todo programa seja executado. Já o compilador traduz para a linguagem de máquina todo o programa fonte e só então ele é executado.

Existem linguagens de programação interpretadas e compiladas. O COBOL é compilado, o BASIC pode ser tanto compilado como interpretado. A linguagem Pascal é tradicionalmente compilada.

O Turbo Pascal é um ambiente integrado de desenvolvimento (IDE) porque traz um editor, um compilador e um linkeditor integrados.

1.3. Primeiro programa em Pascal

```
program Ex1Cap1;
uses crt;
label
   fim;
const
   Meu_Nome = 'Alberto';
   Minha_Idade = 50;
type
```

```
   nacionalidade = (BRASILEIRA, PORTUGUESA, INGLESA, ALEMA, AMERICANA);
var idade  : integer;
    altura : real;
    nome   : string[30];
    sexo   : char;
    n      : nacionalidade;

procedure Linha;
var i:integer;
begin
   for i:=1 to 80 do
      write('-');
end;

function Soma(x,y:integer):integer;
begin
   Soma:=x+y;
end;

BEGIN
   ClrScr;
   Linha;
   writeln('Meu nome e -------> ',Meu_Nome);
   Linha;
   write('Qual o seu nome ----> ');
   readln(Nome);
   Linha;
   write('Qual a sua idade ---> ');
   readln(idade);
   Linha;
   writeln('Nossas idades somam --> ',Soma(Minha_Idade, idade));
   Linha;
   if Minha_Idade >= idade then
      goto fim;
   writeln('Eu sou mais novo que voce');
   Linha;
fim:
   write('Prazer em conhece-lo');
END.
```

1.4. Notação Sintática

Notação	Descrição
Negrito	Palavras reservadas do Pascal, operadores e pontuações
<palavra>	Os caracteres < > indicam que a palavra entre eles deve ser substituída por itens definidos pelo usuário
[item]	Indicam que os itens são opcionais

...	Indicam que o item precedente pode ser repetido uma ou mais vezes

Exemplo 1: Definição de um `if-then-else`

```
if <exp-logica> then
   <comando>
[else
   <comando>]
```

Exemplo 2: Declaração de variáveis

```
var <variavel>[,<variavel>]...:<tipo>;
```

1.5. Identificadores

Servem para dar nomes a variáveis, tipos, procedimentos, funções, constantes, programas, units e campos de registros.

Podem ter qualquer tamanho mas somente os 63 primeiros caracteres são significativos (restrição do Turbo Pascal).

O primeiro caractere pode ser uma letra ou caractere sublinhado, os demais podem ser também números.

Não existe distinção entre maiúsculas e minúsculas.

1.5.1. Identificadores predefinidos

O Turbo Pascal traz vários identificadores predefinidos, tais como: `ClrScr` e `DelLine`. Não é permitido reutilizar esses identificadores para declarar novos elementos no programa.

1.5.2. Identificadores definidos pelo usuário

Ao criar um programa é necessário declarar vários identificadores para definir variáveis, procedimentos, funções etc. É permitido utilizar qualquer identificador (obedecendo às regras) desde que não exista um identificador predefinido com o mesmo nome.

Exemplos de identificadores válidos: `Nome_Cliente`, `_Numero` e `Linha10`.

Exemplos de identificadores inválidos: `80hifens` (começa com número) e `Endereco Residencial` (inclui um branco).

1.6. Palavras reservadas

As palavras reservadas do Turbo Pascal são palavras que fazem parte da sua estrutura e têm significados predeterminados. Elas não podem ser redefinidas e não podem ser utilizadas como identificadores de variáveis, procedures, functions etc. Algumas das palavras reservadas são:

and	**array**	**begin**	**case**	**const**	**div**	**do**
downto	**else**	**end**	**file**	**for**	**forward**	**function**
goto	**if**	**in**	**label**	**mod**	**nil**	**not**
of	**or**	**procedure**	**program**	**record**	**repeat**	**set**
then	**to**	**type**	**until**	**var**	**while**	**with**

Algumas palavras reservadas específicas do Turbo Pascal:

absolute	**external**	**shl**	**shr**	**string**	**xor**

1.7. Comentários

São textos colocados no programa fonte que servem para facilitar seu entendimento. No Turbo Pascal, todo texto que estiver entre { e } ou (* e *) é considerado comentário.

```
Contador := 10;
while true do (* Laço infinito *)
begin
   if Contador <= 0 then { Identifica o final da iteração }
      break;
   writeln(Contador);
   Contador := Contador - 1;
end;
```

Um caractere { seguido de um caractere $ indica uma diretiva de compilação e não um comentário.

1.8. Estrutura de um programa Pascal

Um programa Pascal contém:

1. **Cabeçalho** (opcional): Indica o nome do programa;

2. **Cláusula Uses** (opcional): Declara as units usadas pelo programa;

3. **Seção de Declarações** (opcional): Contém a declaração dos identificadores usados no programa, tais como: rótulos, constantes, tipos, variáveis e subprogramas;

4. **Corpo:** Contém todos os comandos a serem executados.

A estrutura de um programa é mostrada abaixo:

```
(* cabeçalho do programa *)
program <identificador>;

 [<cláusula uses>]

(* declarações *)
[<seção de declarações>]
[<declaração de rótulos>]
[<declaração de constantes>]
[<declaração de tipos>]
[<declaração de variáveis>]
[<declaração de subprogramas>]

(* corpo do programa *)
BEGIN
   <comando>[<comando>]...
END.
```

Exemplo de programa:

```
program Ex2Cap1; { cabecalho }
uses crt; { clausula uses }
var { declaracoes }
   Nome        : string;
   I, N_Letras : byte;
BEGIN  { corpo }
   clrscr;
   write('Qual eh o seu nome? ');
   readln(Nome);
   N_Letras := 0;
   for I := 1 to Length(Nome) do
      if Nome[I] in ['a'..'z', 'A'..'Z'] then
         inc(N_Letras);
   writeln('Seu nome contem ', N_Letras, ' letras');
END.
```

1.9. Units

Uma unit é um conjunto de constantes, tipos de dados, variáveis e subprogramas, os quais podem ser utilizados por outros programas.

O Turbo Pascal traz várias units, tais como:

- **Crt**: Traz funções que permitem manipular o vídeo.

- **Dos**: Contém rotinas para acessar o sistema operacional e manipular arquivos.

- **Graph**: Oferece uma coleção de subprogramas que permitem trabalhar com gráficos.

Além das predefinidas, é possível declarar novas units específicas para a aplicação sendo desenvolvida ou a ser desenvolvida.

1.10. Tipos de Dados

Um tipo de dado especifica as características de um dado. Toda variável e constante usada em um programa Pascal tem um tipo associado.

Podemos dividir os tipos de dados em três categorias:

- **Tipo Escalar**: representa um único item de dado. Pode ser ordinal ou real. No tipo ordinal existe uma relação de ordem entre os itens. O real é usado para representar números em ponto flutuante.

- **Tipo Estruturado**: representa uma coleção de itens de dados.

- **Tipo Apontador**: faz referência ou aponta para uma variável.

Tipos de dados predefinidos

A tabela abaixo contém os tipos de dados predefinidos:

Tipo	Descrição	Tamanho
boolean	Escalar ordinal que pode assumir os valores true (verdadeiro) e false (falso)	1 byte
char	Escalar ordinal que pode engloba todos os caracteres ASCII	1 byte
byte	Escalar ordinal cujos valores são número inteiros que variam de 0 a 255	1 byte

word	Escalar ordinal cujos valores são número inteiros que variam de 0 a 65535	2 bytes
shortint	Escalar ordinal cujos valores são número inteiros que variam de -128 a 127	1 byte
integer	Escalar ordinal cujos valores são número inteiros que variam de -32768 a 32767	2 bytes
longint	Escalar ordinal cujos valores são número inteiros que variam de -2147483648 a 2147483647	4 bytes
single	Escalar real cujos valores variam de 1.5e–45 a 3.4e38. Contém de 7 a 8 dígitos significativos	4 bytes
real	Escalar real cujos valores variam de 2.9e–39 a 1.7e38. Contém de 11 a 12 dígitos significativos	6 bytes
double	Escalar real cujos valores variam de 5.0e–324 a 1.7e308. Contém de 15 a 16 dígitos significativos	8 bytes
comp	Escalar real cujos valores variam de –9.2e18 a 9.2e18. Contém de 19 a 20 dígitos significativos	8 bytes
extend	Escalar real cujos valores variam de 3.4e–4932 a 1.1e4932. Contém de 19 a 20 dígitos significativos	10 bytes
string	Tipo estruturado que contém itens de dados do tipo char	-
text	Tipo estruturado que é definido como file of char	-

Tipos Escalar Enumerado

Um tipo escalar enumerado é um escalar ordinal cujos valores que as variáveis deste tipo podem assumir são definidos através de uma lista valores. Exemplos:

```
(SEG, TER, QUA, QUI, SEX, SAB, DOM)
(MINIMO, NORMAL, MAXIMO)
```

A ordem é crescente da esquerda para a direita, ou seja, `MINIMO<NORMAL<MAXIMO`.

As constantes definidas em um tipo enumerado devem ser exclusivas, isto é, não devem ser declaradas em outros tipos enumerados.

Tipo Subintervalo

O tipo subintervalo é um tipo escalar ordinal que representa um conjunto de valores consecutivos. Um subintervalo é definido da seguinte forma:

```
<constante>..<constante>
```

As constantes devem ser do mesmo tipo ordinal, sendo a primeira menor do que a segunda.

Exemplos de declarações de tipos subintervalo:

```
'a'..'z' { valores: a,b,c,d,e,f,...,z }
0..10    { valores: 0,1,2,3,4,5,6,7,8,9,10 }
SEG..SEX { valores: SEG,TER,QUA,QUI,SEX }
```

1.11. Constantes

Uma constante consiste em um valor que não muda durante toda a execução do programa. Podem ser classificadas de acordo com o tipo predefinido a que pertencem:

Inteiras: São números inteiros e podem ser representadas da seguinte forma:

- Dígitos decimais sem ponto decimal: 12345 e –12345
- Dígitos hexadecimais que devem ser precedidos do caractere $: $3039 (12345)

Reais: São números fracionários que podem ser representadas da seguinte forma:

- Dígitos decimais com ponto decimal: 12.345 (representa o número 12,345)
- Dígitos decimais seguidos pela letra E ou e, acompanhados por dígitos decimais para indicar a potência de 10: 0.12345e2 (representa o número 12,345)

Literais: São seqüências de caracteres colocados entre apóstrofos. Exemplo: `'Nome da empresa'`. Para incluir um apóstrofo em uma constante literal, deve-se colocar um outro apóstrofo (os dois formam um único).

Para representar uma constante literal que não tem nenhum caractere, colocam-se os dois apóstrofos juntos (`''`).

Para inserir caracteres de controle usa-se o caractere # seguindo do número correspondente na tabela ASCII. Exemplo: As constantes #65 e #13 representam, respectivamente, o caractere 'A' e uma quebra de linha.

Constantes nomeadas

Uma constante pode ser associada a um identificador, fazendo com que ele se torne um sinônimo do valor da constante. Para definir uma constante nomeada utiliza-se o comando `const`, que tem a seguinte sintaxe:

```
const
   <identificado> = <constante>;
  [<identificado> = <constante>;]...
```

Exemplos:

```
Numero_Dias = 30;
PI = 3.141519265;
Nome = 'Alberto'
```

As constantes `Numero_Dias`, `PI` e `Nome` são, respectivamente, dos tipos `integer`, `real` e `string`.

1.12. Variáveis

Uma variável contém um valor que pode ser manipulado (lido ou gravado) durante a execução de um programa. No Pascal, todas as variáveis são declaradas através do comando `var` dentro da seção de declarações. Abaixo encontra-se a sintaxe do comando `var`.

```
var
   <identificador> [,<identificador>]... : <tipo-do-dado>;
  [<identificador> [,<identificador>]... : <tipo-do-dado>;]...
```

Onde: identificador é o nome que será dado à variável cujo tipo e indicado por tipo-do-dado. É possível declarar mais de uma variável separando-as através de vírgulas.

Exemplo:

```
var
   X, Y, Z : integer;
   Taxa_Juros : double;
   Nome : string;
   Mes : 1..12;
   Dia_Util : (SEG,TER,QUA,QUI,SEX);
```

1.13. Strings

O Turbo Pascal provê um tipo de dados para se trabalhar com cadeias de caracteres, o tipo `string`. É um tipo estruturado semelhante a um `array`. A diferença básica é que o primeiro byte do `array` é usado para armazenar o tamanho corrente da `string`, ou seja, qual é o número de caracteres contidos na `string`. Assim, uma `string` pode conter de 0 a 255 caracteres (que é a faixa de valores que um tipo `byte` pode conter).

Ao declarar uma `string` deve-se informar seu tamanho máximo, que deve estar entre 1 e 255. Quando não informado, o valor assumido será 255. O `byte` de comprimento informa quantos caracteres estão efetivamente sendo usados.

Exemplos:

```
var
   Nome        : string[30];
   Endereco    : string[60];
   CEP         : string[8];
   Complemento : string;
```

A figura abaixo mostra a estrutura da variável `CEP` definida acima:

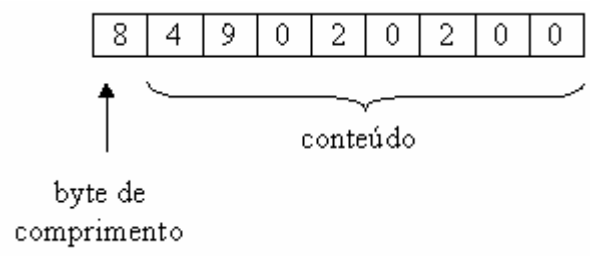


Figura 1 – Estrutura de uma string

O programa abaixo mostra a manipulação do byte de comprimento de uma variável do tipo `string`.

```
program Ex3Cap1;
var
   Str : string;
BEGIN
   Str := '12345';
   writeln(Str);     { Imprime 12345 }
   Str := 'ABC';
   writeln(Str);     { Imprime ABC }
   Str[0] := CHR(5); { Modifica o byte de comprimento para 5 }
   writeln(Str);     { Imprime ABC45 }
END.
```

1.14. Declaração de tipos

Uma declaração de um tipo de dado consiste em associar um identificador a uma descrição de tipo de dado. Esse identificador pode ser usado como se fosse um tipo predefinido.

Sintaxe para declaração de tipos:

```
type
   <identificador> = <descrição-do-tipo>;
  [<identificador> = <descrição-do-tipo>;]...
```

Exemplo de declaração de tipos e de variáveis:

```
type
   Upper_Char : 'A'..'Z';
   Prioridade : (MINIMA, NORMAL, MAXIMA);
var
   Inicial_Do_Nome     : Upper_Char;
   Prioridade_Execucao : Prioridade;
```

1.15. Constantes tipadas

As constantes são variáveis que têm um valor inicial associado antes do início da execução do programa. Apesar de serem denominadas constantes, podem ser modificadas durante a execução do programa. Ao contrário das constantes nomeadas, as tipadas têm seu tipo de dado declarado explicitamente.

Sintaxe de declaração de constantes tipadas:

```
const
   <identificador> : <tipo-do-dado> = <valor>;
  [<identificador> : <tipo-do-dado> = <valor>;]...
```

Exemplos:

```
const
   Fim     : boolean = true;
   PontoX  : integer = 0;
   PontoY  : integer = 0;
   Msg     : string  = 'Pressione <ENTER> para continuar'
```

1.16. Type Casting

É um mecanismo no qual uma variável de um tipo passa a ser tratada como se fosse de outro tipo de dado. Isso permite resolver alguns problemas de programação.

A sintaxe para fazer type casting é a seguinte:

```
<identificador-do-tipo> (<identificador-da-variável>)
```

Exemplo:

```
program Ex4Cap1;
var
   Caractere : char;
BEGIN
   write('Digite um caractere qualquer:');
   readln(Caractere);
   write('Codigo ASCII de ''', Caractere,  '''=');
   writeln(byte(Caractere));
END.
```

1.17. Rótulos

Os rótulos são referenciados pelos comandos `goto` com a finalidade de efetuar um desvio incondicional. Um rótulo é um identificador ou um número inteiro entre 0 e 9999. Os rótulos são declarados através do comando `label`.

Sintaxe de declaração de rótulos:

```
label <rótulo>[,<rótulo>]...;
```

Exemplo:

```
program Ex5Cap1;
label
   Inicio;
var
   Numero : byte;
BEGIN
   write('Digite um numero entre 0 e 255:');
   readln(Numero);
Inicio:
   writeln(Numero);
   Numero := Numero - 1;
   if Numero > 0 then
      goto Inicio;
END.
```

2. Estilo de Programação

2.1. Nomes

Toda palavra-chave, nome de subprogramas (funções e procedimentos) serão escritos com letras minúsculas, com exceção do `BEGIN` e `END` do programa principal que terão exclusivamente letras maiúsculas.

2.2. Identificadores

Nos identificadores criados no programa podem ser usadas letras maiúsculas e minúsculas, começando-se cada palavra sempre por maiúscula. Exemplos: `Soma`, `Contador`, `SaldoMedio` e `Taxa_Maxima`.

2.3. Seções

Na seção de declarações deve-se colocar a palavra-chave (`var`, `type`, `label`, `const`) sozinha em uma linha e os identificadores significativos serão definidos um por linha, dando uma tabulação de 3 posições em relação à palavra-chave.

```
type
   DiaDaSemana = (Dom, Seg, Ter, Qua, Qui, Sex, Sab);
   Vetor       = array[1..50] of real;
var
   Soma,
   Contador : real;
   I, J, K  : integer;
```

2.4. Blocos

O `begin` e o `end` dos blocos devem ser escritos em linhas separadas, começando na mesma coluna e todos os comandos contidos no bloco devem ser deslocados 3 posições para a direita.

```
while Contador > 0 do
begin
   writeln(Contador);
   Contador := Contador - 1;
end
```

2.5. Comando condicional if-then-else

Para o comando `if-then-else` deve-se usar a seguinte estrutura:

```
if Contador > 1 then
   writeln('positivo')
else
```

```
   writeln('nao positivo');
```

2.6. Atribuições

Quando há linhas consecutivas contendo comandos de atribuição, deve-se alinhar o operador := na mesma coluna.

```
Soma     := 0.0;
Contador := 10;
K        := 1;
```

2.7. Quebra de Linha

Quando uma linha lógica tiver que ser quebrada em várias linhas, as linhas de continuação devem ser deslocadas para a direita para realçar o comando da primeira linha.

```
if contador > 0 then
   begin
      writeln('-------------------------------------------------',
              '-------------------------------------------------',
              '-------------------------------------------------');
      writeln;
   end;
```

3. Expressões

Uma expressão é composta por operandos e operadores que representam uma fórmula que quando avaliada sempre gera um resultado.

Operandos podem ser variáveis, constantes ou resultados de funções

No Pascal, os operadores podem ser unários (`not, -`) ou binários (`and, *, +, -`), conforme o número de operandos.

O Pascal tem quatro tipos básicos de expressões:

- Numéricas
- Literais
- Relacionais
- Booleanas ou Lógicas

3.1. Ordem de avaliação de expressões

A ordem de avaliação das expressões obedece à seguinte regra:

1º) Parênteses (maior prioridade)

2º) Hierarquia dos Operadores

3º) Esquerda para a Direita (menor prioridade)

Os operadores do Pascal podem ser classificados da seguinte forma:

- Negação (maior prioridade)
- Multiplicativos
- Aditivos
- Relacionais (menor prioridade)

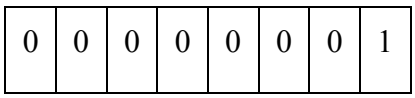
3.2. Expressões Numéricas

As expressões numéricas são caracterizadas por terem operadores numéricos e resultam em valores numéricos.

Operador de Negação

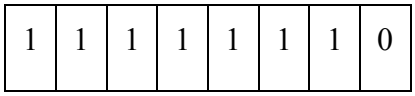
Inverte os bits do operando, ou seja, transforma 0 em 1 e 1 em 0. Para uma variável `X` do tipo `byte` que contém o valor 1, a expressão `not X` retornaria 254.

Variável `X` contendo o valor 1



⇓

Resultado da avaliação da expressão `not X` (valor 254)



Sintaxe: `not <operando>`

Exemplo:

```
program Ex1Cap3;
var
   b : byte;
BEGIN
   write('Digite um numero entre 0 e 255:');
   readln(b);
   writeln('not ', b, ' = ', not b);
END.
```

Operadores Multiplicativos

a) Multiplicação (`*`)

Quando algum dos operandos for real, o resultado será real. Caso contrário, o resultado da multiplicação será um valor inteiro.

Sintaxe: `<operando> * <operando>`

b) Divisão real (`/`)

Independentemente do tipo dos operandos, o resultado será sempre real.

Sintaxe: `<operando> / <operando>`

c) Divisão inteira (`div`)

O resultado será sempre inteiro, já que este operador retorna o quociente da divisão.

Sintaxe: `<operando> div <operando>`

Exemplo: `5 div 2 => 2`

d) Resto da divisão (`mod`)

Retorna um número inteiro que representa o resto da divisão.

Sintaxe: `<operando> mod <operando>`

Exemplo: `5 mod 2 => 1`

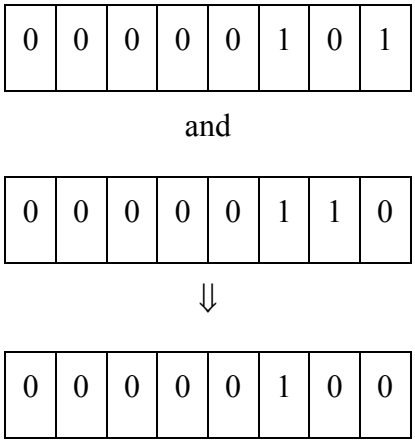
e) E lógico (`and`)

Retorna um número inteiro resultante da operação lógica "E" sobre os operandos.

Operando 1	**Operando 2**	**AND**	**OR**	**XOR**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Sintaxe: `<operando> and <operando>`

Exemplo: Aplicando o and sobre inteiros contendo os valores 5 e 6.



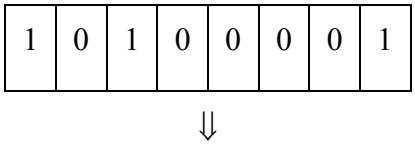
f) Deslocamento lógico para a esquerda (`shl`)

Desloca os bits do primeiro operando para a esquerda o número de vezes indicado pelo segundo operando.

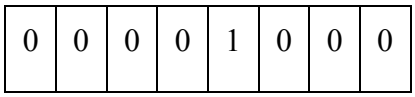
Sintaxe: `<operando> shl <operando>`

Exemplo:

Variável `X` contendo o valor 161



Resultado da avaliação da expressão `X shl 3 => 8`



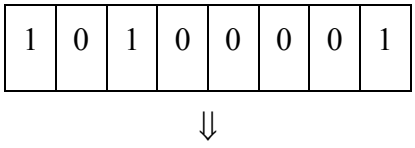
g) Deslocamento lógico para a direita (`shr`)

Desloca os bits do primeiro operando para a direita o número de vezes indicado pelo segundo operando.

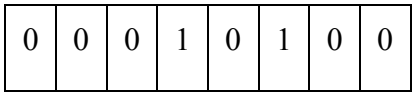
Sintaxe: `<operando> shr <operando>`

Exemplo:

Variável `X` contendo o valor 161



Resultado da avaliação da expressão `X shr 3 => 20`



Operadores aditivos

a) Adição e Subtração (`+` e `-`)

Quando algum dos operandos é real, o resultado também é real. Caso contrário, será sempre inteiro.

Sintaxe: `<operando> + <operando>`

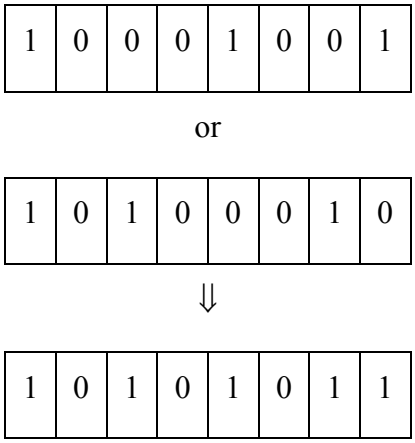
`<operando> - <operando>`

b) OU lógico (`or`)

Aplica para cada bit correspondente dos operandos o "OU" lógico.

Sintaxe: `<operando> or <operando>`

Exemplo:

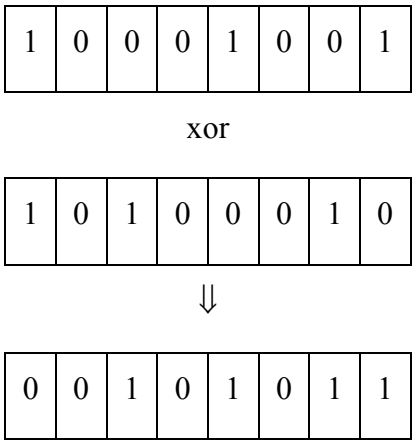


c) OU exclusivo (`xor`)

Aplica para cada bit correspondente dos operandos o "OU EXCLUSIVO" lógico.

Sintaxe: `<operando> xor <operando>`

Exemplo:



3.3. Expressões Literais

São expressões cujos operandos são do tipo `string` ou `char`.

O operador de concatenação (+) é o único que pode ser usado em uma expressão literal. Sua função é retornar uma `string` com o conteúdo do primeiro operando seguindo pelo do segundo.

Sintaxe: `<operando> + <operando>`

Exemplo: `'Alberto ' + 'Costa ' + 'Neto' => 'Alberto Costa Neto'`

3.4. Expressões Relacionais

O resultado da avaliação de um expressão relaciona é do tipo `boolean`.

Uma expressão relacional consiste de dois termos (tipos escalares ou `string`) separados por um dos operadores abaixo:

Operador	Significado
=	Igual
>	Maior
>=	Maior ou igual
<	Menor
<=	Menor ou igual
<>	Diferente

Em relação a strings, a comparação é feita caractere por caractere da esquerda para a direita até que sejam encontrados dois caracteres diferentes. Nesse momento, a ordem é definida através da comparação de seus códigos ASCII.

Todas as expressões relacionais abaixo resultam no valor `true`:

```
'ABC' <> 'XYZ'
'ABC' < 'XYZ'
'A' < 'XYZ'
'X' > 'ABC'
'12345' < '66'
'abc' > 'ABC'
'abc 123' < 'abc123'
```

3.5. Expressões Booleanas

Em uma expressão booleana os operandos são do tipo `boolean` , assim como o resultado da sua avaliação.

Operador de Negação

O operador de negação `not` é unário e retorna `true` quando o operando sobre o qual é aplicado é `false`. Caso contrário, retorna `false`.

Operador Multiplicativo

O operador multiplicativo `and` é binário e retorna `true` quando ambos os operandos contêm o valor `true`. Caso contrário, retorna `false`.

Operadores Aditivos

a) OU (`or`)

Retorna `true` quando algum dos operandos for `true`.

b) OU exclusivo (`xor`)

Retorna `true` quando apenas um dos operados for `true`.

Avaliação Completa e Parcial

Freqüentemente, o resultado de uma expressão booleana pode ser obtido antes de avaliar o restante da expressão. Quando isso ocorre, é possível otimizar a avaliação da expressão. Essa forma de avaliação é conhecida como avaliação **parcial** ou **curto circuito** (*short circuit*) e é o padrão (*default*) do Turbo Pascal.

É possível usar a avaliação **completa** através da diretiva de compilação `{$B+}`, como mostra o exemplo abaixo:

```
program Ex2Cap3;
var
   X : byte;
BEGIN
   write('Digite um numero qualquer: ');
   readln(X);

   {$B+} { Habilita a avaliacao completa }
   if (X mod 2 = 0) and (X mod 3 = 0) then
      writeln('Numero divisivel por 2 e 3');
END
```

Para habilitar a avaliação parcial, utiliza-se a diretiva de compilação `{$B-}`.

3.6. Expressões Constantes

São expressões que podem ser avaliadas em tempo de compilação, ou seja, seu resultado é conhecido mesmo sem executar o programa.

Em uma expressão constante não deve haver referência a variáveis ou funções, excetuando-se as funções predefinidas: `abs`, `chr`, `hi`, `length`, `lo`, `odd`, `ord`, `pred`, `round`, `sizeof`, `succ`, `swap` e `trunc`.

Exemplo:

```
program Ex3Cap3;
const
   Y   : char = succ('X');
   Tam : byte = length('ABC123');
   Dez : byte = 110 mod 50;
BEGIN
   writeln('Y   = ', Y);
   writeln('Tam = ', Tam);
   writeln('Dez = ', Dez);
END.
```

4. Funções e Procedimentos do Turbo Pascal

4.1. Funções

Uma função é um conjunto de comandos que realizam alguma operação e retornam um resultado. O resultado de uma função pode ser utilizado em uma expressão da mesma forma que uma variável ou constante.

O Turbo Pascal traz um conjunto de funções predefinidas, das quais podemos destacar as seguintes:

abs(X)

Retorna o valor absoluto do número passado (seja ele inteiro ou real).

```
abs(-2.5) => retorna 2.5
abs(-100) => retorna 100
```

chr(X: integer): char

Retorna o caractere correspondente ao código ASCII fornecido como argumento.

```
chr(70) => retorna 'F'
chr(33) => retorna '!'
```

copy(S: string; Index: integer; Count: integer): string

Retorna uma substring da string S contendo o número de caracteres especificados por Count partindo da posição Index.

Quando o valor de Index é maior que o tamanho da string, é retornada uma string vazia.

Quando Count especifica um número de caracteres superior ao que pode ser obtido, é retornada substring que começa na posição Index e termina no final da string passada.

```
copy('12345', 1, 3) => Retorna '123'
copy('12345', 2, 4) => Retorna '2345'
copy('12345', 6, 1) => Retorna ''
copy('12345', 5, 2) => Retorna '5'
```

cos(X: real): real

Retorna o coseno do ângulo especificado (em radianos).

```
cos(0)      => Retorna 1
cos(PI/2)   => Retorna 0
```

```
cos(PI)     => Retorna -1
cos(3*PI/2) => Retorna 0
```

exp(X: real): real

Retorna o valor da base dos logaritmos naturais elevado a X, ou seja, e^X.

```
exp(0) => Retorna 1
exp(1) => Retorna 2.7182818285e00
exp(2) => Retorna 7.3890560989e00
```

frac(X: real): real

Retorna a parte fracionária de um número real.

```
frac(100.5) => Retorna 0.5
frac(-9.9)  => Retorna -0.9
```

hi(X): byte

Retorna o byte de mais alta ordem do parâmetro passado.

```
hi(255)   => Retorna 0
hi(65535) => Retorna 255
```

high(X)

Retorna o maior valor que o parâmetro pode assumir. O parâmetro pode ser uma variável ou um tipo escalar ordinal.

```
type
   dia_util = (SEG, TER, QUA, QUI, SEX);
var
   dia : dia_util;
...
high(dia)     => Retorna SEX
high(byte)    => Retorna 255
high(longint) => Retorna 2147483647
```

int(X: real): real

Retorna a parte inteira do argumento passado.

```
var
   X : real;
...
X := 123.45;
int(X) => Retorna 123.0
```

length(S: string): integer

Retorna o comprimento dinâmico de uma string.

```
length('abc'); => Retorna 3
length('');    => Retorna 0
```

ln(X: real): real

Retorna o logaritmo natural do número passado como parâmetro.

```
ln(1)      => Retorna 0
ln(10)     => Retorna 2.3025850930
ln(exp(1)) => Retorna 1
```

lo(X): byte

Retorna o byte de mais baixa ordem do parâmetro passado.

```
lo(255)   => Retorna 255
lo(65535) => Retorna 255
```

low(X)

Retorna o menor valor que o parâmetro pode assumir. O parâmetro pode ser uma variável ou um tipo escalar ordinal.

```
type
   dia_util = (SEG, TER, QUA, QUI, SEX);
var
   dia : dia_util;
...
low(dia)     => Retorna SEG
low(byte)    => Retorna 0
low(longint) => Retorna -2147483648
```

odd(X: longint): boolean

Retorna `true` quando o parâmetro passado é um número é ímpar.

```
odd(1) => true
odd(2) => false
odd(0) => false
```

ParamCount: word

Retorna o número de parâmetros passados para programa na linha de comando.

ParamStr(Index): string

Retorna um dos parâmetro especificados na linha de comando.

O parâmetro que está na posição 0 é o caminho para o arquivo executável.

Para especificar na linha de comando parâmetros que contêm espaços em branco, pode-se utilizar aspas como um delimitador (Ex: "A B C").

```
program Ex1Cap4;
var
   I : word;
BEGIN
   for I := 0 to ParamCount do
      writeln(ParamStr(I));
END.
```

pi: real

Retorna o valor de PI (3.1415926535897932385)

pos(Substr: string; S: string): byte

Busca por uma substring em uma string, retornando a posição a partir da qual se encontra. Caso não encontre, retorna 0.

```
Pos('ABC', '123ABC') => Retorna 4
Pos('123', '123ABC') => Retorna 1
Pos('X', '123ABC')   => Retorna 0
```

pred(X)

Retorna o predecessor do parâmetro passado. Funciona apenas com tipos escalares ordinais.

```
type
   Dia_Util = (SEG, TER, QUA, QUI, SEX);
const
   dia : Dia_Util = QUI;
...
succ(100) => Retorna 101
succ('B') => Retorna 'C'
succ(dia) => Retorna SEX
pred(100) => Retorna 99
pred('B') => Retorna 'A'
pred(dia) => Retorna QUA
```

random [(X: word)]

Retorna um número aleatório. A função pode ter tipos de retorno diferentes:

- Quando não é especificado um parâmetro, a função retorna um número maior ou igual a zero e menor do que 1.

- Quando é especificado um parâmetro inteiro (X), a função retorna um valor maior ou igual a zero e menor do que X.

Obs: Antes de chamar a função `random` é necessário chamar o procedimento `randomize` ou atribuir um valor à variável `randSeed`. Para um mesmo valor atribuído a `randSeed` há a garantia de que a seqüência de números aleatórios será sempre a mesma.

```
program Ex2Cap4;
const Max = 5;
var
   I : word;
BEGIN
   randomize;

   for I := 0 to 10 do
      writeln(random:1:10);

   for I := 0 to 10 do
      writeln(random(Max));
END.
```

```
program Ex3Cap4;
const Max = 5;
var
   I : word;
BEGIN
   randSeed := 1;

   for I := 0 to 10 do
      writeln(random:1:10);

   for I := 0 to 10 do
      writeln(random(Max));
END.
```

round(X: real): longint

Retorna um número inteiro que é resultado do arredondamento do parâmetro real especificado.

```
round(5.6)  => Retorna 6
round(5.5)  => Retorna 6
round(5.49) => Retorna 5
```

sin(X: real): real

Retorna o seno do ângulo especificado (em radianos).

```
sin(0)      => Retorna 0
sin(PI/2)   => Retorna 1
sin(PI)     => Retorna 0
sin(3*PI/2) => Retorna -1
```

SizeOf(X): integer

Retorna o número de bytes ocupados pelo parâmetro.

```
type
   STR50 = string[50];
var
   X : real;
...
SizeOf(X)       => Retorna 6
SizeOf(integer) => Retorna 2
SizeOf(string)  => Retorna 256
SizeOf(STR50)   => Retorna 51
```

sqr(X)

Retorna o número passado elevado ao quadrado.

```
sqr(3)   => Retorna 9
sqr(2.5) => Retorna 6.25
```

sqrt(X)

Retorna a raiz quadrada do número passado.

```
sqrt(9)    => Retorna 3
sqrt(6.25) => Retorna 2.5
```

succ(X)

Retorna o sucessor do parâmetro passado. Funciona apenas com tipos escalares ordinais.

swap(X)

Coloca os bytes de baixa ordem no lugar dos de alta ordem e vice-versa.

```
swap(256) => Retorna 1
swap(2)   => Retorna 512
```

trunc(X: real): longint

Trunca o número real passado, retornando um número inteiro.

```
trunc(0.5)    => Retorna 0
trunc(1.9999) => Retorna 1
trunc(2.5)    => Retorna 2
```

UpCase(Ch: char): char

Converte um caractere em seu correspondente maiúsculo. O mesmo caractere é retornado quando se tratar de um caractere maiúsculo ou especial.

```
UpCase('a') => Retorna 'A'
UpCase('Z') => Retorna 'Z'
UpCase('+') => Retorna '+'
```

4.2. Procedimentos

Uma procedimento é um conjunto de comandos que realizam alguma operação sem retornar um resultado. Quando é necessário retornar algum valor produzido pelo procedimento, pode-se fazer através de parâmetros passados por variável (referência).

O Turbo Pascal traz um conjunto de procedimentos predefinidos, dos quais podemos destacar as seguintes:

dec(var X [; N: longint])

Subtrai (decrementa) do valor de X o valor especificado em N ou 1 caso N não seja informado.

Gera um código mais otimizado que os comandos equivalentes X := X - N ou X := X - 1.

delete(var S: string; Index: integer; Count: integer)

Remove uma substring de uma string. S é a string cujo conteúdo será modificado. Index contém a posição inicial e Count o tamanho máximo da substring que será retirada da string.

```
program Ex4Cap4;
var
   S : string;
BEGIN
   S := 'abc123xyz';
   delete(S, 7, 2); { S = 'abc123z' }
   writeln(S);
   delete(S, 7, 1); { S = 'abc123' }
   writeln(S);
   delete(S, 7, 1); { S = 'abc123' }
   writeln(S);
   delete(S, 1, 3); { S = '123' }
   writeln(S);
END.
```

exit

Faz com que o bloco corrente seja terminado imediatamente. O programa só encerrado usando `exit` quando o bloco corrente é o próprio programa principal.

FillChar(var X; Count: word; Value)

Preenche a memória usando o valor especificado em **Value**, iniciando a partir do endereço da variável **X** e repetindo o valor o número de vezes definido por **Count**. **Value** pode ser tanto do tipo `byte` como `char`.

```
program Ex5Cap4;
var
   Linha : string;
BEGIN
   FillChar(Linha[0], 81, '-');
   Linha[0] := CHR(80);
   write(Linha);
END.
```

halt [(Exitcode: word)]

Provoca a saída imediata do programa retornando ao sistema operacional.

Pode ser especificado um valor de retorno para o sistema operacional como uma forma de indicar o tipo do erro ocorrido. Em geral, qualquer valor diferente de zero indica um erro.

inc(var X [; N: longint])

Adiciona (incrementa) ao valor de X o valor especificado em N ou 1 caso N não seja informado.

Gera um código mais otimizado que os comandos equivalentes X := X + N ou X := X + 1.

insert(Source: string; var S: string; Index: integer)

Insere uma substring em uma string. A variável **Source** contém a substring que será inserida na string **S** a partir do índice especificado por **Index**.

```
program Ex6Cap4;
var
   Nome : string;
BEGIN
   Nome := 'Alberto Neto';
   Insert(' Costa', Nome, 8);
   writeln(Nome);  { Imprime 'Alberto Costa Neto' }
END.
```

move(var Source, Dest; Count: word)

Copia um número especificado de bytes contíguos (**Count**) de **Source** para **Dest**.

```
program Ex7Cap4;
var
   C : char;
   I : byte;
BEGIN
   I := 0;
   C := 'A';
   move(C, I, 1);
   writeln(I); { Imprime 65 }
END.
```

str(X[: Width[: Decimals]]; var S: string)

Converte o número **X** para uma string usando a mesma representação que pode ser usada no `Write`.

```
program Ex8Cap4;
var
   X      : real;
   S1, S2 : string;
BEGIN
   X := 12345.6789;
   str(X:5:2, S1);
   str(X:1:0, S2);
   writeln(X);   { Imprime  1.2345678900E+04 }
   writeln(S1);  { Imprime 12345.68 }
   writeln(S2);  { Imprime 12346 }
END.
```

val(S; var V; var Code: integer)

Converte a string **S** em um número e armazena na variável **V**. Se a conversão for bem sucedida **Code** contém o valor 0. Caso contrário (a string não puder ser convertida para um número) **Code** é diferente de 0.

```
program Ex9Cap4;
var
   Num_Int  : integer;
   Num_Real : real;
   S1, S2   : string;
   Code     : integer;
BEGIN
   write('Entre com um numero inteiro: ');
   readln(S1);
   val(S1, Num_Int, Code);
   if Code <> 0 then
      writeln('Numero invalido (Erro:', Code, ')')
```

```
   else
      writeln('Numero valido (', Num_Int, ')');

   write('Entre com um numero real: ');
   readln(S2);
   val(S2, Num_Real, Code);
   if Code <> 0 then
      writeln('Numero invalido (Erro:', Code, ')')
   else
      writeln('Numero valido (', Num_Real, ')');
END.
```

4.3. Procedimentos de Entrada

O Pascal traz dois procedimentos para entrada de dados: `read` e `readln`.

Os procedimentos `read` e `readln` recebem uma lista de variáveis. A essas variáveis são atribuídos os dados lidos da unidade de entrada (como padrão é o teclado).

O procedimento `readln`, após fazer a leitura dos dados correspondentes às variáveis, ignora os dados restantes da linha e passa para a próxima linha.

O procedimento `read`, após fazer a leitura dos dados correspondentes às variáveis, continua na posição onde a leitura foi encerrada.

Leitura de caracteres

A uma variável do tipo `char` é atribuído o próximo caractere da linha corrente.

Leitura de strings

Quando é especificada uma variável do tipo `string`, a leitura é iniciada no próximo caractere da linha corrente e é encerrada quando se alcança o tamanho máximo da `string` ou se chega ao final da linha.

Leitura de números

Na leitura de dados numéricos todos os espaços em branco à esquerda são ignorados. Antes do primeiro dígito pode vir um sinal (`+` ou `-`). A leitura é encerra quando é encontrado um espaço em branco ou ao atingir o final da linha.

Exemplo: Lendo números, caracteres e strings

```
program Ex10Cap4;
var
   R    : real;
   X, Y : integer;
   C    : char;
   Str  : string[5];
BEGIN
   write('R (real):');
   readln(R);
   write('X e Y (integer):');
   readln(X,Y);
   write('C (char):');
   readln(C);
   write('Str (string[5]):');
   readln(Str);
   writeln;
   writeln('R=', R, ' | X=', X, ' | Y=', Y,
           ' | C=', C, ' | Str=', Str);
END.
```

Alteração da entrada padrão para arquivo

Para que a entrada de dados seja feita através da leitura de um arquivo podem ser utilizados os comandos:

```
assign(input, '<nome-do-arquivo>');
reset(input);
```

O arquivo especificado no comando `assign` deve existir para não provocar um erro durante a execução do programa.

A leitura do arquivo é feita da mesma forma que pelo teclado, isto é, através dos procedimentos `read` e `readln`.

Existe uma função chamada `eof` que retorna `true` se já foi atingido o final do arquivo. Quando se lê dados do teclado e as teclas CTRL + Z são pressionadas, a função `eof` também retorna `true`.

Para voltar a ler dados do teclado no mesmo programa é necessário usar os seguintes comandos:

```
assign(input, '');
reset(input);
```

Exemplo: Lendo de um arquivo os dados dos clientes de um banco

```
program Ex11Cap4;
const
   Num_Clientes : word = 0;
   Soma         : real = 0;
var
   Nome  : string[30];
   Conta : string[6];
   Saldo : real;
   Linha : string;
BEGIN
   assign(input, 'Ex11.dat');
   reset(input);

   if eof then
      exit;

   FillChar(Linha[1], 48, '-');
   Linha[0] := Chr(48);

   writeln('Cliente', ' ':24,'Conta', ' ':7, 'Saldo');
   writeln(Linha);

   while not eof do
   begin
      readln(Nome, Conta, Saldo);
      writeln(Nome:30, ' ', Conta:6, ' ', Saldo:10:2);
      Soma := Soma + Saldo;
      inc(Num_Clientes);
   end;

   writeln(Linha);
   writeln('Saldo Medio = ', (Soma/Num_Clientes):4:2);
END.
```

4.4. Procedimentos de Saída

O Pascal traz dois procedimentos para saída de dados: `write` e `writeln`.

Os procedimentos `write` e `writeln` recebem uma lista de variáveis as quais terão seus valores direcionados para a unidade de saída (como padrão é o vídeo).

Após direcionar o conteúdo de todas as variáveis para a saída, o procedimento `writeln` posiciona a unidade de saída no início da próxima linha. Já o procedimento `write` faz com que a unidade de saída permaneça posicionada logo após o último valor.

Tanto no `write` como no `writeln` é possível especificar o comprimento mínimo e o número de casas decimais (reais) ocupado pela valor.

Exemplo:

```
program Ex12Cap4;
BEGIN
   writeln('xyz':4);     { ' xyz' }
   writeln(54321:6);     { ' 54321' }
   writeln(54321:2);     { '54321' }
   writeln(2.5:3);       { ' 2.5E+00'}
   writeln(-2.5:3);      { '-2.5E+00'}
   writeln(2.5:10);      { '  2.500E+00'}
   writeln(2.5:5:2);     { ' 2.50' }
   writeln(-2.5:5:2);    { '-2.50' }
   writeln(' ':4);       { '    ' }
   writeln('-':4);       { '   -' }
END.
```

Alteração da saída padrão para arquivo

Para que a saída de dados seja direcionada para um arquivo podem ser utilizados os comandos:

```
assign(output, '<nome-do-arquivo>');
rewrite(output);
```

Se o arquivo especificado no comando `assign` já existir, o comando `rewrite` apaga e criaa um novo arquivo, deixando-o aberto.

Para voltar a imprimir dados no vídeo no mesmo programa é necessário usar os seguintes comandos:

```
assign(output, '');
rewrite(output);
```

Exemplo:

```
program Ex13Cap4;
BEGIN
   assign(output, 'Ex13.dat');
   rewrite(output);
   writeln('Joao da Silva                 1234-5  1.00');
   writeln('Jose da Silva                 4321-0 10.00');
   writeln('Maria da Silva                9876-5100.00');
END.
```

Enviando dados para a impressora

O Turbo Pascal possui uma `unit` chamada `printer`. Esta `unit` define um identificador `lst` que é um arquivo texto associado com a porta LPT1.

Através dos procedimentos `write` e `writeln` é possível enviar dados para a impressora. A única diferença em relação ao vídeo é a obrigatoriedade de fornecer como primeiro argumento o identificador `lst`.

Existem alguns caracteres que servem como comandos para a impressora:

#10: Faz a mudança de linha.

#12: Provoca uma quebra de págin.

#13: Faz o retorno do carro

Exemplo: Imprimindo um arquivo texto.

```
program Ex14Cap4;
uses printer;
const
   LINHAS_POR_PAGINA = 10;
var
   Nome_Arq             : string;
   Linha                : string[70];
   Num_Linhas, Num_Pags : word;
BEGIN
   write('Entre com o nome do arquivo: ');
   readln(Nome_Arq);
   assign(input, Nome_Arq);
   reset(input);

   Num_Linhas := 0;
   Num_Pags   := 0;

   while not eof do
   begin
      if Num_Linhas mod LINHAS_POR_PAGINA = 0 then
         begin
            if Num_Pags <> 0 then
               write(lst, #12);

            inc(Num_Pags);
            writeln(lst,'Pag:', Num_Pags:2, '(', Nome_Arq, ')',#10,#13);
         end;

      inc(Num_Linhas);
      readln(Linha);
      writeln(lst, Num_Linhas:5, ': ', Linha);
   end;

   if Num_Linhas > 0 then
      writeln(lst, #12);
END.
```

5. Comandos do Turbo Pascal

5.1. Tipos de Comandos

Podemos classificar os comandos em comandos simples e estruturados. Os primeiros são comandos que não contêm outros comandos. Os comandos estruturados podem conter outros comandos.

Podemos dividir os comandos estruturados em três categorias: Composto (executados em seqüência), Condicional (executados condicionalmente) e Repetitivos (executados repetitivamente).

- Comandos Simples: Atribuição, Procedimento, Goto e Vazio.
- Comandos Estruturados
  - Composto
  - Condicionais: `if`, `case`
  - Repetitivos: `for`, `while`, `repeat`, `break` e `continue`

5.2. Comando de Atribuição

Permite atribuir um valor a uma variável. Sintaxe:

```
<var> := <expressão>
```

Para que expressão seja compatível com uma variável, ou seja, é possível atribuir seu resultado à variável, é necessário que uma das condições abaixo seja satisfeita:

- O resultado da expressão é do mesmo tipo da variável;
- A variável é um número real e a expressão retorna um número inteiro;
- A variável é um subintervalo compatível com o resultado da expressão.

5.3. Comando Procedimento

Consiste na chamada de um procedimento predefinido ou definido pelo usuário. Sintaxe:

```
<nome-do-procedimento> [(<arg>[,<arg>]...)]
```

5.4. Comando goto

Apesar da linguagem Pascal trazer esse comando, deve-se evitar utilizá-lo.

5.5. Comando Vazio

Um comando vazio consiste na ausência de um comando onde poderia haver algum comando. Não provoca qualquer efeito no programa. É útil, por exemplo para eliminar a ambigüidade em comandos `if`.

5.6. Comando Composto

É um conjunto de comandos que devem ser executados de forma seqüencial. Em Sintaxe:

```
begin
   <comando>[;<comando>]...
end
```

Exemplo:

```
if Resp = 'S' then
      begin
         write('Deseja realmente sair (S/N)? ');
         readln(Resp);
         if Resp = 'S' then
            halt
      end
```

5.7. Comando If

O comando `if` permite condicionar a execução de um comando à avaliação de uma expressão. Além disso, é possível selecionar um dos dois comandos disponíveis a depender do resultado da expressão. Sintaxe:

```
if <expressão-booleana> then
   <comando>
[else
   <comando>]
```

Exemplos:

```
if N1 > N2 then
   Maior := N1
```

```
if Sexo = 'M' then
   writeln('Masculino')
else
   writeln('Feminino')
```

```
readln(C);
C := UpCase(C);
if C <> 'S' then
   begin
      if C = 'N' then
         writeln('Nao')
   end
else
   writeln('Sim')
```

O exemplo abaixo mostra como o comando vazio ajuda a eliminar a ambigüidade que é tratada no exemplo acima com um comando composto:

```
readln(C);
C := UpCase(C);
if C <> 'S' then
   if C = 'N' then
      writeln('Nao')
   else
      { Comando vazio }
else
   writeln('Sim')
```

5.8. Comando Case

Consiste de uma expressão e uma lista de comando. Cada comando é precedido por constantes ou subintervalos separados por vírgulas (rótulos de case) do mesmo tipo da expressão. A expressão deve retornar um tipo escalar ordinal com 1 ou 2 bytes.

Ao executar o `case` primeiro avalia-se a expressão. Em seguida, procura-se de cima para baixo por um rótulo que contenha um valor igual ao resultado da avaliação da expressão e executa-se o comando associado. Se não for encontrado nenhum rótulo com o valor há duas alternativas: (a) executar o comando contido no `else` (caso exista) e encerrar o `case`; (b) sair do `case` sem executar comandos.

Sintaxe do comando `case`:

```
case <exp> of
   <rótulo-do-case> : <comando>;
  [<rótulo-do-case> : <comando>;]...
  [else
      <comando> [;<comando>]...
end
```

Sintaxe do rótulo do `case`:

```
<const>[..<const>][,<const>[..<const>]]...
```

Exemplo: Permite executar as operações de soma, subtração, multiplicação e divisão sobre números inteiros.

```
program Ex1Cap5;
var
   A, B : integer;
   Op   : char;
BEGIN
   writeln('Entre com uma expressao (+-/*)');
   readln(A);
   readln(Op);
   readln(B);

   write(A, Op, B, ' = ');

   case Op of
      '+' : writeln(A + B);
      '-' : writeln(A - B);
      '*' : writeln(A * B);
      '/' : writeln(A / B);
      else
         writeln('expressao invalida');
   end;
END.
```

5.9. Comando For

É o comando de repetição mais apropriado quando se conhece o número de iterações antecipadamente. Sintaxe:

```
for <var> := <exp1> {to|downto} <exp2> do
   <comando>
```

`<var>` é a variável de controle cujo valor será incrementando (ou decrementado quando downto for usado). Os incrementos e decrementos têm, respectivamente, o mesmo resultado da execução das funções `succ` e `pred`.

As expressões `<exp1>` e `<exp2>` são avaliadas no início da execução do laço e funcionam, respectivamente, como os valores inicial e final da variável `<var>`.

Apesar de ser possível modificar o valo da variável de controle durante a execução do laço, isso não é aconselhável.

Exemplo: Escreve as letras do nome digitado em maiúsculas e separadas por um espaço em branco.

```
program Ex2Cap5;
var
   Nome : string[30];
   I, J : byte;
BEGIN
   writeln('Entre com o seu nome (ate 30 caracteres)');
   readln(Nome);

   for I := 1 to 5 do
   begin
      writeln;
      for J := 1 to length(Nome) do
         write(UpCase(Nome[J]), ' ');
      writeln;
      for J := length(Nome) downto 1 do
         write(UpCase(Nome[J]), ' ');
   end;
END.
```

5.10. Comando While

É um comando de repetição faz com que o comando seja executado enquanto sua condição for verdadeira (`true`). Sintaxe:

```
while <exp> do
   <comando>
```

A expressão booleana `<exp>` é avaliada antes do início de cada iteração.

Exemplo: Gera um número aleatório e dá ao usuário 3 chances de adivinhá-lo.

```
program Ex3Cap5;
const
   CHANCES = 3;
var
   Num_Aleatorio, Tot_Chutes, Chute : byte;
   Acertou : boolean;
BEGIN
   randomize;
   Num_Aleatorio := random(10);

   Tot_Chutes := 0;
   Acertou := false;

   while (Tot_Chutes < CHANCES) and (not Acertou) do
   begin
      write('Chute outro numero entre 0 e 9:');
      readln(Chute);
      inc(Tot_Chutes);
```

```
      if Chute = Num_Aleatorio then
         Acertou := true;
   end;

   if Acertou then
      writeln('Parabens! Voce precisou de ', Tot_Chutes, ' chutes');
END.
```

5.11. Comando Repeat

É um comando de repetição faz com que os comandos sejam executados até que uma condição seja verdadeira (`true`). Sintaxe:

```
repeat
   <comando>[;<comando>]...
until <exp>
```

A expressão booleana `<exp>` é avaliada no final de cada iteração. Isso faz com que os comandos sejam executados pelo menos uma vez.

Exemplo: Solicita número inteiros e calcula a soma dos mesmos.

```
program Ex4Cap5;
const
   Soma : longint = 0;
var
   Numero : integer;
   Resp : char;
BEGIN
   repeat
      write('Digite um numero inteiro:');
      readln(Numero);
      inc(Soma, Numero);
      write('Continua (S/N)?');
      read(Resp);
   until UpCase(Resp) = 'N';

   writeln('Soma dos numeros digitados = ', Soma);
END.
```

5.12. Comando Break

Pode ser usado dentro de um comando de repetição e provoca a saída incondicional do laço mais interno. Não está disponível do Pascal padrão.

Utilização do `break`:

```
while <exp1> do
begin
   ...
   repeat
      ...
      break;
      ...
   until <exp2>;
   {desvia para este ponto}
   ...
end
```

5.13. Comando Continue

Pode ser usado dentro de um comando de repetição e provoca um desvio para o final do laço mais interno. Não está disponível do Pascal padrão.

Utilização de `continue` dentro de um `for`:

```
for I := <exp1> {to|downto} <exp2> do
begin
   ...
   continue;
   ...
   {desvia para para este ponto}
end
```

Utilização de `continue` dentro de um `while`:

```
while <exp> do
begin
   ...
   continue;
   ...
   {desvia para para este ponto}
end
```

Utilização de `continue` dentro de um `repeat`:

```
repeat
   ...
   continue;
   ...
   {desvia para para este ponto}
until <exp>
```

6. Arrays

6.1. Introdução

É um tipo de dado estruturado que define um número fixo de elementos, todos do mesmo tipo de dado.

Cada elemento do *array* é referenciado através de um índice (ou subscrito).

Os *arrays* podem ser classificados em unidimensionais e multidimensionais. No primeiro, também chamado de vetor, é necessário apenas um índice para referenciar um elemento. O segundo exige um índice para cada dimensão. Quando um *array* tem apenas duas dimensões é chamado de matriz.

6.2. Sintaxe de Definição do Tipo Array

```
array [<tipo-do-índice>[,<tipo-do-índice>]...] of <tipo-do-dado>
```

Onde:

- **tipo-do-índice** pode ser o identificador de um tipo escalar ordinal, um tipo enumerado ou um subintervalo.

- **tipo-do-dado** é o tipo de dado de todos os elementos do *array*.

Observação: Quando definimos uma variável do tipo `string[n]` o Turbo Pascal trata a variável da mesma forma que um `array[0..n] of char`, no qual o primeiro elemento é usado como byte de comprimento.

6.3. Referenciando os Elementos de um Array

Para referenciar um elemento de um *array* é necessário informar o nome da variável, seguindo de uma lista de índices (delimitada por colchetes) separados por ',' (caso seja multidimensional).

Exemplo:

```
program Ex1Cap6;
type
   Vetor_Inteiros = array [byte]            of integer;
   Matriz_Strings = array ['A'..'Z',1..100] of string[20];
   Dia_da_Semana  = (SEG, TER, QUA, QUI, SEX, SAB, DOM);
   Valor_por_Dia  = array [Dia_da_Semana] of real;
var
```

```
   i : byte;
   Vet1, Vet2    : Vetor_Inteiros;
   Mat_Nomes      : Matriz_Strings;
   Horas_Trab : Valor_por_Dia;
BEGIN
   Vet1[0] := 0;

   for i := 1 to 255 do
      Vet1[i] := i;

   Vet2 := Vet1;

   for i:= 255 downto 0 do
      write(Vet2[i], ' ');

   Mat_Nomes['A',1] := 'Alberto';
   Mat_Nomes['A',2] := 'Antonio';
   Mat_Nomes['J',1] := 'Jose';
   Mat_Nomes['D',9] := 'Daniela';

   Horas_Trab[SEG] :=  8.0;
   Horas_Trab[TER] := 10.0;
   Horas_Trab[QUA] :=  8.5;
   Horas_Trab[QUI] :=  9.0;
   Horas_Trab[SEX] :=  Horas_Trab[SEG];
   Horas_Trab[SAB] :=  4.0;
   Horas_Trab[DOM] :=  0.0;
END.
```

6.4. Arrays Constantes

Seguem o mesmo conceito das constantes escalares, mas como estão associadas a uma coleção de valores, são denominadas estruturadas.

Os *arrays* constantes são definidos da seguinte forma:

```
const
   <ident-da-const> : <tipo-do-array> = (<const>[,<const>]...)
```

Onde:

- **ident-da-const** é o identificador associado ao *array*
- **tipo-do-array** é tipo do *array* constante
- **const** são os valores dos elementos do *array*

Exemplo:

```
program Ex2Cap6;
type
   ArrayBid = array [1..3,1..3] of integer;
const
   Matriz : ArrayBid = ( (1,0,0),
                         (0,1,0),
                         (0,0,1) );
var
   I, J : byte;
BEGIN
   for I := 1 to 3 do
   begin
      for J := 1 to 3 do
         write(Matriz[I,J], ' ');
      writeln;
   end
END.
```

6.5. Pesquisa Seqüencial

Uma das atividades mais comuns em computação é pesquisar por valores.

A pesquisa seqüencial em *array* consiste em buscar por um elemento a partir do início do mesmo até que o elemento seja encontrado ou o último elemento do *array* tenha sido atingido.

Exemplo: Pesquisa seqüencial em um *array* que contém dados lidos de um arquivo.

```
program Ex3Cap6;
const
   Tam_Max = 1000;
type
   TApelido = string[10];
var
   Apelidos : array [1..Tam_Max] of TApelido;
   Apelido  : TApelido;
   N, I     : integer;
   Achou    : boolean;
BEGIN
   write('Digite o apelido que deseja procurar: ');
   readln(Apelido);

   assign(input, 'Ex3.dat');
   reset(input);

   N := 0;
   while not eof do
   begin
      inc(N);
      readln(Apelidos[N]);
```

```
   end;

   if N = 0 then
      begin
         writeln('O arquivo esta vazio');
         exit
      end;

   Achou := false;
   for I := 1 to N do
      if Apelido = Apelidos[I] then
         begin
            Achou := true;
            break;
         end;

   if Achou then
      writeln('O apelido ja existe')
   else
      writeln('O apelido nao existe');
END.
```

6.6. Classificação

Classificar informações é uma atividade muito importante na computação. Quando os dados estão classificados, a busca pode ser feita de forma mais eficiente.

Há vários métodos de classificação, dentre os quais será mostrado o de Seleção Direta, o qual é bastante simples (porém ineficiente).

Este método consiste em buscar pelo menor valor e colocá-lo na primeira posição do *array*. Em seguida, o menor valor do conjunto desordenado restante é encontrado e colocado na segunda posição. Esses passos são repetidos até que não haja um único elemento no conjunto desordenado.

Dado um *array* contendo os valores 4, 3, 1 e 2, teríamos os seguintes passos:

P1	**P2**	**P3**	**P4**
4	3	1	2

Passo 1: Assume-se que o elemento em P1 (P1 é posição 1) é o menor valor da sublista desordenada que começa em P1. Procura-se, a partir de P2, algum valor menor do que o contido em P1. Troca-se o menor valor encontrado (no caso P3) pelo contido em P1.

P1	P2	P3	P4
1	3	4	2

Passo 2: Assume-se que o elemento em P2 é o menor valor da sublista desordenada que começa em P2. Procura-se, a partir de P3, algum valor menor do que o contido em P2. Troca-se o menor valor encontrado (no caso P4) pelo contido em P2.

P1	P2	P3	P4
1	2	4	3

Passo 3: Assume-se que o elemento em P3 é o menor valor da sublista desordenada que começa em P3. Procura-se, a partir de P4, algum valor menor do que o contido em P3. Troca-se o menor valor encontrado (no caso P4) pelo contido em P3.

P1	P2	P3	P4
1	2	3	4

Passo 4: Como a sublista desordenada só contém 1 (um) elemento, estando portanto ordenada, o *array* já está totalmente ordenado.

Exemplo: Lê os dados de um arquivo, coloca em um *array*, ordena e imprime.

```
program Ex4Cap6;
const
   Tam_Max = 1000;
type
   TItem = string[10];
var
   Itens    : array [1..Tam_Max] of TItem;
   ItemTemp : TItem;
   N, I, J,
   Menor    : integer;
BEGIN
   assign(input, 'Ex4.dat');
   reset(input);

   N := 0;
   while not eof do
   begin
      inc(N);
```

```
      readln(Itens[N]);
   end;

   for I := 1 to N - 1 do
   begin
      Menor := I;
      for J := I + 1 to N do
         if Itens[J] < Itens[Menor] then
            Menor := J;
      if Menor <> I then
         begin
            ItemTemp     :=  Itens[I];
            Itens[I]     := Itens[Menor];
            Itens[Menor] := ItemTemp;
         end
   end;

   for I := 1 to N do
      writeln(Itens[I]);
END.
```

7. Registros

Um registro (`record`) é um tipo de dado estruturado que pode conter elementos heterogêneos, diferentemente dos *arrays* que só armazenam elementos homogêneos.

Registros são muito úteis em situações onde é possível associar um nome a um conjunto de campos, como por exemplo: uma pessoa (registro) possui um CPF, um nome e telefones.

Sintaxe de declaração:

```
record
   <ident-do-campo> : <tipo-do-campo>
 [;<ident-do-campo> : <tipo-do-campo>]...
end
```

Exemplo: Declarando um tipo pessoa com CPF, Nome e Telefones

```
type
   Reg_Pessoa = record
                   CPF       : string[11];
                   Nome      : string[30];
                   Telefones : array[1..3] of string[10];
                end;
var
   Pessoa1,
   Pessoa2 : Reg_Pessoa;
```

7.1. Referenciando campos

Para referenciar um campo de um registro, é necessário informar o identificador do registro, seguido de um ponto e o identificador do campo. O resultado pode ser usado da mesma forma que uma variável do tipo definido no campo.

```
program Ex1Cap7;
type
   Reg_Pessoa = record
                   CPF       : string[11];
                   Nome      : string[30];
                   Data_Nasc : string[8];
                   Pre_Tels  : array[1..3] of string[2];
                   Num_Tels  : array[1..3] of string[8]
                end;
var
   Pessoa : Reg_Pessoa;
   I      : byte;
BEGIN
   Pessoa.CPF  := '12345678901';
   Pessoa.Nome := 'Alberto';
   Pessoa.Data_Nasc := '25122000';
   Pessoa.Pre_Tels[1] := '79';
   Pessoa.Num_Tels[1] := '12345678';
```

```
   Pessoa.Pre_Tels[2] := '71';
   Pessoa.Num_Tels[2] := '23456789';
   Pessoa.Pre_Tels[3] := '83';
   Pessoa.Num_Tels[3] := '34567890';
   writeln('CPF.: ', Pessoa.CPF);
   writeln('Nome: ', Pessoa.Nome);
   writeln('Data: ', copy(Pessoa.Data_Nasc, 1, 2), '/',
                     copy(Pessoa.Data_Nasc, 3, 2), '/',
                     copy(Pessoa.Data_Nasc, 5, 4));
   for I := 1 to 3 do
      writeln('Tel[', I, ']: (', Pessoa.Pre_Tels[I] + ')' +
                                 Pessoa.Num_Tels[I]);
END.
```

7.2. Registros aninhados

Freqüentemente é interessante que um campo de um registro seja definido como um outro registro (registro aninhado). Para isso, basta usar o identificador do registro como tipo do campo durante a declaração do registro ao qual o campo pertence.

Exemplo: Redefinindo a data de nascimento como um registro.

```
program Ex2Cap7;
type
   Data = record
             Dia : string[2];
             Mes : string[2];
             Ano : string[4]
          end;
   Reg_Pessoa = record
                   CPF       : string[11];
                   Nome      : string[30];
                   Data_Nasc : Data;
                   Pre_Tels  : array[1..3] of string[2];
                   Num_Tels  : array[1..3] of string[8];
                end;
var
   Pessoa : Reg_Pessoa;
   I      : byte;
BEGIN
   Pessoa.CPF  := '12345678901';
   Pessoa.Nome := 'Alberto';
   Pessoa.Data_Nasc.Ano := '2000';
   Pessoa.Data_Nasc.Mes := '12';
   Pessoa.Data_Nasc.Dia := '25';
   Pessoa.Pre_Tels[1] := '79';
   Pessoa.Num_Tels[1] := '12345678';
   Pessoa.Pre_Tels[2] := '71';
   Pessoa.Num_Tels[2] := '23456789';
   Pessoa.Pre_Tels[3] := '83';
   Pessoa.Num_Tels[3] := '34567890';
   writeln('CPF.: ', Pessoa.CPF);
   writeln('Nome: ', Pessoa.Nome);
   writeln('Data: ', Pessoa.Data_Nasc.Dia, '/',
```

```
                     Pessoa.Data_Nasc.Mes, '/',
                     Pessoa.Data_Nasc.Ano);
   for I := 1 to 3 do
      writeln('Tel[', I, ']: (', Pessoa.Pre_Tels[I] + ')' +
                                 Pessoa.Num_Tels[I]);
END.
```

7.3. Arrays de Registros

Uma forma de evitar `arrays` paralelos é utilizar `arrays` contendo registros. Dessa forma, cada elemento do `array` é um registro, o pode conter vários campos.

Exemplo: Redefinindo os `arrays` de prefixos e de números de telefones para usar registros.

```
program Ex3Cap7;
type
   Data = record
             Dia : string[2];
             Mes : string[2];
             Ano : string[4]
          end;
   Telefone = record
                 Pre : string[2];
                 Num : string[8]
              end;
   Reg_Pessoa = record
                   CPF       : string[11];
                   Nome      : string[30];
                   Data_Nasc : Data;
                   Tels      : array[1..3] of Telefone
                end;
var
   Pessoa : Reg_Pessoa;
   I      : byte;
BEGIN
   Pessoa.CPF  := '12345678901';
   Pessoa.Nome := 'Alberto';
   Pessoa.Data_Nasc.Ano := '2000';
   Pessoa.Data_Nasc.Mes := '12';
   Pessoa.Data_Nasc.Dia := '25';
   Pessoa.Tels[1].Pre := '79';
   Pessoa.Tels[1].Num := '12345678';
   Pessoa.Tels[2].Pre := '71';
   Pessoa.Tels[2].Num := '23456789';
   Pessoa.Tels[3].Pre := '83';
   Pessoa.Tels[3].Num := '34567890';
   writeln('CPF.: ', Pessoa.CPF);
   writeln('Nome: ', Pessoa.Nome);
   writeln('Data: ', Pessoa.Data_Nasc.Dia, '/',
                     Pessoa.Data_Nasc.Mes, '/',
                     Pessoa.Data_Nasc.Ano);
   for I := 1 to 3 do
      writeln('Tel[', I, ']: (', Pessoa.Tels[I].Pre + ')' +
                                 Pessoa.Tels[I].Num);
```

```
END.
```

7.4. Usando o comando With

O comando `with` facilita a referências aos campos de um registro. É bastante útil quando é necessário acessar vários campos de um mesmo registro pois elimina a necessidade de informar o nome do registro antes do nome campo.

Sintaxe do comando `with`:

```
with <variável> [,<variável>]... do
   <comando>
```

Exemplo: Incluindo o comando `with` no programa de exemplo anterior

```
program Ex4Cap7;
type
   Data = record
             Dia : string[2];
             Mes : string[2];
             Ano : string[4]
          end;
   Telefone = record
                 Pre : string[2];
                 Num : string[8]
              end;
   Reg_Pessoa = record
                   CPF       : string[11];
                   Nome      : string[30];
                   Data_Nasc : Data;
                   Tels      : array[1..3] of Telefone
                end;
var
   Pessoa : Reg_Pessoa;
   I      : byte;
BEGIN
   with Pessoa, Data_Nasc do
   begin
      CPF  := '12345678901';
      Nome := 'Alberto';
      Ano  := '2000';
      Mes  := '12';
      Dia  := '25';
      Tels[1].Pre := '79';
      Tels[1].Num := '12345678';
      Tels[2].Pre := '71';
      Tels[2].Num := '23456789';
      Tels[3].Pre := '83';
      Tels[3].Num := '34567890';
      writeln('CPF.: ', CPF);
      writeln('Nome: ', Nome);
      writeln('Data: ', Dia, '/', Mes, '/', Ano);
      for I := 1 to 3 do
```

```
         with Tels[I] do
            writeln('Tel[', I, ']: (', Pre + ')' + Num);
   end
END.
```

Quando há uma colisão nos nomes dos campos, o que será acessado ou modificado é o mais à direita na lista de registros do comando `with`.

```
program Ex5Cap7;
type
   Data = record
             Dia : string[2];
             Mes : string[2];
             Ano : string[4]
          end;
var
   Data1, Data2 : Data;
BEGIN
   Data1.Dia := '01';
   Data1.Mes := '01';
   Data1.Ano := '2001';
   Data2.Dia := '02';
   Data2.Mes := '02';
   Data2.Ano := '2002';

   with Data1, Data2 do
   begin
      Dia := '03';
      Mes := '03';
      Ano := '2003';

      Data1.Ano := '1001';
   end;

   with Data1 do
      writeln('Data1: ', Dia, '/', Mes, '/', Ano); {Imprime 01/01/1001}
   with Data2 do
      writeln('Data2: ', Dia, '/', Mes, '/', Ano); {Imprime 03/03/2003}
END.
```

7.5. Registros com Variantes

Às vezes é necessário que os nomes e os tipos dos campos possam ser modificados. Uma alternativa para esse problema é declarar todos os campos necessários para que estejam disponíveis na situação em que são requeridos. Essa alternativa, porém, pode provocar muito desperdício de memória.

Uma outra abordagem é dividir um registro em duas partes: uma fixa e outra variante. Na parte fixa são colocados os campos que sempre são requeridos. Na parte variante são colocadas as combinações dos campos que sempre aparecem juntas na memória mas não simultaneamente.

Como exemplo, um registro que guarda informações sobre um cliente deve ser variante pois os dados necessários para um cliente pessoa física não são iguais aos de um cliente pessoa jurídica.

Parte Fixa	**Parte Variante**	
	Pessoa Física	Pessoa Jurídica
Nome	CPF Data de Nascimento	CPNJ

Sintaxe de definição de registros com variantes:

```
record
   <ident-campo> : <tipo-do-campo>
 [;<ident-campo> : <tipo-do-campo>]...
   case [<ident-do-tag> : ] <tipo-do-tag> of
      <const>[,<const>]... : (<ident-campo> : <tipo-do-campo>
                              [;<ident-campo> : <tipo-do-campo>]...);
     [<const>[,<const>]... : (<ident-campo> : <tipo-do-campo>
                              [;<ident-campo> : <tipo-do-campo>]...);]...
```

Onde:

- **ident-do-tag** indica qual parte variante está ativa.
- **tipo-do-tag** corresponde ao tipo do ident-do-tag. Pode ser qualquer tipo escalar ordinal.
- **const** indica o valor que o ident-do-tag deve assumir para ativar a parte variante.

O registro `Reg_Cliente` (definido abaixo) está organizado na memória da seguinte forma:

Parte Fixa	
Nome (string [30]) – 31 bytes	Tipo (char) – 1 byte

Parte Variante (com pessoa física)	
CPF (string [11]) – 12 bytes	Data_Nasc (string[8]) – 9 bytes

Parte Variante (com pessoa jurídica)	
CNPJ (string [14]) – 15 bytes	(não ocupado) – 6 bytes

Exemplo: Definindo um registro com variante para cliente (pessoa física ou jurídica) para fazer a leitura de um arquivo.

```
program Ex6Cap7;
type
   Reg_Cliente = record
                    Nome : string[30];
                    case Tipo : char of
                       'F': ( CPF       : string[11];
                              Data_Nasc : string[8] );
                       'J': ( CNPJ      : string[14] );
                 end;
const
   N : word = 0;
var
   Cliente : Reg_Cliente;
   Clientes_A : array[1..10] of Reg_Cliente;
   I : word;
BEGIN
   assign(input, 'ex6.dat');
   reset(input);

   if eof then
      exit;

   while not eof do
   begin
      read(Cliente.Nome, Cliente.Tipo);
      if Cliente.Tipo = 'F' then
         readln(Cliente.CPF, Cliente.Data_Nasc)
      else
         readln(Cliente.CNPJ);

      if Cliente.Nome[1] = 'A' then
      begin
         inc(N);
         Clientes_A[N] := Cliente;
      end;
   end;

   writeln('Nome', ' ':27, 'Tipo', ' ', 'CPF/CNPJ');
   writeln('-------------------------------------------------');
   for I := 1 to N do
      with Clientes_A[I] do
      begin
         write(Nome, ' ':1, Tipo, ' ':4);
         if (Tipo = 'F') then
            writeln(CPF)
         else
            writeln(CNPJ);
      end;
END.
```

Um registro com variante não precisa especificar um identificador para o campo de `tag` pois ele é usado apenas para indicar logicamente que parte variante está sendo usada. Entretanto, ele não impede o acesso aos campos de outra parte variante.

Exemplo: Usando um registro para visualizar um caractere como se fosse um número.

```
program Ex7Cap7;
type
   ByteChar = record
                 case char of
                    'C' : (Caractere : char);
                    'N' : (Numero    : byte);
              end;
var
   Num_Char : ByteChar;
BEGIN
   write('Digite uma letra: ');
   readln(Num_Char.Caractere);
   write('Codigo ASCII de (', Num_Char.Caractere, '): ');
   writeln(Num_Char.Numero);
END.
```

7.6. Registros Constantes

É possível definir registros constantes usando a seguinte sintaxe:

```
const
   <ident-da-const> : <tipo-do-reg> = (<ident-do-campo : <const>
                                      [;<ident-do-campo : <const>]...);
```

Exemplo: Definindo um valor padrão para um registro data

```
const
   Data_Padrao : Data = (Dia:'01'; Mes:'02'; Ano:'2003');
```

Exemplo: Definindo registros constantes para um registro com variante

```
const
   Cliente_PF : Reg_Cliente = (Nome:'Alberto';
                               Tipo:'F';
                               CPF:'12345678901';
                               Data_Nasc:'19450518');
   Cliente_PJ : Reg_Cliente = (Nome:'Alberto';
                               Tipo:'J';
                               CNPJ:'12345678901234');
```

8. Conjuntos

O tipo conjunto provêm da idéia de conjuntos em matemática. No Pascal há restrições em relações aos membros de um conjunto, que devem ser escalares ordinais de um mesmo tipo.

Sintaxe de declaração:

```
set of <tipo-base>;
```

Onde **tipo-base** é um tipo escalar enumerado, escalar ordinal ou um subintervalo.

Exemplos:

```
type
   Caracteres       = set of char;
   Carac_Maiusculos = set of 'A'..'Z';
   Digitos          = set of 0..9;
   Dias_da_Semana   = set of (Seg, Ter, Qua, Qui, Sex, Sab, Dom);
```

8.1. Criando conjuntos

Para definir conjuntos em Pascal utiliza-se o construtor de conjunto cuja sintaxe é mostrada a seguir:

```
[<exp>[..<exp>][,<exp>[..<exp>]]...]
```

Onde **exp** é qualquer tipo de expressão que retorne um valor compatível com o tipo do conjunto.

Exemplos:

```
const
   Resp_Valida = ['s','S','n','N'];
var
   Vogais_Minusculas : Caracteres;
   Numeros_Sorteados : Digitos;
   Dias_de_Trabalho  : Dias_da_Semana;
...
   Vogais_Minusculas := ['a','e','i','o','u'];
   Numeros_Sorteados := [1, 3, 4, 6, 8];
   Dias_de_Trabalho  := [Seg, Ter, Qua, Qui, Sex];
...
```

Seguindo a idéia dos conjuntos em matemática, no Pascal também há como definir conjuntos vazios (representados por um par de colchetes [ ]). Além disso, não há a idéia de ordenação entre os elementos de um conjunto.

8.2. Operações sobre conjuntos

As operações que podem ser executadas sobre conjuntos são as seguintes:

Igualdade (=)

Retorna `true` quando os dois conjuntos são iguais, ou seja, ambos possuem os mesmos elementos.

Ex: `['a','b'] = ['b','a']` retorna `true`

Desigualdade (<>)

Retorna `true` quando os dois conjuntos são diferentes, ou seja, um dos conjuntos possui pelo menos um elemento diferente do outro.

Ex: `['a','b','c'] <> ['b','a']` retorna `true`

União (+)

Retorna um outro conjunto contendo a união dos elementos dos dois conjuntos (sem repetição).

Ex: `['a','b','c'] + ['b','a','d']` retorna `['a','b','c','d']`

Interseção (*)

Retorna um conjunto contendo os elementos comuns a ambos os conjuntos.

Ex: `['a','b'] * ['b','c','d']` retorna `['b']`

Diferença (-)

Retorna um conjunto contendo apenas os elementos do primeiro conjunto que não estão presentes no segundo conjunto.

Ex: `['a','b','e'] - ['b','c','d']` retorna `['a','e']`

Subconjunto (<=)

Retorna `true` quando todos os elementos do primeiro conjunto encontram-se no segundo conjunto.

Ex: `['a','b'] <= ['b','a','c']` retorna `true`

Superconjunto (>=)

Retorna `true` quando todos os elementos do segundo conjunto encontram-se no primeiro conjunto.

Ex: `['b','a','c'] >= ['a','b']` retorna `true`

Pertence (`in`)

Retorna `true` quando o valor especificado está presente no conjunto.

Ex: `'a' in ['a','b']` retorna `true`

8.3. Utilização da Memória

O Turbo Pascal utiliza 1 bit para indicar a presença de um elemento no conjunto. Um conjunto pode abranger no máximo 256 valores (32 bytes).

Supondo a existência do seguinte conjunto:

```
type
   Dias_da_Semana = set of (Seg, Ter, Qua, Qui, Sex, Sab, Dom);
var
   Final_de_Semana : Dias_da_Semana;
...
   Final_de_Semana := [];
   Final_de_Semana := [Sab,Dom];
...
```

Observações:

- `Final_de_Semana` só precisa de um byte pois o conjunto só pode conter até 7 valores.

- Após a primeira atribuição a variável `Final_de_Semana` contém somente bits zero.

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

- Após a segunda atribuição a variável `Final_de_Semana` contém 2 bits 1 e 6 bits 0.

7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0

Exemplo: Validando os caracteres digitados, aceitando apenas números e ponto.

```
program Ex1Cap8;
const
   Caracteres_Validos = ['0'..'9','.'];
var
   Valor_Lido : string[15];
   I : byte;
   Ok : boolean;
BEGIN
   write('Digite um numero: ');
   readln(Valor_Lido);

   Ok := true;
   for I := 1 to length(Valor_Lido) do
      if not (Valor_Lido[I] in Caracteres_Validos) then
         begin
            writeln('Erro na posicao ', I);
            Ok := false;
         end;

   if Ok then
      writeln('Numero valido');
END.
```

9. Subprogramas

9.1. Introdução

A linguagem Pascal permite declarar trechos de código fora do programa principal e associados a um identificador que podem ser executados sempre que invocados. Esses trechos de código são chamados de subprogramas, módulos ou rotinas.

Os principais motivos para se usá-los são:

- **Evitar codificação:** trocar certos trechos de programas que se repetem por chamadas de um subprograma que será codificado apenas uma vez.

- **Modularizar o programa:** particionando-o em módulos (subprogramas) logicamente coerentes, cada uma com função bem definida. Isto facilita a organização do programa, bem como o entendimento dele.

Essa técnica de programação é denominada programação estruturada.

Na linguagem pascal, é possível declarar dois tipos de subprogramas: procedimentos (`procedure`) e funções (`function`).

9.2. Procedimentos

Em Pascal, os procedimentos são declarados na seção de declaração de subprogramas, onde são associados a um identificador.

Para utilizar um subprograma basta chamá-lo no corpo do programa, como se faz com procedimentos predefinidos (write, read, delete, insert, val, str, etc.), fazendo com que os comandos definidos nos subprogramas sejam executados.

Sintaxe de declaração de procedimentos sem parâmetros:

```
procedure <nome-do-proc>;
[declarações]
begin
  [<comando>[;<comando>]]
end;
```

Exemplo: Programa que usa um procedimento que imprime uma linha com 80 hífens

```
program Ex1Cap9;

procedure ImprimirLinha;
```

```
begin
   write('----------------------------------------');
   write('----------------------------------------');
end;

BEGIN
   ImprimirLinha;
   writeln('Mensagem entre as linhas');
   ImprimirLinha;
END.
```

9.3. Passagem de Parâmetros

Muitas vezes o comportamento do subprograma depende de algum dado, como a maioria dos procedimentos predefinidos. O `delete`, por exemplo, precisa saber a de que `string` irá remover, da posição a partir de onde irá iniciar a remoção e o número de caracteres. Todas essas informações são fornecidas através de parâmetros.

Ao chamar um procedimento, são passados valores ou variáveis como parâmetro. Esses são chamados de **parâmetros atuais**. Internamente, esses valores ou variáveis são tratados por identificadores definidos no procedimento, sendo conhecidos como **parâmetros formais**.

Sintaxe de definição de procedimentos com parâmetros:

```
procedure <nome-do-proc>
   [([var|const| ] <ident-param>[,<ident-param>]... : <tipo>
   [;[var|const| ] <ident-param>[,<ident-param>]... : <tipo>]...)];
 [declarações]
begin
  [<comando>[;<comando>]]
end;
```

Passagem de Parâmetros por Variável

Essa forma de passagem de parâmetro deve ser usada quando se deseja que o subprograma utilize a variável como forma de retornar algum valor a quem o chamou. Nesse tipo de passagem de parâmetro, só é permitido especificar variáveis.

Nesse tipo de passagem de parâmetro, um parâmetros formal é apenas um outro identificador para o parâmetro atual (informado na chamada do subprograma), ou seja, compartilham a mesma memória.

Para especificar um parâmetro passado por variável é necessário apenas informar a palavra reservada `var` antes do identificador do parâmetro. Quando essa palavra é omitida, a passagem de parâmetros é feita por valor.

Exemplo: Utilizando um procedimento que calcula a média de três notas

```
program Ex2Cap9;

procedure CalcularMedia(var N1, N2, N3, Med : real);
begin
   Med := (N1 + N2 + N3) / 3.0;
end;

var
   Nota1,
   Nota2,
   Nota3,
   Media : real;

BEGIN
   write('Entre com 3 notas: ');
   readln(Nota1, Nota2, Nota3);
   CalcularMedia(Nota1, Nota2, Nota3, Media);
   writeln('Media das 3 notas: ', Media:3:1);
END.
```

Passagem de Parâmetros por Valor

Na passagem de parâmetros por valor é permitido passar tanto uma variável como um valor (constante ou resultado de uma expressão).

Deve-se ter cuidado com esse tipo de passagem de parâmetro pois ela provoca a cópia do valor do parâmetro atual em uma nova área de memória, acessível através dos parâmetros formais.

Na passagem de parâmetros por valor nenhuma das modificações nos parâmetros formais é refletida nos parâmetros atuais.

Exemplo: Passagem por valor e Passagem por variável

```
program Ex3Cap9;

procedure IncPorVariavel(var I : longint; N : longint);
begin
   I := I + N;
end;

procedure IncPorValor(I : longint; N : longint);
begin
   I := I + N;
end;

var
   X : longint;
```

```
BEGIN
   X := 0;
   IncPorVariavel(X, 10);
   writeln(X); { Imprime 10 };

   X := 0;
   IncPorValor(X, 10);
   writeln(X); { Imprime 0 }
END.
```

Passagem de Parâmetros por Constante

É uma forma de passagem de parâmetros parecida com a passagem por valor. A principal diferença é que na passagem por constante, o valor do parâmetro não pode ser alterado durante o subprograma.

Para especificar um parâmetro passado por constante é necessário apenas informar a palavra reservada `const` antes do identificador do parâmetro.

É interessante usar esse tipo de passagem de parâmetro para garantir que os valores dos parâmetros não serão modificados acidentalmente durante a execução do subprograma.

```
program Ex4Cap9;
type
   TComparacao = char;
var
   R    : TComparacao;
   X, Y : integer;

procedure Comparar(const V1, V2 : integer; var Res : TComparacao);
begin
   if V1 < V2 then
      Res := '<'
   else
      if V1 > V2 then
         Res := '>'
      else
         Res := '='
end;

procedure Testar(const V1, V2 : integer);
begin
   Comparar(V1, V2, R);
   writeln(V1, R, V2);
end;

BEGIN
   write('Digite 2 valores inteiros: ');
   readln(X, Y);
   Testar(X, Y);
END.
```

9.4. Identificadores Locais

Muitas vezes é necessário utilizar além dos parâmetros formais algumas outras variáveis. Uma solução é declará-las em uma seção de declaração de variáveis do próprio programa. Assim, qualquer subprograma declarado abaixo da variável poderá ter acesso a ela.

Além de variáveis, é possível declarar constantes nomeadas, tipos e outros subprogramas.

Exemplo: Usando em um procedimento uma variável declarada no programa.

```
program Ex5Cap9;

var I : word;

procedure Imprimir(c : char; n : word);
begin
   for I := 1 to n do
      write(c);
end;

BEGIN
   Imprimir('-', 80);
   Imprimir('+', 80);
END.
```

Porém, essa abordagem não é aconselhável por várias razões, dentre as quais pode-se citar:

- Pode-se perder o controle de quais subprogramas estão acessando uma mesma variável.

- É mais difícil aproveitar o subprograma para colocá-lo em outros programas pois isso exige descobrir quais variáveis está usando e declará-las no novo programa, podendo gerar conflito com variáveis já existentes.

- Ajuda a poluir o espaço de nomes de variáveis como conseqüência da criação de muitas variáveis globais.

Pelas razões acima, é necessário restringir ao mínimo o acesso às variáveis definidas no programa. Quando a variável só é usada no escopo do procedimento, ela pode ser declarada como uma variável local. Porém, se ela precisar ficar no escopo do programa a melhor solução é passá-la como parâmetro para o subprograma.

```
program Ex6Cap9;

procedure Imprimir(c : char; n : word);
var I : word;
begin
   for I := 1 to n do
      write(c);
```

```
end;

BEGIN
   Imprimir('-', 80);
   Imprimir('+', 80);
END.
```

9.5. Funções

Uma função é muito parecida com um procedimento, exceto que uma função deve retornar um valor. Existem 3 diferenças entre funções e procedimentos:

- Troca-se a palavra reservada `procedure` por `function`.
- Após a lista de parâmetros deve ser informado o tipo do dado retornado pela função.
- No corpo da função deve ser especificado um valor de retorno para a função.

Sintaxe de definição de funções:

```
function <nome-da-função>
   [([var|const| ] <ident-param>[,<ident-param>]... : <tipo>
   [;[var|const| ] <ident-param>[,<ident-param>]... : <tipo>]...)]
   : <tipo-retorno>;
[declarações]
begin
  [<comando>[;<comando>]]
end;
```

Exemplo: Converte todos os caracteres de uma `string` para caracteres maiúsculos.

```
program Ex7Cap9;

function Upper(s : string) : string;
var
   I : word;
begin
   for I := 1 to length(s) do
      s[I] := UpCase(s[I]);

   Upper := s;
end;

BEGIN
   writeln(Upper('aBc123!')); { Imprime ABC123! }
END.
```

9.6. Parâmetro Tipo String Aberto

Quando uma variável do tipo `string` com tipo diferente do especificado no parâmetro formal é passada por variável para o subprograma, ocorrerá um erro de compilação *type mismatch*. Isso não ocorre quando é passado por valor ou por constante.

Exemplo: Exemplo de *type mismatch* devido à diferenças entre os tipos dos parâmetros formal e atual.

```
program Ex8Cap9;

type
  str50 = string[50];
var
  Nome : string[30];

function Upper(var s : str50) : string;
var
   I    : word;
   Temp : string;
begin
   for I := 1 to length(s) do
      s[I] := UpCase(s[I]);

   Upper := s;
end;

BEGIN
   Nome := 'alberto';
   writeln(Upper(Nome)); { type mismatch nesta linha ao compilar }
END.
```

Uma forma de solucionar esse problema é definir um tipo (através de um `type`) e utilizá-lo ao definir a variável e o parâmetro formal.

Outra abordagem é especificar na assinatura (cabeçalho) do subprograma o tipo como sendo `OpenString`, o qual aceita qualquer tipo de `string`. Ao utilizar o este tipo, as funções `low` retorna 0, `high` retorna o tamanho máximo para o parâmetro atual e `SizeOf` o número de bytes ocupados pelo parâmetro atual (incluindo o byte de comprimento).

```
program Ex9Cap9;

function Posicao(c : char; var str : OpenString) : byte;
var
   I : word;
begin
   Posicao := 0;
   for I := 1 to length(str) do
      if str[I] = c then
         begin
            Posicao := I;
```

```
            exit
         end
end;

var Nome : string[50];
BEGIN
   Nome := 'alberto';
   writeln(Posicao('r', Nome));
END.
```

9.7. Parâmetro Tipo Array Aberto

Uma das dificuldades ao escrever subprogramas que usam *arrays* é definir o seu tamanho, dificultando a criação de rotinas eficientes (parâmetro formal muito maior que o necessário).

É possível usar parâmetros do tipo *array* com tamanho indeterminado (aberto). Os índices variam de 0 a N - 1, onde N é o tamanho do *array* passado como parâmetro atual.

Ao utilizar-se um *array* aberto a função `low` retorna 0, `high` retorna o valor de N - 1 e `SizeOf` retorna o número de bytes ocupados pelo parâmetro atual.

Exemplo: Criando um procedimento para imprimir um array de interiros

```
program Ex10Cap9;

procedure Imprimir(vetor : array of integer);
var I : integer;
begin
   for I := 0 to high(vetor) do
      write('[', I, ']=', vetor[I], ' ');
end;

var
   array10  : array[1..10] of integer;
   array100 : array[1..100] of integer;
   I        : integer;

BEGIN
   for I := 1 to 10 do
      array10[I] := I;

   for I := 1 to 100 do
      array100[I] := I;

   writeln('array com 10');
   Imprimir(array10);
   writeln;
   writeln('array com 100');
   Imprimir(array100);
END.
```

9.8. Parâmetros sem Tipo

O Turbo Pascal permite especificar parâmetros sem tipos definidos. Portanto, é necessário fazer *type casting* no subprograma para se trabalhar com os parâmetros passados. Essa característica permite fazer rotinas mais genéricas, mas requer muito cuidado pois pode ser fonte de erros durante a execução do programa (caso haja algum *type casting* errado).

Exemplo: Comparando dois parâmetros de até 64 Kbytes.

```
program Ex11Cap9;

function Igual(var A, B; NBytes : word) : boolean;
type
   bytes = array[0..65534] of byte;
var
   N : word;
begin
   Igual := true;
   for N := 0 to NBytes - 1 do
      if bytes(A)[N] <> bytes(B)[N] then
         begin
            Igual := false;
            exit;
         end;
end;

const
   c1 : char = 'c';
   c2 : char = 'c';
   I  : integer = 1;
   J  : integer = 2;
   A  : string = 'alberto';
   B  : string = 'alberto';

BEGIN
   writeln(Igual(c1, c2, 1)); { Imprime TRUE }
   writeln(Igual(I, J, 2));   { Imprime FALSE }
   writeln(Igual(A, B, 8));   { Imprime TRUE }
END.
```

10. Units

A utilização de subprogramas permite reutilizá-los várias vezes dentro de um programa. Porém, não é possível utilizá-los de fora dos programas onde foram declarados, obrigando a copiá-los diretamente para os outros programas, gerando duplicação de código e consequentemente problemas de manutenção.

Para resolver essa questão, o Turbo Pascal traz um recurso chamado `unit`. Uma `unit` permite criar subprogramas, variáveis, tipos e constantes separadamente. As units são independentes dos programas, podendo ser compiladas a qualquer momento.

O Turbo Pascal traz diversas units predefinidas, como CRT, Dos, Graph, Printer e System (incluída como padrão).

10.1. Sintaxe de definição de uma unit

A sintaxe de definição de units é mostrada abaixo:

```
unit <ident-unit>;

interface
[uses <lista-de-units>;]
[Declarações públicas]

implementation
[uses <lista-de-units>;]
[Declarações privadas]
[Implementações de subprogramas]

[begin
   [<comando>[;<comando>]]]
end.
```

Uma `unit` está dividida em três seções: seção de interface, seção de implementação e seção de inicialização.

10.2. A seção de Interface

Esta seção, que inicia-se após a palavra reservada `interface` e termina antes de `implementation`, especifica quais funções e procedimentos implementados na seção de implementação serão públicos (visíveis fora do escopo da seção de implementação).

Tudo que for definido dentro desta seção é acessível a partir dos programas que utilizam a `unit`.

10.3. A seção de Implementação

Esta seção contém as declarações que são privadas à `unit`, ou seja, não podem ser acessadas dos programas que as utilizam. A única exceção são os subprogramas cujos cabeçalhos (assinaturas) encontram-se na seção de interface.

Quando as assinaturas dos subprogramas estão declaradas na interface, não é obrigatório (apesar de ser aconselhável) incluí-los na seção de implementação na forma completa, isto é, basta especificar o tipo (`procedure` ou `function`) junto com o nome.

10.4. A seção de Inicialização

A seção de inicialização contém comandos que são executados antes do início da execução do programa que a utiliza. Essa seção é mais utilizada para inicializar variáveis e abrir arquivos necessários para os subprogramas nela definidos.

10.5. Utilizando units em programas

Exemplo: `unit` que define algumas funções para manipular strings.

```
unit strbib;

interface

type str50 = string[50];

function Upper(s : str50) : str50;
function Limpar(s : str50) : str50;

implementation

var CharsParaLimpar : set of char;

function Upper(s : str50) : str50;
var I : word;
begin
   for I := 1 to length(s) do
      s[I] := UpCase(s[I]);

   Upper := s;
end;

procedure LimparEsquerda(var s : str50);
begin
   while (length(s) > 0) and (s[1] in CharsParaLimpar) do
      delete(s, 1, 1);
end;

procedure LimparDireita(var s : str50);
begin
```

```
   while (length(s) > 0) and (s[length(s)] in CharsParaLimpar) do
      delete(s, length(s), 1);
end;

function Limpar(s : str50) : str50;
begin
   LimparEsquerda(s);
   LimparDireita(s);
   Limpar := s;
end;

begin
   CharsParaLimpar := [' '];
end.
```

Um exemplo de utilização da `unit` definida acima é mostrada no programa a seguir:

```
program Ex1Cp10;

uses strbib;

var s : str50;

BEGIN
   write('Entre com um nome: ');
   readln(s);
   write('Nome com letras maiusculas: ');
   writeln(Upper(Limpar(s)));
END.
```

10.6. Colisão de identificadores

Um problema que pode acontecer quando se utiliza units é haver colisão dos identificadores, ou seja, um identificador declarado em um programa (ou unit) ser igual a outro identificador definido na seção `interface` de alguma das units que utiliza.

Quando isso ocorre, não há erro de compilação, mas apenas um dos identificadores vai poder ser utilizado. Nesse caso, prevalece o identificador definido no programa principal ou na unit que está utilizando uma outra unit.

É possível declarar explicitamente que identificar utilizar qualificando-o com o nome da `unit` onde foi definido, isto é, colando-se o nome da `unit` seguido de um ponto mais o nome do identificador.

Exemplo: Programa que utiliza 3 units e demostra as regras de resolução de colisão de identificadores

```
program Ex2Cap10;
uses unit1, unit3, unit2;
var A : integer;
BEGIN
```

```
   A := 10;
   writeln('A = ', A); { Imprime 10 }
   writeln('A (unit1) = ', unit1.A); { Imprime 1 }

   writeln('B = ', getB); { Imprime 0 }
   setB(1);
   writeln('B = ', getB); { Imprime 1 }
   writeln('C = ', C);    { Imprime 2 }

   writeln('C (unit2) = ', unit2.C); { Imprime 2 }
   writeln('C (unit3) = ', unit3.C); { Imprime 3 }

   writeln('B = ', B); { Imprime 2 }
   writeln('D = ', D); { Imprime 3 }
END.
```

```
unit unit1;
interface

var A : integer;

function getB : longint;
procedure setB(novoB : longint);

implementation

var B : string;

function getB : longint;
var
   Code : integer;
   Temp : longint;
begin
   val(B, Temp, Code);
   getB := Temp;
end;

procedure setB(novoB : longint);
begin
   str(novoB, B);
end;

begin
   A := 1;
   B := '0';
end.
```

```
unit unit2;
interface

var B, C : integer;

implementation

var D : integer;

begin
   B := 2;
   C := 2;
   D := 2;
end.
```

```
unit unit3;
interface

var C, D : integer;

implementation

begin
   C := 3;
   D := 3;
end.
```

11. Unit CRT

A `unit crt` possui funções e procedimentos que permitem trabalhar com o teclado e o vídeo.

No Turbo Pascal há dois formas de se trabalhar com vídeo:

- **Modo texto:** trabalha com os caracteres da tabela ASCII. Os caracteres que compõem a tela a ser exibida no vídeo são mantidos na memória de forma que seja possível apresentá-los quando a tela precisar ser redesenhada.

- **Modo gráfico:** requer a utilização d `unit graph`. O vídeo é tratado através da manipulação de pixels. Permite criar vários tipos de gráficos, o que não é possível no Modo texto.

No modo texto o vídeo pode apresentar até 2000 caracteres, sendo organizados em 25 linhas de 80 colunas. Cada caractere tem uma cor de *foreground* (frente) e uma de *background* (fundo). As características de um caractere (cores de *foreground* e *background*, por exemplo) são mantidas em um byte de atributo.

Os caracteres e seus bytes de atributo são mantidos na memória em uma área denominada *memória de vídeo*.

11.1. Byte de Atributo

O byte de atributo permite especificar 16 cores de *foreground* (frente), 8 cores de *background* (fundo) e se o caractere deve piscar.

A cor de *background* é definida através de 3 bits que indicam a presença das cores vermelho (*red*), verde (*green*) e azul (*blue*). Através da combinação dessas cores chega-se a outras cores. Esta forma de especificar cores normalmente é chamada de RGB.

A cor de *foreground* segue a mesma idéia da de *background* exceto que usa adicionalmente um bit para indicar o brilho (totalizando 4 bits), sendo possível obter cores mais claras ou escuras (verde claro e escuro, por exemplo).

O bit restante do byte de atributo é utilizado para indicar se o caractere deve piscar.

11.2. Procedimentos para manipulação de cores

A `unit crt` possui constantes predefinidas que facilitam a especificação das cores de frente e fundo dos caracteres. Estas constantes são mostradas na tabela abaixo:

Cores escuras (Foreground &Background)		**Cores claras (Foreground)**	
`Black`	0	`DarkGray`	8
`Blue`	1	`LightBlue`	9
`Green`	2	`LightGreen`	10
`Cyan`	3	`LightCyan`	11
`Red`	4	`LightRed`	12
`Magenta`	5	`LightMagenta`	13
`Brown`	6	`Yellow`	14
`LightGray`	7	`White`	15

Além das constantes definidas na tabela acima, há a constante `Blink` que pode ser somada a qualquer cor de *foreground* e faz com que o caractere pisque.

O Turbo Pascal traz 5 procedimentos que permitem manipular as cores dos caracteres apresentados no vídeo, são eles:

- **TextColor(<cor>)**: Define a cor de *foreground* atual.
- **TextBackground(<cor>)**: Define a cor de *background* atual.
- **NormVideo**: Retorna à configuração de cores existente no início do programa.
- **LowVideo**: Muda a cor atual para a cor escura correspondente. Transforma, por exemplo, a cor `Yellow` em `Brown`.
- **HighVideo**: Muda a cor atual para a cor clara correspondente. Transforma, por exemplo, a cor `Brown` em `Yellow`.

Exemplo: Mudando as cores de *foreground* e *background*.

```
program Ex1Cap11;
uses crt;
BEGIN
   { Muda a cor de foreground }
   TextColor(Yellow);
   writeln('Frente Amarela');

   { Muda a cor de background }
   TextBackground(Blue);
   writeln('Frente Amarela e Fundo Azul');

   { Muda a cor atual para a cor escura correspondente }
   LowVideo;
   writeln('Frente Marrom e Fundo Azul');

   { Muda a cor atual para a cor clara correspondente }
   HighVideo;
   writeln('Frente Amarela e Fundo Azul');

   { Restaura a configuracao de cor inicial }
   NormVideo;
END.
```

Deve-se tomar o cuidado de sempre chamar o procedimento NormVideo no final do programa pois caso não seja chamado, a configuração das cores de vídeo permanecerá mesmo após o final do programa.

11.3. Procedimentos para limpeza de tela

Os procedimentos para limpar a tela disponíveis na `unit crt` são:

- **ClrScr**: Limpa a tela completamente usando a cor de *background* atual. Move o cursor para o canto superior esquerdo.

- **ClrEol**: Limpa a linha atual partindo da posição corrente até o caractere mais à direita usando a cor de background atual. O cursor permanece na posição inicial.

Exemplo: Limpando a tela usando ClrScr e ClrEol

```
program Ex2Cap11;
uses crt;
BEGIN
   { Faz com que toda a tela fique azul }
   TextBackground(Blue);
   ClrScr;

   { Imprime uma string usando uma cor de fundo e apaga o restante
     da linha usando uma cor de fundo diferente }
```

```
   write('ABC');
   TextBackground(LightGray);
   ClrEol;

   { Espera pela digitação de uma tecla antes de limpar a tela }
   Readkey;
   NormVideo;
   ClrScr;
END.
```

11.4. Produzindo sons

A `unit crt` traz os comandos `Sound` e `NoSound` que servem, respectivamente para ativar a emissão de um som em uma freqüência (indefinidamente) e desativar o som. O procedimento `Sound` recebe como parâmetro a freqüência.

Exemplo: Programa que solicita a freqüência e o tempo de emissão do som e o reproduz.

```
program Ex3Cap11;
uses crt;
var
   Freq, Tempo : integer;
BEGIN
   write('Entre com a frequencia:');
   readln(Freq);
   write('Entre com o tempo:');
   readln(Tempo);

   Sound(Freq);
   Delay(Tempo);
   NoSound;
END.
```

O procedimento `Delay` permite parar a execução do programa pelo tempo (em milisegundos) especificado como parâmetro.

11.5. Posicionamento do cursor

A `unit crt` divide a tela em 25 linhas cada uma com 80 colunas, totalizando 2000 caracteres. Cada caractere é referenciado através de uma coordenada composta pelo número da linha (entre 1 e 25) e da coluna (entre 1 e 80). O caractere no canto superior esquerdo, por exemplo, tem a coordenada (1,1), enquanto que o caractere no canto inferior direito tem a coordenada (80, 25).

A `unit crt` tem um procedimento chamado `GotoXY(<X>, <Y>)` que move o cursor para a posição especificada pelos parâmetros `X` (coluna) e `Y` (linha).

Para identificar a linha e a coluna onde se encontra o cursor há duas funções chamadas `WhereY` e `WhereX` que retornam esses valores.

Exemplo: Usando o GotoXY, WhereX e WhereY

```
program Ex4Cap11;
uses crt;
var X,Y : integer;
BEGIN
   ClrScr;
   GotoXY(5,2);
   write('Linha 2, Coluna 5');

   X := WhereX;
   Y := WhereY;
   GotoXY(1,1);
   writeln('X = ', X, ' e Y = ', Y); { Imprime X = 22 e Y = 2 }
END.
```

11.6. Apagando e Inserindo Linhas

O procedimento `DelLine` apaga a linha atual onde se encontra o cursor, mantendo-o na mesma posição. Todas as linhas abaixo sobem para preencher o espaço deixado pela linha removida.

Exemplo: Apagando uma linha da tela

```
program Ex5Cap11;
uses crt;
const
   NUM_LINHAS = 5;
var
   I : integer;
BEGIN
   ClrScr;
   for I := 1 to NUM_LINHAS do
      writeln('Linha ', I);

   { Remove a linha 2, fazendo com que as demais subam }
   GotoXY(3,2);
   DelLine;

   GotoXY(1, NUM_LINHAS + 1);
   writeln('Linha ', NUM_LINHAS + 1);
END.
```

O procedimento `InsLine` insere uma linha em branco na posição corrente e faz com que a linha atual, junto com todas abaixo dela sejam deslocadas para baixo.

Exemplo: Inserindo uma linha na tela

```
program Ex6Cap11;
uses crt;
const
   NUM_LINHAS = 5;
```

```
var
   I : integer;
BEGIN
   ClrScr;
   for I := 1 to NUM_LINHAS do
      writeln('Linha ', I);

   { Insere uma linha em branco no lugar da Linha 3 }
   GotoXY(10,3);
   InsLine;

   { Mensagem iniciada na coordenada (10, 3) }
   writeln('Linha Inserida');

   GotoXY(1, NUM_LINHAS + 2);
   writeln('Linha ', NUM_LINHAS + 2);
END.
```

11.7. Manipulação de Teclado

A função `KeyPressed` retorna `true` quando alguma tecla foi pressionada e `false` caso contrário.

A função `ReadKey` para a execução do programa até que alguma tecla seja pressionada. Quando a tecla é pressionada, ela retorna o caractere que foi digitado.

Exemplo: Fazendo uma contagem regressiva que pode ser encerrada apertando uma tecla.

```
program Ex7Cap11;
uses crt;

procedure ContagemRegressiva(Cont : integer);
begin
   while (cont > 0) and not KeyPressed do
   begin
      writeln(Cont);
      Delay(1000);
      dec(Cont);
   end;
end;

BEGIN
   ClrScr;
   writeln('Pressione alguma tecla...');
   ContagemRegressiva(5);
   if KeyPressed then
      writeln('Tecla (', Ord(ReadKey), ') pressionada')
   else
      writeln('Nenhuma tecla pressionada');
END.
```

11.8. Janelas

Em algumas situações pode ser interessante trabalhar com uma região específica da tela. A `unit crt` permite definir uma janela através do procedimento `Window(<X1>,<Y1>,<X2>,<Y2>)`, onde `X1` e `Y1` determinam o canto superior esquerdo e `X2` e `Y2` o canto inferior direito.

As funções `WhereX` e `WhereY` e os procedimentos `GotoXY` , `ClrScr`, `ClrEol`, `DelLine`, `InsLine`, funcionam de forma relativa à janela corrente, ou seja, um comando `ClrScr` faz a limpeza da janela corrente, deixando intacta a área ao seu redor.

Exemplo: Imprimindo a hora na última linha da tela.

```
program Ex8Cap11;
uses crt, dos;
var
  X, Y : integer;
  H, M, S, MS : word;
BEGIN
   ClrScr;

   Window(1, 25, 80, 25);
   TextColor(Yellow);
   TextBackground(Blue);

   ClrScr;
   write('Hora: ');
   X := WhereX;
   Y := WhereY;

   while not KeyPressed do
   begin
     GetTime(H, M, S, MS);
     GotoXY(X, Y);
     ClrEol;
     write(H, ':', M, ':', S);
     Delay(1000);
   end;

   Window(1,1,80,25);
   NormVideo;
   ClrScr;
END.
```

12. Arquivos

```
Os procedimentos vistos nas seções 4.3 e 4.4;
END.
```

13. Arquivos

Os procedimentos vistos nas seções 4.3 e 4.4 permitiam direcionar a entrada e a saída padrão para arquivos e utilizar normalmente os comandos de leitura e escrita. Essa abordagem, porém tem limitações como:

- Impossibilidade de ler/gravar mais de um arquivo ao mesmo tempo;
- Impossibilidade de ler do teclado e enviar dados para o vídeo ao mesmo tempo em que os arquivos são processados;
- A forma de acesso é seqüencial, ou seja, bastante limitada.

13.1. Arquivos Tipo Texto

O arquivo tipo texto é bastante fácil de trabalhar. É composto por linha(s) formadas por caractere(s). Uma linha pode ter um número infinito de caracteres.

Uma das principais vantagens desse tipo de arquivo é que um programa escrito em Pascal pode ler um arquivo criado por um outro programa escrito em outra linguagem ou até mesmo usando um editor de texto. Além disso, pode gravar um arquivo que pode ser lido por um editor de texto ou outro programa.

Associando identificadores a arquivos

O Turbo Pascal permite associar um identificador a arquivos do tipo texto. Essa associação é feita através da definição de uma variável do tipo `text` e do uso do procedimento `assign`, como mostrado abaixo:

```
var <ident-do-arquivo> : text
...
assign(<ident-do-arquivo>, <nome-do-arquivo>)
```

Onde:

- **ident-do-arquivo** é o identificador que será usado dentro do programa Pascal (também conhecido como nome interno).
- **nome-do-arquivo** é o nome utilizado pelo sistema de arquivos para localizá-lo (também conhecido como nome externo).

Abrindo arquivos

O Turbo Pascal traz 3 procedimentos para abrir arquivos tipo texto: `rewrite`, `append` e `reset`. A sintaxe de utilização desses três procedimentos é idêntica:

```
rewrite(<ident-do-arquivo);
append(<ident-do-arquivo);
reset(<ident-do-arquivo);
```

- **rewrite**: Cria um novo arquivo. Quando já existe algum arquivo com o mesmo nome, ele é apagado e um novo é criado. O arquivo criado é mantido aberto exclusivamente para escrita.

- **append**: Abre o arquivo exclusivamente para escrita. Todos os dados enviados para o arquivo serão incluídos no final, mantendo o conteúdo existente. Caso o arquivo não exista, ocorrerá um erro.

- **reset**: Abre o arquivo exclusivamente leitura. Caso o arquivo não exista, ocorrerá um erro.

Fechando arquivos

O procedimento `close` serve para fechar um arquivo aberto. Ao fechar um arquivo ele passa a ficar disponível para outros programas ou até para o mesmo programa acessá-lo de uma forma diferente. A sintaxe do procedimento `close` é mostrada abaixo:

```
close(<ident-do-arquivo>)
```

É importante fechar os arquivos assim que possível pois alguns dados mantidos no *buffer* podem não ser transferidos para o disco, acarretando perda de dados. Essas perdas podem ser ocasionadas por:

- Falta de energia durante a execução do programa;

- Erro inesperado no programa (como divisão por zero e falta de memória).

Leitura e Escrita de arquivos

A leitura de arquivos tipo texto é feita através dos procedimentos `read` e `readln`, usando a seguinte sintaxe:

```
read(<ident-do-arquivo>, [<variável>[,<variável>]]...)
readln(<ident-do-arquivo>, [<variável>[,<variável>]]...)
```

A escrita de arquivos tipo texto é feita através dos procedimentos `write` e `writeln`, usando a seguinte sintaxe:

```
write(<ident-do-arquivo>, [<exp>[,<exp>]]...)
writeln(<ident-do-arquivo>, [<exp>[,<exp>]]...)
```

É possível detectar o final de um arquivo durante a sua leitura através da função `eof`. A função `eoln` indica se foi atingido o final da linha. A sintaxe dessas funções é:

```
eof(<ident-do-arquivo>)
eoln(<ident-do-arquivo>)
```

Detectando erros de Entrada e Saída

O comportamento padrão do Pascal é encerrar o programa caso ocorra algum erro ao manipular arquivos (abrir um arquivo para leitura que não existe, por exemplo).

Pode-se alterar esse comportamento padrão fazendo com que o Pascal guarde em uma variável predefinida chamada `IOResult` um código (número inteiro) que indica o tipo do erro detectado ou 0 caso a última operação tenha sido feita com sucesso.

Para mudar o comportamento padrão do Pascal, deixando o tratamento de erro a cargo do programador, é necessário utilizar as diretivas de compilação `{$I-}` e `{$I+}` para delimitar o trecho de código sem tratamento automático de erros.

Exemplo: Programa que faz a leitura de dois arquivos e cria um terceiro com o conteúdo de ambos.

```
program Ex1Cap12;

function ObterNome(Msg : OpenString) : string;
var Nome : string;
begin
   write(Msg);
   readln(Nome);
   ObterNome := Nome;
end;

procedure Merge(var A1, A2, A3 : text);

  procedure CopiarChars(var X, Y : text);
  var c : char;
  begin
     while not eof(X) do
     begin
        read(X, c);
        write(Y, c);
     end;
  end;
```

```
begin
   CopiarChars(A1, A3);
   CopiarChars(A2, A3);
end;

var
   Arq1, Arq2, Arq3             : text;
   NomeArq1, NomeArq2, NomeArq3 : string;
   Resp                         : char;

BEGIN
   NomeArq1 := ObterNome('Arquivo 1: ');
   NomeArq2 := ObterNome('Arquivo 2: ');
   NomeArq3 := ObterNome('Arquivo 3: ');

   assign(Arq1, NomeArq1);
   assign(Arq2, NomeArq2);
   assign(Arq3, NomeArq3);

   {$I-} reset(Arq1); {I+}
   if IOResult <> 0 then
   begin
      writeln('Arquivo 1 ''', NomeArq1, ''' nao encontrado');
      exit;
   end;

   {$I-} reset(Arq2); {I+}
   if IOResult <> 0 then
   begin
      writeln('Arquivo 2 ''', NomeArq2, ''' nao encontrado');
      exit;
   end;

   {$I-} reset(Arq3); {I+}
   if IOResult = 0 then
   begin
      writeln('Arquivo 3 ''', NomeArq3, ''' ja existe');
      write('Deseja sobrepor(S/N)? ');
      readln(Resp);
      if UpCase(Resp) <> 'S' then
         exit
   end;
   rewrite(Arq3);

   Merge(Arq1, Arq2, Arq3);

   close(Arq1);
   close(Arq2);
   close(Arq3);
END.
```

Renomeando e Apagando arquivos

Quando for necessário renomear um arquivo (mudar seu nome externo), deve-se utilizar o procedimento `rename`.

Para apagar um arquivo há um procedimento chamado `erase` cuja sintaxe é mostrada abaixo, junto com a do procedimento `rename`:

```
rename(<ident-do-arquivo>, <novo-nome-do-arquivo>)
erase(<ident-do-arquivo>)
```

Exemplo: Apagando um arquivo tipo texto.

```
program Ex2Cap12;
var
   Arq  : text;
   Nome : string;
BEGIN
   if ParamCount <> 1 then
      begin
         writeln('Informe o nome do arquivo a ser apagado');
         halt(1);
      end;

   Nome := ParamStr(1);
   assign(Arq, Nome);

   {$I-} reset(Arq); {$I+}
   if IOResult <> 0 then
      writeln('Arquivo ''', Nome, ''' nao encontrado ')
   else
      begin
         close(Arq);
         erase(Arq);
         writeln('Arquivo ''', Nome, ''' apagado com sucesso');
      end;
END.
```

13.2. Arquivos Tipados ou Binários

Os arquivos tipados ou binários, ao contrário dos arquivos texto que só aceitam caracteres, podem ter como componentes qualquer tipo Pascal, incluindo os estruturados (array e record).

Além dessa diferença, o acesso a arquivos tipados pode ser feito de forma randômica.

Definindo arquivos tipados

Para definir um arquivo tipado, deve-se utilizar a seguinte sintaxe:

```
file of <tipo-do-componente>
```

Exemplo: Definindo um arquivos tipados que contêm inteiros e caracteres

```
type
   TArqInt = file of integer;
var
   ArqInt  : TArqInt;
   ArqChar : file of char;
```

Lendo e Escrevendo arquivos tipados

A sintaxe dos procedimentos `read`, `readln`, `write` e `writeln` é idêntica à utilizada para leitura de arquivos tipo texto. Entretanto, quando são utilizados em arquivos tipados passam para a posição seguinte (próximo componente), permitindo que a próxima leitura ou escrita seja feita no local apropriado. Os índices dos componentes são baseados em 0.

Abertura e Fechamento de arquivos tipados

Os procedimentos disponíveis para abrir arquivos são `reset` e `rewrite`, também disponíveis para arquivos texto, porém têm semântica diferente.

O procedimento `reset` abre o arquivo para leitura e escrita simultaneamente. Quando o arquivo não existe provoca um erro.

O procedimento `rewrite` também abre o arquivo para leitura e escrita simultaneamente mas sempre cria um novo arquivo (apagando qualquer um já existente), não provocando erros quando algum já existe.

Para fechar um arquivo tipado, utiliza-se o procedimento `close`, que também segue a mesma sintaxe quando aplicado a arquivos tipo texto.

Acesso Randômico

Os arquivos tipados podem ser acessados seqüencialmente (utilizando-se `read's` consecutivos) ou de forma randômica.

O procedimento `Seek`, cuja sintaxe é mostrada abaixo, move o "apontador" que indica a posição corrente para uma posição qualquer dentro do arquivo.

```
Seek(<ident-do-arquivo>, <pos>)
```

Diferentemente dos arquivos tipo texto, pode-se utilizar os procedimentos `FileSize` e `FilePos` para descobrir, respectivamente, o tamanho do arquivos e a posição corrente no

arquivo. Ambos recebem como parâmetro o identificador do arquivo, como mostrado abaixo:

```
FileSize(<ident-do-arquivo>)
FilePos(<ident-do-arquivo>)
```

Exemplo: Escrevendo e Lendo um arquivo onde os componentes são arrays de strings usando acesso randômico.

```
program Ex3Cap12;
uses crt;
type
   TVetStr = array[1..3] of string;
var
   ArqArray     : file of TVetStr;
   ElemArqArray : TVetStr;
   I, J         : integer;

function ToStr(X : byte) : string;
var S : string;
begin
   Str(X, S);
   ToStr := S;
end;

BEGIN
   ClrScr;
   assign(ArqArray, 'ArqArray.dat');
   rewrite(ArqArray);

   for I := 1 to 5 do
   begin
      for J := 1 to 3 do
         ElemArqArray[J] := 'Componente(' + ToStr(I) + ','
                                          + ToStr(J) + ')';
      write(ArqArray, ElemArqArray);
   end;

   { Imprimindo invertido }
   for I := FileSize(ArqArray) downto 1 do
   begin
      seek(ArqArray, I - 1);
      read(ArqArray, ElemArqArray);

      for J := 1 to 3 do
         writeln(ElemArqArray[J]);
   end;

   close(ArqArray);
END.
```

Truncando arquivos tipados

Existe um procedimento chamado `truncate` cuja finalidade é remover todos os componentes de um arquivo tipado, a partir da posição atual do arquivo. A sintaxe do procedimento `truncate` é mostrada abaixo:

```
Truncate(<ident-do-arquivo>)
```

13.3. Arquivos Sem Tipo

Arquivos sem tipo são úteis quando se deseja mais desempenho nas operações de entrada e saída. A transferência é baseada em blocos de dados que podem ler lidos ou gravados no meio de armazenamento secundário.

Para definir um arquivo sem tipo é necessário declará-lo como do tipo `file`, como mostrado abaixo:

```
var F : file;
```

Abrindo arquivos sem tipo

Para abrir um arquivo sem tipo utiliza-se os procedimentos `Reset` e `Rewrite`, onde o primeiro abre o arquivo para leitura e o segundo para gravação. A sintaxe desses comandos é mostrada a seguir:

```
Reset(<ident-do-arquivo>, [<tamanho-do-registro>])
Rewrite(<ident-do-arquivo>, [<tamanho-do-registro>])
```

Se o parâmetro que indica o tamanho do registro não for informado, o padrão adotado é 128 bytes. Os registros são lidos em conjunto, formando blocos.

Para ler blocos de dados de arquivos sem tipo, utiliza-se o procedimento `BlockRead` e para gravar `BlockWrite`, cuja sintaxe é apresentada abaixo:

```
BlockRead(<ident-do-arquivo>, <buffer>, <num-reg>, <num-reg-lidos>)
BlockWrite(<ident-do-arquivo>, <buffer>, <num-reg>, <num-reg-escritos>)
```

Onde:

- **ident-do-arquivo**: Identificador do arquivo de onde os blocos estão sendo lidos ou gravados.

- **buffer**: Variável que contém os dados lidos ou que serão gravados.

- **num-reg**: Número de registros que se pretende ler ou gravar.

- **num-reg-lidos**: Número de registros efetivamente lidos. Esse parâmetro só é diferente de `num-reg` quando se atinge o final do arquivo e não há mais dados para preencher totalmente o *buffer*.

- **num-reg-escritos**: Número de registros efetivamente gravados. Esse parâmetro só deve ser diferente de `num-reg` se o disco estiver cheio, não sendo possível gravar todo o conteúdo do *buffer*.

Exemplo: Comparando dois arquivos byte por byte

```
program Ex4Cap12;
const
   TAM_BUFFER = 32;
type
   TBuffer = array[1..TAM_BUFFER] of byte;

function BuffersIguais(var B1, B2   : TBuffer;
                       TamB1, TamB2 : integer) : boolean;
var I : integer;
begin
   BuffersIguais := false;

   if TamB1 <> TamB2 then
      exit;

   for I := 1 to TamB1 do
      if B1[I] <> B2[I] then
         exit;

   BuffersIguais := true;
end;

var
   NomeArq1, NomeArq2   : string;
   Arq1, Arq2           : file;
   Buf1, Buf2           : TBuffer;
   LidosBuf1, LidosBuf2 : integer;
   Iguais               : boolean;

BEGIN
   if ParamCount <> 2 then
   begin
      writeln('Entre com os nomes dos dois arquivos');
      halt(1);
   end;

   assign(Arq1, ParamStr(1));
   assign(Arq2, ParamStr(2));

   reset(Arq1, 1);
   reset(Arq2, 1);
```

```
   Iguais := true;
   repeat
      BlockRead(Arq1, Buf1, SizeOf(TBuffer), LidosBuf1);
      BlockRead(Arq2, Buf2, SizeOf(TBuffer), LidosBuf2);

      if not BuffersIguais(Buf1, Buf2, LidosBuf1, LidosBuf2) then
      begin
         Iguais := false;
         break;
      end;

   until (LidosBuf1 = 0) or (LidosBuf2 = 0)
      or (LidosBuf1 <> SizeOf(TBuffer))
      or (LidosBuf2 <> SizeOf(TBuffer));

   close(Arq1);
   close(Arq2);

   if Iguais then
      writeln('Os arquivos sao iguais')
   else
      writeln('Os arquivos sao diferentes');
END.
```

14. Recursão

É uma técnica de programação bastante poderosa. Pode ser utilizada, por exemplo, na definição do fatorial de um número:

```
N! = 1, se N = 0
N! = N x (N-1)!, se N > 0
```

Da definição, temos que

```
0! = 1
1! = 1 x 0! = 1 x 1
2! = 2 x 1! = 2 x 1 x 0! = 2 x 1 x 1 = 2.
```

0! É conhecido como **caso base**, **caso simples** ou **caso de parada**.

Uma função ou procedimento são recursivos quando definidos em função de si mesmos. Além disso, deve partir de casos complexos para casos mais simples até chegar ao caso simples ou base, onde não é mais necessário usar recursão.

Exemplo: Função que calcula o fatorial de um número

```
program Ex1Cap13;

function Fatorial (N : integer) : longint;
begin
   if N = 0 then
      Fatorial := 1
   else
      Fatorial := N * Fatorial(N - 1)
end;

procedure Teste(N : integer);
begin
   writeln('Fatorial(', N, ') = ', Fatorial(N));
end;

var I : integer;

BEGIN
   for I := 0 to 5 do
      Teste(I);
END.
```

Ao executar a função fatorial com o valor 3 teríamos:

```
Fatorial(3) ⇔ 3 * Fatorial(2) ⇔ 3 * 2 * Fatorial(1) ⇔
3 * 2 * 1 * Fatorial(0) ⇔ 3 * 2 * 1 * 1 ⇔ 6
```

Exemplo: Executa a soma dos N primeiros elementos de um vetor de inteiros usando recursão.

```
program Ex2Cap13;
type
   Vetor = array[1..100] of integer;
var
   V : Vetor;

function SomaVetor (V : Vetor; N : integer) : integer;
begin
   if N = 1 then
      SomaVetor := V[1]
   else
      SomaVetor := V[N] + SomaVetor(V, N - 1)
end;

BEGIN
   V[1] := 3;
   V[2] := 4;
   V[3] := 7;
   V[4] := 1;
   V[5] := 8;

   { Imprime 23 }
   writeln('SomaVetor = ', SomaVetor(V, 5));
END.
```

O valor do N-ésimo elemento da seqüência de Fibonacci é calculado através da função:

```
Fibo(N) = N - 1, se N = 1 ou N = 2
Fibo(N) = Fibo(N - 2) + Fibo(N - 1), se N > 2
```

Seguindo essa função a seqüência seria:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Exemplo: Função que retorna o N-ésimo elemento da seqüência de Fibonacci.

```
program Ex3Cap13;

function Fibo (N : word) : word;
begin
   if (N = 1) or (N = 2) then
      Fibo := Pred(N)
   else
      Fibo := Fibo(N - 2) + Fibo(N - 1)
end;

var I : integer;

BEGIN
   for I := 1 to 10 do
```

```
   begin
      write(Fibo(I));

      if I < 10 then
         write(',')
      else
         writeln;
   end
END.
```