UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

PROCESSAMENTO DE DADOS 1
LINGUAGEM DE PROGRAMAÇÃO PASCAL

Francisco Rapchan
rapchan@writeme.com
http://www.inf.ufes.br/~rapchan/pd1/

Capítulo 1 INTRODUÇÃO

O objetivo desta pequena apostila é servir de material de apoio para os cursos introdutórios de Processamento de Dados ministrados pelos professores do Departamento de Informática do Centro Tecnológico da UFES – Universidade Federal do Espírito Santo. Deve ficar claro que não há aqui nenhuma pretensão em substituir os livros texto. Ao contrário, os livros devem ser adquiridos e estudados com afinco pois trazem informações valiosas e de uma forma muito mais abrangente e completa do que esta pequena apostila.

Os capítulos seguem a ordem natural do aprendizado da programação. Começamos no capítulo 2 tratando de algoritmos de uma forma bem superficial. Em muitas instituições de ensino é usado um semestre inteiro só neste tópico tamanha é a sua importância.

O capítulo 3 mostra as primeiras características da linguagem Pascal. Embora esta linguagem tenha sido adotada muito mais pelo seu aspecto didático do que por sua aplicabilidade em sistemas reais já neste capítulo começamos a construir programas úteis para aplicação nas ciências exatas.

O capítulo 4 mostra dos tipos primitivos da linguagem Pascal tais como real, integer, char, string, etc.

O capítulo 5 mostra a importância da modularização como elemento de qualidade de desenvolvimento de software. Neste capítulo mostramos como a modularização pode ser feita em Pascal usando funções e procedimentos.

No capítulo 6 apresentamos uma forma muito comum de tratar dados: os vetores.

Abaixo listamos uma pequena bibliografia para consulta. Grandes partes desta apostila foram baseadas ou extraídas desta bibliografia.

- Programação em Pascal. Byron S. Gottfried. McGraw-Hill - Coleção Schaum (livro texto principal adotado para a disciplina).
- Apostila de Técnicas de Programação. Osmar de Oliveira Braz Jr. http://tec1.unisul.rct-sc.br/osmarjr/
- Apostila de Pascal. José Maria Rodrigues Santos Júnior. DI - Universidade Tiradentes.
- http://www.inf.pucrs.br/~malu/estrut/algo198.html
- http://www.inf.pucrs.br/~fldotti/lapro1/index.htm
- http://www.dcc.unicamp.br/~hans/mc111/98s1/#program

Capítulo 2 NOÇÕES BÁSICAS DE ALGORITMOS

O algoritmo é certamente um dos conceitos mais importantes na programação de computadores. Durante este curso daremos ênfase no desenvolvimento de algoritmos utilizando estruturas simples de dados.

2.1 O QUE SÃO ALGORITMOS?

Algoritmo é a descrição, de forma lógica, dos passos a serem executados no cumprimento de determinada tarefa. É a forma pela qual descrevemos soluções de problemas do nosso mundo, afim de serem implementadas utilizando os recursos do mundo computacional. Dito de outra forma, um algoritmo é uma seqüência lógica de ações a serem executadas para se executar uma determinada tarefa.

Algoritmo não é a solução de um problema, pois, se assim fosse, cada problema teria um único algoritmo. Algoritmo é um caminho para a solução de um problema, e em geral, os caminhos que levam a uma solução são muitas.

Um dos grandes teóricos da computação e criador da linguagem de programação Pascal, Niklaus Wirth define que

Programas = Algoritmos + Estrutura de dados

O algoritmo não é uma etapa na formação de um programa. Não é um rascunho do programa, nem mesmo um programa em uma linguagem informal. O algoritmo é o conjunto de instruções que manipula os dados para obtenção de um resultado.

Por exemplo suponha o seguinte algoritmo para fazer um bolo (atenção não execute este algoritmo sozinho, peça ajuda de alguém que realmente saiba cozinhar!)

1. Pegue os ingredientes: ovos, trigo, sal, leite, açúcar e fermento
2. Bata em uma tigela a manteiga com o açúcar até virar uma pasta branca
3. Acrescente as gemas, o leite, o trigo e o sal.
4. Bata até formar um creme homogêneo.
5. Acrescente o fermento e misture.
6. Coloque o creme na assadeira
7. Coloque para assar por 20 minutos

Neste caso temos os dados como os ingrediente (ovos, trigo, sal, leite, açúcar e fermento) e o algoritmo é a própria receita que manipula estes ingredientes.

Um mesmo algoritmo pode ser escrito de formas diferentes.

Esta mesma receita escrita de outra forma poderia ser:

1. Pegue os ingredientes: ovos, trigo, sal, leite, açúcar e fermento
2. Coloque a manteiga e o açúcar em uma tigela
3. Enquanto não formar um creme branco:

   Bata a manteiga e o açúcar da tigela
4. Acrescente as gemas, o leite, o trigo e o sal.

5. Enquanto não formar um creme homogênio:

   Bata as gemas, o leite, o trigo e o sal.

6. Acrescente o fermento e misture.
7. Coloque o creme na assadeira
8. Coloque no forno
9. Aguarde :

   Assar

   até passar 20 minutos

2.2 ALGORITMOS COMPUTACIONAIS

Pode-se aprender a construir programas fazendo os algoritmos usando uma linguagem de programação como C ou Pascal. Entretanto estas linguagens apresentam uma série de dificuldades extras para quem está começando a programar tais como:

- São extremamente formais. O estudante pode se perder na sintaxe da linguagem perdendo o algoritmo de vista.
- Usam termos em Inglês. Outra fonte de confusão para quem inicia a programar e não conhece bem o idioma inglês.

Uma outra forma de aprender a fazer algoritmos é utilizar uma linguagem hipotética e mais tarde, assim que os principais conceitos tenham sido entendidos, mostrar a "tradução" desta linguagem em uma linguagem real.

Neste curso, utilizarmos inicialmente uma linguagem parecida com o Pascal porém em português e muito menos formal. Dizemos que uma linguagem não formal é uma pseudo – linguagem de programação. Há várias pseudo – linguagens, algumas muito conhecidas como o PORTUGOL e o Fluxograma. A primeira é uma mistura de português com Pascal e a segunda é uma forma gráfica de representação de algoritmos.

Exemplo

Faça um algoritmo em pseudo linguagem que leia dois números que o usuário do computador digitará no teclado, some-os e mostre o resultado na tela do computador.

Uma solução seria:

```
Leia do teclado dois valores
Some os dois valores
Mostre o resultado da soma na tela
```

Uma solução um pouco mais formal seria:

```
Leia o valor do teclado e armazene na memória A
Leia o valor do teclado e armazene na memória B
Some os valores da memória A e B e coloque o resultado na memória
                                                                SOMA
Mostre na tela o valor da memória SOMA.
```

Nesta solução vemos o uso de 3 memórias: A, B e SOMA. Aqui o conceito de memória é semelhante ao das memórias existentes nas calculadoras. Elas servem para acumular o resultado de cálculos intermediários.

Uma solução ainda mais formal seria:

```
INÍCIO
  Leia (A)
  Leia (B)
  SOMA ← A + B
  Mostre(SOMA).
FIM
```

Aqui as palavras INÍCIO e FIM foram usadas para delimitar o algoritmo.

Observe que na última solução, usamos uma notação bem mais resumida. Embora aqui cada linha corresponda àlinha do exemplo anterior, vemos que utilizamos bem menos palavras.

- Na primeira linha por exemplo, ao invés de dizer: Leia o valor do teclado e armazene na memória A, dissemos apenas Leia (A), significando a mesma coisa.
- Na terceira linha, ao invés de dizer: Some a memória A com a memória B e coloque o resultado na memória SOMA, dissemos apenas SOMA ← A + B, significando a mesma coisa.

De fato, mesmo em pseudo - linguagens de programação é necessário algum formalismo. Caso contrário teremos muita dificuldade para adaptar (traduzir) os algoritmos construídos na pseudo – linguagem em algoritmos de uma linguagem rela de programação.

2.3 O USO DE VARIÁVEIS EM ALGORITMOS

Hoje em dia qualquer calculadora tem pelo menos uma memória para armazenar valores intermediários dos cálculos. As calculadoras um pouco mais sofisticadas têm dezenas destas memórias. Nestas calculadoras, para diferenciar uma memória da outra, normalmente são usadas letras do alfabeto (A, B, C, D....) ou símbolos como M1, M2, M3... representando a memória 1, memória 2 e assim por diante.

No computador a idéia de memória é um pouco diferente do que em uma calculadora. Quando vamos usar memória em um algoritmo para armazenar algo como um resultado de uma operação ou mesmo um número ou uma palavra, precisamos informar primeiro ao computador que precisaremos desta memória.

Informalmente é como se disséssemos ao computador: “Reserve memória para que eu possa usar em um cálculo com números inteiros” ou “ Reserve memória para que eu possa usar em um cálculo com números reais” e assim por diante.

Alem de reservar a memória, temos que dizer ao computador como iremos referenciar aquela memória. Então, diríamos de alguma forma para o computador: “Reserve memória para que eu possa usar em um cálculo com números inteiros e chame esta memória de A” ou então: “Reserve memória para que eu possa usar em um cálculo com números inteiros e chame esta memória de SOMA” e assim por diante. Chamamos os endereços de memória respectivamente de A e de SOMA. Poderíamos ter chamado de B, de VALOR, de CONTADOR, de SALÁRIO, de DÍVIDA ou do que quiséssemos.

Dizemos que estes endereços nomeados de memória são as **variáveis** do programa.

Então poderíamos ter dito: “Crie a variável A do tipo inteiro” que seria o equivalente a dizer: “Reserve memória para que eu possa usar em um cálculo com números inteiros e chame esta memória de A”. Uma forma ainda mais sintética seria dizer simplesmente “Variável A : inteiro”. Se tivermos que usar as variáveis A, B e SOMA faríamos:

```
Variáveis
  A, B, C: inteiro
```

Uma variável representa uma localização de memória do computador utilizada para armazenar valores. Uma variável simples pode assumir diversos valores ao longo do tempo, mas em um dado instante ela

representa exatamente um. Sempre encontramos na posição de memória representado por uma variável o último valor lá depositado. Eventuais valores anteriores não são mais recuperáveis. Uma vez atribuído um valor a uma variável, o valor anteriormente armazenado se perde.

Exemplo

Faça um algoritmo em pseudo linguagem que leia dois números que o usuário do computador digitará no teclado, some-os e mostre o resultado na tela do computador.

```
Variáveis
      A, B, C : do tipo inteiro.

Início
    Leia (A)
    Leia (B)
    SOMA ← A + B
    Mostre(SOMA)
Fim
```

Capítulo 3 INTRODUÇÃO À LINGUAGEM PASCAL

A linguagem de programação PASCAL foi criada para ser uma ferramenta educacional, isto no início da década de 70 pelo Professor Niklaus Wirth do Techinical University em Zurique. Foi batizada pelo seu idealizador de PASCAL, em homenagem ao grande matemático Blaise Pascal, inventor de uma das primeiras máquinas lógicas conhecidas. Foi baseada em algumas linguagens estruturadas existentes então, ALGOL e PLI, tentando facilitar ao máximo o seu aprendizado.

O PASCAL somente ganhou popularidade quando foi adotado pela Universidade da Califórnia, San Diego, em 1973. No mesmo período, em seus cursos, também foram feitas implementações para minis e microcomputadores.

Nas suas primeiras implementações, não era muito amigável ao programador, pois eram necessários vários passos para se obter um programa executável. Primeiro devia se escrever o programa em um editor de texto, depois compila-lo, "lincá-lo" e montá-lo. Quando era feita uma manutenção no mesmo, todos estes passos deviam ser refeitos e que não estimulava os programadores.

Apesar de todas as dificuldades iniciais, de seu propósito educacional e a facilidade de programação, o PASCAL começou a ser utilizado por programadores de outras linguagens, tornando-se para surpresa do próprio Nicklaus, um produto comercial. Contudo somente ao final do ano de 1983, é que a softhouse americana Borland International, lançou o TURBO PASCAL para microcomputadores, aliado ao lançamento no mercado de microcomputadores.

3.1 PRIMEIRO PROGRAMA EM PASCAL – OLÁ MUNDO!

Aprender a programar em uma linguagem nem sempre é uma tarefa linear, que vai do mais fácil para o mais difícil. As vezes temos que ver algo um pouco mais difícil, estudar um pouco e então conseguir entender tudo. O objetivo deste item é mostrar alguns programas em Pascal, dando uma breve descrição de seu funcionamento para que o leitor possa se “acostumar” a forma dos programas Pascal. As explicações completas sobre o funcionamento e uso de cada detalhe do programa será visto durante o curso.

EXEMPLO

Escreve na tela do computador: Olá mundo! .

```
program OlaMundo;
begin
  writeln(´Olá mundo!´);
end.
```

EXEMPLO

Escreve na tela os números de 1 até 100, um em cada linha.

```
program Numeros;
var
  I : integer;
begin
  for I := 1 to 100 do
       writeln(I);
```

```
end.
```

3.2 A FORMA GERAL DE UM PROGRAMA PASCAL

Um programa em Pascal consiste de um Cabeçalho (program header) seguido por uma Seção de Declarações, onde todos os objetos locais são definidos, e de um Corpo, onde são especificados, através de comandos, as ações a serem executadas sobre os objetos.

```
PROGRAM Nome_Do_Programa;            { Cabeçalho }
[ declaração de units         ]
[ declaração de rótulos  ]
[ declaração de constantes    ]     { Seção de Declarações }
[ declaração de tipos         ]
[ declaração de variáveis     ]
[ declaração de subprogramas ]
begin
  comando [ ; comando] ...           { Corpo do Programa }
end.
```

3.2.1 CABEÇALHO DO PROGRAMA

O cabeçalho consiste da palavra reservada Program seguida de um identificador que corresponde ao nome do programa, terminando com um ponto e vírgula.

Exemplos:

```
program Circulo;
program Relatório;
```

3.2.2 SEÇÃO DE DECLARAÇÕES

A sintaxe do pascal requer que todos os identificadores usados no programa sejam predefinidos ou que sejam declarados antes que você possa usa-los.

A declaração de identificadores é feita na seção de declaração onde são associados nomes aos objetos que serão usados no programa, tais como tipos de dados, variáveis e sub-programas.

3.2.3 CORPO DO PROGRAMA

Onde são especificados, através dos comandos da linguagem as ações do programa. É a onde fica lógica do programa.

3.3 IDENTIFICADORES

Os identificadores são nomes a serem dados a variáveis, tipos definidos, procedimentos, funções e constantes nomeadas.

Os identificadores são formados com letras de “a” até “z”, dos dígitos de “0” até “9” e dos caracteres especiais “_” . Um identificador deve começar com uma letra ou com “_” e deve ser único nos primeiros 63 caracteres.

Não existe distinção entre letras maiúsculas e minúsculas no nome de um identificador. Por exemplo, os nomes ALPHA, alpha e Alpha são equivalentes. Atenção para o fato de que identificadores muito longos são mais fáceis de serem lidos pelas as pessoas quando se usa uma mistura de letras maiúsculas e minúsculas; por exemplo, SalarioMinimo é mais fácil de se ler do que SALARIOMINIMO.

EXEMPLOS DE IDENTIFICADORES VÁLIDOS

```
PAGAMENTO, Soma_Total, MaiorValor, Media1, _Media
```

EXEMPLOS DE IDENTIFICADORES INVÁLIDOS

```
%Quantidade     O símbolo % não é permitido
4Vendedor       Não pode começar com um número
Soma Total      Não pode ter espaços entre as letras
```

Observação : Um identificador deverá ficar inteiramente contido em uma linha do programa, ou seja você não pode começar a digitar o nome do identificador numa linha e acabar em outro.

3.4 PALAVRAS RESERVADAS

Pascal reconhece certo grupo de palavras como sendo reservadas. Essas palavras tem significado especial e não podem ser usadas como identificadores em um programa. A seguir listamos todas as palavras reservadas do Pascal Padrão:

and	downto	In	packed	to
array	else	Inline	procedure	type
asm	End	Interface	program	unit
begin	File	Label	record	until
case	For	Mod	repeat	until
const	Foward	Nil	set	uses
constructor	Function	Not	shl	var
destructor	Goto	object	shr	while
div	If	of	string	with
do	implementation	or	then	xor

3.5 COMENTÁRIOS

Comentários são textos escritos dentro do código-fonte do programa para explicar ou descrever alguns aspectos relativos ao mesmo. Os comentários podem ser colocados em qualquer lugar do programa onde um espaço em branco possa existir.

Você pode colocar comentários de duas formas: ou envolvendo o texto entre chaves **{..}** ou entre uma combinação parêntese asterisco e asterisco parêntese **(* .. *)**.

Quando o compilador encontra o símbolo **{** ele salta todos os caracteres até encontrar um **}**. Da mesma forma, todos os caracteres que seguem **(*** são pulados até ser detectado o símbolo ***)**.

3.6 LENDO E MOSTRANDO DADOS

Para a entrada elementar de dados através do dispositivo padrão de entrada, que, via de regra, é representado pelo teclado, existem os comandos **read** e **readln**.

O dispositivo padrão de saída é, via de regra, o monitor do computador. A saída neste dispositivo é gerada com os comandos **write** e **writeln**. Estes comandos adicionam caracteres ao fluxo de dados associado ao dispositivo padrão, onde são devidamente visualizados. Os dois comandos adicionam os caracteres correspondentes à representação simbólica dos valores associados aos seus parâmetros.

Além deste comportamento, o comando **writeln** ainda adiciona os caracteres **CR** e **LF** ao fluxo de saída e causa com isto uma mudança de linha na visualização dos dados na tela do monitor do computador após o último caractere escrito pelo comando. O cursor na tela indica a próxima posição para a escrita de novos valores.

Suponha que a variável k contenha o valor 18 (10010, em binário) e a variável m o valor 20 (10100, em binário). Se você executar o comando

```
write(k,m);
```

e o cursor se encontrar no início de uma linha, então os valores binários das variáveis são convertidos para a sua representação simbólica e você terá, nesta linha, o seguinte texto:

```
1820_
```

onde _ representa a posição do cursor após esta operação. Conforme você pode observar, os dois valores encontram-se justapostos. Se você quiser evitar isto, você terá que escrever um ou mais brancos entre estes dois valores. Isto pode ser feito da seguinte forma:

```
write(k,' ',m);
```

O segundo argumento representa este espaçamento de uma posição. Na situação acima, você obterá a seguinte linha de saída:

```
18 20_
```

Se você tivesse optado pelo comando writeln ao invés do comando write, o comando executado seria

```
writeln(k,' ',m);
```

e a saída correspondente seria

```
18 20
_
```

Como você pôde observar nos exemplos acima, o espaço para representar os valores dos argumentos na sua forma simbólica é a menor possível e, portanto, o espaço ocupando depende destes valores. Se você quiser ter mais controle sobre o espaço a ser utilizado, você usa *parâmetros formatados.*. Por exemplo, o comando

```
write(k:5);
```

indica que a representação simbólica do valor de k será utilizado um "campo" de 5 posições. Neste campo também deverá ser acomodado um possível valor negativo. Se menos do que 5 posições são necessárias para a representação do valor de k, então a representação é alinhada à esquerda e as posições não ocupadas à direita são "preenchidas" com o caractere branco. A formatação para valores inteiros é indicada pelo caractere : seguido de uma expressão que deve resultar em um valor inteiro.

Para reais, temos duas expressões, cada uma precedida por um caractere :. Neste caso, o valor da primeira expressão define o tamanho do campo, onde será representado o valor, e o segundo o numero de casas decimais desejadas. No campo dimensionado pelo valor da primeira expressão devem ser acomodados o sinal do número, a parte inteira, o ponto decimal e a parte fracionária.

Suponha que x contenha o valor 17.527. Se o cursor estiver no início de uma linha e você executar o comando

```
writeln(x:5:2);
```

então você obterá o seguinte resultado:

```
 17.52
 –
```

como no caso de inteiros, valores que não ocuparem todo o campo alocado são precedidos por caracteres brancos. Se, por outro lado, o campo estiver subdimensionado, tanto no caso de inteiros como de reais, então o campo é preenchido com asteriscos.

3.7 DECLARAÇÃO DE VARIÁVEIS

Uma variável deve ser declarada na seção apropriada (início do programa principal, procedimento ou função) de um programa onde ela será usada. Através de uma declaração associamos ao nome da variável um tipo de dados que restringe os valores que a variável em questão pode assumir. Em uma declaração de variáveis não associamos valores particulares a uma variável específica. Ao iniciar-se a execução do programa, dizemos que as variáveis declaradas têm valores indefinidos. Eventualmente o sistema de execução atribui um determinado valor, mas não convém confiar nisso. Inicialize as suas varíaveis na primeira oportunidade. Isto é uma **boa prática de programação**.

EXEMPLO

```
var
  ano : integer;
```

3.7.1 ATRIBUIÇÃO DE VALORES A VARIÁVEIS

Quando definimos uma variável é natural atribuirmos a ela uma informação. Uma das formas de colocar um valor dentro de uma variável é através da atribuição direta do valor desejado que a variável armazena.

Para isto utilizaremos o símbolo := que equivale ao ← no pseudo-código

EXEMPLO

```
PROGRAMA Teste
VARIÁVEIS
      Número:  INTEIRO
INICIO
  Número ¬ 10
FIM
```

```
PROGRAM Teste;
VAR
      Numero:  INTEGER;
BEGIN
  Numero := 10;
END.
```

O Exemplo acima nos informa que:

- Foi definido uma variável, a qual demos o Nome de “Numero”, e informamos que esta variável, ou posição de memória, só poderá aceitar dados, que sejam numéricos e que estejam entre -32768 a +32767 (tipo INTEGER).
- Atribuímos àvariável “Numero” o valor 10

Observação: Quando se atribui um valor a uma variável, seu antigo valor é **perdido** para sempre!

3.8 COMANDO COMPOSTO

Consiste de um ou vários comandos que devam ser executados sequencialmente, sendo usado em situações onde a sintaxe do Pascal permite apenas um único comando.

Os comandos que fazem parte do comando composto são separados por ponto-e-vírgula, devendo começar e terminar pelas palavras reservadas **begin** e **end;**, respectivamente, de acordo com a sintaxe :

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

EXEMPLO

```
if A > B then
begin
  Temp := A;
  A    := B;
  B    := Temp;
end;
```

3.9 COMANDO DE SELEÇÃO *IF*

O comando IF permite efetuarmos um desvio bidirecional na lógica do programa, segundo uma determinada condição booleana (uma condição booleana refere-se àlógica de Boole – veja mais a frente o tipo booleano para mais informações).

Há duas formas para o comando IF em Pascal. Uma sem o ELSE:

```
if <expressão-booleana>  then
   <comando>;
```

E outra com o ELSE:

```
if <expressão-booleana>  then
   <comando>
else
  <comando>;
```

Observe que nestas formas só é possível executar um comando seja dentro do IF, seja dentro do ELSE. Para executarmos conjuntos de comandos devemos usar o begin-end. Exemplo:

```
if <expressão-booleana>  then
  begin
    <comando1>;
    <comando2>;
    ...
    <comandoN>
```

```
  end;
```

Com o ELSE:

```
if <expressão-booleana>  then
  begin
    <comando1>;
    <comando2>;
    ...
    <comandoN>
  end
else
  begin
    <comando1>;
    <comando2>;
    ...
    <comandoN>
  end;
```

Observação: Note que não há sinal de ponto e vírgula (;) entre o **end** e o **else**.

EXEMPLOS

```
if A <= B then
  A := ( A + 1 ) / 2;
```

```
if A > B then
  writelen ('O maior vale:',A)
else
  writelen ('O maior vale:',B);
```

```
if Nome1 = ´Jose´ then
begin
  J := J div 2 + 1;
  writeln( J * 2 );
end;
```

```
if Media >= 5 then
begin
  writeln( ´Aluno Aprovovado´ );
  writeln( ´Parabéns !!!´ );
end
else
  writeln( ´Aluno Reprovado´ );
```

```
if Sexo = MASCULINO then
  if Idade > 18 then
```

```
  begin
    writeln( ´Jovem, aliste-se no Exército, Marinha ou Aeronautica!´ );
    writeln( ´Sirva à sua pátria, venha fazer uma carreira brilhante´ );
  end
  else
    writeln( ´Você ainda é muito jovem para o serviço militar´ );
```

```
if Sexo = MASCULINO then
begin
  if Idade > 18 then
  begin
    writeln( ´Jovem, aliste-se no Exército, Marinha ou Aeronautica!´ );
    writeln( ´Sirva à sua pátria, venha fazer uma carreira brilhante´ );
  end
  else
    writeln( ´Você ainda é muito jovem para o serviço militar´ );

  writeln( ´Fim do programa! ´);
end;
```

EXEMPLO

Dado dois valores A e B quaisquer, faça um algoritmo que imprima se A > B, ou A < B , ou A = B

```
PROGRAM Maior;
VAR
   A,B   :  INTEGER;
BEGIN
   WRITE('Digite os valores A e B');
   IF A > B THEN
         WRITE('A é maior que B')
   ELSE
         IF A < B THEN
               WRITE('A é menor que B')
         ELSE
               WRITE('A é igual a B');
END.
```

3.10 COMANDO DE REPETIÇÃO *WHILE*

O comando WHILE faz com que um comando seja executado enquanto a expressão de controle permanecer verdadeira (TRUE).

Sintaxe :

```
while expressao do
   comando;
```

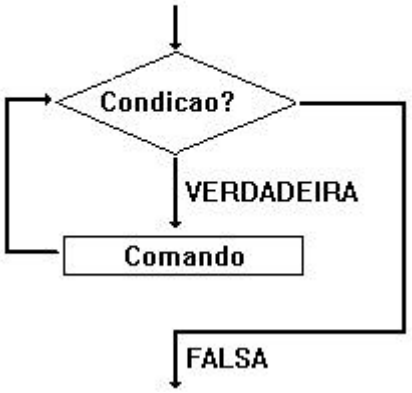
ou

```
while <condição> do
begin
  <comando-1>;
  <comando-2>;
  ...
  <comando-n>
end
```

Se o comando **while** controlar um comando composto (seqüência de comandos delimitados por um **begin** e um **end**.)

A expressão que controla a repetição deverá ser do tipo boolean, sendo a mesma avaliada antes que o comando do laço seja executado. Isto quer dizer que, se logo no início o resultado da expressão for FALSE, o laço não será executado nenhuma vez.

DO-WHILE:



Conforme esboçado acima, inicialmente é testado o valor de <condição>. Se ela for verdadeira, então são executados os comandos <comando-1>; <comando-2>; ... <comando-n> contidos no comando composto controlado pelo comando **while** (ou apenas o <comando>, se o comando **while** controlar apenas a execução de um comando simples). Depois testa-se novamente o valor de <condição>. Se esta continuar ainda verdadeira, então a seqüência de comandos acima é novamente executada e assim sucessivamente, até que <condição> seja falsa.

Observe que a <condição> é sempre testada antes dos comandos controlados serem executados. Se na primeira vez ela já for falsa, então os comandos controlados não chegam a ser executados nenhuma vez. Observe também que para terminar a execução do comando **while**, em algum momento, através de algum dos efeitos causados pela execução dos n comandos, a <condição> deve-se tornar falsa. Caso contrário, o processo de repetição ocorrerá indefinidamente e a execução do programa, no qual encontra-se inserido o comando **while**, não termina. Se seu programa estiver correto, você pode assumir que a <condição> negada é válida após a execução do comando **while**.

Por exemplo, o programa abaixo não vai terminar nunca:

```
program Loop;
var i: integer;
begin
  i := 1;
  while i <> 10 do
  begin
     writeln (i);
     i := i + 2
  end;
end.
```

O que há de errado com o trecho de programa acima?

EXEMPLO

```
program TesteWhile;
var I, J : integer;
BEGIN
  I := 0;
  while I < 5 do
    I := I + 1;
  writeln( I );
  J := 0;
  while J < 5 do
  begin
    J := J + 1;
    writeln( J );
  end;
END.
```

EXEMPLO

Suponha que precisássemos calcular o produto de dois números inteiros x e y sem utilizarmos o operador de multiplicação. Poderíamos fazer isto somando y vezes o valor de x. Desta forma utilizaríamos repetidamente a função de soma para implementar uma função mais complexa multiplicação:

```
program multiplicacao;
var x, y, produto, auxiliar: integer;
begin
  readln(x, y);
  produto:= 0;
  auxiliar:= y;
  while auxiliar > 0 do
  begin
    produto:= produto + x;
    auxiliar:= auxiliar - 1
  end;
  writeln(produto)
end.
```

Após o término da execução do comando **`while`** (`auxiliar` = 0), teremos `produto = x*y`, como desejávamos.

3.11 COMANDO DE REPETIÇÃO *REPEAT – UNTIL*

O comando **repeat** possui a seguinte sintaxe:

repeat

<comando-1>;

<comando-2>;

…

<comando-n>

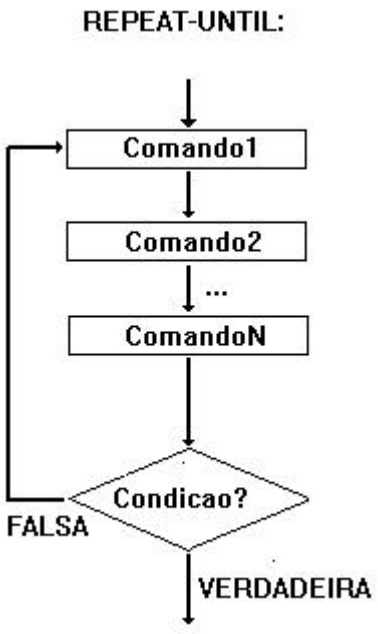
until <condição>

O comando **repeat** age de forma muito parecida com a do comando **while**. A principal diferença é que aqui os comandos são executados antes de se testar o valor de **<condição>**. Assim, a seqüência de comandos <comando-1>; <comando-2>; ...; <comando-n> é executada pelo menos uma vez, independentemente do valor inicial da **condição**.

Outro fato a ser mencionado é que o comando **repeat**, ao contrário do comando **while**, admite uma seqüência de comandos entre as palavras chaves **repeat** e **until**, não havendo, portanto, necessidade de delimitar a seqüência a ser controlada pelo comando **repeat** por um **begin** e um **end**.

A seqüência de comandos controlada, portanto, é executada pelo menos uma vez sendo o processo repetitivo interrompido quando a avaliação da <condição>, ao final de uma execução da seqüência de comandos controlada, for verdadeira. Quando termina a execução do comando **repeat**, você pode assumir que a <condição> é verdadeira.

A figura abaixo ilustra o fluxo de controle do comando **repeat**:



EXEMPLO

Considere o seguinte trecho de programa:

```
i = 1;
while i <= 5 do
begin
    <comando>;
    i := i + 1
end;
```

Para obtermos os mesmos resultados do trecho acima:

```
i := 1;
repeat
  <comando>;
  i:= 1 + 1
until i > 5 ;
```

EXEMPLO

Podemos calcular o fatorial de um número `N` da seguinte forma:

fatorial(0) = 1

fatorial(1) = 1 * fatorial(0) = 1 * 1 = 1

fatorial(2) = 2 * fatorial(1) = 2 * 1 = 2

fatorial(3) = 3 * fatorial(2) = 3 * 2 = 6

...

fatorial(N) = N * fatorial(N-1).

Assim, baseado nesse algoritmo, o seguinte programa implementa uma solução iterativa para o problema proposto:

```
program Fatorial;
var numero, fatorial, auxiliar: integer;
begin
  readln(numero);
  auxiliar:= 1;
  fatorial:= 1;
  repeat
    fatorial:= fatorial * auxiliar;
    auxiliar:= auxiliar + 1
  until auxiliar > numero;
  writeln(fatorial)
end.
```

3.12 COMANDO DE REPETIÇÃO *FOR*

O comando **for** tem o seguinte formato:

```
for <variavel> := <valor1> to <valor2> do <comando>
```

A instrução acima é equivalente ao seguinte comando **while**:

```
<variavel> := <valor1>;
```

```
while <valor2> <= <variavel> do
begin
  <comando>;
  <variavel> := <variavel> + 1
end
```

Existe uma forma alternativa do comando **for** que tem o seguinte formato:

```
for <variavel> := <valor1> downto <valor2> do <comando>
```

Esse comando é equivalente a:

```
<variavel> := <valor1>;
while <variavel> >= <valor2> do
begin
   <comando>;
   <variavel> := <variavel> - 1
end
```

As variantes do comando **for** acima são utilizadas quando *sabemos o número de vezes que o <comando> (comando controlado) deverá ser executado.* Note, também, que <variavel> (chamada de variável de controle) possui um valor diferente a cada vez que <comando> é executado (no primeiro formato, a <variavel> é incrementada por uma unidade a cada iteração (repetição) e, no segundo, decrementada). Os valores assumidos pela <variavel> ao longo da execução do comando **for** normalmente são utilizados em <comando>.

EXEMPLO

```
program somatoria;            {preambulo}
{inicio do bloco}
const n = 10;                 {declaracao de constante}
var i,n,s,termo: integer;     {declaracao de variaveis}
begin
  s := 0;                     {s   d   c}
  for i := 1 to n do          {e   e   o}
  begin                       {q       m}
    write('termo ',i,' -> '); {u       a}
    read(termo);              {e       n}
    s := s + termo;           {n       d}
  end;                        {c       o}
  writeln;                    {i       s}
  write('somatoria = ',s);    {a        }
end.
```

Como será a saída se, ao invés da seqüência de comandos,

```
writeln;
write('somatoria = ',s);
```

tivéssemos

```
writeln('somatoria = ',s);
```

no final do código do programa acima?

EXEMPLO

Suponha que precisássemos calcular a soma S dos N primeiros números naturais:

`S = 0 + 1 + 2 + 3 ... + N.`

Não existe um comando tão específico em Pascal que execute de uma só vez a soma dos `N` primeiros números naturais. Entretanto, podemos realizar esta operação somando dois números por vez, `N` vezes, da seguinte forma:

```
S(0) = 0
S(1) = S(0) + 1 = 0 + 1 = 1
S(2) = S(1) + 2 = 1 + 2 = 3
S(3) = S(2) + 3 = 3 + 3 = 6
S(4) = S(3) + 4 = 6 + 4 = 10
    ...
S(N) = S(N-1) + N.
```

Temos, então, a seguinte solução do problema baseado no algoritmo acima:

```
program Soma;
var N, S, auxiliar: integer;
begin
  readln(N);
  S := 0;
  auxiliar:= 1;
  while N >= auxiliar do
  begin
    S := S + auxiliar;
    auxiliar := auxiliar + 1
  end;
  writeln(S)
end.
```

No programa acima, `auxiliar` assume os valores de `1` a `N` sendo que, a cada execução do comando **`while`**, soma-se o valor corrente de `auxiliar` ao acumulador de soma `S` (inicialmente com valor zero). Como `auxiliar` assume inicialmente o valor `1` e é acrescido de uma unidade a cada repetição, chegará o momento em que `auxiliar` assumirá o valor `N + 1`, o que torna falsa a condição entre as palavras chave **`while`** e **`do`** e causa o término da execução do comando **`while`**. Nesse ponto, `S = 0 + 1 + 2 ... + N`, da forma que desejávamos.

Poderíamos também implementar o mesmo algoritmo utilizando um comando **`repeat`**:

```
program Soma;
var N, S, auxiliar: integer;
begin
  readln(N);
  S := 0;
  auxiliar := 1;
  repeat
    S := S + auxiliar;
    auxiliar := auxiliar + 1
  until auxiliar > N
  writeln(S)
end.
```

ou um comando **`for`**

```
program Soma;
var N, S, auxiliar: integer;
begin
  readln(N);
  S := 0;
  for auxiliar := 1 to N do S := S + auxiliar;
  writeln(S)
end.
```

3.13 COMANDO DE SELEÇÃO MÚLTIPLA *CASE*

O comando CASE permite efetuarmos um desvio multidirecional na lógica do programa. Ele consiste de um expressão (chamada *seletor*) e uma lista de comandos, cada um precedido por constantes ou subintervalos separados por vírgulas (chamados *rótulos de case*), de mesmo tipo do seletor, que pode ser qualquer escalar ordenado (integer, char, boolean, enumerated, faixa exceto real).

```
case expressão of
  rotulo-case : comando;
  [rotulo-case : comando;]...
  [else
  comando [; comando ]...
end;
```

EXEMPLO

Mostre a opção escolhida.

```
program Figuras;
var
Tipo   : integer;
BEGIN
  writeln( 'Qual o tipo da figura ? ');
  writeln( '0-Triangulo' );
  writeln( '1-Retangulo' );
  writeln( '2-Quadrado'  );
  writeln( '3-Circulo'   );
  readln( Tipo );

  Figura := TFigura( Tipo );
  case Figura of
    0 : writeln( 'Você escolheu a figura Triangulo' );
    1 : writeln( 'Você escolheu a figura Retangulo' );
    2 : writeln( 'Você escolheu a figura Quadrado'  );
    3 : writeln( 'Você escolheu a figura Circulo'   );
  end;
END.
```

EXEMPLO

Mostre se a idade é referente a um bebê, criança ou adolescente, seguindo o seguinte critério:

- 0 a 3: ESCREVA('BEBÊ')
- 4 a 10 : ESCREVA('CRIANÇA')
- 11 a 18: ESCREVA('ADOLESCENTE')
- caso contrário: ESCREVA('ADULTO')

Podemos fazer este exemplo simplesmente usando IF's:

```
IF ( IDADE >= 0 ) AND ( IDADE <= 3 ) THEN
      WRITE( 'BEBÊ')
ELSE
```

```
IF ( IDADE >= 4 ) AND ( IDADE <= 10 ) THEN
      WRITE( 'CRIANÇA')
ELSE
      IF ( IDADE >= 11 ) AND ( IDADE <= 18 ) THEN
            WRITE( 'ADOLESCENTE')
      ELSE
            WRITE( 'ADULTO');
```

O Exemplo mostrado poderia ser escrito da seguinte forma usando o comando CASE:

```
CASE Idade OF
      0,1,2,3     :  WRITE('BEBÊ');
      4,5,6,7,8,9,10    :  WRITE('CRIANÇA');
      11,12,13,14,15,16,17,18 :  WRITE('ADOLESCENTE');
ELSE
      WRITE('ADULTO');
END;
```

Ou ainda, de uma forma mais compacta:

```
CASE Idade OF
      0..3  :  WRITE('BEBÊ');
      4..10 :  WRITE('CRIANÇA');
      11..18      :  WRITE('ADOLESCENTE');
ELSE
      WRITE('ADULTO');
END;
```

Observação: O Comando CASE pode ser substituído por um conjunto de IF-THEN-ELSE aninhados, no entanto, nesses casos, onde há muitas condições, o comando CASE, torna o programa mais legível.

EXEMPLO

O programa abaixo imprime na tela a situação do aluno após a sua nota ser digitada.

```
Program Nota;
Var
  nota: integer;
Begin
  writeln('Digite sua nota:');
  readln(nota);
  Case nota of
    0..2: writeln('Nota péssima');
    3..4: writeln('Nota ruim');
    5..6: writeln('Nota pouco ruim');
    7: writeln('Nota boa');
    8..9: writeln('Nota muito boa');
    10: writeln('Nota excelente!!');
    else writeln ('Nota inválida');
  end;
end.
```

EXEMPLO

Mostre a tecla que foi digitada.

Observação: Neste programa utilizamos a variável Tecla com o tipo char. Este tipo de variável permite armazenar valores de teclas.

```
program TestaTecla;
var Tecla : char;
BEGIN
  writeln( 'Digite um número ' );
  readln( Tecla );

  case Tecla of
    'A'..'Z', 'a'..'z':  writeln('Você pressionou uma Letra');
    '0'..'9':            writeln('Você pressionou um Numero');
    '+', '-', '*', '/':  writeln('Você pressionou um Sinal Aritmetico');
  else
    writeln( 'Você pressionou uma outra tecla qualquer' );
  end;
END.
```

3.14 EXERCÍCIOS RESOLVIDOS

Apresentamos abaixo uma série de programas resolvidos. Como exercício, procure refazer estes exercícios.

3.14.1 ALGORITMO DO TRIÂNGULO

Faça um algoritmo para ler a base e a altura de um triângulo. Em seguida, escreva a área do mesmo.

Área = (Base * Altura) / 2

```
program triangulo;
var
  area, base, altura: real;
begin
  { Entrada }
  write ('Digite a base: ');
  readln (base);
  write ('Digite a altura: ');
  readln (altura);

  { Calculos }
  area:= (base*altura)/2;

  { Saida }
  writeln ('A area do triangulo e: ',area:10:2);
end.
```

3.14.2 ALGORITMO PREÇO AUTOMÓVEL

O preço de um automóvel é calculado pela soma do preço de fábrica com o preço dos impostos (45% do preço de fábrica) e a percentagem do revendedor (28% do preço de fábrica). Faça um algoritmo que leia o nome do automóvel e o preço de fábrica e imprima o nome do automóvel e o preço final.

```
program preco;
var
  nome: string;
  Precofabrica,  PrecoFinal, imposto : real;
begin
  write ('Digite nome: ');
  readln (nome);
  write ('Digite preco de fabrica: ');
  readln (Precofabrica);
  Imposto := Precofabrica * (0.45 + 0.28);
  PrecoFinal := PrecoFabrica + imposto;
  writeln ('Automovel:', nome);
  writeln ('Preco de venda:', PrecoFinal:10:2);
end.
```

3.14.3 ALGORITMO MEDIA VALOR

Dado uma série de 20 valores reais, faça uma algoritmo que calcule e escreva a média aritmética destes valores, entretanto se a média obtida for maior que 8 deverá ser atribuída 10 para a média.

```
Program MEDIA_20;
var
     conta:integer;
     media,num,soma:real;
Begin
     conta := 0;
     soma  := 0;
     Writeln ('Digite 20 numeros');
     While conta < 20 do Begin
        read (num);
        soma  := soma + num;
        conta := conta + 1;
     End;
     media := (soma / 20);
     if media > 8 then
         Writeln ('media =10')
     Else Begin
        Writeln ('A media ,');
        Writeln (media);
     End;
End.
```

3.14.4 ALGORITMO 3 MENORES

Faça um algoritmo que leia 3 números inteiros e imprima o menor deles.

```
Program MENOR_N;
var
   n1,n2,n3,menor:integer;
Begin
  Writeln ('Digite 3 valores');
  Read (N1,N2,N3);
  If (N1<N2) And (N1<N3) Then
    menor:=N1
  Else
    If (N2<N1) And (N2<N3) Then
       menor:=N2
    Else
       menor:=N3;
  Writeln ('O menor valor ,');
  Writeln (menor);
End.
```

3.14.5 ALGORITMO MEDIA MAIOR QUE 4

Dado um conjunto de n registros cada registro contendo um valor real, faça um algoritmo que calcule a média dos valores maiores que 4.

```
program maior4;
 var
    n, aux: integer;
    soma, media, numero: real;

begin
   write ('Digite valores diferentes de 999');
    writeln;
    aux := 0;
    soma := 0;
    read (numero);

    while numero <> 999 do
    begin
       if numero > 4 then
       begin
          soma:= soma + numero;
          aux := aux + 1;
       end;
       read (numero);
    end;

    media := (soma/aux);
    write ('MEDIA=');
    write (media);
end.
```

3.14.6 ALGORITMO SALÁRIO

Uma empresa tem para um determinado funcionário uma ficha contendo o nome, número de horas trabalhadas e o n^0 de dependentes de um funcionário.

Considerando que a empresa paga 12 reais por hora e 40 reais por dependentes.

Sobre o salário são feito descontos de 8,5% para o INSS e 5% para IR.

Faça um algoritmo para ler o Nome, número de horas trabalhadas e número de dependentes de um funcionário. Após a leitura, escreva qual o Nome, salário bruto, os valores descontados para cada tipo de imposto e finalmente qual o salário líquido do funcionário.

```
Program Salario;
Var
      Nome:String;
      Numhora, Salbruto, Salliq:Real;
      Numdep:Integer;
Begin
     Write ('Digite O Nome Do Funcionario:');
     Read (Nome);
```

```
      Writeln;
      Write ('Numero De Horas Trabalhadas:');
      Read (Numhora);
      Writeln;
      Write ('E Numero De Dependentes:');
      Read (Numdep);
      Writeln;
      Salbruto:=(12*Numhora)+(40*Numdep);
      Salliq:=Salbruto-((Salbruto*0.085)+(Salbruto*0.05));
      Write ('Nome Do Funcionario:');
      Write (Nome);
      Writeln;
      Write ('Salario Liquido:');
      Write (Salliq);
End.
```

3.14.7 ALGORITMO 50 TRIANGULOS

Faça um algoritmo para ler base e altura de 50 triângulos e imprimir a sua área.

```
program triangulo;
var
   base,altura,area:real;
   contador:integer;
begin
   clrscr;
   contador:=1;
   while contador < 51 do
     begin
       writeln('Digite a Base');
       read(base);
       writeln('Digite a Altura');
       read(altura);
       area:=(base*altura)/2;
       write('Esta e a area do triangulo   ');
    writeln (area);
       contador:=contador+1;
    end;
end.
```

3.14.8 ALGORITMO MEDIA MENORES

Dado um conjunto de 20 valores reais, faça um algoritmo que:

a) Imprima os valores que não são negativos.

b) Calcule e imprima a média dos valores < 0.

```
program numeros;
var
   valor,media,soma:real;
   cont,contpos:integer;
begin
cont:=1;
   while cont < 21 do
      begin
```

```
        writeln('Digite um valor real');
        read(valor);
        if valor < 0 then
          begin
            media:=media+valor;
            contpos:=contpos+1;
          end
        else
          begin
            write (valor);
            writeln (' e um valor maior que Zero');
          end;
        cont:=cont+1;
   end;
   media:=media/contpos;
   write ('O numero de valores menores que Zero sao  ');
   writeln (contpos);
   write ('A media do valores menores que Zero e  ');
   writeln (media);
end.
```

Capítulo 4 TIPOS PRIMITIVOS DE DADOS

Quando definimos variáveis, devemos indicar ao compilador quanto de memória a variável irá usar e como esta memória será usada. Até agora temos usado apenas dois tipos de variáveis: tipo ingeger e tipo real. Entretanto a linguagem Pascal define vários outros tipos de dados.

Alguns tipos são chamados de tipos básicos e são suportados por qualquer compilador Pascal. Há porém outros tipos que também são muito úteis mas sua definição pode variar um pouco de compilador para compilador.

Um tipo de dados especifica as características, ou seja os valores e operações possíveis de serem utilizados com um dado desse tipo. Toda variável e constante usada em um programa tem um tipo associado a ela.

Os tipos podem ser divididos em quatro categorias :

Tipo escalar. Representa uma única peça de dados, podendo ser ordenados, isto é, seus valores mantêm um certa ordem. Os tipos integer, char, boolean, enumerado e subintervalo são ordinais, já o tipo real não é ordenado.

Exemplo

Integer	Números inteiros
Real	Números reais
Boolean	Valores lógico TRUE ou FALSE
Char	Caracteres da Tabela ASCII
Enumerado	Relação de Valores em ordem
Subintervalo	Faixa de valores ordenados

Tipo estruturado. Representa uma coleção de itens de dados.

Exemplo

string	Cadeia de caracteres
array	Conjunto de elementos de mesmo tipo
record	Conjunto de elementos de tipos diferentes
set	Conjunto de elementos
file	Arquivos de registro
text	Arquivos texto

Tipo apontador. Representa uma peça de dados que referencia ou aponta para outra peça de dados.

Exemplo

pointer	Referência para uma variável dinâmica

Tipo definido pelo usuário. Definidos a partir da associação dos tipos de dados anteriores.

Exemplo

```
type
  TMaiusculas = 'A'..'Z';              {Tipo baseado em char}
  TContador   = 1..100;                          {Tipo baseado em
  integer}
  Boolean     = ( FALSE, TRUE ); (Tipo baseado em boolean }
```

Neste curso vamos ver apenas os tipos escalares e alguns tipos estruturados.

4.1 *INGEGER* E *REAL*

O Turbo Pascal permite cinco tipo predefinidos de números inteiros, cada um abrange um subconjunto dos números inteiros. Todos os tipos inteiros são ordinais (ordenados).

Tipo	Faixa de valores	Número de bytes
shortint	-128..127	1
integer	-32768..32767	2
longint	-2147483648..2147483647	4
byte	0..255	1
word	0..65535	2

O Turbo Pascal permite cinco tipo predefinidos de números reais, cada um com um faixa de valores e precisão específicas. Todos os tipos reais não são ordinais.

Tipo	Faixa de valores	Dígitos significativos	Número de bytes
real	2.9e-39..1.7e38	11-12	6
single	1.5e-45..3.4e38	7-8	4
double	5.0e-324..1.7e308	15-16	8
extended	3.4e-4932..1.1e4932	19-20	10
comp	-9.2e18..9.2e18	19-20	8

4.1.1 OPERADORES ARITMÉTICOS

Usados para efetuar operações aritméticas com número inteiros e reais.

Subtração	-
Adição	+
Multiplicação	*
Divisão Real	/

Divisão Inteira (truncada)	div
Resto da Divisão Inteira	mod

EXEMPLO

```
var
  A, B : integer;
  C, D : real;

BEGIN
  A := 1;
  B := 3;
  C := 5;
  D := 10;
  A := 1 + B;
  A := B + D;             { errado, D é real }
  B := 10 div 3;
  A := 10 mod 3;
  C := D / C;
  D := 10 div C;          { errado, o operado div é só para inteiros }
  A := -1;
  B := 5 + A;
  B := -A;
  C := D * A;
  B := C * B;  { errado, C é real }
END.
```

4.2 *BOOLEAN*

O tipo BOOLEAN representa os valores lógicos TRUE e FALSE. O tipo BOOLEAN é ordinal, onde : FALSE < TRUE.

Exemplo:

```
var
  Aprovado : boolean;
  Confirma : boolean;
```

4.2.1 OPERADORES RELACIONAIS DO TIPO BOOLEAN

Os operadores relacionais são usados para definir uma condição, uma comparação entre dados de mesmo tipo. O resultado do uso de operadores relacionais é uma expressão booleana.

Maior que	>
Menor que	<
Maior ou igual	>=
Menor ou igual	<=
Igual	=

Diferente	<>
E	and
OU	or
NÃO	not

EXEMPLO

```
var
  Nota1, Nota2 : real;
  A, B, C : integer;

BEGIN
  A := 2;
  B := 3;
  C := 1;

  if ( B = A + C ) and ( NomeAluno1 <> NomeAluno2 ) then
    writeln(´Maria Luiza´, B );

  if ( A = C ) or ( NomeAluno1 = NomeAluno2 ) then
    writeln(´Maria Jose´);

  if not( A = C ) then
    writeln(´Carmem Antónia´ );
END.
```

4.3 *CHAR* E *STRING*

O tipo CHAR representa um único caracter pertencente àtabela ASCII.

```
var
  Sexo : char;
```

O tipo STRING Armazena uma cadeia de caracteres com o tamanho máximo de até 255 caracteres, mas podemos ser especificar um tamanho menor que 255. A posição [0] da string armazena o seu comprimento. Esse tipo permite a concatenação utilizando-se o operador +.

```
var
  Frase : string;
   Nome  : string[45];
```

As definições de variáveis como sendo do tipo CHAR e STRING, possuem algumas curiosidades que merecem um cuidado especial por parte do usuário.

4.3.1 USO DAS ASPAS SIMPLES (APÓSTROFO) (‘)

Quando estivermos fazendo a atribuição de um valor para uma variável do tipo CHAR (Caracter) ou STRING (Cadeia), temos que ter o cuidado de colocar o valor (dado) entre aspas (‘), pois esta é a forma de informar que a informação é caracter.

EXEMPLO:

```
program Teste;
      var
            Letra : char;
            Nome  : string [10];
      begin
            Letra := 'A';
            Nome  := 'João';
      End.
```

4.3.2 MANIPULAÇÃO DE CARACTERES INDIVIDUAIS EM STRING'S (CADEIAS)

Muitas vezes é necessário manipular caracteres individuais em uma STRING (Cadeia) . O Pascal possui uma forma toda especial de permitir tal operação, através do uso de colchetes([ ]) logo após o Nome da variável do tipo STRING (Cadeia) , e o número do caracter que se deseja manipular entre os colchetes.

EXEMPLO

Atribuir o primeiro caracter de uma STRING a uma variável do tipo CHAR.

```
program AtribuiString;
var
      letra : char;
      Nome  : string;
begin
      Nome := 'Joao';
      letra := Nome[1];
end.
```

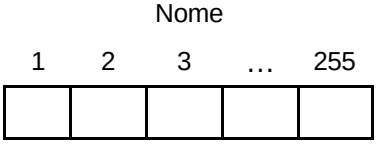
Quando definimos uma variável como sendo do tipo STRING não estamos alocando 1 posição de memória apenas (uma caixa, pela analogia inicial), mas na verdade, estamos alocando até 255 caixas, uma para cada caracter da STRING (lembre-se que uma STRING pode ter no máximo 255 caracteres). Ao utilizarmos o símbolo de colchete, estamos na verdade indicando qual o caracter (qual a caixa) desejamos manipular.

De acordo com o Exemplo acima, teríamos na memória a seguinte situação:

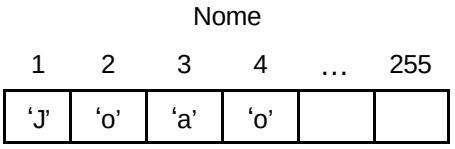
a. Alocamos 255 bytes (caracteres) na memória. A estas posições de memória é dado o Nome de "Nome". Inicialmente estas posições de memória possuem o conteúdo indefinido.

b. Alocamos 1 byte (caracter) na memória. A este caracter é dado o Nome de "Letra". Inicialmente esta posição de memória possuí o conteúdo indefinido.

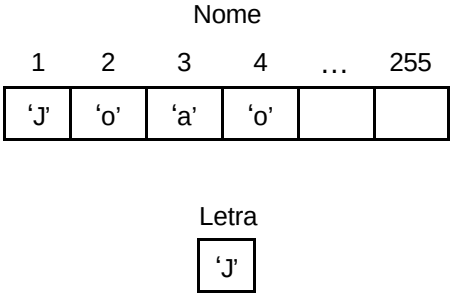
c. Na memória temos a seguinte situação:



d. Atribuímos a variável “Nome” o valor “João”, obtendo na memória a seguinte configuração



e. Atribuímos a variável “Letra” o primeiro caracter da variável “Nome”, ou seja, o conteúdo da primeira posição de memória (caixa). Na memória teremos a seguinte configuração:



Observação. É possível definir variáveis do tipo STRING(Cadeia) com menos de 255 caracteres. Para isto, basta colocar, após a palavra STRING(Cadeia), o número de caracteres desejados entre colchetes ([ ]).

Exemplo:

```
program Define;
var
      Nome: string[80];
```

Desta forma, o espaço ocupado por uma variável STRING(Cadeia) passa de 255 bytes para apenas 80 bytes, na memória.

4.4 TIPO ENUMERADO (*ENUMERATED*)

O tipo escalar enumerado é um escalar ordenado onde os valores que as variáveis deste tipo podem assumir são descritos através de uma lista de valores. Cada valor é um identificador o qual é tratado como uma constante. Isto permite que nomes significativos sejam associados a cada valor usado para as variáveis.

A definição de um tipo enumerado é feita colocando-se entre parênteses os identificadores que as variáveis podem assumir, separados por vírgulas, como mostrado a seguir:

```
var
  Dias : ( Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado );
```

Nesse caso, os valores têm a seguinte ordem :

Domingo < Segunda < Terca < Quarta < Quinta < Sexta < Sabado

EXEMPLO

Abaixo segue um trecho de um programa que lê as horas trabalhadas a cada dia. Observe que Dias vai variar de Segunda até Sabado.

```
program Totaliza_Horas_De_Trabalho;
var
   Dias   :(Domingo,Segunda,Terca,Quarta,Quinta,Sexta,Sabado);
   Meses  :(Jan,Fev,Mar,Abr,Mai,Jun,Jul,Ago,Set,Out,Nov,Dez);
  TotalHoras, HorasDeTrabalho  : integer;

begin
     . . .
  TotalHoras := 0;
  for Dias := Segunda to Sabado do
  begin
    readln( HorasDeTrabalho );
    TotalHoras := TotalHoras + HorasDeTrabalho;
  end;
     . . .
end.
```

4.5 TIPO SUBINTERVALO (*SUBRANGE*)

O Pascal admite também um tipo denominado subintervalo (*subrange*) que representa um subconjunto de valores de tipos escalares ordenados.

Uma variável do tipo subintervalo é declarada da seguinte forma :

```
var
  NumDiadoMes       : 1..31;
  LetrasMaiusculas  : ´A´..´Z´;
  DiaDoAno          : 1..365;
```

4.6 FUNÇÕES PRÉ-DEFINIDAS

O Pascal oferece um conjunto de funções pre-definidas (*built-in functions*), que são usadas com vários tipos de dados simples. As funções, na maioria das vezes, necessitam de dados como parâmetro (dados de entrada). Vejamos algumas dessas funções.

Nome da Função	Objetivo	Tipo do Parâmetro	Tipo do Retorno
abs(x)	Calcula o valor absoluto de x.	inteiro ou real	o mesmo que x
arctan(x)	Calcula o arco-tangente de x em radianos	inteiro ou real	real
cos(x)	Calcula o coseno de x em radianos	inteiro ou real	real
sin(x)	Calcula o seno de x em radianos	inteiro ou real	real
sqr(x)	Calcula o quadrado de x	inteiro ou real	o mesmo que x
sqrt(x)	Calcula a raiz quadrada de x (x>=0)	inteiro ou real	real
odd(x)	Determina se x é par ou impar. TRUE é impar.	inteiro	boolean
exp(x)	Calcula e^x, em que e=2.7182818 é sistema natural de logaritmos neperianos.	inteiro ou real	real
ln(x)	Calcula o logaritmo natural de x (x>0)	inteiro ou real	real
exp(ln(x)*y)	Retorna x elevado a y {utilizando regras de logaritmos}.	Inteiro ou real	real
pi	Retorna o valor de PI (3.1415...)	Nenhum	Real
length(x)	Determina o número de caracteres de x	string	inteiro
concat(x1, x2,...)	Concatena duas ou mais strings (máx 255 caracteres)	string	string
copy(x, y, z)	Retorna uma subcadeia da cadeia x, com z caracteres, começando no caracter y.	string, inteiro, inteiro	string
pos(x, y)	Retorna a posição da cadeia x dentro da cadeia y, se não for encontrada retorna 0.	String, string	inteiro
delete(x, y, z)	Remove z caracteres da cadeia x a partir da posição y	string, inteiro, inteiro	nenhum
Insert(x, y, z)	Insere a cadeia de caracteres x na cadeia y a partir da posição z (max 255)	string, string, inteiro	nenhum
UpCase(x)	Retorna x convertido para maiúscula	char	char
trunc(x)	Trunca x para um número inteiro	real	inteiro
int(x)	Retorna a parte inteira de x	real	real
frac(x)	Retorna a parte fracionária de x	real	real
round(x)	Arredonda x para um inteiro	real	inteiro
chr(x)	Determina o caracter ASCII representado por x	inteiro	char
ord(x)	Determina o inteiro que é usado para codificar x	char	inteiro
pred(x)	Determina o predecessor de x	tipo ordenado	o mesmo de x
succ(x)	Determina o sucessor de x	tipo ordenado	o mesmo de x
sizeof(x)	Retorna o número de byte de x	qualquer tipo	inteiro
inc (x,[y])	Incrementa x de y unidade	tipo ordenado	

Capítulo 5 MODULARIZAÇÃO

Um matemático uma vez disse que um grande problema se resolve dividindo-o em pequenas partes e resolvendo tais partes em separado. Estes dizeres servem também para a construção de programas.

O conjunto de sentenças básicas da linguagem Pascal visto até agora é adequado para escrevermos pequenos programas. Entretanto, se for necessário escrever e testar programas mais sofisticados, que envolvam mais operações, deveremos usar técnicas que nos permitam, de alguma forma, organizar o código fonte.

Alguns aspectos devem ser levados em consideração:

- Programas complexos são muitas vezes compostos por um conjunto de segmentos de código não tão complexos.
- Muitas vezes usamos em um programa trechos que já desenvolvemos em outros programas.

É comum, em programação, decompor a lógica de programas complexos em programas menores e, depois, juntá-los para compor o programa final. Essa técnica de programação é denominada *programação modular*.

A programação modular é num método para facilitar a construção de programas grandes e complexos, através de sua divisão em pequenos módulos, ou subprogramas, mais simples. Estes subprogramas alem de serem mais simples de serem construídos, são também mais simples de serem testados.

Esta tecnica também possibilita o reaproveitamento de código pois podemos utilizar um módulo quantas vezes for necessário. eliminando a necessidade de escrever-lo repetidamente.

Outra importância da modularização é a possibilidade de vários programadores trabalhem simultaneamente na solução de um mesmo problema, através da codificação separada dos diferentes módulos. Assim, cada equipe pode trabalhar em um certo conjunto de móduolos ou subprogramas, construído e testando. Posteriormente estes módulos são integrados formando o programa final. Em outras palavras, quando for necessário construir um grande programa, devemos dividi-lo em partes e então desenvolver e testar cada parte separadamente. Mais tarde, tais partes serão acopladas para formar o programa grande.

Para falarmos um pouco mais sobre as vantagens da modularização necessitamos definir alguns conceitos sobre qualidade de programas. A norma **ISO 9126** define as características gerais de um software de qualidade. Esta norma define 6 características primordiais em um software de qualidade:

Funcionalidade	O programa satisfaz as necessidades para as quais ele foi construído? Ele faz o que se propõem a fazer?
Confiabilidade	O programa é imune a falhas ou, pelo menos, o tempo entre falhas é longo?
Usabilidade	O programa é fácil de usar? É fácil aprender a usar o programa? Há documentação adequada sobre o uso do programa?
Eficiência	O programa é suficientemente rápido? Usa muitos recursos (memória, disco, rede, CPU...) ?
Manutenibilidade	Há documentação adequada sobre o código do programa? É fácil encontrar a origem da falha quando ela ocorre? É fácil corrigi-la? Há grande risco quando se faz alterações? É fácil testar quando se faz alterações?
Portabilidade	É fácil usar em outro ambiente? É fácil modificar e adaptar o programa para outros ambientes?

Estas 6 características são importantíssimas para construirmos código de qualidade. Para alcançarmos este padrão de qualidade precisamos levar muito a sério a questão da modularização.

A modularização, em Pascal, pode ser feita através de *procedimentos (**procedures**) e funções (**functions**).* Isso é feito associando-se um nome a uma seqüência de comandos através do que chamamos *Declaração do Procedimento ou da Função*. Pode-se, então, usar o nome do procedimento ou da função dentro do corpo do programa, sempre que desejarmos que o seu bloco de comandos seja executado, isso é o que chamamos de *Chamada do Procedimento ou da Função*.

Vamos começar tratando as funções. Mais tarde veremos que os procedimentos são na realidade formas mais simples de funções.

5.1 FUNÇÕES

Na matemática fazemos uso intenso de funções. Por exemplo, podemos definir uma função seno: f(x)=sem(x); quadrado: f(x) = x^2; inverso: f(x) = 1/x e muitas outras. Todas estas funções possuem apenas um parâmetro. Na matemática podemos também definir funções com mais de um parâmetro. Por exemplo uma função hipotenusa:

$$f(x, y) = \sqrt{x \ + y}$$

O resultado desta função para x = 3 e y = 4 é 5. Assim dizemos que f(3,4) da função hipotenusa retorna 5. Dizemos que x e y são os parâmetros e os valores 3 e 4 são os argumentos da função hipotenusa.

Nas linguagens de programação também temos funções e são bem parecidas com as funções da matemática.

Por exemplo, em um programa Pascal a função hipotenusa definida acima ficaria da seguinte forma:

```
program teste;
var
   i : real;

function f (x: real; y: real) : real;
var
   h : real;
begin
   h := sqrt(x*x + y*y);
   f := h;
end;

begin
   i := f(3.0,4.0);
   writeln (i);
end.
```

Neste programa definimos uma função chamada simplesmente de **f**. Embora chamar uma função de **f** ou **g** seja prática comum na matemática, na informática normalmente são usados nomes mais expressivos, de preferência que façam menção ao uso ou comportamento da função. No nosso exemplo ficaria melhor então chamar a função de **hipotenusa** ao invés de chama-la simplesmente de **f**.

Observe que definimos a função **f** , seus parâmetros x do tipo real e y também do tipo real. Definimos também o tipo de retorno da função; neste caso, coincidentemente real também.

A figura abaixo explica as principais partes deste programa.

`program teste;`	Cabeçalho do programa .
`var` `  i : real;`	Declaração de variáveis do programa
`function f (x: real; y: real) : real;`	Cabeçalho da função
`var` `  h : real;`	Declaração de variáveis da função

`begin`
`  h := sqrt(x*x + y*y);`
`  f := h;`
`end;`
	Corpo da função

`begin`
`  i := f(3.0,4.0);`
`  writeln (i);`
`end.`
	Corpo do programa principal

Observe que no corpo da função os parâmetros **x** e **y** são usados como variáveis comuns. Outro elemento usado como variável é o próprio nome da função que neste caso chama-se **f**.

Esta é um ponto importante ao tratarmos de funções. O nome da função deve ser usado para receber o valor que será retornado pela função. É assim que o Pascal permite indica o que a função irá retornar.

IMPORTANTE: O nome da função não é uma variável qualquer como as outras. Ele é usado para indicar o valor que a função vai retornar.

5.1.1 SEQÜÊNCIA DE EXECUÇÃO

A seqüência de execução do programa teste do exemplo acima é:

1. O programa teste é carregado na memória e é definido (alocado) espaço de memória para a variável **i**.
2. Começa-se a executar as instruções que estão no corpo do programa. A primeira instrução é uma atribuição de f(3.0,4.0) para a variável i.
3. Os valores dos argumentos da função f são calculados. Neste caso, não há nada o que calcular pois os valores já estão na forma numérica.
4. O programa é paralisado e o código da função f começa a ser executado.
5. É definido (alocado) espaço de memória para o retorno da função. No exemplo acima este espaço é nomeado **f**. São também definidos (alocados) espaços de memória para as variáveis x, y e h. Observe que x e y são parâmetros da função.
6. É atribuído o valor 3.0 para x e 4.0 para y.
7. Começa-se a executar as instruções que estão no corpo da função. A primeira instrução é uma atribuição de sqrt(x*x + y*y) para a variável h.
8. É calculado o valor que será passado como argumento para a função sqrt. O valor calculado é x*x + y*y = 3.0 * 3.0 + 4.0 * 4.0 = 25.0 .
9. O código da função **f** é paralisado e a função **sqrt** passa a ser executada com o argumento 25.0 . Não detalharemos a execução da função sqrt porque segue a mesma idéia da execução da função f que já estamos detalhando.

10. Assim que a função sqrt termina sua execução a função f continua sendo executada. O valor de retorno da função sqrt, 5, é atribuído à variável h.
11. O valor da variável h é atribuído ao espaço de memória alocado para a função f com a instrução f := h;
12. A execução da função f termina e o programa principal continua exatamente do ponto em que havia parado.
13. O valor de retorno da função **f** é atribuído à variável **i** (ou seja, i recebe 5).
14. O programa é paralisado e o código do procedimento writeln é executado...

5.2 ESCOPO DE VARIÁVEIS

Observe o programa abaixo.

```
program escopo;
var
   s : integer;
   a : integer;

function soma(i : integer) : integer;
var
   v : integer; {
begin
   v := s + i;
   s := v;
   soma := v;
end;

begin
   s := 1;
   a := soma(5);
   writeln (a, '   ',s);
end.
```

Este é um programa bem simples. Serve apenas para propósitos didáticos. Nele, a função **soma** usa 3 variávies:

- **i**: definida como parâmetro da função.
- **v**: definida na área de definição de variáveis da função.
- **s**: definida na área de definição de variáveis do programa principal.

Observa que embora a variável **s** tenha sido definida para o programa principal ela também é enxergada de dentro da função **soma**. Desta forma, os vares das variáveis definidas para o programa podem ser usadas ou mesmo alteradas de dentro de uma função interna à este programa.

Por outro lado, vemos que há 2 variáveis locais à função **soma**: **i** e **v**. Estas duas variáveis só são enxergadas pelo código da própria função **soma.** Não é possível ao programa principal acessar o valor destas variáveis.

Dizemos que as variáveis definidas para o programa então são **globais** (possuem escopo ou âmbito global) em relação à função e que as variáveis definidas dentro da função (i e v neste caso) são **locais** (possuem escopo ou âmbito local) à função.

Uma pergunta comum com relação à variáveis globais e locais é a seguinte: E se for definida uma variável local com o mesmo nome de uma variável global? Neste caso teremos 2 variáveis completamente diferentes. Localmente não será possível acessar a variável global e vice–versa.

Observação: De forma geral deve-se evitar o acesso a variáveis globais no âmbito local. Este tipo de prática é considerado má prática de programação porque torna o código muito interdependente.

5.3 PROCEDIMENTOS

Procedimentos são basicamente funções que não retornam valor.

A sintaxe é a seguinte:

```
PROCEDURE <Nome> [(parâmetros)]
      <definições>
BEGIN
      <comandos>;
END;
```

Observe que há duas diferenças na sintaxe com relação à função:

- O procedimento é definido com a palavra reservada **procedure**.
- Após a lista de parâmetros não há no procedimento um tipo de retorno como acontece com as funções.

Já temos usado alguns procedimentos em nossos programas tais como o read, readln, write e writeln.

Em algumas linguagens os procedimentos são chamados de sub-rotinas.

EXEMPLO

O programa abaixo altera o valor de **s**.

```
program escopo;
var
   s : integer;

procedure soma(i : integer);
   s := s + i;
end;

begin
   s := 1;
   soma(5);
   writeln (s);
end.
```

Este programa vai mostrar o valor **6** na tela.

Observe que um procedimento não retorna valor, portanto, não podemos fazer atribuição de um procedimento àuma variável nem passa-lo como argumento.

Por exemplo, não é possível fazer as operações abaixo:

```
program exemplo;
var
   s, a : integer;

procedure soma(i : integer);
   s := s + i;
end;

begin
   s := 1;
   a := soma(5);        {errado!!!}
   writeln (soma(5));   {errado!!!}

end.
```

EXEMPLO

O programa abaixo lê uma string e a mostra na tela.

```
PROGRAM Teste;
VAR
      Nome :  STRING[80];     {variável global}

PROCEDURE LeNome;
BEGIN
      READLN(Nome);
END;

BEGIN
      LeNome;
      WRITE(Nome);
END
```

No Exemplo acima, a variável “Nome” , por ser definida como global, pode ser manipulada dentro de qualquer ponto do programa, sendo que qualquer mudança no seu conteúdo, será visível nas demais partes do programa.

Observe também que o procedimento **LeNome** não tem parâmetros. Deve-se evitar procedimentos (ou funções) que usem variáveis globais pois torna o código muito restrito e interdependente.

EXEMPLO

O programa abaixo lê um número N e mostra os números de 1 até N na tela.

```
PROGRAM Teste;
PROCEDURE EscreveNoVídeo;
VAR
      Número, N  : INTEGER;
BEGIN
      READ(N);
      FOR número : = 1 TO N DO
            WRITE(Número);
```

```
END;

BEGIN
      EscreveNoVídeo;
END.
```

5.4 PASSAGEM DE PARÂMETRO

Há dois tipos de passagem de parâmetros em Pascal:

- Passagem de parâmetros por valor.
- Passagem de parâmetros por referência.

A **passagem de parâmetros por valor** é a forma que temos usado em todos os exemplos até agora. Dizemos que parâmetros passados por valor são parâmetros de entrada. O valor do argumento é passado para dentro da função ou do procedimento.

Por exemplo, o código apresentado abaixo usa o procedimento **soma** para incrementar o valor de **s** em **i** unidades. Na função **soma** o parâmetro **i** é passado "por valor". Dizemos que **i** é um parâmetro de entrada.

```
program escopo;
var
   s : integer;
   a : integer;

procedure soma(i : integer);
begin
   s := s + i;
end;

begin
   s := 1;
   a := 1;
   soma(5);
   writeln (a, '   ',s);
end.
```

Quando o programa executa o procedimento **soma(5)** o argumento 5 é passado para o parâmetro **i** que irá armazenar este valor. O parâmetro i é uma variável que ocupa um espaço em memória e, neste caso, coloca o valor 5 neste espaço que ela possui. Observe que o procedimento **soma** é muito pouco versátil. Ela só pode incrementar o valor de **s**. Caso quiséssemos alterar o valor de **a** teríamos que criar um outro procedimento **soma** exclusivo para **a**. Seria interessante um protótipo de procedimento em que tivéssemos dois parâmetros: um indicando a variável que desejamos alterar e outro o valor que queremos somar.

Mas não seria tão fácil. O código abaixo, por exemplo, não funcionaria adequadamente:

```
program escopo;
var
   s : integer;
   a : integer;

procedure soma(n: integer, i : integer);
begin
   n := n + i;
```

```
end;

begin
   s := 1;
   a := 1;
   soma(s,5);
   soma(a,3);
   writeln (a, '   ',s);
end.
```

Neste caso, o procedimento **soma** é chamado duas vezes. Na primeira o parâmetro **n** recebe o argumento **s** que vale 1. Então **n** passa a valer 1 e é somado com 5 dando 6 como resultado. Mas **n** não altera o valor de **s**. O parâmetro **n** existe apenas durante o breve tempo em que o procedimento **soma** é executado. Assim que o procedimento termina sua execução a área de memória ocupada por **n** é liberada (apagada). Desta forma, o resultado impresso será 1 1.

Aí entra a **passagem de parâmetro por referência**! Neste tipo de passagem de parâmetro não é criado um novo espaço de memória para o parâmetro. O que acontece é que o parâmetro vai usar o mesmo espaço de memória usado pelo argumento.

Para indicar que um parâmetro é passado por valor e não por argumento, colocamos a expressão **var** na frente dele no cabeçalho da função ou do procedimento.

Corrigindo o exemplo dado acima:

```
program escopo;
var
   s : integer;
   a : integer;

procedure soma(var n: integer, i: integer);
begin
   n := n + i;
end;

begin
   s := 1;
   a := 1;
   soma(s,5);
   soma(a,3);
   writeln (a, '   ',s);
end.
```

Neste caso o procedimento **soma** tem o parâmetro **n** passado por referência e o parâmetro **i** passado por valor.

Na chamada do procedimento **soma(s, 5)** o parâmetro **n** irá compartilhar a mesma área de memória do argumento **s**. Assim, alterar **n** equivale a alterar **s.** Na chamada seguinte, **soma(a, 3)** o parâmetro **n** irá compartilhar a mesma área de memória do argumento **a**. Assim, alterar **n** equivale a alterar **a.** O resultado impresso será 4 6.

EXEMPLO

O programa abaixo usa o procedimento pTroca que troca os valores de dois argumentos.

```
program trocaValores;
var val1,val2:integer;

procedure pTroca(var a,b:integer);
   var aux:integer;
begin
```

```
    aux:=a;
    a:=b;
    b:=aux
end;

begin
  read(val1,val2);
  writeln(val1,val2);
  pTroca(val1,val2);
  writeln(val1,val2);
end.
```

5.5 EXEMPLOS DE FUNÇÕES E PROCEDIMENTOS

EXEMPLO 1

O exemplo abaixo mostra o valor do fatorial dos números 1 até 10. Note que temos duas variáveis com nome i definidas neste programa. Uma é definida para o programa principal e outra para a função. O compilador trata ambas separadamente. Dizemos que o escopo das duas variáveis é diferente. Uma tem como escopo o programa principal e outra tem como escopo apenas a função ou seja ela é enxergada apenas dentro da função.

```
{ Mostra o valor do fatorial dos números de 1 até 10 }
program teste;
var
   i: longint;

{ Calcula o fatorial de um numero inteiro
  n            número do qual será calculado o fatorial
  Retorno:     retorna o fatorial do parâmetro n
}
function fatorial (n : integer) : longint;
var
   i : byte;      { i só vai de 1 até 10, não precisa ser integer}
   aux: longint;  { aux calcula o fatorial. Deve ser grande}
begin
   aux := 1;
   for i := 1 to n do
      aux := aux * i;
   fatorial := aux;
end;

begin {corpo do programa principal}
   for i := 1 to 10 do
      writeln (i,'! = ', fatorial (i) );
end.
```

A saída deste programa será:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

```
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

EXEMPLO 2

O programa abaixo faz uso de uma função que serve para passar os caracteres de uma string para letra maiúscula.

```
program exemplo;
var
   s: string;

{ Converte para maiúscula os caracteres de uma string.
  frase    string que será convertida.
  retorno: retorna a string que está em frase em maiúsculas
}
function maiuscula (frase : string) : string;
var
   aux : string;
   i : byte;
begin
   aux := frase;
   {Percorre todos os caracteres da frase}
   for i := 1 to length(frase) do
       aux[i] := upcase(frase[i]); {upcase retorna o maiúsculo}

   maiuscula := aux;
end;
begin{corpo do programa principal}
   Write   ('Digite uma frase:     '); readln(s);
   s := maiuscula(s);
   writeln ('Em letras maiusculas: ',s);
end.
```

O resultado deste programa será:

```
Digite uma frase:     Bon dia
Em letras maiusculas: BOM DIA
```

A função maiuscula usa a função upcase que recebe como parâmetro um caracter e retorna este caracter em maiúscula.

Vale lembrar que em Pascal uma string é formada por uma sequência de caracteres. Nós chamamos esta sequência de vetor. Assim, em outras palavras, uma string em Pascal é um vetor de caracteres. Cada caracter da string pode ser obtido usando-se um índice. Assim, se uma string frase recebe o valor 'Bom dia', então frase[1] é o caracter 'B' , frase[2] é o caracter 'o' e assim por diante até frase[7] que é o caracter 'a'.

Poderíamos ter optado por fazer ao invés de uma função **maiuscula**, um procedimento. Neste caso teríamos que passar o parâmetro **frase** por referência. O único problema é que perderemos o valor inicial da variável passada como argumento.

```
program exemplo;
```

```
  var
     s: string;

  { Converte para maiúscula os caracteres de uma string.
    frase    string que será convertida.
  }
  procedure maiuscula (var frase : string);
  var
      i : byte;
  begin
     {Percorre todos os caracteres da frase}
     for i := 1 to length(frase) do
         frase[i] := upcase(frase[i]); {upcase retorna o maiúsculo}

  end;
  begin{corpo do programa principal}
     Write   ('Digite uma frase:     '); readln(s);
     maiuscula(s);
     writeln ('Em letras maiusculas: ',s);
  end.
```

Observação: Aproveitamos também para fazer o código sem a variável **aux**.

EXEMPLO 3

O programa abaixo lê um número do teclado e informa se o número lido é um ano bissexto ou não. Para isto ele faz uso de uma função que retorna TRUE se ano é bissexto ou FALSE caso contrário.

```
program teste;
var
   i: integer;

function bissexto (a:integer) : boolean;
Begin
   if ((a mod 4 = 0) and (a mod 100<>0)) or (a mod 400 = 0)then
      bissexto:= TRUE
   else
      bissexto:= FALSE;
End;

begin
   write (‘Digite um ano:’);
   readln (i);
   if bissexto(i) then
      writeln (‘O ano é bissexto’)
   else
      writeln (‘O ano não é bissexto’);

end.
```

Observe que o if foi construído em uma única linha.

EXEMPLO 4

O programa abaixo lê uma frase do teclado e depois mostra a metade desta frase.

```
    program teste;
```

```
var
   s: string;


{
  Retorna os n primeiros caracteres de uma string dada
  s        string original.
  n        número de caracteres
  retorno: retorna os n primeiros caracteres da string s
}
function direita (s: string; n:integer):string;
var
   aux : string;
   i : integer;
begin
   aux := '';
   for i := n to length(s) do
      aux := aux + s[i];
   direita := aux;
end;

begin{corpo do programa principal}
   Write   ('Digite uma frase:'); readln(s);
   s := direita(s, length(s) div 2);
   writeln ('A metada da frase: ',s);
end.
```

O resultado deste programa é:

```
Digite uma frase: Bom dia
A metada da frase: Bom
```

A função esquerda retorna os n primeiros elementos da string.

Este programa traz um aspecto interessante das strings: pode-se concatenar variáveis do tipo string ou do tipo char simplesmente somando-as. Assim, se tivermos duas strings `s1:='Bom '` e `s2:='dia'`, podemos concatena-las simplesmente usando o operador `+` . Então poderíamos ter `s3:=s1+s2`. e o valor de `s3` seria `'Bom dia'`.

EXEMPLO 5

O programa abaixo mostra a frase 'aid moB' (que é o inverso de 'Bom dia') no vídeo.

```
program teste;
var
   i: integer;
   s: string;
{
  Inverte os caracteres de uma string.
  s        string original.
  retorno: retorna a string s invertida
}
function inverte (s: string) : string;
var
   i, tamanho : integer;
   aux : string;
```

```
begin
   tamanho := length(s);
   aux := '';
   for i := tamanho downto 1 do
       aux := aux + s[i];
   inverte := aux;
end;

begin {corpo do programa principal}
   s := 'Bom dia';
   writeln (inverte(s));
end.
```

Este programa mostra o uso do comando **for** com o **downto**. O downto permite que o valor de i varie de um valor grande para um valor pequeno. Exemplo: for i := 10 downto 1 do .

Poderíamos ter optado por fazer ao invés de uma função **inverte**, um procedimento. Neste caso teríamos que passar o parâmetro **s** por referência. O único problema é que perderemos o valor inicial da variável passada como argumento. O código ficaria assim:

```
program teste;
var
   i: integer;
   s: string;
{
  Inverte os caracteres de uma string.
  s        string que será invertida.
}
procedure inverte (var s: string);
var
   i, tamanho : integer;
   aux : string;
begin
   tamanho := length(s);
   aux := '';
   for i := tamanho downto 1 do
       aux := aux + s[i];
   s := aux;
end;

begin {corpo do programa principal}
   s := 'Bom dia';
   inverte(s);
   writeln (s);
end.
```

Observe que neste caso não podemos mais usar o comando `writeln (inverte(s));` porque inverte(s) é agora um procedimento e portanto não retorna valor algum.

EXEMPLO 6

O programa abaixo mostra que podemos ter mais de uma função definida em um programa. Mostra também que podemos chamar uma função de dentro de outra função.

```
program teste;
var
```

```
   s: string;

{
  Retorna os n últimos caracteres de uma string dada
  s        string original.
  n        número de caracteres
  retorno: retorna os n últimos caracteres da string s
}
function esquerda (frase: string; n:integer):string;
var
   aux : string;
   i : integer;
begin
   aux := '';
   for i := 1 to n do
       aux := aux + frase[i];
   esquerda := aux;
end;


{
  Retorna os n primeiros caracteres de uma string dada
  s        string original.
  n        número de caracteres
  retorno: retorna os n primeiros caracteres da string s
}
function direita (s: string; n:integer):string;
var
   aux : string;
   i : integer;
begin
   aux := '';
   for i := n to length(s) do
      aux := aux + s[i];
   direita := aux;
end;

{
  Retira os brancos do início e do fim de uma string dada
  frase    string original.
  retorno: retorna a string frase sem brancos no início ou no fim
}
function TiraBrancos (frase : string):string;
var
   i : integer;
begin

   { Retira brancos do in¡cio da string }
   i := 1;
   while (i < length(frase)) and (frase[i] = ' ')  do
         i:= i+1;

   frase := direita(frase,i);

   { Retira brancos do final da string }
   i := length (frase);
```

```
   while (i >= 1 ) and (frase[i] = ' ')  do
         i:= i-1;

   frase := esquerda(frase,i);

   TiraBrancos := frase;

end;

begin {corpo do programa principal}
   Write('Digite uma frase:'); readln(s);
   s := TiraBrancos(s);
   write ('Sem brancos     :',s);
end.
```

O resultado deste programa é a frase lida, sem brancos no início ou no fim.

Poderíamos ter optado por fazer ao invés de uma função **TiraBrancos**, um procedimento. Neste caso teríamos que passar o parâmetro **frase** por referência. O único problema é que perderemos o valor inicial da variável passada como argumento. O código da função **TiraBrancos** e do porgrama principal ficariam como mostrado abaixo. As demais funções não serial alteradas:

```
. . .
function TiraBrancos (var frase : string);
var
   i : integer;
begin

   { Retira brancos do in¡cio da string }
   i := 1;
   while (i < length(frase)) and (frase[i] = ' ')  do
         i:= i+1;

   frase := direita(frase,i);

   { Retira brancos do final da string }
   i := length (frase);
   while (i >= 1 ) and (frase[i] = ' ')  do
         i:= i-1;

   frase := esquerda(frase,i);

end;

begin {corpo do programa principal}
   Write('Digite uma frase:'); readln(s);
   TiraBrancos(s);
   write ('Sem brancos     :',s);
end.
```

Poderíamos também alterar as funções direita e esquerda para procedimentos.

EXEMPLO 7

Escrever um predicado (isto é, uma função que retorna um valor booleano e um programa que exercite tal predicado) que verifique se um número é primo ou não.

```
program sePrimo;
  var num:integer;

  {
    Verifica se um dado número é primo.
    n        número que será verificado.
    retorno: TRUE se n for primo. FALSE caso contrário.
  }
  function primo(n:integer):boolean;
    var
      i:integer;
      condPrimo:boolean;
  begin
    condPrimo:=false;
    if n=2
    then condPrimo:=true
    else
      if (n>0) and (n mod 2=1)
      then
      begin
        i:=1;
        repeat
          i:=i+2
        until (n mod i=0) or (i>n div 2);
        condPrimo:=n mod i<>0
      end;
    primo:=condPrimo
  end;

begin
  read(num);
  if primo(num)
  then writeln('sim')
  else writeln('nao')
end.
```

Poderíamos ter optado por fazer ao invés de uma função **primo**, um procedimento. Neste caso teríamos que passar um parâmetro de tipo booleano por referência.

```
program sePrimo;
  var
      num:integer;
      ehprimo: boolean;


  {
    Verifica se um dado número é primo.
    n          número que será verificado.
    condPrimo  TRUE se n for primo. FALSE caso contrário.
  }
```

```
  procedure primo(n:integer, condPrimo: boolean);
    var
      i:integer;
  begin
    condPrimo:=false;
    if n=2 then
       condPrimo:=true
    else
      if (n>0) and (n mod 2=1) then
      begin
        i:=1;
        repeat
          i:=i+2
        until (n mod i=0) or (i>n div 2);
        condPrimo:=n mod i<>0
      end;
   end;

begin
  read(num);
  primo(num, ehprimo);
  if ehprimo then
      writeln('sim')
  else
      writeln('nao');
end.
```

Observe o uso do procedimento primo no programa principal:

```
primo(num, ehprimo);
```

É necessário passar uma variável booleana como argumento. Esta variável receberá TRUE se o número **num** for primo ou FALSE caso contrário. Neste caso, parece mais natural construir uma função do que um procedimento.

EXEMPLO 8

Escrever um programa que calcule o máximo divisor comum de dois números inteiros positivo através de sua decomposição em fatores primos.

```
program mdc4;
  var a,b,res,prim:integer;


  {
    Retorna o mdc de dois números.
    a,b      números para encontrar o mdc.
    retorno: o valor do mdc entre a e b.
  }
  function mdc(a,b:integer):integer;


   {
     Retorna o próximo número primo.
     p        números primo corrente.
     retorno: o próximo número primo a partir de p.
   }
```

```
  function proximoPrimo(p:integer):integer;


    {
      Verifica se um número é primo.
      n        números que será verificado.
      retorno: TRUE se n for primo. FALSE caso contrário.
    }
    function primo(n:integer):boolean;
    var
        i:integer;
        condPrimo:boolean;
    begin
      condPrimo:=false;
      if n=2 then
         condPrimo:=true
      else
        if (n>0) and (n mod 2=1) then begin
          i:=1;
          repeat
            i:=i+2
          until (n mod i=0) or (i>n div 2);
          condPrimo:=n mod i<>0
        end;
      primo:=condPrimo
    end;


  begin { proximoPrimo }
    if p=2 then
       proximoPrimo:=3
    else begin
      repeat
        p:=p+2;
      until primo(p);
      proximoPrimo:=p
    end
  end;

begin { mdc }
  prim:=2;
  res:=1;
  while (a<>1) and (b<>1) do
    if a mod prim=0 then begin
      a:=a div prim;
      if b mod prim=0 then begin
        b:=b div prim;
        res:=res*prim
      end
    end
    else
      if b mod prim=0 then
         b:=b div prim
      else
         prim:=proximoPrimo(prim);
  mdc:=res
```

```
  end;


begin { programa principal }
  read(a,b);
  writeln(mdc(a,b))
end.
```

Este programa exemplifica algo que é muito comum: definir uma função ou um procedimento dentro de outra função ou procedimento. Neste exemplo, a função **primo** está definida dentro de **proximoPrimo** que por sua vez está definida dentro da função **mdc**. Neste caso, o programa principal só enxerga a função **mdc**. A função **mdc** por sua vez só enxerga **proximoPrimo**.

EXEMPLO 9

O n-ésimo número da seqüência de Fibonacci F_n é dado pela seguinte fórmula de recorrência:

$F_1 = 1$
$F_2 = 1$
$F_i = F_{i-1} + F_{i-2}$, para $i>=3$

Escrever uma função (e um programa que exercite tal função) que calcule F_n para $n>=1$.

```
program Fibonacci;
  var n:integer;

  { Retorna n-ésimo valor da série de Fibonacci.
    n        número do elemento da série.
    retorno: n-ésimo valor da série de Fibonacci.}
  function fib(n:integer):integer;
    var i,fib1,fib2,soma:integer;
  begin
    fib1:=1; fib2:=1;
    for i:=3 to n do begin
      soma:=fib1+fib2;
      fib1:=fib2;
      fib2:=soma
    end;
    fib:=fib2
  end;
begin
  read(n);
  writeln(fib(n))
end.
```

Capítulo 6 MATRIZES UNIDIMENSIONAIS OU VETORES

Como na matemática, em programação, uma matriz é composta basicamente de um conjunto de elementos organizados em linhas e colunas. Um vetor é uma matiz de apenas uma linha. Dito de outra forma, um vetor é uma matriz unidimensional (que só tem uma dimensão ou seja; uma linha). Neste capítulo trataremos apenas de vetores. Em outra oportunidade veremos as matrizes com mais dimensões.

6.1 VETORES

Vimos, ser possível dar um Nome para uma posição de memória, sendo que a esta será associado um valor qualquer. Muitas vezes porém, esta forma de definição (de alocação de memória) não é suficiente para resolver certos problemas computacionais. Imagine por exemplo, como construir um programa, para ler o nome de 1000 pessoas e que mostrasse estes mesmos nomes, mas ordenados alfabeticamente? Não seria uma tarefa simples, teríamos que definir 1.000 variáveis do tipo STRING.

Considere o tamanho do programa, e o trabalho braçal necessário para construi-lo. Isto só com 1.000 nomes. Imagine agora 1.000.000 de nomes. A construção deste algoritmo seria inviável na pratica. Para resolver problemas como este, e muitos outros, usamos um novo conceito para alocação de memória, o vetor.

Um vetor é uma estrutura de dados composta e homogênea. Dizemos que é composta porquê possui mais de um elemento e homogênea, porquê todos os elementos devem ser do mesmo tipo.

Já temos trabalhado com um tipo especial de vetor; o tipo string. Uma string é um vetor pois possui um conjunto de elementos que são do tipo char.

Os elementos de um vetor é acessado através de um índice.

A sintaxe é:

```
PROGRAM Define;
VAR
      <Nome>:  ARRAY [INICIO..FIM] OF <tipo>;
BEGIN
      <Comandos>;
END.
```

Observação:

- "ARRAY" é uma palavra reservada do Pascal e quer dizer matriz.
- Os valores "INICIO" e "FIM" correspondem aos índices inicial e final do vetor.
- Uma variável indexada pode ser apenas de um tipo de dado.

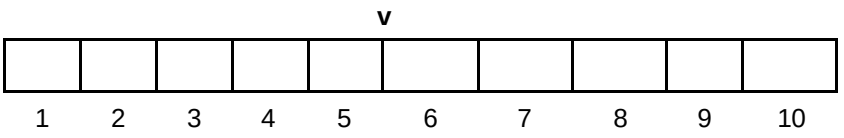
Os elementos do vetor são acessado através de um índice.

EXEMPLO

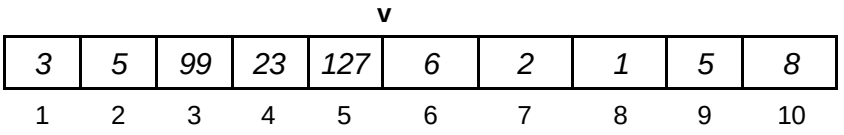
Faça um programa que crie um vetor de números de 10 posições e leia os elementos do teclado. Depois mostre todos os elementos lidos.

```
program teste;
var
   v : array [1..10] of integer; { cria o vetor v }
   i : integer;                  { servirá como índice do vetor }
begin
   for i := 1 to 10 do    {lê 10 números do teclado}
      readln ( v[i] );

   for i := 1 to 10 do
      writeln ( v[i] );     {mostra os 10 números lidos na tela}
end.
```

No exemplo acima, após a definição da variável, a memória estará como mostrado no esquema abaixo:



Suponha que o usuário entre com os seguintes números: 3, 5, 99, 23, 127, 6, 2, 1, 5, 8. Neste caso teríamos a memória de **v** preenchida da seguinte forma:



O elemento de posição 1 é o 3, o elemento de posição 2 é o 5, o elemento de posição 3 é o 99, o elemento de posição 4 é o 23 e assim por diante, até o elemento de posição 10 que é o 8.

Não devemos confundir o valor do elemento com sua posição. No exemplo anterior, o elemento de posição 6 é o 6 por coincidência. Poderia ser qualquer outro número.

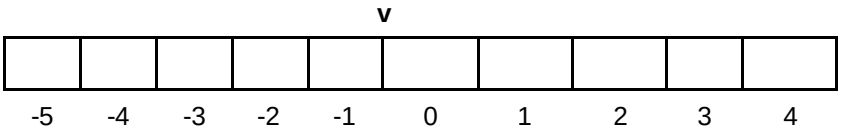
É permitido definir vetores que comecem com valores diferente de 1. Poderíamos ter construído o exemplo anterior criando uma variável v que começasse em 0 e fosse até 9 ou que começasse em 5 e fosse até 14 ou mesmo que começasse em –5 e fosse até 4.

Neste último caso teríamos o seguinte programa:

```
program teste;
var
   v : array [-5..4] of real;
   i : integer;
```

```
begin
   for i := -5 to 4 do
      readln (v[i]);

   for i := -5 to 4 do
      writeln (v[i]);
end.
```

No exemplo acima, após a definição da variável, a memória estará como mostrado no esquema abaixo:



EXEMPLO

O programa abaixo usa um vetor para ler 5 valores e depois mostra a soma dos números.

```
program vet_soma;
var
   v: array [1..5] of integer;
   soma,cont:integer;
Begin
   soma:=0;
   writeln('digite os 5 valores');

   for cont:=1 to 5 do
      readln(v[cont]);

   for cont:=1 to 5 do
      soma:=soma + v[cont];

   writeln('Soma=',soma);
end.
```

EXEMPLO

Leia 4 números, coloque-os em um vetor e mostre-os na ordem inversa de sua leitura.

```
program ex2;
var
   vet:array[1..4] of integer;
   pos:integer;
Begin
   writeln('Digite os valores');
   for pos:=1 to 4 do
      readln(vet[pos]);

   writeln('Ordem inversa:');
   for pos:=4 downto 1 do
      writeln(vet[pos]);
end.
```

6.2 DEFINIÇÃO DE NOVOS TIPOS DE DADOS

Em Pascal é possível ao programador definir seus próprios tipos de dados. Isto é feito da seguinte forma:

```
TYPE <nome_do_tipo>  = < definição_do_tipo >;
```

A definição do TYPE deve vir antes da definição das variáveis, ou seja, antes da partícula VAR.

EXEMPLO

O trecho de programa abaixo define alguns novos nomes para alguns tipos conhecidos.

```
program teste;
type
   TipoInteiro = integer;
   T_Nota = real;
   TDia = (seg, ter, qua, qui, sex, sab, dom);
var
   x : TipoInteiro;
   y : integer;
   prova1 : T_Nota;
   hoje : TDia;

begin
   ...
   x := 5;
   hoje = ter;
   ...
end.
```

Podemos também definir tipos usando arrays.

EXEMPLO

O programa abaixo define o tipo TNota como um array de reais de 20 posições.

```
program teste;
type
   TNotas = array [1..20] of real;
var
   nota : TNotas;
   i : integer;
   soma : real;

begin
   soma := 0;
   for i := 1 to 20 do
      read (nota[i]);

   for i:= 1 to 20 do
      soma := soma + nota[i];

   writeln ('Total:', soma);
```

```
end.
```

Podemos usar um vetor como argumento de uma função ou um procedimento. Entretanto uma função não pode retornar um vetor.

Observação: Uma função **não** pode retornar um vetor.

6.3 EXEMPLOS DE PROGRAMAS USANDO VETORES

EXEMPLO 1

Dada uma seqüência de n números reais, fazer um programa que determine os números que compõem a seqüência e a freqüência de tais números na seqüência dada.

Exemplo: `n = 10` Seqüência: `3.2 8.1 2.5 3.2 7.4 8.1 3.2 6.3 7.4 3.2`

Saída:

```
Num  Freq
3.2     4
8.1     2
2.5     1
7.4     2
6.3     1
```

```
program numeroDeOcorrencias;
const
   lim=20;
var
   n,i,j,k: integer;
   num    : real;
   achou  : boolean;
   numero : array [1..lim] of real;
   freq   : array [1..lim] of integer;

begin
  read(n);
  if n<=lim then begin
    k:=0;
    { Inicia o valor do vetor freq com zeros }
    for j:=1 to lim do
       freq[j]:=0;

    { Lê cada número e verifica sua frequência}
    for i:=1 to n do begin
      read(num);
      achou:=false;
      j:=1;

      { Procura o número lido no vetor de frequência }
```

```
      while not achou and (j<=k) do begin
         if numero[j]=num then begin
            achou:=true;
            freq[j]:=freq[j]+1
         end
         else
            j:=j+1;
      end;


    { Se não encontrou o número no vetor de frequência,
      significa que é a primeira vez que ele apreceu.
      Então, coloque-o no vetor freq com valor 1}
    if not achou then begin
         k:=k+1;
         numero[k]:=num;
         freq[k]:=1
      end
    end;

    { Mostra os valores na tela }
    writeln('   Num   Freq');
    for j:=1 to k do
       writeln(numero[j],freq[j]);
  end
end.
```

EXEMPLO 2

O programa abaixo calcula a média de 10 notas.

```
program teste;
type
   TNotas = array [1..10] of real;
var
   nota : TNotas;
   media : real;

procedure LeNotas (var n : TNotas);
var
   i : integer;
begin
   for i := 1 to 10 do
      read (n[i]);
end;

function CalculaMedia (n : TNotas):real;
var
   i : integer;
   soma : real;
begin
   soma := 0;
   for i := 1 to 10 do
      soma := soma + n[i];
```

```
   CalculaMedia := soma/10;
end;

begin

   LeNotas(nota);
   media := CalculaMedia(nota);

   writeln ('Valor da media: ',media);
end.
```

EXEMPLO 3

Dado um polinômio $p(x) = a_0 + a_1x + a_2x^2 + ... + {}_nx^n$, faça um programa que calcule $p(x)$. São dados o grau **n** do polinômio, os coeficientes do polinômio $\mathbf{a_0, a_1, a_2, ..., a_n}$, e o valor de **x**.

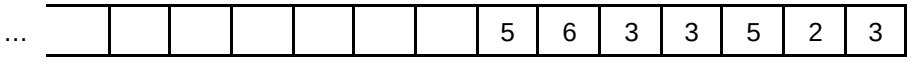
```
program polinomio;

  const lim = 10;     { cria uma constante chamada lim }

  type  vetCoef = array [0..lim] of real;

  var   i,n,j,k:integer;
        x:real;
        coef:vetCoef;

  { Calcula x elevado a y }
  function elevado (x,y : real): real;
  begin
     elevado := exp(ln(x)*y);
  end;

  { Calcula o valor do polinômio
    x : valor de x em p(x)
    n : grau do polinômio
    a : vetor de coeficientes de p(x)
    retorno : retorna o valor de p(x) }
  function p(x:real; n:integer; var a:vetCoef):real;
  var
    pParcial :real;
    i :integer;
  begin
    pParcial:=0.0;
    for i:=0 to n do
      pParcial := pParcial + (a[i] * elevado(x,i));
    p:=pParcial
  end;

begin {programa principal }

  write('Grau do polinomio: ');
  readln (n);

  writeln('Coeficientes do polinomio:');
  for i:=n downto 0 do begin
     write ('a',i,' = ');
     readln(coef[i]);
  end;

  write('Valor de X: ');
  read(x);

  writeln('O valor de p(X) = ',p(x,n,coef))
end.
```

Neste exemplo definimos uma constante chamada **`lim`** que define o limite superior do índice do vetor. Embora até agora não tenhamos usado muito as constantes (que em Pascal são definidas com a palavra reservada `const`) elas são muito úteis pois permites que tenhamos definido em um único lugar um valor que aparece em várias partes do programa. Por exemplo, se por algum motivo desejássemos mudar o valor do limite superior do vetor para 50, só precisaríamos alterar o valor de **lim** para 50 e recompilar o programa.

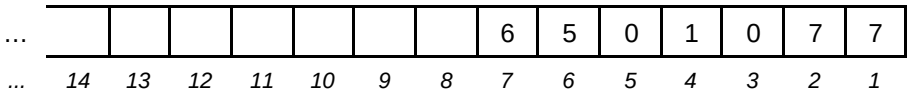
EXEMPLO 4

O programa abaixo permite somar dois números inteiros longos com até 200 dígitos. Para isto é usada manipulação numérica através de vetores.

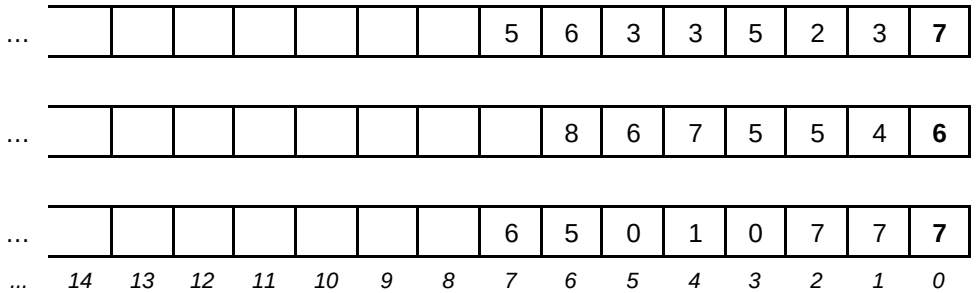
A idéia é colocar cada algarismo do número que desejamos somar em um vetor e depois somar os dois vetores. Suponha que desejamos somar 5633523 com 867554:



Soma:



Observamos que o primeiro número possui 7 algarismos e o segundo apenas 6 algarismos. No programa, guardamos o número de algarismos de cada número na posição 0 do vetor, da seguinte forma:




```
program SomaInteirosLongos;

const
   LIM = 100;

type
   t_vet = array [0..LIM] of byte;

var
   s : string;
   n1, n2, r: t_vet;
```

```
{ Coloca zero em todas as posições do vetor para inicia-lo }
procedure ZeraVet (var v: t_vet);
var
   i : integer;
begin
   for i:= 0 to LIM do
      v[i] := 0;
end;

{ Copia os algarismos de uma string para o vetor }
procedure StrToVet (s: string; var v:t_vet);
var
   i : byte;
begin
   { Guarda o numero de digitos }
   v[0] := length(s);

   { Coloca os digitos em de s em v }
   for i := length(s) downto 1 do
      v[length(s)-i+1] := ord(s[i])-48; { 48 é o ASCII de 0 (zero) }

end;

{Copia os algarismos do vetor para uma string}
procedure VetToStr (v : t_vet; var s: string);
var
   i : byte;
begin
   s := '';

   { Coloca os digitos em de v em s }
   for i := v[0] downto 1 do
      s := s + chr(v[i]+48);

end;

{Soma dois vetores de algarismos }
procedure SomaVet (v1, v2: t_vet; var r: t_vet);
var
   soma, vai1, i: byte;
   vAux : t_vet;
begin

   { Identifica quem é o maior numero }
   if v1[0] < v1[0] then begin
      vAux := v2;
      v2   := v1;
      v1   := vAux;
   end;

   { Faz a soma }
   vai1 := 0;
   for i := 1 to v1[0] do begin
      soma := v1[i] + v2[i] + vai1 ;
      if soma >= 10 then begin
         soma := soma - 10;
```

```
         vai1 := 1;
      end
      else
         vai1 := 0;

      r[i] := soma;
   end;

   { Se ainda tem vai um, coloca na última posição }
   if (vai1 = 1) then begin
      r[i+1] := 1;
      r[0] := v1[0] + 1;
   end
   else
      r[0] := v1[0]
end;

{Programa principal }
begin

   { Inicia com zeros os valores de n1 e n2 }
   ZeraVet(n1);
   ZeraVet(n2);

   { Pula uma linha}
   writeln;

   { Lê do teclado uma string contendo o número
     e coloca este número em n1 e depois repete para n2 }
   readln (s);
   StrToVet (s, n1);

   readln (s);
   StrToVet (s, n2);

   { Soma n1 e n2 colocando o resultado em r }
   SomaVet (n1, n2, r);

   { Coloca os algarismos que estão em r em uma string e mostra }
   VetToStr(r,s);
   writeln(s);
end.
```