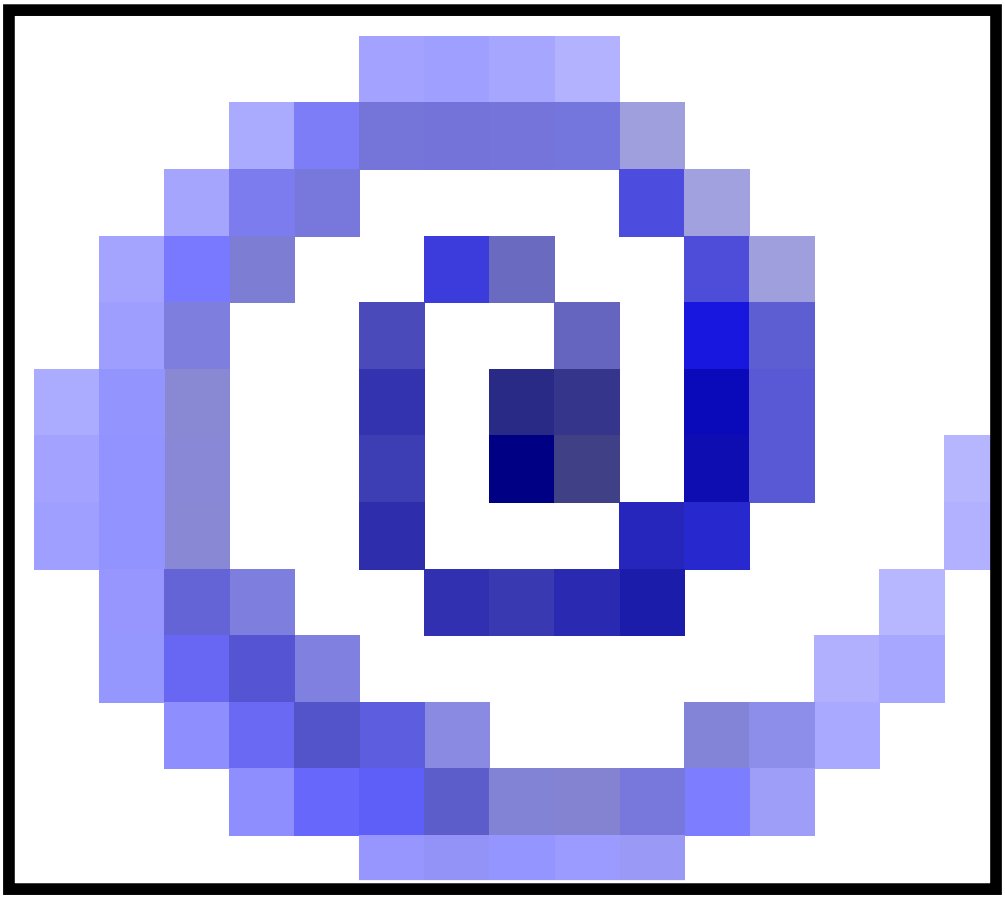
LINGUAGEM PASCAL

NOÇÕES BÁSICAS USANDO TURBO PASCAL



Silvio do Lago Pereira

SUMÁRIO

1. INTRODUÇÃO

A linguagem *Pascal*, cujo nome é uma homenagem ao matemático francês *Blaise Pascal*, foi desenvolvida na década de 60 pelo professor *Niklaus Wirth*. Inicialmente, sua finalidade era ser uma linguagem para uso didático, que permitisse ensinar com clareza os principais conceitos envolvidos na programação estruturada de computadores. Hoje, numa versão mais moderna denominada *Delphi*, essa linguagem é também utilizada por profissionais de diversas áreas tais como processamento de dados, computação e engenharia.

1.1. *O Sistema Turbo Pascal*

Para criar um programa na linguagem *Pascal*, usaremos o *Turbo Pascal*. Esse *software* engloba:

① um *editor de textos* que nos permite digitar e salvar em disco o programa codificado em *Pascal*;

② um *compilador*, que traduz o programa escrito em Pascal para a linguagem de máquina.

Após ter digitado o programa, usamos:

① **F2**: para salvar em disco o programa digitado;

② **Ctrl+F9**: para compilar e executar o programa;

③ **Alt+F5**: para ver o resultado da execução.

1.2. *Um Primeiro Exemplo*

Vamos criar um programa que "*dadas as duas notas obtidas por um aluno, informe a sua média*". Para realizar esta tarefa, o programa deverá executar os seguintes passos:

1º *Solicitar a primeira nota*;
2º *Ler o valor e armazená-lo na memória*;
3º *Solicitar a segunda nota*;
4º *Ler o valor e armazená-lo na memória*;
5º *Calcular a média com os valores fornecidos*;
6º *Exibir o resultado do cálculo*;

Em Pascal, esses passos são indicados assim:

```
program media;
var  p1, p2, m : real;
begin
   write('Primeira nota? ');
   readln(p1);
   write('Segunda nota? ');
   readln(p2);
   m := (p1+p2)/2;
   writeln('A média é ', m:0:1);
end.
```

1.3. *Algumas Explicações*

① A palavra *program* serve para definir o nome do programa, que deve ser digitado logo em seguida;

② Os locais de memória onde os dados são armazenados, denominados *variáveis*, são identificados por *nomes* que devem iniciar com letras;

③ A palavra *var* serve para declarar o tipo das variáveis que serão usadas no programa. Variáveis que armazenam números inteiros devem ser do tipo *integer* e aquelas que permitem números com parte fracionária devem ser do tipo *real*;

④ A palavra *begin* marca o início do programa;

⑤ Todas as mensagens a serem exibidas na tela do computador devem ser colocadas entre apóstrofos. Uma variável, entretanto, não; caso contrário, o computador exibirá o seu nome e não o seu valor.

⑥ O comando *write* exibe uma informação na tela e deixa o cursor na mesma linha em que a informação foi exibida. Já o comando *writeln*, faz com que o cursor salte para o início da linha seguinte.

☛ *Para que não sejam exibidas em notação científica, variáveis do tipo real devem ser formatadas da seguinte maneira:*
variável **:** *tamanho_campo* **:** *casas_decimais*

⑦ O comando *readln* aguarda até que o usuário digite um valor e pressione <*enter*> e, em seguida, armazena o valor na variável especificada.

⑧ O final do programa é indicado pela palavra *end*.

1.4. *Exercícios*

Para cada problema, codifique um programa *Pascal*:

1.1. Dada a medida dos lados de um quadrado, informe a sua área.

1.2. Dada a medida do raio de uma circunferência, informe o seu perímetro.

1.3. Dada uma distância (*km*) e o total de combustível (ℓ) gasto por um veículo para percorrê-la, informe o consumo médio (km/ℓ).

1.4. Sabe-se que com uma lata de tinta pinta-se $3m^2$. Dadas a largura e a altura de uma parede, em metros, informe quantas latas de tinta serão necessárias para pintá-la completamente.

1.5. Dadas as medidas dos catetos de um triângulo retângulo, informe sua hipotenusa. [*Dica*: em *Pascal*, $\sqrt{x}$ escreve-se *sqrt*(*x*)].

1.5. *A Unidade CRT*

A unidade *CRT* consiste de um conjunto de comandos adicionais que são oferecidos no *Turbo Pascal*. Como esses comandos não fazem parte da linguagem *Pascal* padrão, para utilizá-los, precisamos incluir a seguinte instrução no programa:

```
uses crt;
```

Essa instrução indica ao compilador que iremos usar os comandos adicionais contidos na unidade *CRT*. Sem ela, os comandos adicionais não podem ser reconhecidos pelo compilador.

☛ *Essa instrução deve ser a segunda linha do programa, conforme indicado a seguir:*

```
program exemplo;
uses crt;
...
```

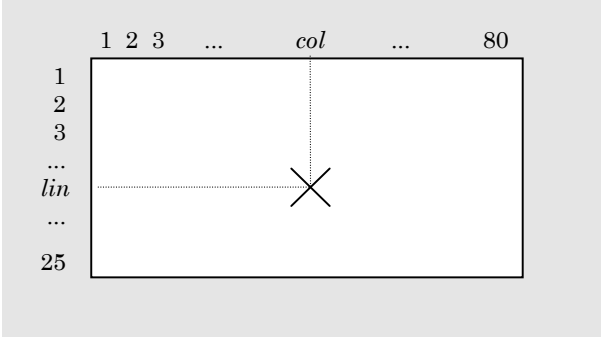
Temos a seguir a descrição dos comandos definidos em *CRT* que são mais utilizados em *Turbo Pascal*:

① *clrscr*: limpa a tela e posiciona o cursor no início da primeira linha;

② *textcolor*(*cor*): seleciona a *cor* na qual os textos serão exibidos no vídeo. As cores são representadas por números de 0 a 15 ou, então, por palavras em inglês (*red*, *blue*, ...). Para exibir texto piscante, adicione a palavra *blink* à cor selecionada; por exemplo, *red+blink*;

③ *textbackground*(*cor*): seleciona a *cor* do fundo sobre o qual os textos serão exibidos. As cores são representadas da mesma maneira que no comando anterior. Para que a tela toda apareça na cor de fundo selecionada, execute o comando *clrscr* logo após o *textbackground*;

④ *gotoxy*(*col, lin*): posiciona o cursor na posição de tela indicada. Caso o valor de *col* (1 a 80) ou *lin* (1 a 25) esteja fora do intervalo permitido, o cursor não é movimentado.



1.6. *Exercícios*

Para cada problema a seguir, codifique um programa *Pascal*. Use os comandos adicionais disponíveis em *CRT* e, para não ter que ficar digitando **Alt+F5** para ver os resultados, inclua *readln*[1] como a última instrução do programa (antes do *end*).

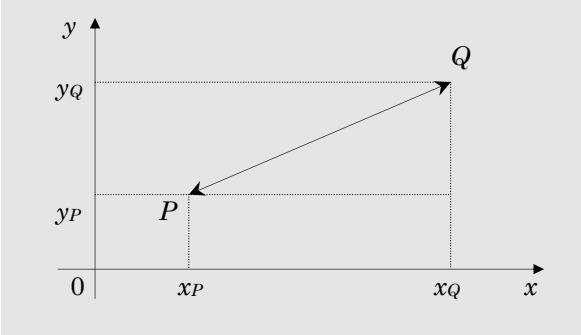
1.6. Dada uma distância (*km*) e o tempo (*h*) gasto por um veículo para percorrê-la, informe a velocidade média do veículo (*km/h*).

1.7. Dada uma medida em centímetros (*c*), informe o valor correspondente em polegadas (*p*). Utilize a fórmula: $p = c / 2{,}54$.

1.9. Dada uma temperatura em graus *Celsius* (*ºC*), informe a correspondente em graus *Fahrenheit* (*ºF*). Utilize a fórmula: $F = (9/5) \times C + 32$.

1.10. Sabe-se que $1m^2$ de carpete custa R$ 35,00. Dados o comprimento e a largura de uma sala, em metros, informe o valor que será gasto para forrar todo o seu piso.

1.11. Dadas as coordenadas de dois pontos *P* e *Q* do plano cartesiano, informe a distância entre eles. [*Dica*: use o teorema de *Pitágoras*]



2. A Estrutura De Seleção

A estrutura de *seleção*, ou *condicional*, é uma estrutura que nos permite selecionar e executar apenas um entre dois comandos possíveis. Para decidir qual comando deverá ser executado, emprega-se uma *expressão lógica*, denominada *condição*: se esta for verdadeira, seleciona-se a primeira alternativa; caso contrário, se for falsa, seleciona-se a segunda.

```
if condição
   then comando1
  else comando2;
```

[1] *Quando usado sem variáveis, o comando readln apenas aguarda que o usuário pressione a tecla <enter>.*

Observe que os comandos no *if-else* são mutuamente exclusivos, isto é, a seleção de um deles impede que o outro seja executado, e vice-versa.

☛ *À vezes, não há duas alternativas, apenas uma: ou o comando é executado ou, então, nada é feito. Nestas ocasiões, pode-se omitir a parte else do comando if-else.*

A expressão lógica, usada como *condição*, pode ser formada pelos seguintes tipos de operadores:

① *aritméticos*: +, –, *, /, *div* e *mod*;

② *relacionais*: =, <>, <, >, <= e >=;

③ *lógicos*: *not*, *and* e *or*.

2.1. *Um Exemplo de Aplicação*

Temos a seguir um exemplo de como empregar a estrutura de seleção ao codificar um programa em *Pascal*. Neste exemplo, são fornecidas as duas notas de um aluno e o programa informa se ele está aprovado ou reprovado:

```
program situacao;
uses crt;
var  p1, p2, m : real;
begin
   clrscr;
   write('Informe as duas notas: ');
   readln(p1,p2);
   m := (p1+p2)/2;
   writeln('Média: ', m:0:1);
   write('Situação: ');
   if m >= 7.0
      then writeln('aprovado')
      else writeln('reprovado');
   readln;
end.
```

2.2. *Utilizando Blocos de Comandos*

Sempre que for necessário executar mais que um comando quando a condição for verdadeira (ou falsa) é preciso agrupar os diversos comandos em um único *bloco*, isto é, precisamos colocá-los entre as palavras *begin* e *end*. Por exemplo, suponha que fosse necessário alterar o programa anterior de modo que a mensagem *aprovado* fosse exibida em azul e *reprovado*, em vermelho. Então, teríamos que codificar a estrutura de seleção do seguinte modo:

```
...
if m >= 7.0 then
   begin
      textcolor(blue);
      writeln('aprovado');
   end
 else
   begin
      textcolor(red);
      writeln('reprovado');
   end;
...
```

2.3. *Exercícios*

Para cada problema, codifique um programa *Pascal*:

2.1. Dado um número, informe se ele é par.

2.2. Dados dois números distintos, informe qual deles é o maior.

2.3. Dada a idade de uma pessoa, informe se ela pode ou não ter carta de motorista.

2.4. O índice de massa corporal (*imc*) de uma pessoa é dado pelo seu peso dividido pelo quadrado da sua altura. Se o *imc* é superior a 30, a pessoa é considerada obesa. Dados o peso (*kg*) e a altura (*m*) de uma pessoa, informe seu *imc* e indique se ela precisa de regime.

2.5. Numa empresa paga-se R$ 14,50 por hora e recolhe-se 15% dos salários acima de R$ 1.200,00 para o imposto de renda. Dado o número de horas trabalhadas por um funcionário, informe o valor do seu salário bruto, do desconto de I.R. e do seu salário líquido.

2.6. Numa faculdade, os alunos com média pelo menos 7,0 são aprovados, aqueles com média inferior a 3,0 são reprovados e os demais ficam de recuperação. Dadas as duas notas de um aluno, informe sua situação. Use as cores azul, vermelho e amarelo para as mensagens *aprovado*, *reprovado* e *recuperação*, respectivamente.

2.7. Dados os coeficientes ($a \neq 0$, b e c) de uma equação do 2º grau, informe suas raízes reais. Utilize a fórmula de *Báskara*:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4.a.c}}{2.a}$$

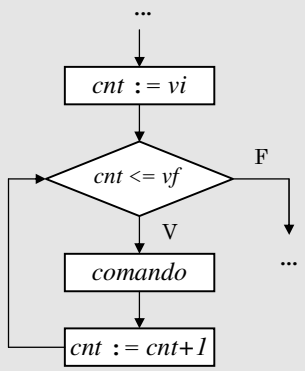
2.8. Dados três números, verifique se eles podem ser as medidas dos lados de um triângulo e, se puderem, classifique o triângulo em *equilátero*, *isósceles* ou *escaleno*.

3. REPETIÇÃO COM CONTADOR

A estrutura de *repetição com contador* é uma estrutura que nos permite executar uma ou mais instruções, um *determinado* número de vezes. Para controlar o número de repetições, empregamos uma variável denominada *contador*.

```
for cnt := vi to vf do
    comando;
```

Ao encontrar o *for*, o computador primeiro atribui o valor inicial *vi* ao contador *cnt*. A partir daí, o *comando* associado à estrutura é executado repetidamente, até que o contador atinja o valor final *vf*. A cada vez que o *comando* é executado, o contador é incrementado automaticamente pelo computador. O fluxograma a seguir ilustra o funcionamento do comando *for*:



☛ *O Pascal exige que a variável utilizada como contador seja declarada como integer.*

3.1. *Fazendo Tabuadas...*

A seguir, mostramos como o comando *for* pode ser usado para criar um programa que exibe a tabuada de um número fornecido pelo usuário:

```
program tabuada;
var  n, c, r : integer;
begin
   write('Digite um número: ');
   readln(n);
   for c:=1 to 10 do
      writeln(n,' x ',c,' = ',n*c);
   readln;
end.
```

☛ *Para repetir vários comandos no for, agrupe-os em um bloco, usando begin e end.*

3.2. *Contagem Decrescente*

Algumas vezes, ao invés de fazer o contador aumentar, é preferível fazê-lo diminuir. Para indicar que a contagem deve ser decrescente, usamos a palavra *downto* em vez de *to*. O programa a seguir exibe uma contagem decrescente usando *downto*:

```
program regressivo;
var n, c: integer;
begin
   write('Digite um número: ');
   readln(n);
   for c:=n downto 1 do
      writeln(c);
   readln;
end.
```

3.3. *Exercícios*

Para cada problema, codifique um programa *Pascal*:

3.1. Faça um programa para ler 10 números e exibir sua média aritmética.

3.2. Dado um número n, informe a soma dos n primeiros números naturais. Por exemplo, se n for igual a 5, o programa deverá exibir a soma 1+2+3+4+5, ou seja, 15.

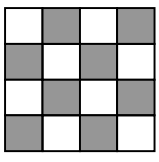
3.3. O quadrado de um número n é igual à soma dos n primeiros números ímpares. Por exemplo, $2^2 = 1+3 = 4$ e $5^2 = 1+3+5+7+9 = 25$. Dado um número n, informe seu quadrado usando essa idéia.

3.4. Dados um número real x e um número natural n, calcule e informe o valor de x^n. [*Dica*: $x^n = x.x...x$ *(com n termos x)*]

3.5. Dado um número natural n, calcule e informe o valor do seu fatorial. [*Dica*: $n! = (n-1).n...3.2.1$]

3.6. Dadas as temperaturas registradas diariamente durante uma semana, informe que dia da semana foi mais frio e que dia foi mais quente.

3.7. Dado um número n, desenhar um tabuleiro de xadrez $n \times n$. Por exemplo, para n=4, temos:

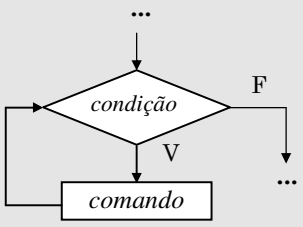


4. REPETIÇÃO COM PRECONDIÇÃO

Através da estrutura de repetição com *precondição*, um comando pode ser executado repetidamente, sem que tenhamos que escrevê-lo diversas vezes no programa. Para controlar a repetição, a execução do comando fica condicionada ao valor de uma expressão lógica, denominada *condição*, que é avaliada a cada nova repetição, sempre antes que o comando seja executado. Enquanto essa expressão é verdadeira, o *comando* é repetido.

```
while condição do
   comando;
```

Veja a seguir o fluxograma que ilustra o funcionamento desta estrutura de repetição:



> ☛ *Para repetir vários comandos no while, agrupe-os em um bloco com begin e end.*

4.1. *Exibindo os Dígitos de um Número...*

No exemplo a seguir mostramos como a estrutura de repetição com precondição pode ser usada para criar um programa que exibe um a um os dígitos que compõem um número natural. Por exemplo, sendo fornecido como entrada o número 8315, o programa deverá exibir como saída os dígitos 5, 1, 3 e 8. A estratégia será dividir o número sucessivamente por 10 e ir exibindo os restos obtidos. O processo pára somente quando o quociente obtido numa das divisões é zero:

```
8315 |10
8310  831 |10
   5  830   83 |10
         1  80    8 |10
            3  0    0  <-- pára
               8
```

```
program digitos;
var  n, r : integer;
begin
   write('Digite um número: ');
   readln(n);
   while n>0 do
      begin
         r := n mod 10;
         n := n div 10;
         writeln(r);
      end;
   readln;
end.
```

4.2. *Exercícios*

Para cada problema, codifique um programa *Pascal*:

4.1. Dado um número n, exibir todos os ímpares menores que n. Por exemplo, para n=10 deverão ser exibidos os ímpares: 1, 3, 5, 7 e 9.

4.2. A soma de n ímpares consecutivos, a partir de 1, é equivalente a n^2. Por exemplo, 1^2=1, 2^2=1+3, 3^2=1+3+5, 4^2=1+3+5+7, ... Inversamente, o número n de ímpares consecutivos que podem ser subtraídos de um número x (sem produzir resultado negativo) é igual à raiz quadrada inteira de x. Por exemplo, se tivermos x=18, poderemos subtrair dele no máximo 1+3+5+7=16, e a resposta será n=4. Dado um número x, informe sua raiz quadrada inteira n, usando essa idéia.

4.3. A *Série de Fibonacci* é: 1, 1, 2, 3, 5, 8, 13, ... Note que os dois primeiros termos desta série são iguais a 1 e, a partir do terceiro, o termo é dado pela soma dos dois termos anteriores. Dado um número $n \geq 2$, exiba todos os termos da série que sejam menores ou iguais a n.

4.4. Numa certa agência bancária, as contas são identificadas por números de até quatro dígitos. Por motivo de consistência, a cada número n é associado um dígito verificador d calculado da seguinte maneira:

> Seja n = 7314 o número da conta.
>
> 1º Adiciona-se todos os dígitos de n, obtendo-se a soma s. Para o n considerado, a soma seria: s = 4+1+3+7 = 15;
>
> 2º O dígito verificador d é dado pelo resto da divisão de s por 10. Para o s considerado, o dígito verificador seria d = 5

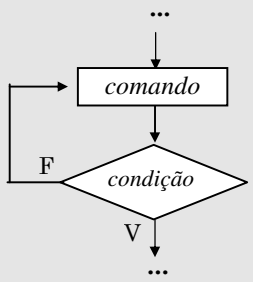
Dado um número de conta n, informe o dígito verificador correspondente.

5. REPETIÇÃO COM POSCONDIÇÃO

A estrutura de repetição com *poscondição* permite que um comando seja executado até que uma determinada condição seja satisfeita. Esse tipo de repetição garante que o comando seja executado pelo menos uma vez antes que a repetição termine.

```
repeat
    comando;
until condição;
```

O fluxograma a seguir ilustra o funcionamento do comando *repeat-until*:



5.1. *Automatizando o Caixa*

Considere o problema: "*Dada uma série de valores, representando os preços dos itens comprados por um cliente, informe o total a ser pago*". Como cada cliente pode comprar um número diferente de itens, não há como prever o número exato de valores de entrada. Então, vamos convencionar que o final da série será determinado por um valor nulo. Isso é razoável, já que não é possível que um item custe R$ 0,00. Assim, o programa deverá solicitar os preços e adicioná-los até que um preço nulo seja informado; nesse momento, o valor total deverá ser exibido.

```
program caixa;
uses crt;
var  s, p : real;
begin
   clrscr;
   s := 0;
   repeat
      write('Preço? ');
      readln(p);
      s := s+p;
   until p=0;
   writeln('Total a pagar: R$',s:0:2);
   readln;
end.
```

☛ *Note que as palavras repeat e until servem como delimitadores de bloco e, assim, dispensam o uso de begin e end.*

5.2. *Forçando uma Entrada*

O comando *repeat-until* é muito utilizado em situações em que o usuário deve ser forçado a digitar um valor dentro de um certo intervalo. Por exemplo, poderíamos alterar o programa da *tabuada*, visto na seção 3.1, de tal forma que o usuário fosse obrigado a digitar um valor entre 1 e 10.

```
program tab;
uses crt;
var  n, c, r : integer;
begin
   clrscr;
   repeat
     gotoxy(1,1);
     write('Digite um número: ');
     clreol;
     readln(n);
   until (n>=1) and (n<=10);
   for c:=1 to 10 do
      writeln(n,' x ',c,' = ',n*c);
   readln;
end.
```

5.3. *Exercícios*

Para cada problema, codifique um programa *Pascal*:

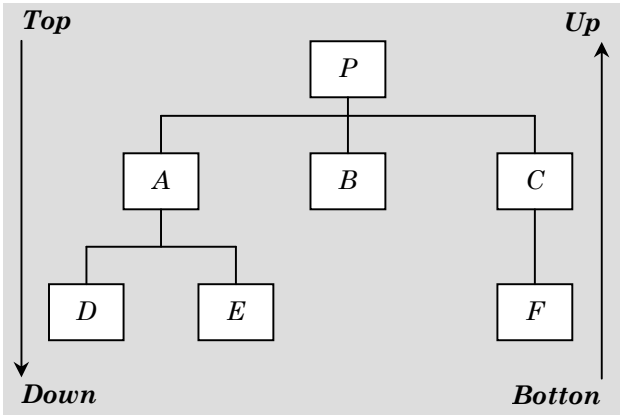
5.1. Dado o saldo inicial e uma série de operações de crédito/débito, informe o total de créditos, o total de débitos, a C.P.M.F. paga (0,40% do total de débitos) e o saldo final da conta.

```
Saldo inicial? 1000.00
Operação? 200
Operação? -50
Operação? -10
Operação? 170
Operação? -500
Operação? 0
--------------
Créditos..: R$ 370.00
Débitos...: R$ 560.00
C.P.M.F...: R$   2.24
Saldo.....: R$ 807.76
```

5.2. Refaça o exercício 2.7, sobre equações do 2º grau, de tal modo que o usuário seja forçado a informar um valor diferente de zero para o coeficiente *a* da equação.

6. MODULARIAÇÃO

A *modularização* é uma técnica de desenvolvimento baseada no princípio de "*divisão e conquista*": para resolvermos um problema complexo, primeiro identificamos e resolvemos subproblemas mais simples e, então, combinamos as soluções desses subproblemas para obter a solução do problema original. Esse processo de decomposição de problemas em subproblemas é representado através de um *diagrama hierárquico funcional* (D.H.F.), conforme segue:



No exemplo acima, o problema original *P* foi decomposto em três subproblemas: *A*, *B* e *C*. Eventualmente, após uma decomposição, podem surgir subproblemas que ainda são muito complexos. Nesse caso, fazemos mais uma decomposição. No nosso exemplo, o subproblema *A* foi decomposto em *D* e *E* e o subproblema *C* foi decomposto em *F*. Não há limite para esse processo de decomposição e ele só termina quando atingimos subproblemas suficientemente simples para que sejam resolvidos diretamente. Cada módulo do D.H.F. corresponde a uma *rotina* que deverá ser codificada no programa *Pascal*. Note que o *projeto* é feito de cima para baixo (*top-down*), enquanto *a implementação* deve ser feita de baixo para cima (*botton-up*). Isso quer dizer que, no programa *Pascal*, um módulo só deve aparecer quando todos aqueles dos quais ele depende já foram codificados anteriormente. Sendo assim, o módulo principal será sempre o último a ser codificado no programa.

6.1. *Módulos*

Um *módulo*[2] pode ser codificado como uma função ou como um procedimento. O módulo será uma *função* se ele deve *devolver um valor* como resultado de sua execução e será um *procedimento* se apenas deve *produzir um efeito* quando é executado. Uma maneira interessante de diferenciar funções de procedimentos é observar que enquanto *funções representam perguntas*, *procedimentos representam ordens*. Por exemplo, considere os comandos a seguir:

```
...
clrscr;
write('Num? ');
readln(n);
r := sqrt(n);
writeln('Raiz de ',n:0:1,' é ',r:0:1);
...
```

O comando *clrscr ordena* que o computador limpe a tela e, portanto, é um procedimento. Já o comando *sqrt*(*n*) *pergunta* ao computador qual é a raiz quadrada de *n* e, portanto, é uma função. O procedimento *clrscr produz um efeito* como resultado de sua execução, enquanto a função *sqrt devolve um valor*. Evidentemente, os comandos *write* e *writeln* são procedimentos e o efeito que produzem é alterar o estado da tela, exibindo informações. Mas e o comando *readln*? É uma função ou um procedimento? Note que *readln*(*n*) corresponde à ordem "*leia n*" e, portanto, é também um procedimento; seu efeito é alterar o valor da variável *n*.

6.2. *Funções*

Ao definir uma função, usamos o seguinte formato:

```
function nome(parâmetros) : tipo;
variáveis locais;
begin
    instruções;
    ...
    nome := resposta;
end;
```

Toda função deve ter um *nome*, através do qual ela possa ser chamada no programa. Para executar, uma função pode precisar receber alguns *parâmetros* de entrada e, ao final de sua execução, devolve uma resposta do *tipo* especificado. Se a função precisa de outras variáveis, além dos parâmetros, essas devem ser declaradas como *variáveis locais*. Para indicar a *resposta* final devolvida pela função, devemos atribuir seu valor ao *nome* da função.

☛ *As variáveis locais e os parâmetros são criados na memória apenas no momento em que a rotina entra em execução e deixam de existir assim que a execução é concluída.*

Por exemplo, a função a seguir calcula a medida da hipotenusa *c* de um triângulo retângulo a partir das medidas dos seus catetos *a* e *b*.

[2] *O mesmo que rotina ou comando.*

```
function hip(a, b : real) : real;
var c : real;
begin
    c := sqrt(sqr(a)+sqr(b));
    hip := c;
end;
```

Para usar essa função num programa, basta lembrar que a implementação deve ser *botton-up* e que, portanto, o código da função deve aparecer *antes* do código do *módulo principal*[3] do programa.

```
program TesteHip;
uses crt;
var x, y, h : real;

{ cálculo da hipotenusa }
function hip(a, b : real) : real;
var c : real;
begin
    c := sqrt(sqr(a)+sqr(b));
    hip := c;
end;

{ módulo principal }
begin
   write('Catetos? ');
   readln(x,y);

   h := hip(x,y);
   write('Hipotenusa: ',h:0:1);
end.
```

A execução do programa inicia-se sempre no módulo principal e se o comando não é chamado nesse módulo ele não é executado. No exemplo acima, o comando *hip*() é chamado com os argumentos *x* e *y*, cujos valores são copiados, respectivamente, para os parâmetros *a* e *b* da função. Esses parâmetros são então utilizados no cálculo da hipotenusa *c*. Para indicar que o valor em *c* deve ser devolvido como resposta da função, devemos atribuí-lo ao *nome* da função. Quando a função termina, as variáveis *a*, *b* e *c* deixam de existir; entretanto, o valor de *c* não é perdido, já que foi preservado no momento em que foi atribuído como resposta da função.

☛ *Em Pascal, comentários são indicados entre chaves e são ignorados na compilação.*

[3] *No Pascal, o módulo principal é o bloco finalizado com end seguido de ponto final.*

6.3 *Exercícios*

Para cada problema a seguir, crie uma função e codifique um programa para testá-la:

6.1. Dado um número natural, determine se ele é par ou ímpar.

6.2. Dado um número real, determine seu valor absoluto, ou seja, seu valor sem sinal.

6.3. Dados dois números reais, determine o máximo entre eles.

6.4. Dados dois números reais, determine a média aritmética entre eles.

6.5. Dados um número real x e um número natural n, determine x^n.

6.6. Dado um número natural n, determine $n!$.

6.7. Dado um número n, determine seu dígito verificador conforme definido no exercício 4.4.

6.4. *Procedimentos*

Procedimentos têm o seguinte formato:

```
procedure nome(parâmetros);
variáveis locais;
begin
    instruções;
    ...
end;
```

Assim como funções, procedimentos devem ter um *nome*, através do qual ele possa ser chamado. Procedimentos podem receber *parâmetros* de entrada, mas não fornecem um valor de saída e, portanto, não precisamos declarar o tipo de resposta.

```
procedure msg(c,l:integer; m:string);
begin
    gotoxy(c,l);
    write(m);
end;
```

O comando *msg*(*c*,*l*,*m*), definido acima, exibe uma dada mensagem *m* numa determinada posição (*c*,*l*) da tela. Ao codificar um programa completo contendo esse comando, é preciso lembrar que um módulo não pode se referir a outro que não tenha ainda sido codificado e, portanto, qualquer outro módulo do programa que utilize o comando *msg*() deve aparecer codificado após ele (similarmente ao que foi feito no programa *TesteHip*).

6.8. O comando *sound*(*f*), cujo parâmetro *f* é um número inteiro, liga o alto-falante para emitir som com freqüência de *f hertz*. Para desligar o alto-falante, temos o comando *nosound* e, para dar uma pausa, temos *delay*(*t*), cujo parâmetro *t* indica o número de milissegundos a aguardar. Usando esses comandos disponíveis em *CRT*, crie o comando *beep*, que emite um "*bip*".

6.9. Usando os comandos de som, crie um comando para simular o som de uma *sirene*. [*Dica*: sons graves têm freqüências baixas e sons agudos têm freqüências altas]

6.10. Crie o comando *centraliza*(*l*,*m*), que exibe a mensagem *m* centralizada na linha *l*, e faça um programa para testá-lo. [*Dica*: a função *length*(*s*) devolve comprimento da *string s*.]

6.11. Crie o comando *horiz*(*c*,*l*,*n*), que exibe uma linha horizontal com *n* caracteres de comprimento, a partir da posição (*c*,*l*) da tela. [*Dica*: para obter uma linha contínua, use o caracter cujo código ASCII é 196]

6.12. A rotina a seguir tem como objetivo desenhar uma moldura cujo canto esquerdo superior está na posição (C_i,L_i) e cujo canto direito inferior está na posição (C_f,L_f); entretanto, ela contém alguns erros. Codifique um programa para testar seu funcionamento e corrija os erros que você observar:

```
procedure M(Ci,Li,Cf,Lf,cor:integer);
var  i: integer;
begin
   textcolor(cor);
   gotoxy(Ci,Li); write(#191);
   gotoxy(Cf,Li); write(#192);
   gotoxy(Ci,Lf); write(#217);
   gotoxy(Cf,Lf); write(#218);

   for i:= Ci+1 to Cf-1 do
      begin
         gotoxy(i,Li); write(#179);
         gotoxy(i,Lf); write(#179);
      end;

   for i:= Li+1 to Lf-1 do
      begin
         gotoxy(Ci,i); write(#196);
         gotoxy(Cf,i); write(#196);
      end;
end;
```

6.13. Baseando-se na rotina acima, crie o comando *vert*(*c*,*l*,*n*), que exibe uma linha vertical com *n* caracteres de comprimento, a partir da posição (*c*,*l*) da tela. Usando essa rotina e *horiz*(), codifique uma nova versão do comando *M*, que desenha molduras.

Um parâmetro pode ser passado a uma rotina de duas maneiras distintas: por valor ou por referência. Quando é *passado por valor*, o parâmetro é criado numa nova área de memória, ainda não utilizada, e o valor do argumento é copiado para essa área. Quando é *passado por referência*, o parâmetro compartilha o mesmo espaço de memória já utilizado pelo argumento. Na passagem por valor, a rotina tem acesso apenas a uma *cópia do argumento*; já na passagem por referência, a rotina acessa diretamente o *argumento original*. Para indicar que a passagem deve ser feita por referência, devemos prefixar a declaração do parâmetro com a palavra *var*.

Para entender a utilidade da passagem por referência, vamos considerar um exemplo: criar um comando que permuta os valores de duas variáveis que lhe são passadas como argumentos. Supondo que esse comando se chame *troca*, o código

```
...
x := 5;
y := 7;
troca(x,y);
write(x,y);
...
```

deverá produzir 75 como saída.

Primeiro vamos entender por que a passagem por valor não funciona nesse caso.

```
...
procedure troca(a,b:integer);
var c : integer;
begin
   c := a;
   a := b;
   b := c;
end;
...
begin
   ...
   x := 5;
   y := 7;
   troca(x,y);
   write(x,y);
   ...
end.
```

a: 5
b: 7
c: ?

Os parâmetros são cópias dos argumentos!

x: 5
y: 7

Argumentos originais.

Como podemos ver acima, parâmetros passados por valor são criados como cópias dos argumentos originais. Quando a rotina é executada, de fato, ela troca os valores das variáveis *a* e *b*, que são destruídas assim que a execução da rotina termina. Retornando ao programa principal, os valores de *x* e *y* estarão inalterados e, portanto, teremos 57 como saída.

Para funcionar como esperado, o comando *troca*() deve permutar diretamente os valores das variáveis *x* e *y*. Isso só é possível se a passagem for feita por referência. Para indicar essa alteração, basta acrescentar a palavra *var* na declaração dos parâmetros:

```
...
procedure troca(var a,b:integer);
var c : integer;
begin
   c := a;          c: ?   } variável local
   a := b;
   b := c;
end;
...
begin
   ...
   x := 5;          a,x: 5 }
   y := 7;                 } Parâmetros e argumentos compartilham o mesmo espaço!
   troca(x,y);      b,y: 7 }
   write(x,y);
   ...
end.
```

Agora, os parâmetros compartilham a mesma área de memória já utilizada pelos argumentos passados à rotina. Evidentemente, qualquer alteração feita nos parâmetros também é feita nos argumentos. Ao final da execução da rotina, as variáveis *a*, *b* e *c* são destruídas; entretanto, somente o espaço de memória utilizado pela variável local *c* pode ser liberado, já que os espaços utilizados por *a* e *b* foram apenas tomados "emprestados" e devem continuar sendo usados pelos argumentos *x* e *y*. Com essa nova versão, obteremos a saída `75`, como era esperado.

6.7. *Valor versus Referência*

A passagem por valor tem como vantagem o fato de que ela garante a *segurança* dos argumentos que são passados à rotina, preservando seus valores originais para uso futuro no programa. Essa segurança, porém, é obtida a custa de um gasto adicional de *espaço*, para criar a cópia na memória, e *tempo*, para transferir os valores do original para a cópia. Por outro lado, a passagem por referência não garante a segurança dos argumentos, já que esses podem ser alterados pela rotina. Porém, é *mais eficiente* em termos de espaço e tempo, já que os argumentos não precisam ser duplicados na memória.

A decisão entre usar passagem por valor ou referência deve ser feita com base na seguinte regra: *se é esperado que um argumento seja alterado pela execução de uma rotina, então a sua passagem deve ser feita por referência.*

6.8. *Exercícios*

6.14. No comando *readln*(*n*), o argumento *n* é passado por valor ou por referência? Por quê?

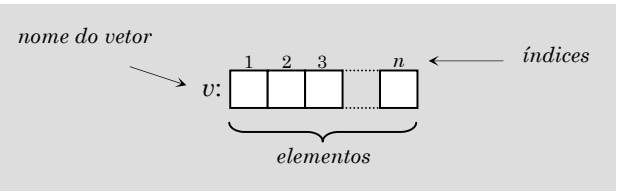
6.15. Explique por que o parâmetro *n* da função para cálculo de dígito verificador, criada no exercício 6.7, deve ser passado por valor.

6.16. Crie o comando *incr*(*v*,*n*), que incrementa o valor da variável *v* em *n* unidades. Por exemplo, se *v* vale 5, após a execução de *incr*(*v*,3), a variável *v* deverá estar valendo 8.

6.17. As funções *wherex* e *wherey*, definidas em *CRT*, devolvem a coluna e a linha em que se encontra o cursor. Usando essas funções, crie a rotina *cursor*(*x*,*y*), que devolve a posição corrente do cursor através dos parâmetros *x* e *y*.

7. VETORES

Um *vetor* é uma coleção de variáveis de um mesmo tipo, que compartilham o mesmo nome e que ocupam posições consecutivas de memória. Cada variável da coleção denomina-se *elemento* e é identificada por um *índice*. Se *v* é um vetor indexado de 1 a *n*, seus elementos são *v*[1], *v*[2], *v*[3], ..., *v*[*n*].



Antes de criar um vetor é preciso declarar o seu tipo, ou seja, especificar a quantidade e o tipo dos elementos que ele irá conter. Isso é feito através do comando *type*, conforme exemplificado a seguir.

```
type vetor = array[1..5] of integer;
var v : vetor;
```

A primeira linha define um novo tipo de vetor, denominado *vetor*, cujos elementos indexados de 1 a 5 são do tipo *integer*. A segunda linha declara uma variável, denominada *v*, do tipo recém definido.

Em geral, um vetor *v* pode ser indexado com qualquer expressão cujo valor seja um número inteiro. Essa expressão pode ser uma simples constante, uma variável ou então uma expressão propriamente dita, contendo operadores aritméticos, constantes e variáveis. Por exemplo, seja *i* uma variável do tipo *integer* e *v* o vetor criado no exemplo anterior. Se $i=3$, então $v[i \text{ div } 2] \equiv v[1]$, $v[i-1] \equiv v[2]$, $v[i] \equiv v[3]$, $v[i+1] \equiv v[4]$ e $v[2*i-1] \equiv v[5]$. Entretanto, $v[i/2]$ causará um erro de compilação; já que a expressão $i/2$ tem valor igual a 1.5, que não é um índice permitido.

7.1. *As temperaturas Acima da Média*

Dadas as temperaturas que foram registradas diariamente, durante uma semana, deseja-se determinar em quantos dias dessa semana a temperatura esteve acima da média. A solução para esse problema envolve os seguintes passos:

① *obter os valores das temperaturas;*
② *calcular a média entre esses valores;*
③ *verificar quantos deles são maiores que a média.*

Lembrando da técnica de modularização, cada um desses passos representa um subproblema cuja solução contribui para a solução do problema originalmente proposto. Então, supondo que eles já estivessem solucionados, o programa poderia ser codificado como segue:

```
program TAM;
const max = 7;
Type  Temp = array[1..max] of real;
var   T : Temp;
      m : real;
...
begin
   obtem(T);
   m := media(T);
   writeln('Total = ',conta(T,m));
end.
```

☛ *A declaração const é usada para criar constantes, disponíveis a todo o programa.*

O programa ficou extremamente simples; mas, para ser executado, é preciso que as rotinas *obtem*, *media* e *conta* sejam definidas. A rotina para a obtenção dos dados pode ser codificada da seguinte maneira:

```
procedure obtem(var T : Temp);
var i : integer;
begin
   writeln('Informe temperaturas: ');
   for i:=1 to max do
      begin
         write(i,'º valor? ');
         readln(T[i]);
      end;
end;
```

Note que o parâmetro *T* é passado por referência, já que desejamos preencher o vetor originalmente passado à rotina e não uma cópia dele. Lembre-se de que as cópias são destruídas ao final da execução da rotina e que, portanto, os valores armazenados serão perdidos se a passagem for por valor. Note também que, na rotina que calcula a média a seguir, a passagem poderia ser por valor, pois não pretendemos alterar a variável original; entretanto, a passagem por referência torna o código mais eficiente.

```
function media(var T : Temp) : real;
var i : integer;
    S : real;
begin
   S := 0;
   For i:=1 to Max do
      S := S + T[i];
   media := S/max;
end;
```

Finalmente, a rotina que faz a contagem das temperaturas acima da média fica assim:

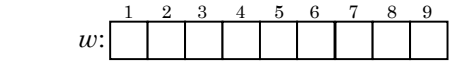
```
function conta(var T:Temp;
                  m:real) : integer;
var i, c : integer;
begin
   c := 0;
   for i:=1 to Max do
      if T[i] > m then
         c := c+1;
   Conta := c;
end;
```

7.2. *Exercícios*

7.1. Crie tipos de vetores para armazenar:

① as letras vogais do alfabeto;
② as alturas de um grupo de 10 pessoas; e
③ os nomes dos meses do ano.

7.2. Considere um vetor *w* cujos 9 elementos são do tipo *integer*:



Supondo que *i* seja uma variável do tipo *integer* e que seu valor seja 5, que valores estarão armazenados em *w* após a execução das atribuições a seguir?

```
① w[1] := 17;
② w[i div 2] := 9;
③ w[2*i-1] := 95;
④ w[i-1] := w[9] div 2;
⑤ w[i] := w[2];
⑥ w[i+1] := w[i]+ w[i-1];
⑦ w[w[2]-2] := 78;
⑧ w[w[i]-1] := w[1]* w[i];
```

7.3. Codifique a rotina *minimax(T,x,y)*, que devolve através dos parâmetros *x* e *y*, respectivamente, a mínima e a máxima entre as temperaturas armazenadas no vetor *T*.

7.4. Codifique a rotina *Histograma(T)*, que exibe um histograma da variação da temperatura durante a semana. Por exemplo, se as temperaturas em *T* forem 19, 21, 25, 22, 20, 17 e 15°C, a rotina deverá exibir:

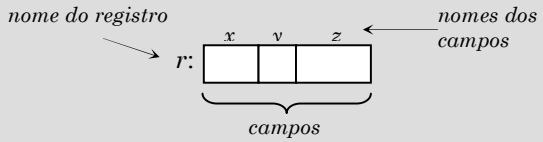
```
D: ■■■■■■■■■■■■■■■■■
S: ■■■■■■■■■■■■■■■■■■■■
T: ■■■■■■■■■■■■■■■■■■■■■■■■
Q: ■■■■■■■■■■■■■■■■■■■■
Q: ■■■■■■■■■■■■■■■■■■
S: ■■■■■■■■■■■■■■■■
S: ■■■■■■■■■■■■■■■
```

Suponha que as temperaturas em *T* sejam todas positivas e que nenhuma seja maior que 80°C. [*Dica*: crie uma rotina que exibe uma linha com tamanho proporcional à temperatura.]

7.5. Usando as rotinas desenvolvidas nos dois exercícios anteriores, altere o programa *TAM* para exibir a temperatura média, a mínima, a máxima e também o histograma de temperaturas.

8. REGISTROS

Um *registro* é uma coleção de variáveis logicamente relacionadas que não precisam ser do mesmo tipo. Como no vetor, essas variáveis também compartilham o mesmo nome e ocupam posições consecutivas de memória. Cada variável da coleção é um *campo* do registro e é identificada por um *nome* de campo. Se *x*, *y* e *z* são nomes de campo válidos e *r* é um registro, então *r.x*, *r.y* e *r.z*. são os campos de *r*.



Por exemplo, as declarações a seguir permitem a criação de uma variável capaz de armazenar datas:

```
type Data = record
              dia : integer;
              mes : integer;
              ano : integer;
            end;
var hoje : Data;
```

Para atribuir valores aos campos do registro *hoje*, podemos escrever:

```
hoje.dia := 25;
hoje.mes := 2;
hoje.ano := 2000;
```

8.1. *Registros Aninhados*

É possível criar um registro em que um ou mais de seus campos também sejam registros, desde que tais registros tenham sido previamente definidos. Por exemplo, temos a seguir a criação de um registro contendo um campo do tipo *Data* já definido:

```
type Pessoa = record
                nome : string[31];
                fone : string[20];
                nasc : Data;
              end;
var amigo : Pessoa;
```

Para atribuir valores aos campos do registro *amigo*, podemos escrever:

```
amigo.nome := 'Itivaldo Buzo';
amigo.fone := '850-9973';
amigo.nasc.dia := 27;
amigo.nasc.mes := 7;
amigo.nasc.ano := 1970;
```

8.2. *Tabelas*

Também é possível combinar vetores e registros de muitas maneiras interessantes. A combinação mais comum é um vetor cujos elementos são registros. Como exemplo, vamos criar uma variável para armazenar uma agenda contendo informações sobre vários amigos:

```
const max = 10;
type Agenda = array[1..max] of Pessoa;
Var  a : Agenda;
```

Por exemplo, para atribuir valores ao segundo elemento do vetor *a*, escrevemos:

```
a[2].nome := 'Roberta Soares';
a[2].fone := '266-0879';
a[2].nasc.dia := 15;
a[2].nasc.mes := 11;
a[2].nasc.ano := 1971;
```

Um vetor cujos elementos são registros é denominado *tabela* e é representado com os elementos dispostos em linhas e os campos em colunas.

	nome	fone	nasc
1	Itivaldo Buzo	850-9973	27/07/1970
2	Roberto Soares	266-0879	15/11/1971
3	Maria da Silva	576-8292	09/05/1966
10	Pedro Pereira	834-0192	04/08/1973

8.3. *Exercícios*

8.1. Defina um tipo de registro para armazenar dados de um vôo, como por exemplo os nomes das cidades de origem e de destino, datas e horários de partida e chegada. Crie uma variável desse tipo e atribua valores aos seus campos.

8.2. Usando o tipo já definido no exercício anterior, defina um tipo de tabela para armazenar os dados de todos os vôos de um aeroporto (suponha que o total de vôos seja 5) e codifique uma rotina para preencher uma tabela dessas.

8.3. Crie uma rotina que receba uma tabela contendo as informações de vôos e a exiba na tela.

8.4. Crie uma rotina que receba uma tabela contendo as informações de vôos e uma data e exiba na tela todos os vôos para a data indicada.

8.5. Usando as rotinas já definidas, codifique um programa completo para (1) preencher a tabela de vôos, (2) exibir a tabela de vôos na tela e (3) permitir ao usuário realizar consultas sobre os vôos de uma determinada data até que ele deseje parar.

9. ARQUIVOS

Um *arquivo* é semelhante a um vetor, exceto por dois motivos: primeiro, o vetor fica armazenado na memória RAM, enquanto o arquivo fica armazenado em disco; segundo, o vetor deve ter um tamanho fixo, definido em sua declaração, enquanto o tamanho do arquivo pode variar durante a execução do programa. A vantagem no uso de arquivos é que, diferentemente do que ocorre com vetores, os dados não são perdidos entre uma execução e outra do programa. A desvantagem é que o acesso a disco é muito mais lento do que o acesso à memória e, conseqüentemente, o uso de arquivos torna a execução do programa mais lenta. Para melhorar a eficiência, em geral, partes do arquivo são carregadas em tabelas armazenadas na memória antes de serem manipuladas. Isso diminui o número de acessos em disco e aumenta a velocidade do programa.

Para usar um arquivo, é preciso primeiro definir o seu tipo. Por exemplo, a seguir definimos um arquivo para armazenar dados de funcionários

```
type data = record
              dia, mes, ano: integer;
            end;
type func = record
              nome    : string[30];
              salario : real;
              admissao: Data;
            end;
type cadastro = file of func;
var A : cadastro;
    R : func;
```

9.1. *Principais Operações em Arquivos*

Seja *A* uma variável arquivo e *R* uma variável registro, do tipo daqueles armazenados em *A*. Para manipular arquivos, temos os seguintes comandos básicos:

- *assign*(*A*,*'nome do arquivo'*): associa à variável *A* o *nome* do arquivo que ela irá representar;
- *reset*(*A*): abre arquivo representado por *A*, no modo de *leitura*, e posiciona o primeiro registro como corrente; caso o arquivo não exista em disco, ocorre um erro de execução;
- *rewrite*(*A*): abre o arquivo representado por *A*, no modo de *gravação*, e torna corrente sua primeira posição; caso o arquivo já exista, ele é apagado.
- *eof*(*A*): devolve *true* se já foi atingido o *final do arquivo* representado por *A*.
- *close*(*A*): *fecha* o arquivo representado por *A*.
- *read*(*A*,*R*): lê em *R* o registro corrente no arquivo representado por *A* e pula para o próximo;
- *write*(*A*,*R*): grava o registro *R* na posição corrente do arquivo representado por *A* e pula para a próxima posição;
- *seek*(*A*,*P*): torna *P* (os registros são numerados a partir de zero) a posição corrente no arquivo representado por *A*;
- *filepos*(*A*): devolve a posição corrente no arquivo representado por *A*;
- *filesize*(*A*): devolve o tamanho (número de registros armazenados) do arquivo representado por *A*;
- *truncate*(*A*): trunca (elimina todos os registros) a partir da posição corrente, até o final, do arquivo representado por *A*.

9.2. *Gravando um Arquivo*

Vamos codificar um simples programa que solicita os dados dos funcionários ao usuário e os armazena em um arquivo em disco. Como a quantidade exata de funcionários é desconhecida, vamos convencionar que a entrada de dados termina se o usuário digita ponto final quando lhe é solicitado o nome do funcionário:

```
program cadastra;
type data = record
              dia, mes, ano: integer;
            end;
     func = record
              nome    : string[30];
              salario : real;
              admissao: Data;
            end;
     cadastro = file of func;
var A : cadastro;
    R : func;
begin
   assign(A,'FUNC.DAT');
   rewrite(A);
   repeat
      write('Nome? ');
      readln(R.nome);
      if R.nome<>'.' then
         begin
            write('Salario? ');
            readln(R.salario);
            writeln('Admissao: ');
            write('Dia? ');
            readln(R.admissao.dia);
            write('Mês? ');
            readln(R.admissao.mes);
            write('Ano? ');
            readln(R.admissao.ano);
            write(A,R);
         end;
   until R.nome='.';
   close(A);
end.
```

9.3. *Lendo um Arquivo*

A execução do programa *cadastra*, codificado acima, fará com que seja criado em disco um arquivo chamado FUNC.DAT. Agora, vamos criar um programa que seja capaz de ler esse arquivo e listar seus dados em tela. Para que o arquivo seja lido corretamente, é preciso usar a *mesma* definição de tipo.

```
program lista;
type data = record
              dia, mes, ano: integer;
            end;
     func = record
              nome    : string[30];
              salario : real;
              admissao: Data;
            end;
     cadastro = file of func;
var A : cadastro;
    R : func;
begin
   assign(A,'FUNC.DAT');
   reset(A);

   while not eof(A) do
      begin
         read(A,R);
         write(R.nome,', ');
         write(R.salario:0:2,', ');
         write(R.admissao.dia,'/');
         write(R.admissao.mes,'/');
         writeln(R.admissao.ano);
      end;
   close(A);
end.
```

9.3. Exercícios

9.1. Altere o programa sobre horários de vôos, codificado no exercício 8.5, de tal forma que o usuário tenha a opção de gravar a tabela em disco ou carregar a tabela com dados lidos do disco.

9.2. Com o comando *truncate* não é possível remover um registro do meio do arquivo, já que ele apaga tudo desde a posição corrente até o final do arquivo. A solução é a seguinte: para remover o registro de uma posição *p*, no meio do arquivo, copie o último registro do arquivo para a posição *p* e então elimine apenas o último registro. Com base nessa idéia, codifique um programa, denominado *exclui*, que permite ao usuário excluir qualquer um dos registros do arquivo FUNC.DAT, criado pelo programa *cadastra*. Para cada operação de exclusão, o programa deve listar os registros, e suas respectivas posições, e solicitar ao usuário o número do registro a ser excluído. [*Dica*: abra o arquivo no modo de leitura para não perder os dados]

9.3. Para acrescentar um registro num arquivo existente, abra o arquivo no modo de leitura, posicione-se na primeira posição disponível no final do arquivo e grave o novo registro. Usando essa idéia, crie um programa para inclusão.