

Curso de Formação Profissional em Técnico em Informática

DISCIPLINA DE LÓGICA DE PROGRAMAÇÃO:

APOSTILA DA LINGUAGEM DE PROGRAMAÇÃO PASCAL

Avaliação

- ♦ Provas escritas
- ♦ Resolução de listas de exercícios
- ♦ Trabalhos

Nome do Aluno: _____

Prof.: Arley Rodrigues

Site: www.portalqigaweb.com.br

LINGUAGEM DE PROGRAMAÇÃO PASCAL

1. INTRODUÇÃO

Para armazenar um algoritmo na memória de um computador e para que ele possa, em seguida, comandar as operações a serem executadas, é necessário que ele seja **programado**, isto é, que seja transcrito para uma **linguagem** que o computador possa entender, direta ou indiretamente.

1.1. LINGUAGENS DE PROGRAMAÇÃO

Linguagem é uma maneira de comunicação que segue uma forma e uma estrutura com significado interpretável. Portanto, **linguagem de programação** é um conjunto finito de palavras, comandos e instruções, escritos com o objetivo de orientar a realização de uma tarefa pelo computador.

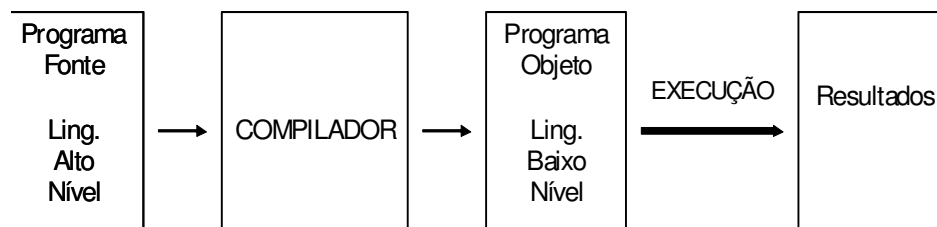
Logicamente, a linguagem que nós utilizamos em nosso cotidiano é diferente da linguagem utilizada pela máquina. A máquina trabalha somente com códigos numéricos (linguagem de máquina), baseados nos números 0 e 1 (sistema binário), que representam impulsos elétricos, ausente e presente.

Este é o conceito de nível de linguagem: alto nível para as mais próximas da linguagem humana; baixo nível para as mais semelhantes à linguagem de máquina.

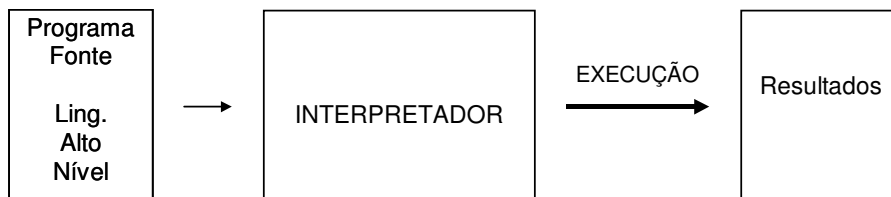
1.2. TRADUTORES

Para que um computador possa "entender" um programa escrito em uma linguagem de alto nível, torna-se necessário um meio de tradução entre a linguagem utilizada no programa e a linguagem de máquina. Este meio pode ser de dois tipos: **compilador** e **interpretador**.

COMPILADOR - traduz o programa escrito em linguagem de alto nível (programa-fonte) para um programa equivalente escrito em linguagem de máquina (programa-objeto).



INTERPRETADOR - traduz e envia para execução, instrução por instrução e o programa permanece na forma fonte.



Exemplos de linguagens de programação: PASCAL, C, CLIPPER, BASIC, COBOL, etc.

1.3 A LINGUAGEM PASCAL

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

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

O **TURBO PASCAL** é um programa que contém, além do compilador PASCAL, um ambiente completo de programação, com editor de programa, depurador de erros, sistema de ajuda, etc.

Estudaremos, a seguir, os itens fundamentais que compõe a linguagem PASCAL.

2. ELEMENTOS BÁSICOS

2.1. IDENTIFICADORES

São nomes escolhidos para representar constantes, variáveis, tipos, funções, procedimentos, UNITS, programas e campos de um registro. Para definirmos um identificador, devemos observar o seguinte:

- pode ter qualquer comprimento, mas apenas os sessenta e três primeiros caracteres são significativos;
- deve ter como primeiro caracter uma letra;
- após a primeira letra só pode conter letras, dígitos ou sublinha (_);
- não pode conter espaços;
- letras maiúsculas e minúsculas são indiferentes; (não é case sensitive)
- não podem haver identificadores repetidos;
- não pode ser uma palavra reservada.

2.2. PALAVRAS RESERVADAS

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

ABSOLUTE	END	INLINE	PROCEDURE	TYPE
AND	EXTERNAL	INTERFACE	PROGRAM	UNIT
ARRAY	FILE	INTERRUPT	RECORD	UNTIL
BEGIN	FOR	LABEL	REPEAT	USES
CASE	FORWARD	MOD	SET	VAR

CONST	FUNCTION	NIL	SHL	WHILE
DIV	GOTO	NOT	SHR	WITH
DO	IF	OF	STRING	XOR
DOWNT0	IMPLEMENTATION	OR	THEN	
ELSE	IN	PACKED	TO	

3. TIPOS DE DADOS

3.1. SIMPLES

INTEGER - Envolve os números inteiros. Na versão 7.0 do Turbo Pascal, existem também outros tipos de números inteiros: SHORTINT, BYTE, WORD e LONGINT.

Tipo	Valor mínimo	Valor máximo	Bytes ocupados
SHORTINT	-128	127	1
BYTE	0	255	1
INTEGER	-32768	32767	2
WORD	0	65535	2
LONGINT	-2147483648	2147483647	4

Exemplos: -45, 1, 138, 0, -2

REAL - abrange os números reais. Na versão 7.0, existem também outros tipos de números reais: SINGLE, DOUBLE, EXTENDED e COMP.

Tipo	Valor mínimo	Valor máximo	Bytes ocupados	Dígitos Significativos
REAL	2.9×10^{-39}	1.7×10^{38}	6	11-12
SINGLE	1.5×10^{-45}	3.4×10^{38}	4	7-8
DOUBLE	5.0×10^{-324}	1.7×10^{308}	8	15-16
EXTENDED	3.4×10^{-4932}	1.1×10^{4932}	10	19-20
COMP	$-2^{63} + 1$	$2^{63} - 1$	8	19-20

Exemplos: 4.5, -32.0, .5, 7.8E3, 21E+3, -315E-3

CHAR - representa um único carácter, escrito entre apóstrofos ('). A maioria dos computadores utilizam a tabela de códigos ASCII para representar todos os caracteres disponíveis. Exemplos:

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

BOOLEAN - representa um valor lógico. Utiliza apenas duas constantes lógicas: TRUE (verdadeiro) e FALSE (falso).

3.2. ESTRUTURADOS

STRING - formado por um conjunto de elementos do tipo CHAR. O tamanho máximo é de 255 caracteres. Exemplos:

'ASPER', 'Processamento de Dados', '123'

Os outros tipos de dados estruturados são: **ARRAY**, **RECORD**, **FILE**, **SET** e **TEXT**.

4. EXPRESSÕES ARITMÉTICAS

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

4.1. OPERADORES ARITMÉTICOS

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

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

EXEMPLOS:

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

4.2. PRIORIDADE

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

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

OBSERVAÇÕES:

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

EXEMPLOS:

$$2 + 3 / 2 = 2 + 1.5 = 3.5$$

$$(2 + 3) / 2 = 5 / 2 = 2.5$$

4.3. FUNÇÕES E PROCEDIMENTOS NUMÉRICOS PREDEFINIDOS

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

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

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

$$\text{EXP}(\text{LN}(X)*N)$$

EXEMPLOS:

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

5. EXPRESSÕES LÓGICAS

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

5.1. OPERADORES RELACIONAIS

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

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

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

EXEMPLOS:

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

5.2. OPERADORES LÓGICOS

São usados para combinar expressões lógicas.

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

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

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

5.3. PRIORIDADE

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

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

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

(A > B) OR (B = C)

6. FORMATO DE UM PROGRAMA PASCAL

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

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

Um programa escrito em Pascal tem o seguinte formato:

```
PROGRAM <identificador>;
```

```
<bloco>.
```

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

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

Observação: no Turbo Pascal, a cláusula PROGRAM, bem como a seqüência correta das declarações, não são obrigatórios.

6.1. DECLARAÇÃO DE USO DE UNITS

Um programa Pascal pode fazer uso de algumas UNITS padrão que estão disponíveis no Sistema Turbo, tais como: CRT, DOS, PRINTER, GRAPH, etc.

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

```
USES <UNIT> , ... , <UNIT> ;
```

EXEMPLO:

```
USES CRT,PRINTER;
```

6.2. DECLARAÇÃO DE CONSTANTES

Serve para associarmos nomes às constantes utilizadas no programa. Possui o seguinte formato:

```
CONST <identificador>=<valor>;...;<identificador>=<valor>;
```

EXEMPLO:

```
CONST BRANCO = ' ' ; PI = 3.1416 ; MAX = 10 ; OK = TRUE;
```

6.3. DECLARAÇÃO DE TIPOS

Serve para definirmos novos tipos e estruturas de dados. Não detalharemos esse tópico agora.

6.4. DECLARAÇÃO DE VARIÁVEIS

Serve para associarmos tipos às variáveis utilizadas no programa. Possui o seguinte formato:

```
VAR  
  
  <lista-de-identificadores> : <tipo>;  
  
  ...  
  
  <lista-de-identificadores> : <tipo>;
```

Onde:

<lista-de-identificadores> é uma lista de nomes de variáveis de um mesmo tipo, separadas por vírgula.
<tipo> é o nome de um dos tipos predefinidos na linguagem ou criado pelo programador.

EXEMPLO:

```
VAR   X,Y,Z : REAL;  
  
      I,J : INTEGER;  
  
      ACHOU : BOOLEAN;  
  
      LETRA : CHAR;  
  
      PALAVRA,FRASE : STRING;
```

6.5. DECLARAÇÃO DE PROCEDIMENTOS E FUNÇÕES

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

6.6. ÁREA DE COMANDOS

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

```
BEGIN  
  
  <comando> ;  
  
  ... ;  
  
  <comando>  
  
END
```

7. COMENTÁRIOS

Um comentário é usado para aumentar a clareza de um programa, embora não seja analisado pelo computador. Deve ser escrito entre chaves:

```
{ comentário }
```

8. COMANDO DE ATRIBUIÇÃO

O comando de atribuição tem a forma:

<identificador> := <expressão>

No comando de atribuição, a variável e a expressão devem ser do mesmo tipo, exceto nos seguintes casos:

- a) a variável sendo *real*, a expressão pode ser *integer*
- b) a variável sendo *string*, a expressão pode ser *char*

EXEMPLOS:

Var

I : Integer;

R : Real;

S : String;

C : Char;

Begin

I := 5;

R := I;

C := 'A';

S := C

End.

9. COMANDO DE ENTRADA

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

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

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

ou

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

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

EXEMPLOS:

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

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

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


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

11. COMANDOS DE DECISÃO

As estruturas de decisão (condicionais) são utilizadas para tomar uma decisão baseada no resultado da avaliação de uma condição de controle e seleciona uma ou mais ações possíveis (comandos) para serem executados pelo computador.

No Pascal, existe três tipos de estrutura de decisão: O comando IF, que pode ser utilizado de duas formas: simples ou composto; e o comando CASE, que é utilizado para uma decisão seletiva.

11.1 COMANDO DE DECISÃO SIMPLES (IF-THEN)

Utilizado quando se deseja executar uma ação (um comando ou uma seqüência de comandos) caso uma determinada condição seja verdadeira. A estrutura de decisão simples do Pascal é o **IF**, e deve ser utilizada da seguinte forma:

```
IF <condição> THEN <comando>
```

Neste caso, o <comando> só será executado se a <condição> resultar no valor TRUE.

A <condição> deve ser uma expressão lógica. O <comando> pode ser um comando simples ou um comando composto. Um comando composto é formado por dois ou mais comandos, separados por ponto-e-vírgula e delimitados por BEGIN e END.

EXEMPLO:

```
Program EXEMPLO_DE_DECISAO_SIMPLES;  
  
    {Ler um número inteiro e exibí-lo se for positivo}  
  
Var  
  
    N : integer;  
  
Begin  
  
    readln(N);  
  
    if N > 0 then  
  
        writeln(N)  
  
End.
```

No exemplo acima, o comando WRITE só será executado se a condição N>0 for verdadeira. Caso contrário, nenhuma ação será executada.

11.2 COMANDO DE DECISÃO COMPOSTA (IF-THEN-ELSE)

Utilizada quando se deseja executar um entre dois comandos (ou uma entre duas seqüências de comandos) dependendo do resultado de uma condição.

A estrutura de decisão composta do Pascal também é o **IF**, mas executado com a seguinte sintaxe:

```
IF <condição> THEN <comando1> ELSE <comando2>
```

Neste caso, se a <condição> resultar no valor TRUE, será executado o <comando1>; caso contrário, será executado o <comando2>. Também aqui, a <condição> deve ser uma expressão lógica, e <comando1> e <comando2> devem ser um comando simples ou um comando composto.

EXEMPLO:

```
Program EXEMPLO_DE_DECISAO_COMPOSTA;  
  
    {Lê um número e determinar se é maior que zero ou não}  
  
Var  
  
    N : integer;
```

Begin

```
  readln(N);
```

```
  if N > 0 then
```

```
    writeln (N, ' é maior que zero' )
```

```
  else
```

```
    writeln (N, ' não é maior que zero')
```

End.

Neste exemplo, a mensagem que será exibida dependerá do resultado da expressão lógica $N > 0$. Se for verdadeira, será executado o comando WRITE que sucede a palavra THEN. Caso contrário, será executado o WRITE que sucede a palavra ELSE. Em nenhuma hipótese serão executados os dois comandos.

Em algoritmos mais complexos, é comum a utilização de IF's aninhados, ou seja, uma estrutura IF possuindo como <comando> uma outra estrutura IF.

EXEMPLO:

```
Program EXEMPLO_DE_IFS_ANINHADOS;
```

```
{Determinar se um número é maior, menor ou igual a zero}
```

```
Var
```

```
  N : integer;
```

```
Begin
```

```
  readln(N);
```

```
  if N > 0 then
```

```
    writeln (N, ' é maior que zero' )
```

```
  else
```

```
    if N < 0 then
```

```
      writeln (N, ' é menor que zero')
```

```
    else
```

```
      writeln (N, ' é igual a zero')
```

```
End.
```

Pode-se observar que diversas linhas deste programa terminaram sem o ponto-e-vírgula, isto porque o ponto-e-vírgula só é utilizado para separar comandos e/ou estruturas.

Quando for necessário mais de um comando na estrutura, então deve ser colocado begin e end dentro da estrutura, sendo acrescentado um ponto e vírgula no final de cada comando.

No caso de haver Begin e end no IF e não tiver ELSE, deve-se colocar um ponto e vírgula no end

No caso de haver Begin e end no IF e no ELSE, deve-se colocar um ponto e vírgula no end do ELSE

No caso de haver Begin e end apenas no ELSE, deve-se colocar um ponto e vírgula no end do ELSE

No caso de haver Begin e end no IF e ter ELSE, não deve ser colocado um ponto e vírgula no end

Deve-se tomar cuidado quando da utilização de IF's aninhados, pois a cláusula ELSE é sempre relacionada ao último IF. Se, dentro de algum programa, precisarmos contornar este fato, podemos fazê-lo com os delimitadores BEGIN e END.

EXEMPLO:

LINGUAGEM ALGORITMICA	LINGUAGEM PASCAL
se COND1 então	if COND1 then
Se COND2 então	begin
Comando1	if COND2 then
senão Comando2	comando1
	end
	else Comando2

11.3 COMANDO DE DECISÃO MÚLTIPLA (CASE - OF)

Utilizada quando se deseja executar um entre vários comandos (ou uma entre várias seqüências de comandos) dependendo do resultado de uma expressão.

A estrutura de seleção (decisão múltipla) do Pascal é o CASE, e obedece a seguinte sintaxe:

```

CASE <expressão> OF
    <lista-de-constantes-1> : <comando-1>;
    <lista-de-constantes-2> : <comando-2>;
    ...
    [ELSE <comando-n>]
END

```

A <expressão> deve resultar um tipo escalar (outros tipos que não o REAL e o STRING). A <lista-de-constantes-x> devem conter uma ou mais constantes (separadas por vírgula), e devem ser do mesmo tipo da <expressão>. O <comando-x> pode ser um comando simples ou composto.

O resultado de <expressão> é comparado com cada constante da <lista-de-constante> para verificar igualdade. Caso a igualdade seja verificada, o <comando> correspondente é executado e a estrutura finalizada. Caso nenhuma igualdade seja verificada, o <comando> correspondente ao ELSE (optativo) será executado.

EXEMPLO:

```

Program EXEMPLO_DE_DECISAO_MÚLTIPLA;
{Simulador de uma calculadora básica de números inteiros}

Uses
    CRT;

Var
    X,Y : integer;
    OP : char;

Begin
    clrscr;
    write('Digite os operandos: ');
    readln(X,Y);

```

```
write('Digite o operador: ');
readln(OP);

case OP of
    '+' : writeln(X + Y);
    '-' : writeln(X - Y);
    '*', 'x', 'X' : writeln(X * Y);
    '/' : writeln(X div Y);
    else writeln('operação inválida');
end; {case}

readkey;

End.
```

Neste exemplo, a mensagem que será exibida dependerá do conteúdo da variável OP. Se for igual a uma das constantes especificadas, será executado o comando WRITELN correspondente. Se nenhuma constante for igual ao conteúdo de OP, será executado o WRITELN do ELSE.

Podemos também escrever o mesmo programa acima sem utilizar a estrutura CASE, apenas utilizando IF's aninhados.

EXEMPLO:

```
Program EXEMPLO_DE_DECISAO_MÚLTIPLA_2;

{Simulador de uma calculadora básica de números inteiros}

Uses
    CRT;

Var
    X,Y : integer;
    OP : char;

Begin
    clrscr;

    write('Digite os operandos: ');
    readln(X,Y);

    write('Digite o operador: ');
    readln(OP);

    if OP='+' then
        writeln(X + Y)
    else
        if OP='-' then
            writeln(X - Y)
        else
            if (OP='*') or (OP='x') or (OP='X') then
```

```
writeln(X * Y)

else

  if OP='/' then

    writeln(X div Y)

  else

    writeln('operação inválida');

readkey;

End.
```

11.4 Comando Label e Goto

A instrução *Goto* permite desviar a seqüência de execução do programa para um determinado *Label* pré-definido. Para utilizarmos algum *Label*, ele deve, obrigatoriamente, ser declarado na subárea *Label*.

Exemplo:

Program LblGoTo2; { *Determina se os valores x y z digitados formam um triângulo. Supondo que x,y,z, sejam os valores lidos, então:*

1-) Se $x < y + z$ e $y < x + z$ e $z < x + y$ então x,y,z são lados de um triângulo:

```
Label INICIO;

Uses CRT;

Var x,y,z : Real;

    Tecla : Char;

Begin

INICIO:

  ClrScr;

  Write('X = '); Readln(x);

  Write('Y = '); Readln(y);

  Write('Z = '); Readln(z);

  Writeln;Writeln;

  If (x<y+z) and (y<x+z) and (z<x+y)

    Then If (x=y) and (x=z)

      Then Writeln('TRIÂNGULO EQUILÁTERO')

      Else If (x=y) Or (x=z) Or (y=z)

        Then Writeln('TRIÂNGULO ISÓSCELES')

        Else Writeln('TRIÂNGULO ESCALENO')

      Else Writeln('X,Y,Z NÃO SÃO LADOS DE UM TRIÂNGULO');

  Writeln;Writeln;

  Write('Deseja Continuar ? --> ');
```

```
Tecla:=ReadKey;
```

```
If (Tecla='s') Or (Tecla='S')
```

```
Then Goto INICIO;
```

```
End.
```

12. COMANDOS DE REPETIÇÃO

Além de permitir a mudança da seqüência de execução de um conjunto de comandos de um programa, as estruturas de controle de uma linguagem, dispõem de recursos para repetir a execução de um conjunto de comandos.

No Pascal, existe três tipos de estrutura de repetição: com teste no início (WHILE), com teste no final (REPEAT) e automática (FOR).

12.1. REPETIÇÃO COM TESTE NO INÍCIO (WHILE-DO)

A estrutura de controle WHILE permite que um comando simples ou composto seja executado repetidamente, enquanto uma condição de controle seja VERDADEIRA. A forma geral do WHILE é:

```
WHILE <condição> DO <comando>
```

A <condição> deve ser uma expressão lógica. O <comando> pode ser um comando simples ou um comando composto.

Como o teste da <condição> é realizado no início do laço, o <comando> será executado zero ou mais vezes, dependendo da avaliação da <condição>.

EXEMPLO:

```
Program EXEMPLO_DE_WHILE; {escrever os números inteiros de 1 a 100}
```

```
Var
```

```
  N : integer;
```

```
Begin
```

```
  N := 1;
```

```
  while N <= 100 do
```

```
    begin
```

```
      writeln(N);
```

```
      N := N + 1
```

```
    end
```

```
End.
```

Neste exemplo, o comando WRITELN será executado repetidas vezes enquanto a variável N possuir um valor igual ou inferior a 100. O programa terá como saída a seqüência dos números inteiros de 1 a 100.

12.2 REPETIÇÃO COM TESTE NO FINAL (REPEAT-UNTIL)

A estrutura de controle REPEAT permite que um comando simples ou composto seja executado repetidamente até que uma condição de controle seja FALSA. A forma geral do REPEAT é:

```
REPEAT <comando> UNTIL <condição>
```

A <condição> deve ser uma expressão lógica. O <comando> pode ser um comando simples ou um comando composto. Não há a necessidade dos delimitadores BEGIN e END no comando composto em um REPEAT.

Como o teste da <condição> é realizado no final do laço, o <comando> será executado uma ou mais vezes, dependendo da avaliação da <condição>.

EXEMPLO:

```
Program EXEMPLO_DE_REPEAT; {escrever os núm. inteiros de 1 a 100}
```

```
Var  
  N : Integer;  
Begin  
  N := 1;  
  repeat  
    writeln(N);  
    N := N + 1  
  until N > 100  
End.
```

O exemplo anterior é equivalente ao exemplo do WHILE, onde o comando WRITELN será executado repetidas vezes até que a variável N possua um valor superior a 100.

A estrutura REPEAT também é bastante utilizada para repetirmos um programa diversas vezes, até que o usuário deseje sair do mesmo.

EXEMPLO:

```
Program EXEMPLO_DE_REPEAT_2;  
  {calcula a média de 2 números dados repetidas vezes}  
Uses  
  CRT;  
Var  
  N1,N2,MEDIA : real;  
  RESP : char;  
Begin  
  clrScr;  
  repeat  
    write('Digite os dois números: ');  
    readln (N1,N2);  
    MEDIA := (N1+N2)/2;  
    writeln (MEDIA);  
    write ('Deseja repetir o programa (s/n)? ');  
    RESP := readkey;  
  until (RESP='N') or (RESP='n')  
End.
```

12.3 REPETIÇÃO AUTOMÁTICA (FOR)

A estrutura de controle FOR permite que um comando simples ou composto seja repetido um número específico de vezes. A sua forma geral é:

```
FOR <var> := <vi> TO <vf> DO <comando>
```

Onde <var> é uma variável de controle, do tipo inteira, que assumirá inicialmente o valor inicial <vi> e será **incrementada** do valor 1 após cada repetição do laço. A repetição será finalizada quando o conteúdo de <var> for superior ao valor final <vf>. O <comando> também pode ser simples ou composto.

Uma outra forma da estrutura FOR é a seguinte:

```
FOR <var> := <vi> DOWNTO <vf> DO <comando>
```

Neste caso, a variável de controle <var> será **decrementada** do valor 1 após cada repetição do laço e a repetição será finalizada quando o conteúdo de <var> for inferior ao valor final <vf>.

EXEMPLO:

```
Program EXEMPLO_DE_FOR;
{escreve os números inteiros de 1 a 100}
Var
  N : integer;
Begin
  for N := 1 to 100 do
    writeln(N)
  End.
```

Observe, no exemplo acima, que não foi necessário utilizar um comando para atribuir um valor inicial a variável N, nem também um outro comando para incrementá-la com o valor 1. Isto é feito automaticamente pela estrutura FOR.

A estrutura de repetição FOR é especialmente indicada para quando o número de repetições é previamente conhecido. Caso contrário, devemos utilizar o WHILE ou o REPEAT, dependendo do caso.

EXERCÍCIOS RESOLVIDOS

01. Escreva um programa que leia um conjunto 100 números inteiros e exiba o valor médio dos mesmos.

```
Program resolvidofor_1;
Var
  N,SOMA,CONT : integer;
Begin
  SOMA := 0;
  for CONT := 1 to 100 do
    begin
      readln(N)
      SOMA := SOMA + N;
    end;
  writeln(SOMA);
End.
```

02. Escreva um programa que leia um conjunto de números inteiros e exiba o valor médio dos mesmos.

Obs: A condição de saída do laço será a leitura do valor 0.

```
Program resolvidofor_2;
Var
```

N,CONT,SOMA,MEDIA : integer;

Begin

SOMA := 0;

CONT := 0;

readln(N);

while N <> 0 do

begin

SOMA := SOMA + N;

CONT := CONT + 1;

readln(N)

end;

MEDIA := SOMA div CONT;

writeln(MEDIA);

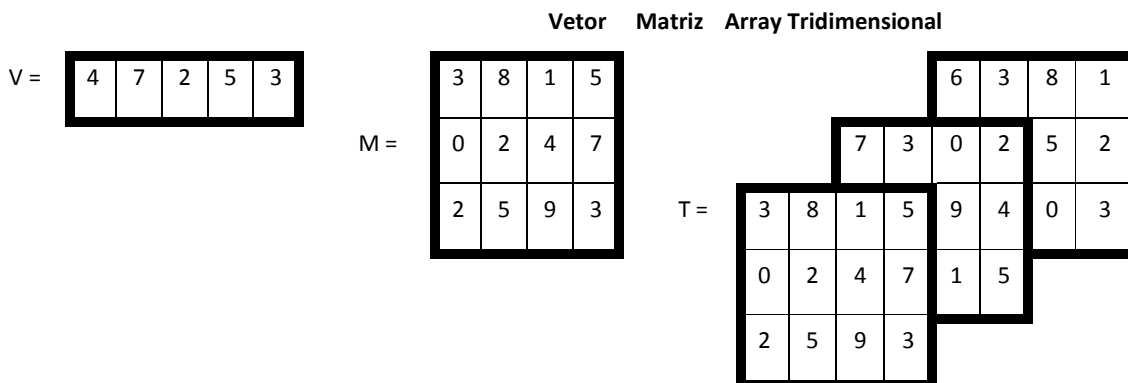
End.

13. ARRAYS

As *variáveis compostas homogêneas*, mais conhecidas como **arrays**, correspondem a conjuntos de elementos de um mesmo tipo, representados por um único nome.

Os arrays podem variar quanto a sua dimensão, isto é, a quantidade de índices necessária para a individualização de cada elemento do conjunto. O array unidimensional também é conhecido por **vetor**, enquanto o array bidimensional é denominado de **matriz**.

EXEMPLOS:



Cada elemento dos arrays podem ser referenciados através de índices. Exemplos:

V[1] = 4 M[1,1] = 3 T[1,1,1] = 3

V[2] = 7 M[2,3] = 4 T[2,3,2] = 9

V[5] = 3 M[3,1] = 2 T[1,2,3] = 3

13.1 VETORES

Vetores são arrays que necessitam de apenas um índice para individualizar um elemento do conjunto.

Declaração:

Para definirmos uma variável do tipo vetor, utilizamos a seguinte sintaxe:

lista-de-identificadores : ARRAY[índice-inicial..índice-final] OF tipo

onde:

lista-de-identificadores são os nomes das variáveis que se deseja declarar;
índice-inicial é o limite inferior do intervalo de variação do índice;
índice-final é o limite superior do intervalo de variação do índice;
tipo é o tipo dos componentes da variável

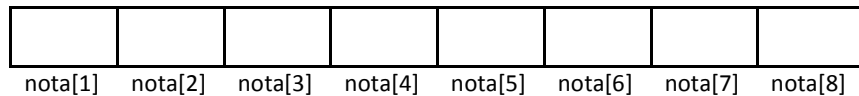
O *índice-inicial* e o *índice-final* devem ser do mesmo tipo escalar (inteiro, caracter ou booleano).

EXEMPLO:

Declarar uma variável composta de 8 elementos numéricos de nome NOTA.

```
var    NOTA : array[1..8] of real;
```

fará com que passe a existir um conjunto de 8 elementos do tipo real, individualizáveis pelos índices 1, 2, 3, ..., 8.



Outros exemplos de declarações de vetores:

```
var    IDADE : array[1..20] of integer;
       NOME : array[1..30] of string;
       QUANT : array['A'..'Z'] of integer;
       RECEITA,DESPESA : array[90..96] of real;
       VETOR: array[-5..5] of char;
```

Para processarmos individualmente todos os componentes de um vetor, é aconselhável o uso da estrutura FOR, para que possamos variar o índice do vetor.

13.1.1 EXERCÍCIOS RESOLVIDOS DE ARRAYS UNIDIMENSIONAIS (VETOR)

01. Dado um vetor A definido com A : array[1..100] of integer;

- a) preenchê-lo com o valor 30;
for I:=1 to 100 do
A[I]:=30;
- b) preenchê-lo com os números inteiros 1,2,3,...,100.
for I:=1 to 100 do
A[I]:=I;

02. Fazer um programa que leia um vetor A contendo 30 números inteiros, calcule e exiba:

- a) o maior elemento;
- b) a posição (índice) do maior elemento.

Program R5_02;

const

N = 30; {número de elementos do vetor}

var

A : array[1..N] of integer;

I,POS,MAIOR : integer;

begin

{leitura do vetor}

for I:=1 to N do

 readln(A[I]);

{inicialização das variáveis}

MAIOR := A[1];

POS := 1;

{comparação dos elementos com a variável MAIOR}

for I:=2 to N do

 if A[I] > MAIOR then

 begin

 MAIOR := A[I];

 POS := I;

 end;

{exibição dos resultados}

 writeln(MAIOR,POS);

end.

03. Fazer um programa para ler 20 números, calcular a média dos mesmos e exibir os números que estiverem acima da média.

Program R5_03;

const

N = 20; {número de elementos do vetor}

var

NUM : array[1..N] of real;

I,SOMA : integer;

MEDIA : real;

begin

{inicialização da variável SOMA}

SOMA := 0;

{leitura e soma dos números}

for I:=1 to N do

 begin

```
    readln(NUM[I]);
    SOMA := SOMA + NUM[I];
end;
{cálculo da média}
MEDIA := SOMA / N;
{exibição dos números maiores que a média}
for I:=1 to N do
    if NUM[I] > MEDIA then
        writeln(NUM[I]);
end.
```

04. Fazer um programa que:

- a) leia um vetor VET de 100 números inteiros;
- b) leia um valor inteiro NUM;
- c) determine e exiba a posição de NUM dentro de VET. Caso NUM não seja encontrado dentro de VET, exiba o valor 0 (zero).

Program R5_04;

const

N = 100; {número de elementos do vetor}

var

VET : array[1..N] of integer;

NUM,I,POS : integer;

begin

{leitura dos dados}

for I:=1 to N do

readln(VET[I]);

readln(NUM);

{determinação da posição}

POS := 0;

for I:=1 to N do

if VET[I] = NUM then

POS := I;

{exibição do resultado}

writeln(POS);

end.

13.2 MATRIZES

Matrizes são arrays que necessitam de dois índices para individualizar um elemento do conjunto. O primeiro índice representa as linhas e o segundo as colunas.

Declaração:

Para definirmos uma variável do tipo matriz, utilizamos a seguinte sintaxe:

```
lista-de-identificadores : ARRAY [índice1-inicial..índice1-final,
índice2-inicial..índice2-final] OF tipo
```

onde:

lista-de-identificadores são os nomes das variáveis que se deseja declarar;
índice1-inicial é o limite inferior do intervalo de variação do primeiro índice;
índice1-final é o limite superior do intervalo de variação do primeiro índice;
índice2-inicial é o limite inferior do intervalo de variação do segundo índice;
índice2-final é o limite superior do intervalo de variação do segundo índice;
tipo é o tipo dos componentes da variável

o *índice1-inicial* e o *índice1-final* devem ser do mesmo tipo escalar (inteiro, caracter ou booleano). O *índice2-inicial* também deve ser do mesmo tipo escalar do *índice2-final*.

EXEMPLO:

Declarar uma matriz M, de 4 linhas por 3 colunas, constituída de elementos numéricos inteiros.

```
var M : array[1..4,1..3] of integer;
```

fará com que passe a existir uma estrutura de dados agrupada denominada M, com 4x3=12 elementos inteiros, endereçáveis por um par de índices, com o primeiro indicando a linha e o outro, a coluna.

M =

m ₁₁	m ₁₂	m ₁₃
m ₂₁	m ₂₂	m ₂₃
m ₃₁	m ₃₂	m ₃₃
m ₄₁	m ₄₂	m ₄₃

Outros exemplos de declarações de matrizes:

```
var
```

```
M1 : array[1..4,80..90] of real;
```

```
M2 : array['A'..'E',0..10] of string;
```

```
M3 : array[-3..3,1..3] of char;
```

13.2.1 EXERCÍCIOS RESOLVIDOS DE ARRAYS BIDIMENSIONAIS (MATRIZ)

01. Fazer um programa para ler uma matriz 3 x 5 de números inteiros e escrevê-la após ter multiplicado cada elemento por 2.

```
Program R5_05;
```

```
const
```

```
NL = 3; {número de linhas}
```

```
NC = 5; {número de colunas}
```

```
K = 2; {fator para multiplicação}
```

```
var
```

```
M : array[1..NL,1..NC] of integer;
```

```
I,J : integer;
```

```
begin
```

```
{leitura da matriz}
```

```
for I:=1 to NL do
```

```
  for J:=1 to NC do
```

```
    begin
```

```
      write('Elemento da linha ',I,' coluna ',J,' : ');
```

```
      readln(M[I,J]);
```

```
    end;
```

```
{cálculo da multiplicação}
```

```
for I:=1 to NL do
```

```
  for J:=1 to NC do
```

```
    M[I,J] := M[I,J] * K;
```

```
{exibição da matriz resultante}
```

```
writeln('Resultado:');
```

```
for I:=1 to NL do
```

```
  begin
```

```
    for J:=1 to NC do
```

```
      write(M[I,J], ' ');
```

```
    writeln;
```

```
  end;
```

```
end.
```

02. Dada uma matriz de 4 x 5 elementos inteiros, calcular a soma de cada linha, de cada coluna e de todos os seus elementos.

Obs: utilize um vetor para armazenar o resultado da soma de cada linha e outro para a soma de cada coluna.

Program resolvido;

```
const
```

```
NL = 4; {número de linhas}
```

```
NC = 5; {número de colunas}
```

```
var
```

```
M : array[1..NL,1..NC] of integer;
```

```
L : array[1..NL] of integer;
```

```
C : array[1..NC] of integer;
```

```
I,J,SOMA : integer;

begin

{leitura da matriz}

  for I:=1 to NL do
    for J:=1 to NC do
      begin
        write('Elemento da linha ',I,' coluna ',J,' : ');
        readln(M[I,J]);
      end;
    }

{cálculo da soma dos elementos de cada linha}

  for I:=1 to NL do
    begin
      L[I] := 0;
      for J:=1 to NC do
        L[I] := L[I] + M[I,J];
      end;
    }

{cálculo da soma dos elementos de cada coluna}

  for J:=1 to NC do
    begin
      C[J] := 0;
      for I:=1 to NL do
        C[J] := C[J] + M[I,J];
      end;
    }

{cálculo da soma de todos os elementos da matriz}

  SOMA := 0;
  for I:=1 to NL do
    for J:=1 to NC do
      SOMA := SOMA + M[I,J];
    }

{exibição dos resultados}

  for I:=1 to NL do
    writeln('Soma da Linha ',I,' : ',L[I]);
  for J:=1 to NC do
    writeln('Soma da Coluna ',J,' : ',C[J]);
  writeln('Soma da Matriz: ',SOMA);

end.
```

14. RECURSIVIDADE

A recursividade é uma característica que alguns problemas apresentam: a de serem definidos em termos deles mesmos. Todo problema que se comporta assim é dito ser recursivo.

A recursão é uma técnica apropriada se o problema a ser resolvido tem as seguintes características:

- 1) a resolução dos casos maiores do problema envolve a resolução de um ou mais casos menores;
- 2) os menores casos possíveis do problema podem ser resolvidos diretamente;
- 3) a solução iterativa do problema (usando enquanto, para ou repita) é complexa.

Exemplo:

O problema do fatorial, que foi visto anteriormente, é recursivo por definição:

$$N! = N \times (N - 1) \times (N - 2) \times (N - 3) \dots \times 1 \quad (\text{Equação 1})$$

Existe um caso especial: 0! é igual a 1, por definição.

A partir da equação 1, podemos concluir que o fatorial de N está expresso em termos do fatorial de N-1.

$$N! = N \times (N - 1) \times (N - 2) \times (N - 3) \dots \times 1$$

$$\underbrace{\hspace{15em}}_{(N - 1)!}$$

Resumindo:

$$N! = N \times (N - 1)! \quad (\text{Equação 2})$$

A equação 2 é válida para todos os números inteiros com exceção do 0 (zero), sendo, portanto, necessário um tratamento especial.

O programa recursivo em Pascal para o cálculo do fatorial de N ficaria assim:

```

Program FATORIAL;

var

  N,F : longint;

{função recursiva que retorna o fatorial de N}

function FAT(N:longint):longint;

begin
  if N = 0 then
    FAT := 1
  else
    FAT := N * FAT (N - 1)
end;

{programa principal}

begin
  write('Digite um número: ');

  readln(N);

  F := FAT(N);

```

```
writeln('O fatorial de ',N,' é ',F)
```

```
end.
```

Note que para cada chamada da função recursiva FAT deve ser criado um novo nível de ativação, guardado em uma pilha, onde para cada um destes níveis de ativação têm-se um parâmetro de chegada e um valor de chamada para uma nova ativação ou um valor de retorno.

Criaremos uma tabela para melhor expor a observação acima e para isso vamos calcular o fatorial de 3:

Número da ativação	Valor de chegada	Rechamada	Valor de retorno
1	3	3 * FAT(3-1)	
2	2	2 * FAT(2-1)	
3	1	1 * FAT(1-1)	
4	0		1
3			1
2			2
1			6

Observe que até ser atendida a condição de retorno ($N = 0$), têm-se apenas reativações de FAT. Quando o valor de chegada é igual a 0 (na ativação 4), começa o processo de retorno, onde cada nível imediatamente superior recebe o resultado do nível inferior. Quando um nível recebe o resultado do seu nível inferior ele o aplica para poder calcular a expressão $FAT := N * FAT(N-1)$ até então indefinida, retornando para o nível de cima o seu valor calculado.

É importante que seja observada a condição de encerramento da recursão para evitar que este processo continue indefinidamente.

P7.30. Escreva um programa que exiba uma tabela de conversão de graus Celsius para Fahrenheit, no intervalo de 1 a 100, variando de 1 em 1, dispostos em 5 colunas na tela.

16. ARQUIVOS E REGISTROS

16.1. REGISTROS

São conjuntos de dados logicamente relacionados, mas de tipos diferentes (inteiro, real, string, etc.).

Exemplo: Numa dada aplicação, podem-se ter os seguintes dados sobre funcionários de uma empresa:

- ⇒ Nome
- ⇒ Endereço
- ⇒ Idade
- ⇒ Salário

Cada conjunto de informações do funcionário pode ser referenciado por um mesmo nome, por exemplo, FICHA. Tais estruturas são conhecidas como registros, e aos elementos dos registros dá-se o nome de campos.

O conceito de registro visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que guardam estreita relação lógica.

16.1.1. DECLARAÇÃO

Criam-se estruturas de dados agrupados na forma de registro através da seguinte declaração:

```
lista-de-identificadores :      RECORD
                                campos
                                END;
```

onde:

lista-de-identificadores são os nomes que estão sendo associados aos registros que se deseja declarar;
campos são declarações de variáveis, separadas por ponto-e-vírgula.

Exemplo: Declarar o registro FICHA especificado anteriormente.

```
Var FICHA : record
            NOME : string[30];
            ENDereco : string[40];
            IDADE : byte;
            SALARIO : real;
end;
```

16.1.2. REFERÊNCIA

A referência ao conteúdo de um dado campo do registro será indicada pela notação:

identificador-do-registro . identificador-do-campo

Exemplo: Sabendo-se que o registro FICHA, em um dado instante, contivesse os valores a seguir:

NOME : Antônio Ajuizado

ENDEREÇO: Rua das Virtudes, s/n

IDADE: 20 anos

SALÁRIO: R\$ 150,00

FICHA.IDADE estaria fazendo referência ao conteúdo do campo IDADE do registro FICHA, isto é, 20.

16.1.3. CONJUNTO DE REGISTROS

Podem-se ter conjunto de registros referenciáveis por um mesmo nome e individualizados por índices, através da utilização de um array de registros.

Exemplo:

```
Var
    TAB : array[1..50] of record
        MATR : integer;
        NOME : string[30];
        MEDIA : real;
    end;
```

EXERCÍCIO RESOLVIDO

R8.01. Considerando o registro de uma mercadoria de uma loja contendo as seguintes informações: código, nome, preço e estoque, fazer um programa que, dado o registro de 50 mercadorias, leia um código exiba o nome, preço e estoque da respectiva mercadoria.

```
Program MERCADORIAS;
Uses Crt;
Const N = 50;
Var    TAB : array[1..N] of record
        COD : string[6];
        NOME : string[15];
        PRECO: real;
        EST : integer;
    end;
    I : integer;
    K : string[6];
    RESP : char;

Begin
    clrscr;
    {Leitura dos dados}
    for I:=1 to N do
        begin
            write('Código: '); readln(TAB[I].COD);
            write('Nome: '); readln(TAB[I].NOME);
```

```
write('Preço: '); readln(TAB[I].PRECO);
write('Estoque: '); readln(TAB[I].EST);
end;
repeat
  {leitura da chave de pesquisa}
  write('entre com o código desejado: ');
  Readln(K);
  {testa em cada registro se o código é igual a chave pesquisada}
  for I:=1 to N do
    if K = TAB[I].COD then
      writeln(TAB[I].NOME, TAB[I].PRECO, TAB[I].EST);
    {verifica se o usuário deseja pesquisar outro código}
    write('Repetir(S/N)?');
    RESP := readkey;
  until upcase(RESP) = 'N';
End.
```

16.1.4. O COMANDO WITH

Este comando permite que os campos de um registro sejam denotados unicamente por seus identificadores, sem a necessidade de serem precedidos pelo identificador do registro. Forma geral:

```
WITH identificador-do-registro DO
  comandos
```

EXERCÍCIO RESOLVIDO

R8.02. Reescrever o programa MERCADORIAS (R8.01), utilizando o comando WITH.

```
Program MERCADORIAS_WITH;
Uses Crt;
Const N = 50;
Var TAB : array[1..N] of record
  COD : string[6];
  NOME : string[15];
  PRECO: real;
  EST : integer;
end;
```

```
I : integer;
K : string[6];
RESP : char;

Begin
  clrscr;
  {Leitura dos dados}
  for I:=1 to N do
    with TAB[I] do
      begin
        write('Código: '); readln(COD);
        write('Nome: '); readln(NOME);
        write('Preço: '); readln(PRECO);
        write('Estoque: '); readln(EST);
      end;
    end;
  repeat
    {leitura da chave de pesquisa}
    write('entre com o código desejado: '); Readln(K);
    {testa em cada registro se o código é igual a chave pesquisada}
    for I:=1 to N do
      with TAB[I] do
        if K = COD then
          writeln(NOME, PRECO, EST);
      end;
    end;
    {verifica se o usuário deseja pesquisar outro código}
    write('Repetir(S/N)?');
    RESP := readkey;
  until upcase(RESP) = 'N';
End.
```

EXERCÍCIOS PROPOSTOS

P8.01. Escreva, na linguagem Pascal, as declarações que definem os seguintes registros:

- a) NOME, ENDEREÇO, CPF, SEXO;
- b) MATRÍCULA, NOTA1, NOTA2, NOTA3, MEDIA.

P8.02. Uma indústria faz a folha mensal de pagamentos de seus 80 empregados baseada no seguinte:

- ⇒ Existe uma tabela com os dados de cada funcionário (matrícula, nome e salário bruto);
- ⇒ Escreva um programa que leia e processe a tabela e emita, para cada funcionário, seu contracheque, cujo formato é dado a seguir:

Matrícula:
Nome:
Salário Bruto:
Dedução INSS:
Salário Líquido:

- ⇒ O desconto do INSS é de 12% do salário bruto.
- ⇒ O salário líquido é a diferença entre o salário bruto e a dedução do INSS.

P8.03. Em certo município, vários proprietários de imóveis estão em atraso com o pagamento do IPTU. Escrever um programa que calcule e escreva o valor da multa a ser paga por estes proprietários, considerando que:

- ⇒ os dados de cada imóvel (identificação, valor do imposto e número de meses em atraso) estão à disposição para leitura;
- ⇒ as multas devem ser calculadas no valor de 1% por mês de atraso.
- ⇒ o último registro lido, que não deve ser considerado, contém identificação do imóvel igual a XXX;
- ⇒ o programa deve exibir: a identificação do imóvel, valor do imposto, número de meses em atraso e multa a ser paga.

P8.04. Escreva um programa que armazene um cadastro de 50 pessoas com os seguintes dados: nome, telefone e data de nascimento (dia, mês, ano) e realize consultas da seguinte forma:

- ⇒ Leia o número de um determinado mês (1 a 12). Obs: a leitura do mês 0 encerra as consultas.
- ⇒ Exiba o nome, o telefone e o dia do aniversário das pessoas daquele respectivo mês.

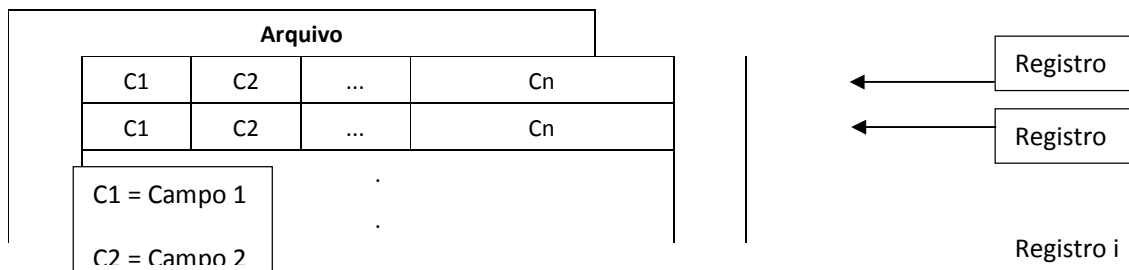
16.2. ARQUIVOS

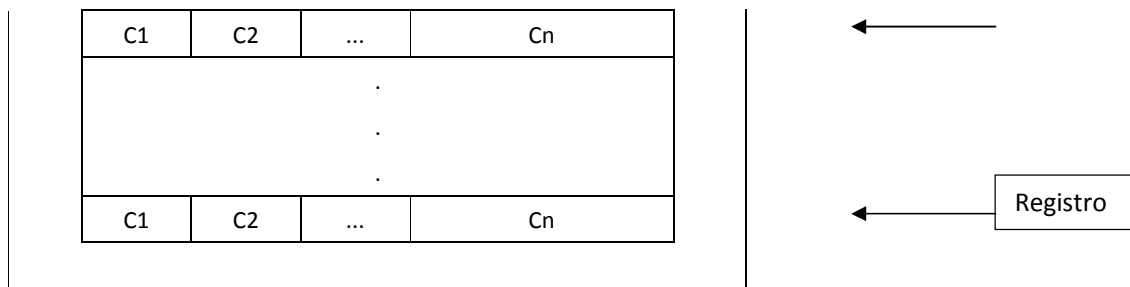
Entende-se por arquivo uma coleção de dados, onde os mesmos possuem alguma relação entre si. Por exemplo, o conjunto de dados sobre os alunos da ASPER. Dentre estes dados sobre os alunos, temos:

- MATRÍCULA (string);
- NOME (string);
- ENDEREÇO (string);
- MEDIA ANUAL (real).

Cada conjunto de dados sobre um determinado aluno recebe o nome de registro. Note-se que um registro é composto por tipos de dados diferentes. Cada dado que compõe o registro (MATRÍCULA, NOME etc) é dito ser um campo.

A figura a seguir, ilustra a relação existente entre campo, registro e arquivo:





Um arquivo necessita de um meio de armazenamento que sirva de depósito (suporte) para seus dados. Os suportes mais comuns para os arquivos são os discos flexíveis, os discos rígidos (winchesters) e as fitas magnéticas.

Esses arquivos podem ser de 3 tipos: seqüencial, indexado e relativo. Essa diferenciação em tipos, está associada a algumas restrições quanto ao acesso aos dados guardados nesses arquivos. Em nosso caso, trataremos apenas com arquivos de organização seqüencial. O arquivo seqüencial é aquele cujo acesso aos seus dados só pode ser feito de forma seqüencial, isto é, um registro de ordem N está localizado após um outro de ordem N-1 e antecede um de ordem N+1.

16.2.1. DECLARAÇÃO DE ARQUIVOS

Um arquivo antes de ser usado precisa ser declarado. Essa declaração cria uma variável que poderá ser associada aos dados gravados no dispositivo físico (disco, fita etc). Para fazermos a declaração de um arquivo usamos o seguinte formato geral:

lista-de-identificadores : FILE OF tipo;

onde:

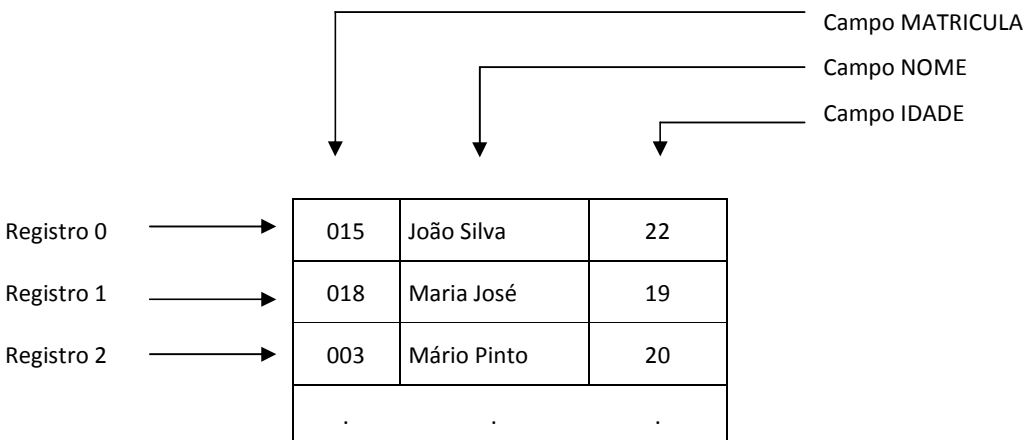
- lista-de-identificadores** são os nomes das variáveis que serão associadas aos arquivos que se deseja declarar;
- tipo** representa o tipo da variável correspondente aos registros do arquivo.

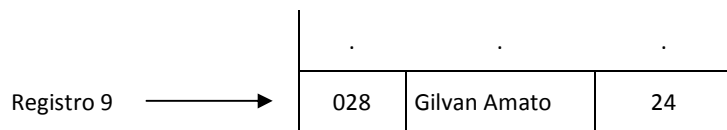
Exemplo:

```

Type
  REG = record
    MATRICULA : string[8];
    NOME : string[30];
    IDADE : byte;
  end;
Var
  ALUNOS : file of REG;
    
```

Estrutura do arquivo ALUNOS, após a sua declaração e utilização, supondo que existam 10 (dez) registros:





No exemplo anterior, podemos observar que o Pascal numera os registros começando pelo número 0 (zero), portanto o último registro terá sempre o número N-1 (sendo N a quantidade de registros do arquivo).

16.2.2. UTILIZAÇÃO DE ARQUIVOS

Para ser utilizado, um arquivo precisa, além de ser declarado, de uma série de tratamentos específicos, que são realizados pelos comandos :

ASSIGN - Este procedimento permite que associemos o nome externo de um arquivo a uma variável do tipo FILE. O nome externo é aquele utilizado pelo sistema operacional, portanto deve ser válido para o mesmo. São possíveis todas as formas usadas no PATH, e quando a unidade ou subdiretórios forem omitidos, estes assumiram o default. Após o uso do ASSIGN, este continuará valendo até que seja dado um novo ASSIGN. O tamanho máximo do nome do arquivo é de 79 caracteres. Este procedimento nunca deve ser usado em um arquivo já aberto. Sua sintaxe:

```
ASSIGN (VAR nome-pascal:FILE; nome-externo:STRING);
```

Exemplo:

```
ASSIGN(CADASTRO,'A:\CADASTRO.ARQ');
```

RESET - Este procedimento permite-nos abrir um arquivo já existente. No caso do uso deste, para a tentativa de abertura de um arquivo não existente, haverá um erro de execução. Para que o procedimento tenha sucesso, é necessário que antes de executá-lo tenhamos utilizado o procedimento ASSIGN. Após o uso do RESET, o ponteiro de registros do arquivo será posicionado no registro 0 (primeiro registro). Sua sintaxe:

```
RESET (VAR arquivo:FILE);
```

Exemplo:

```
RESET(CADASTRO);
```

REWRITE - Este procedimento permite criar e abrir um novo arquivo. Caso o arquivo já exista, terá seu conteúdo eliminado e será gerado um novo arquivo. Antes de executarmos este procedimento, devemos usar o ASSIGN. Sua sintaxe:

```
REWRITE (VAR arquivo:FILE);
```

Exemplo:

```
REWRITE(CADASTRO);
```

CLOSE - Este procedimento permite que se feche um arquivo anteriormente aberto, só sendo permitido o fechamento de um arquivo por vez. Sua sintaxe:

```
CLOSE (VAR arquivo:FILE);
```

Exemplo:

```
CLOSE(CADASTRO);
```

Nos comandos apresentados, vemos uma deficiência: enquanto o RESET apenas nos permite a abertura de arquivos já existentes, o REWRITE apenas abre arquivos novos ou destrói um possível conteúdo anterior. Como poderemos então resolver essa situação? Bem, existe um modo relativamente simples, que consiste no uso de uma diretiva de compilação para checagem de erros de entrada e/ou saída, {\$I}. Esta diretiva retorna um código de erro em uma função do Turbo chamada IORESULT. Vejamos um exemplo prático da utilização da diretiva {\$I}:

```
Program TESTA_ABERTURA;
```

```
Uses crt;
```

```
Type
```

```
REG = record
```

```
COD : string[6];  
DESC : string[20];  
  
end;  
  
Var  
  
ARQ : file of REG;  
  
NOMEARQ : string;  
  
Begin  
  
clrscr;  
  
write('Entre com o nome do arquivo');  
  
readln(NOMEARQ);  
  
assign(ARQ,NOMEARQ);  
  
{!-}  
  
reset(ARQ);  
  
{!+}  
  
if ioresult = 0 then  
  
    writeln('Arquivo aberto sem problemas')  
  
else  
  
    begin  
  
        rewrite(ARQ);  
  
        writeln('Arquivo inexistente. Novo arquivo criado');  
  
    end;  
  
close(ARQ);  
  
writeln('Arquivo fechado');  
  
readkey;  
  
End.
```

Porém, para manipularmos um arquivo não basta apenas abri-lo e fechá-lo. Temos, na maioria das vezes, que ler uma informação contida nele, outras vezes registrar informações novas, ou ainda fazer alterações nas informações já existentes. Vejamos então os comandos que nos permitem tais tarefas:

WRITE - Este procedimento, além de ser usado para exibir dados no vídeo, pode ser usado para gravar informações em um arquivo. Após a execução deste procedimento, o ponteiro de registros do arquivo será deslocado um registro para a frente. Sua sintaxe:

```
WRITE (arquivo:FILE; registro:RECORD);
```

Exemplo:

```
WRITE(CADASTRO,REGISTRO);
```

EXERCÍCIO RESOLVIDO

R8.03. Escreva um programa crie um novo arquivo chamado AGENDA.DAT contendo os campos NOME e FONE, e grave no mesmo 10 registros lidos pelo teclado.

```
Program CRIA_AGENDA;

Uses

  crt;

Type

  REGISTRO = record

    NOME : string[20];

    FONE : string[8];

  end;

Const N = 10;

Var

  AGENDA : file of REGISTRO;

  REG : REGISTRO;

  I : integer;

Begin

  clrscr;

  assign(AGENDA,'AGENDA.DAT');

  rewrite(AGENDA);

  for i:=1 to N do

    begin

      with REG do

        begin

          write('Nome: ');

          readln(NOME);

          write('Fone: ');

          readln(FONE);

          end;

        write(AGENDA,REG);

        end;

    close(AGENDA);

    readkey;

  End.
```

READ - Como já vimos anteriormente, este procedimento permite que atribuamos a uma variável um valor obtido por um dispositivo de entrada. Esse dispositivo pode ser também um arquivo. Após a execução deste procedimento, o ponteiro de registros do arquivo será deslocado um registro para a frente. Sua sintaxe:

```
READ (arquivo:FILE; registro:RECORD);
```

Exemplo:

```
READ(CADASTRO,REGISTRO);
```

EXERCÍCIO RESOLVIDO

R8.04. Escreva um programa que abra o arquivo AGENDA.DAT criado no exercício anterior e exiba os 10 registros na tela do computador.

```
Program EXIBE_10;
```

```
Uses crt;
```

```
Type
```

```
  REGISTRO = record
```

```
    NOME : string[20];
```

```
    FONE : string[8];
```

```
  end;
```

```
Const N = 10;
```

```
Var
```

```
  AGENDA : file of REGISTRO;
```

```
  REG : REGISTRO;
```

```
  I : integer;
```

```
Begin
```

```
  clrscr;
```

```
  assign(AGENDA,'AGENDA.DAT');
```

```
  reset(AGENDA);
```

```
  for i:=1 to N do
```

```
    begin
```

```
      read(AGENDA,REG);
```

```
      with REG do
```

```
        writeln(NOME,' ',FONE);
```

```
      end;
```

```
  close(AGENDA);
```

```
  readkey;
```

```
End.
```

No exercício anterior, não tivemos problema algum ao executar a leitura no arquivo, pois este havia sido gravado por nós mesmos e com uma quantidade de registros predefinidos. Porém, na maioria das vezes, não temos quantidade de registros contidos num arquivo e, para esta situação, devemos saber quando chegarmos ao final de um arquivo. O Turbo Pascal nos fornece tal função:

EOF - Esta função nos retorna o valor TRUE quando for encontrada a marca de fim de arquivo. Sua sintaxe:
EOF (VAR arquivo:FILE) : boolean;

Exemplo:

```
FIM := EOF(CADASTRO);
```

EXERCÍCIO RESOLVIDO

R8.05. Escreva um programa que abra o arquivo AGENDA.DAT criado anteriormente e exiba todos os seus registros na tela do computador (supondo não ser conhecida a quantidade de registros).

```
Program EXIBE_TODOS;
```

```
Uses
```

```
  crt;
```

```
Type
```

```
  REGISTRO = record
```

```
    NOME : string[20];
```

```
    FONE : string[8];
```

```
  end;
```

```
Var
```

```
  AGENDA : file of REGISTRO;
```

```
  REG : REGISTRO;
```

```
Begin
```

```
  clrscr;
```

```
  assign(AGENDA,'AGENDA.DAT');
```

```
  reset(AGENDA);
```

```
  while not eof(AGENDA) do
```

```
    begin
```

```
      read(AGENDA,REG);
```

```
      with REG do
```

```
        writeln(NOME,' ',FONE);
```

```
      end;
```

```
  close(AGENDA);
```

```
  readkey;
```

```
End.
```

Porém, para a manutenção de alguns registros de um determinado arquivo, temos que saber qual a posição deste registro no arquivo e também qual a posição do último registro.

Quando estamos manipulando um arquivo, existe a todo momento um ponteiro que fica deslocando-se de acordo com as leituras e gravações, ou seja, a cada leitura ou gravação este ponteiro avança uma posição. Mas existem algumas funções e procedimentos que permitem a manipulação deste ponteiro. São elas:

SEEK - Este procedimento permite que movamos o ponteiro do arquivo para uma posição preestabelecida. Só pode ser usado em arquivos previamente abertos. Sua sintaxe:

```
SEEK (VAR arquivo:FILE; posição:LONGINT);
```

Exemplo:

```
SEEK(CADASTRO,5); {posiciona o ponteiro no sexto registro do arquivo}
```

FILEPOS - Esta função nos retorna a posição atual do ponteiro do arquivo. Sua sintaxe:

```
FILEPOS (VAR arquivo:FILE) : LONGINT;
```

Exemplo:

```
PONTEIRO := FILEPOS(CADASTRO);
```

FILESIZE - Esta função nos retorna a quantidade de registros do arquivo. Sua sintaxe:

```
FILESIZE (VAR arquivo:FILE) : LONGINT;
```

Exemplo:

```
TAMANHO := FILESIZE(CADASTRO);
```

EXERCÍCIO RESOLVIDO

R8.06. Escreva um programa que faça a inclusão de registros em um arquivo chamado ALUNOS. Se o arquivo não existir, o programa deve criar um arquivo vazio e proceder a inclusão. A leitura da matrícula 0 (zero) indica fim dos dados.

“Lay-out” do arquivo: MATRICULA
 NOME
 MEDIA

Explicação da solução:

Deve-se mover o ponteiro para a posição posterior ao último registro e então efetuar a gravação dos dados lidos pelo teclado.

```
Program INCLUSAO;
```

```
Uses crt;
```

```
Type
```

```
  REGISTRO = record
```

```
    MATR : integer;
```

```
    NOME : string[35];
```

```
    MEDIA: real;
```

```
  end;
```

```
Var
```

```
  ARQ : file of REGISTRO;
```

```
  REG : REGISTRO;
```

```
Begin
```

```
  clrscr;
```

```
  assign(ARQ,'ALUNOS');
```

```
  {$I-}
```

```
reset(ARQ);  
{I+}  
if ioresult > 0 then  
    rewrite(ARQ);  
seek(ARQ, filesize(ARQ));  
write('Matricula: ');  
readln(REG.MATR);  
while REG.MATR <> 0 do  
    begin  
        write('Nome: ');  
        readln(REG.NOME);  
        write('Média: ');  
        readln(REG.MEDIA);  
        write(ARQ, REG);  
        write('Matricula: ');  
        readln(REG.MATR);  
    end;  
close(ARQ);  
End.
```

R8.07. Escreva um programa que altera a média de um aluno no arquivo ALUNOS.

Explicação da solução:

Deve-se solicitar a digitação da matrícula do aluno que se deseja alterar e percorrer todo o arquivo comparando a matrícula de cada aluno com a matrícula digitada. Ao encontrar, exibe-se os dados atuais e solicita-se a digitação da nova média.

Program ALTERACAO;

Uses crt;

Type

REGISTRO = record

 MATR : integer;

 NOME : string[35];

 MEDIA: real;

end;

var

ARQ : file of REGISTRO;

REG : REGISTRO;

K : integer;

RESP : char;

Begin

assign(ARQ,'ALUNOS');

reset(ARQ);

repeat

clrscr;

write('Matricula: ');

readln(K);

seek(ARQ,0);

while not eof(ARQ) do

begin

read(ARQ,REG);

with REG do

if MATR = K then

begin

writeln('Nome: ',NOME);

writeln('Média Atual: ',MEDIA:1:1);

write('Nova Média: ');

readln(MEDIA);

seek(ARQ,filepos(ARQ)-1);

write(ARQ,REG);

end;

end;

write('Repetir(S/N)? ');

RESP :=readkey;

until upcase(RESP) = 'N'

close(ARQ);

End.

R8.08. Escreva um programa que abre o arquivo ALUNOS e exibe todos os alunos com média superior ou igual a 7.0:

Explicação da solução:

Deve-se acessar todos os registros e selecionar aqueles cuja média seja maior ou igual a 7.0.

Program EXIBICAO;

Uses crt;

Type

REGISTRO = record

 MATR : integer;

 NOME : string[35];

 MEDIA: real;

end;

Var

 ARQ : file of REGISTRO;

 REG : REGISTRO;

Begin

 clrscr;

 assign(ARQ,'ALUNOS');

 reset(ARQ);

 while not eof(ARQ) do

 begin

 read(ARQ,REG);

 with REG do

 if MEDIA >= 7.0 then

 writeln(MATR,' ',NOME,' ',MEDIA:1:1);

 end;

 close(ARQ);

 readkey;

End.

R8.09. Escreva um programa que nos permita excluir registros do arquivo ALUNOS.

Explicação da solução:

Para que num arquivo se possa excluir registros é necessário que o mesmo possua um campo que funcione como um "flag", ou seja, um indicador de exclusão. Nesse campo, que normalmente é do tipo lógico (boolean), todos os registros estarão inicialmente com valor FALSE, indicando que nenhum registro deverá ser deletado. Para efetuarmos a exclusão, passaremos por duas etapas:

1ª Etapa: EXCLUSÃO LÓGICA – Ao especificarmos que um determinado registro deva ser deletado, devemos alterar o conteúdo do campo indicador de exclusão para o valor TRUE. Se quisermos excluir mais registros, faremos a mesma operação.

2ª Etapa: EXCLUSÃO FÍSICA – Após definirmos todos os registros que deverão ser deletados, devemos copiar todos os registros não marcados para deleção para um outro arquivo temporário. Feito isto, apagamos o arquivo original e renomeamos o arquivo temporário com o nome do arquivo original.

Program EXCLUSAO;

Uses crt;

Const

 NOMEARQ = 'ALUNOS';

Type

REGISTRO = record

 MATR : integer;

 NOME : string[35];

 MEDIA: real;

 DEL : boolean; {indicador de exclusão}

end;

Var

 ARQ,TMP : file of REGISTRO;

 REG : REGISTRO;

 K : integer;

 EXC,MAIS : char;

Begin

{EXCLUSÃO LÓGICA}

assign(ARQ,NOMEARQ);

reset(ARQ);

repeat

 clrscr;

 write('Matrícula: ');

 readln(K);

 seek(ARQ,0);

 while not eof(ARQ) do

 begin

 read(ARQ,REG);

 with REG do

 if MATR = K then

 begin

 writeln('Nome: ',NOME);

 write('Excluir (S/N)? ');

 EXC := upcase(readkey);

 writeln;

 if EXC = 'S' then

 DEL := true;

 seek(ARQ,FILEPOS(ARQ)-1);

 write(ARQ,REG);

 end;

 end;

 end;

 end;

```
end;
write('Excluir outro (S/N)? ');
MAIS := upcase(readkey);
until MAIS = 'N';
{EXCLUSÃO FÍSICA}
assign(TMP,'TMP');
rewrite(TMP);
seek(TMP,0);
while not eof(ARQ) do
begin
read(ARQ,REG);
if not REG.DEL then
write(TMP,REG);
end;
close(ARQ);
close(TMP);
erase(ARQ); {deleta o arquivo original}
rename(TMP,NOMEARQ); {renomeia o arquivo temporário}
End.
```

EXERCÍCIOS PROPOSTOS

- P8.05. Elabore um programa que copie o arquivo ALUNOS (veja exemplos anteriores) em um outro chamado ALUNOS2.
- P8.06. Elabore um programa que concatena (junta) dois arquivos com o mesmo "lay-out" em um terceiro.
- P8.07. Escreva um programa que, a partir de um arquivo CAD já existente cujo "lay-out" é dado abaixo, determina quantas pessoas são do sexo feminino e quantas são do sexo masculino.

"Layout" do arquivo: MATRICULA
 NOME
 ENDERECO
 SEXO
 SALARIO

- P8.08. Construa um programa que seleciona de um arquivo, gravando em outro, todos os registros cujo campo salário é maior que R\$ 500,00.

"Layout" do arquivo: MATRICULA
 NOME
 SALARIO

