André Carlos Silva

Turbo Pascal para Engenheiros

Turbo Pascal para Engenheiros

André Carlos Silva

Turbo Pascal para Engenheiros

COLEÇÃO
LAVOR

Sumário

À minha esposa e filhos, pela paciência, carinho e amor.

Apresentação

A cada dia, o mundo da computação funde-se mais com a realidade do engenheiro. É praticamente impossível, atualmente, separar essas duas ciências. Não obstante, o conhecimento apenas de ferramentas computacionais não é suficiente para a resolução dos problemas vivenciados no ambiente de trabalho pelo engenheiro. O estudo de uma linguagem de programação, como o Pascal, permite ao engenheiro desenvolver as suas próprias ferramentas computacionais.

Nos capítulos iniciais desta obra são apresentados os conceitos de variáveis, estruturas tomadoras de decisões e de repetição. São apresentadas também técnicas avançadas, tais como: modularização de programas, tipos estruturados de dados, manipulação de arquivos em disco e alocação dinâmica de memória.

Portanto, este livro é uma ferramenta didática pensada para estudantes e profissionais da área de ciências exatas e engenharia, contendo exercícios resolvidos e propostos (quase duzentos no total) sobre problemas comuns a essas áreas. Por esse motivo este livro foi escrito, para auxiliar estudantes e engenheiros a percorrerem o chamado "caminho das pedras" da programação de computadores.

Introdução

A maior revolução já ocorrida na engenharia foi, indubitavelmente, a introdução dos computadores como ferramentas para a resolução de problemas complexos. Se, no passado, um engenheiro civil e/ou um arquiteto ficavam horas desenhando seus projetos em papel vegetal, usando caneta nanquim, hoje o mesmo projeto é realizado em programas CAD (do inglês *Computer-Aided Design,* ou simplesmente Desenho Assistido por Computador) e impresso usando *plotters*, de maneira limpa, rápida e precisa. No planejamento de empreendimentos em mineração e geologia foram adotadas as linguagens de modelamento virtual para representar os modelos de maciços rochosos em três dimensões, o que antes não era possível ser feito em papel, senão usando maquetes. Em suma, para todas as engenharias foram – e ainda estão sendo – desenvolvidas ferramentas para facilitar e acelerar a rotina dos profissionais.

Contudo, todos os dias surgem novos problemas a serem resolvidos e, em muitos casos, novas ferramentas devem ser desenvolvidas; ou, ainda, as pré-existentes devem ser atualizadas. Em ambos os casos, conhecimentos de programação de computadores são requeridos para que o engenheiro possa realizar tal tarefa.

Ao mesmo tempo, tem-se em vista que vários livros sobre programação de computadores dedicam-se também ao ensino de algoritmos estruturados. De acordo com Ziviani (1999), os algoritmos fazem parte do dia a dia das pessoas. As instruções para o uso de medicamentos, as indicações de como montar um aparelho qualquer, uma receita de culinária, estes são alguns exemplos de algoritmos. Um algoritmo, portanto, pode ser visto como uma sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema. Segundo Dijkstra (1971), um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações.

Aqui, optou-se por abordar os conceitos de programação de computadores sem focar exclusivamente em algoritmos – entretanto, ao final deste livro, existe uma breve explanação sobre eles. A importância que se deve dar ao domínio deste conteúdo é evidenciada por Ziviani (1999), ao afirmar que programar é basicamente estruturar dados e construir algoritmos.

Neste livro, são abordados os conceitos fundamentais da linguagem de programação Pascal usando a IDE (do inglês *Integrated Development Environment*), Turbo Pascal Versão 7.0, da Borland, e de lógica de programação de computadores, capacitando o leitor ao desenvolvimento de suas próprias ferramentas computacionais. Outras IDEs, como o Dev-Pascal, Lazaurs ou LW-Pascal, podem ser utilizadas, sem nenhum prejuízo ao leitor. Tais IDEs são abordadas no último capítulo deste livro.

1. Introdução ao Turbo Pascal

A linguagem de programação Pascal foi criada no início da década de 1970 pelo prof. Niklaus Wirth, do Technical University, em Zurique, com o propósito de ser uma ferramenta educacional. Foi batizada pelo seu idealizador de Pascal em homenagem ao grande matemático Blaise Pascal, inventor de uma das primeiras máquinas lógicas conhecidas. A linguagem Pascal foi baseada em linguagens estruturadas existentes então, tais como o ALGOL e PLI, tentando facilitar ao máximo o seu aprendizado. Contudo, essa linguagem só ganhou popularidade quando foi adotada pela Universidade da Califórnia, em San Diego, no ano de 1973. No mesmo período foram feitas implementações da linguagem para mini e microcomputadores.

Apesar de todas as dificuldades iniciais, de seu propósito educacional e da facilidade de programação, o Pascal começou a ser utilizado por programadores de outras linguagens, tornando-se, para surpresa do próprio prof. Wirth, um produto comercial. Somente no final de 1983 é que a *soft-house* americana Borland International lançou o Turbo Pascal para microcomputadores.

O Turbo Pascal é, na verdade, muito mais que um compilador, pois ele é uma associação entre um compilador, um editor de textos e um *link*

editor, o que facilita o ato de programar. Além de tudo isso, o Turbo Pascal permite facilidades e atividades que não foram planejadas pelo prof. Wirth. Assim, levando-se em conta essas considerações, pode-se dizer que o Turbo Pascal consiste em um Ambiente de Desenvolvimento Integrado (ou, em inglês, IDE).

Tendo em vista a história da linguagem Turbo Pascal, é importante entender o que, de fato, é um programa de computador. Um programa de computador (ou do inglês *software*), também chamado de aplicativo ou simplesmente programa, é uma sequência lógica de tarefas atribuídas ao computador. Existem diversas formas de se escrever um programa, contudo o computador só consegue entender os comandos a ele enviados na forma de números binários (compostos apenas pelos dígitos 0 e/ou 1), chamado de código binário ou mesmo código de máquina. Pode-se, então, escrever um programa diretamente em código de máquina com o uso de mnemônicos e montadores (o que é muito difícil e pouco prático). O ideal seria que fosse possível ditar comandos diretamente para o computador em nossa língua-mãe, ou outra de nossa escolha. A solução para esse impasse está nas chamadas linguagens de programação, que consistem em linguagens cujo texto pode, ao mesmo tempo, ser convertido em código de máquina e é inteligível ao ser humano. Ao texto escrito pelo ser humano que será convertido em um programa de computador dá-se o nome de código-fonte (do inglês *source code*).

Para se escrever um programa, pode-se, dessa forma, utilizar uma linguagem de programação de baixo nível (tal como o *Assembly*), que consiste num conjunto diminuto de comandos, ou uma linguagem de alto nível. As linguagens de alto nível, por sua vez, são mais ricas em comandos, de mais fácil manipulação e aprendizagem, sendo preferidas pela maioria dos programadores. As linguagens de alto nível utilizam-se de um interpretador de comandos (também chamado de máquina virtual, que executa o código-fonte sem realizar nenhuma conversão) ou de um compilador (que realiza a conversão do código-fonte em linguagem de baixo nível ou em código de máquina). Um interpretador executa o código-fonte durante a execução do programa, no momento em que o comando for realizado (como exemplo, tem-se a linguagem BASIC e Javascript). Já um compilador fará a conversão

de todo o código-fonte escrito em uma dada linguagem de programação para um programa que, uma vez compilado, poderá ser executado diretamente pelo sistema operacional, sem a necessidade de o programa compilador estar instalado na máquina (exemplos de compiladores são COBOL, FORTRAN e o próprio PASCAL).

O Turbo Pascal é, desse modo, um programa que faz programas. Ele é denominado de ambiente de desenvolvimento, pois não é apenas um compilador, mas também um editor de textos e um *link* editor, todos trabalhando juntos para facilitar o desenvolvimento dos programas desejados. Dentro do ambiente de desenvolvimento do Turbo Pascal é possível desenvolver, testar e depurar os programas, sem a necessidade de usar nenhum outro. Quando o programa estiver pronto, bastará copiar o arquivo executável gerado para a sua distribuição.

1.1 O primeiro programa em Pascal

O código 1.1 apresenta um pequeno programa devidamente escrito na linguagem Pascal. Ele tem a finalidade única e exclusiva de mostrar os diversos componentes de um programa em Turbo Pascal (todos os seus elementos apresentados serão descritos detalhadamente). Todo texto que estiver entre chaves { } refere-se a linhas de comentário, que são ignoradas pelo compilador no ato da compilação.

Código 1.1 Primeiro programa em Pascal

```
Program Primeiro_Exemplo; {este é o cabeçalho do programa}

USES Crt;
{Aqui se utiliza uma UNIT, chamada CRT. Existem várias e o leitor pode inclusive criar as
suas. Nestas units temos procedures e functions previamente compiladas.}

Const
Meu_Nome = 'André';
{Nesta área pode-se definir todas as constantes que quisermos utilizar no programa}
Type
  n = (BRASILEIRA, PORTUGUESA, INGLESA, FRANCESA, ALEMÃ, AMERICANA);
```

```
{O Turbo Pascal possui diversos tipos de variáveis pré-definidas, mas também permite
definir novos tipos na subárea type}

Var
  idade       : integer;
  altura      : real;
  nome        : string[30];
  sexo        : char;
  nacionalidade : n;
{todas as variáveis que forem utilizadas no corpo do programa deverão ser declaradas na
subárea Var}

Procedure Linha;
{A procedure equivale ao conceito de sub-rotina. Sua estrutura pode se tornar tão complexa
como a de um programa. Esta procedure traça uma linha na posição atual do cursor}
Var i:integer;
Begin
  For i:=1 to 80 do
  Write('-');
End;

Function Soma(x,y:integer):integer;
{O Pascal possui diversas funções pré-definidas, mas o programador também pode definir
as suas próprias funções}
Begin
  Soma:=x+y;
End;
{É possível definir quantos procedures e functions quisermos}

{Aqui começa o programa propriamente dito}
Begin
  ClrScr; {limpa a tela}
  Linha;  {executa a procedure linha }
  Writeln('Meu nome e -------> ',Meu_Nome);
  Linha;
  Write('Qual o seu nome ----> ');
  Readln(Nome);
  Linha;
  Write('Qual a sua idade ---> ');
  Readln(idade);
  Linha;
  Writeln('nossas idades somam --> ',Soma(34,idade));
  Linha;
```

```
  nacionalidade := BRASILEIRA;
  Writeln('Minha nacionalidade é brasileira');
  Writeln('Prazer em conhece-lo');
End.
```

1.2 Estrutura de um programa em Pascal

Todo programa em Pascal pode ser subdividido basicamente em três áreas, que são:

- Cabeçalho do programa.
- Área de declarações.
- Corpo do programa.

O cabeçalho do programa é de uso obrigatório, sendo composto pela palavra reservada PROGRAM, seguida de um identificador que representará o nome do programa e o caractere ponto e vírgula (;). O caractere que representa o fim de uma linha de comando na linguagem Turbo Pascal é ponto e vírgula. A área de declarações é subdividida em seis subáreas, todas elas de uso facultativo, a saber:

- USES.
- LABEL.
- CONST.
- TYPE.
- VAR.
- PROCEDURE.
- FUNCTION.

A declaração USES permite utilizar unidades de programa externas previamente desenvolvidas, chamadas de *UNIT*. Uma unidade nada mais é que uma biblioteca de funções e procedimentos, que consistem em pequenos trechos identificados de código-fonte, como pode ser visto no Capítulo 6. Dessa forma, é possível ampliar ainda mais as funcionalidades de uma linguagem de programação, criando novas bibliotecas, reduzindo, assim, o tempo de desenvolvimento, a quantidade de linhas e a complexidade do código-fonte escrito pelo programador.

Na subárea LABEL, deve-se declarar todos os rótulos que forem utilizados no corpo do programa. Os rótulos são aplicados em conjunto com a instrução GOTO. Todas as constantes a serem usadas no programa podem ser definidas na subárea CONST. O Turbo Pascal tem seis tipos básicos de variáveis pré-definidas: INTEGER, REAL, BYTE, BOOLEAN, CHAR e STRING. No entanto, pode-se definir novos tipos de variáveis na subárea TYPE. Todas as variáveis utilizadas no programa devem ser declaradas na subárea VAR. Isso porque a alocação de espaço na memória do computador para o armazenamento das variáveis é feita durante a compilação do programa. Em outras linguagens (como, por exemplo, o C e o C++), pode-se declarar uma variável em diferentes partes do programa, o que não ocorre no Pascal.

Na subárea PROCEDURE, pode-se definir quantas e quais sub-rotinas são necessárias, sendo estas sub-rotinas chamadas durante o programa utilizando os seus respectivos nomes, como será visto. Embora o Turbo Pascal já possua várias funções predefinidas, em alguns casos a criação de uma função pode gerar a redução do número de linhas e da complexidade do programa. Na subárea FUNCTION, é possível definir novas funções.

De acordo com a definição padrão da linguagem Pascal, essas subáreas devem aparecer na sequência mostrada anteriormente, ou seja: LABEL – CONST – TYPE – VAR – PROCEDURE – FUNCTION. Todavia, no Turbo Pascal não existe uma sequência obrigatória dessas áreas.

2. Noções básicas de programação de computadores

Este capítulo apresenta os elementos básicos da linguagem Pascal, bem como a declaração e o tipo de variáveis e constantes, além de uma explanação sobre os operadores usados pela linguagem.

2.1 Elementos básicos da linguagem Pascal

2.1.1 Caracteres utilizados

Os caracteres que podem ser utilizados no Turbo Pascal são divididos em:

- **Letras:** que compreendem os caracteres 'A' até 'Z' e 'a' até 'z'.
- **Números:** que compreendem os algarismos 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9 e as suas combinações.
- **Caracteres especiais**: são caracteres tais como + – * / = ^ < > ( ) [ ] { } . , : ; ' # $, que possuem sentido especial para a linguagem.

Observações:

i. O Pascal não faz distinção entre letras maiúsculas e minúsculas para identificadores, ou seja, ele **não** é *case sensitive* (como, por exemplo, a linguagem C e C++). Dessa forma, a variável ***x*** e a variável ***X*** são consideradas pelo Pascal como uma só.

ii. Embora na maioria das linguagens de programação o sinal de atribuição, que dá um valor a uma variável, seja o sinal de igual (=), no Pascal o símbolo de atribuição é a união de dois caracteres, o dois-pontos e o igual (:=).

Exemplo:

```
A = 100          {em Basic, C e C++}
A := 100;        {em Pascal e Delphi}
```

Todos os dois comandos acima fazem, cada um na sua respectiva linguagem de programação, a atribuição do valor inteiro 100 à variável A.

iii. Dois pontos finais em seguida um do outro (..) indicam um delimitador de faixa, ou de conjunto. Por exemplo, a representação 1..30 indica o intervalo fechado de todos os inteiros entre 1 e 30.

2.1.2 Palavras reservadas

As chamadas palavras reservadas de uma linguagem de programação são identificadores (ou nomes) que fazem parte da sua estrutura da linguagem e possuem significado predeterminado. Uma palavra reservada não pode ser redefinida e não pode ser utilizada como um identificador de variável, procedimento ou função criado pelo programador. A Tabela 2.1 reúne algumas das palavras reservadas utilizadas pelo compilador Turbo Pascal.

Tabela 2.1 - Lista de palavras reservadas utilizadas pelo compilador Turbo Pascal

Absolute*	And	Array	Begin
Case	Const	Div	Do
Downto	Else	End	External*
File	For	Forward	Function
Goto	If	In	Inline*

(continua)

* Palavras reservadas definidas apenas no ambiente Turbo Pascal (não válidas para a linguagem Pascal padrão).

Label	Mod	Nil	Not
Of	Or	Packed	Procedure
Program	Record	Repeat	Set
Shl*	Shr*	String*	Then
To	Type	Until	Var
While	With	Xor*	

2.1.3 Identificadores predefinidos

Tal qual uma palavra reservada, as linguagens de programação possuem vários identificadores predefinidos que não fazem parte da definição padrão da linguagem de programação em si, mas que compõem um conjunto de ferramentas importante para o desenvolvimento de programas. Um identificador é um nome que representa uma variável, uma constante, um procedimento ou uma função, que podem ser utilizados normalmente na construção de programas. Dois exemplos de identificadores utilizados na linguagem Pascal são:

- **CLRSCR** (do inglês *Clear Screen*): este procedimento tem a finalidade de limpar a tela de vídeo.
- **DELLINE** (do inglês *Delete Line*): apaga a linha em que está o cursor.

Dessa forma, o programador não pode usar as palavras CLRSCR ou DELLINE em seus programas senão com as funções supracitadas.

2.1.4 Regras para formação de identificadores

O programador também pode definir seus próprios identificadores. Nomes de variáveis, de rótulos, de procedimentos, de funções e de constantes, por exemplo, são identificadores que devem ser formados pelo programador. Porém, para isso, existem determinadas regras que devem ser seguidas. São elas:

i. O primeiro caractere do identificador deverá ser obrigatoriamente uma letra ou um traço de sublinha (_), também chamado de *underscore* ou *underline*.

ii. Os demais caracteres podem ser letras, dígitos ou um *underscore*, não podendo ser um caractere especial (como, por exemplo, um caractere espaço + – * / = ^ < > ( ) [ ] { } . , : ; ‘ # $).
iii. Um identificador pode ter no máximo 127 caracteres. Como os identificadores serão usados ao longo do programa, é altamente aconselhável que estes sejam autoexplicativos. Contudo, não se recomenda que os identificadores sejam muito extensos.
iv. Como citado anteriormente, um identificador não pode ser palavra reservada.

Podem ser citados como exemplos os identificadores apresentados na tabela a seguir.

Tabela 2.2 - Exemplos de identificadores no Turbo Pascal

Identificadores válidos	Identificadores inválidos
Meu_Nome	2teste
MEU_NOME	Exemplo 23
__Linha	Adi+cão
EXemplo23	Sub-tração

De modo a elucidar a regra *iii*, sobre o tamanho de um identificador, suponha que fosse necessário criar um identificador para trabalhar com a grandeza física Número de Reynolds. A letra R poderia, a princípio, ser escolhida como tal identificador, uma vez que este seria um identificador válido, porém nada explicativo. Já o identificador Numero_de_Reynolds, que é autoexplicativo, é muito grande para ser usado ao longo de um programa. Uma sugestão razoável, nesse caso, seria o identificador NReynolds, por ser um identificador pequeno, mas sem sacrifício ao entendimento de seu sentido.

Quando for necessário criar um identificador composto por duas ou mais palavras (como no exemplo Número de Reynolds), sugere-se o uso de letras maiúsculas para definir o início de uma palavra, sendo as demais letras minúsculas (assim sendo, ter-se-ia no exemplo o identificador NumeroDeReynolds). Atualmente, o uso de um caractere para substituir o espaço em branco

está em desuso, uma vez que a letra maiúscula tem a mesma função, sem o gasto de um caractere adicional, para representar o mesmo identificador.

2.1.5 Linha de comentário

Uma linha de comentário é um texto introduzido ao longo do código-fonte de um programa com a intenção de torná-lo mais inteligível. Uma vez que as linhas de comentário não são consideradas como instruções pelo compilador, são removidas no ato da compilação. Uma boa prática de programação consiste em inserir linhas de comentário nos programas, uma vez que o código-fonte de um programa pode vir a ser revisado por outro programador e tais linhas servirão como um roteiro dentro do programa, mostrando o significado de cada estrutura dele. Todo o texto que estiver entre os símbolos (* e *) ou { e } será considerado pelo compilador como uma linha de comentário.

2.1.6 Números

Na linguagem Pascal, pode-se trabalhar tanto com números inteiros quanto com números reais. Um número inteiro é aquele que não possui parte fracionária. Um número real, por sua vez, possui tanto dígitos antes quanto depois da vírgula. Os números inteiros podem ser representados na base hexadecimal e, para isso, basta precedê-los do símbolo cifrão (por exemplo, o número $A1F). Os números reais podem ser representados em notação científica, por exemplo:

```
125476 = 1.25476e5
```

Em que o número que vem após a letra ***e*** é o expoente da potência de base 10, à qual o número antes da letra está multiplicado, sendo o expoente um número inteiro qualquer.

As tabelas a seguir apresentam os limites (*ranges*) de variação dos tipos mais comuns de variáveis usadas no ambiente Turbo Pascal. Tais limites costumam variar entre versões do Turbo Pascal, sendo as faixas de valores apresentadas nessas tabelas válidas para a versão 7.0 desse ambiente.

A palavra *signed* apresentada na tabela significa "com sinal", o que indica que esse tipo de variável pode armazenar valores negativos ou positivos. Já o tipo de dado indicado como *unsigned* só pode armazenar valores positivos.

Tabela 2.3 - Limites dos tipos de variáveis mais comuns em Turbo Pascal para números inteiros

Tipo	Faixa de variação (*range*)	Formato
Shortint	-128..127	Signed 8-bit
Integer	-32768..32767	Signed 16-bit
Longint	-2147483648..2147483647	Signed 32-bit
Byte	0..255	Unsigned 8-bit
Word	0..65535	Unsigned 16-bit

Tabela 2.4 - Limites dos tipos de variáveis mais comuns em Turbo Pascal para números reais

Tipo	Faixa de variação (*range*)	Dígitos significativos	Bytes
Real	2.9e-39..1.7e38	$11 - 12 * 10^3$	6
Single	1.5e-45..3.4e38	$7 - 8 * 10^3$	4
Double	5.0e-324..1.7e308	$15 - 16 * 10^3$	8
Extended	3.4e-4932..1.1e4932	$19 - 20 * 10^3$	10
Comp	-9.2e18..9.2e18	$19 - 20 * 10^3$	8

Os tipos SHORTINT, INTEGER, LONGINT, BYTE e WORD só podem armazenar valores inteiros, e as diferenças entre esses tipos de dados são: os seus limites de valores (*ranges*) e o fato de uns serem *signed* e outros *unsigned*. Já os tipos REAL, SINGLE, DOUBLE, EXTENDED e COMP podem armazenar valores reais, e a diferença entre esses tipos de dados refere-se apenas aos seus limites de valores (*ranges*).

2.1.7 Caracteres e textos (STRING)

Um caractere na linguagem Pascal é representado pelo tipo de dado CHAR (do inglês caractere), que corresponde a todos os caracteres gráficos que podem ser gerados pelo teclado, tais como dígitos, letras e caracteres especiais (&, #,* etc.). No código-fonte, os caracteres devem vir sempre entre

aspas simples. Assim sendo, para se atribuir a uma variável denominada ***ch*** do tipo CHAR a letra A, deve-se prosseguir da seguinte maneira:

```
ch := 'A';
```

Uma STRING, por sua vez, é uma sequência de caracteres delimitados por aspas simples usadas, em geral, para compor frases ou orações. Esse tipo de dado é chamado de estruturado, ou composto, pois é construído com base no tipo simples caractere. Se uma variável do tipo caractere armazena apenas um caractere, uma variável do tipo STRING armazenará um pequeno texto. Por exemplo:

```
isto é uma string'
'123456'
'Uma string pode conter um ou mais caracteres...'
```

2.1.8 Caracteres de controle

Os chamados caracteres de controle são caracteres que possuem significado especial para a linguagem de programação. No Pascal, tais caracteres devem ser representados pelo seu código da tabela ASCII precedidos do símbolo #, ou então a letra correspondente precedida do símbolo ^. Por exemplo, o comando Control + G, que significa *bell* ou *beep* (soa uma campainha no computador), no Pascal é utilizado como #7 ou ^G. Para usar o comando Control + L, que significa *form feed* (solicita uma alimentação de formulário), basta escrever ^L.

2.2 Definição de variáveis

Todas as variáveis que serão utilizadas no corpo de um programa devem ser declaradas numa subárea específica, iniciada pela palavra reservada VAR. Os tipos de dados predefinidos na linguagem Pascal são divididos em duas categorias:

i. ***Escalares Simples***: são tipos de dados escalares alfanuméricos e que não possuem nenhum tipo de indexador ou campo. Os seguintes tipos de dados pertencem à classe de dados escalares simples:

Chara	Longint	Single
Boolean	Byte	Double
Shortint	Word	Extended
Integer	Real	Comp

Todos os tipos escalares simples mostrados acima já foram vistos, com exceção do tipo BOOLEAN (ou lógico), que só pode assumir os valores lógicos FALSE (falso) ou TRUE (verdadeiro). As palavras FALSE e TRUE são palavras reservadas da linguagem Pascal e, dessa forma, não precisam ser declaradas como identificadores e não é necessário o uso de aspas.

i. ***Escalares estruturados***: são os tipos de dados escalares alfanuméricos que possuem uma estrutura, sendo eles indexados por números inteiros ou definidos pelo nome de seus campos. Os seguintes tipos de dados pertencem à classe de dados escalares estruturados:

String	Record	Set
Array	File	Text

O código 2.1 apresenta um programa cuja finalidade é apenas demonstrar como se realiza a declaração e a inicialização de uma variável em Pascal.

Código 2.1 - Declaração e inicialização de variáveis em Pascal

```
Program Exemplo;      (* cabeçalho do programa *)
Var
  idade, numero_de_filhos : byte;
  altura              : real;
  sexo                : Char;
  nome                 : string[30];
  sim_ou_nao           : Boolean;
  quantidade           : integer;
(* aqui começa o programa propriamente dito*)

Begin
  Idade := 34;
  numero_de_filhos := 2;
```

```
  sexo := 'M';
  nome := 'Felipe';
  sim_ou_nao := TRUE;
  quantidade := 3245;
End.
```

Observações importantes:

i. A palavra reservada VAR pode aparecer mais de uma vez em um programa, o que ocorrerá quando existirem procedimentos e/ou funções no programa.
ii. A sintaxe geral para declaração de variáveis é:

```
variavel_1, variavel_2,...,variavel_n : tipo da variável;
```

iii. Os espaços em branco e quebras de linhas que separam os elementos da linguagem são excluídos no ato da compilação. Dessa forma, podem-se utilizar quantos forem necessários.
iv. As instruções são separadas entre si por ponto e vírgula (;). Pode-se colocar mais de uma instrução em uma única linha. Contudo, o limite de caracteres em uma única linha é de 127.
v. O tipo STRING deve ser procedido da quantidade máxima de caracteres que a variável pode assumir. A alocação de espaço de memória para as variáveis é feita durante a compilação, portanto o compilador precisa desse dado. Por outro lado, o fato de ter sido criada uma STRING de tamanho 30 não obriga o programador a utilizar os 30 caracteres, mas, sim, um máximo de 30.

Além dos tipos de dados predefinidos no Turbo Pascal, pode-se definir novos tipos de dado na subárea TYPE. Uma forma de se utilizar a subárea TYPE é a criação de constantes enumeradas, conforme mostrado abaixo, em que o identificador deve seguir as regras dadas anteriormente e entre os parênteses estão os valores que o identificador poderá assumir.

```
Type
  identificador = (valor1, valor2, valor3, ..., valorN);
```

O código 2.2 mostra a criação de quatro tipos enumerados de dados na subárea TYPE, que são o tipo de dados cor, dia útil, linha e idade. Quando se cria um tipo de dado enumerado, o Turbo Pascal assume, automaticamente, que o elemento da esquerda vale zero e vai incrementando em uma unidade os demais elementos. Para definição do tipo ***cor*** no código 2.2, o elemento amarelo valerá quatro e o elemento verde valerá três.

Código 2.2 - Uso do comando TYPE para a criação de tipos de variáveis

```
Type
(*            0,      1,     2,    3,     4*)
  cor     = (azul, vermelho, branco, verde, amarelo);
(*              0,    1,     2,     3,   4*)
  dia_util = (segunda, terça, quarta, quinta, sexta);
  linha    = string[80];
  idade    = 1..99;
(* a partir deste instante, além dos tipos de dados pré- definidos, podemos também
utilizar os novos tipos de dados definidos cor, dia_útil, linha e idade *)
Var
  i    : integer;
  d    : idade;
  nome  : linha;
  dia   : dia_util;
  cores : cor;
(* etc. *)
```

2.3 Constantes

Constantes são valores que não se alteram durante a execução de um programa, como, por exemplo, o valor de p (3,141592...), ou que raramente se alteram, como é o caso de um programa que armazena as notas de provas de alunos. Pode-se criar uma constante chamada NOTA_MAXIMA e associar a esta o valor 10. Se a nota máxima for alterada para 100, bastará revisar o valor da constante e recompilar o programa, não havendo a necessidade de se revisar todo ele. As constantes são definidas na subárea CONST, por exemplo:

```
Const
  meu_nome     = 'Felipe';
  cor_preferida = 'verde';
  numero_maximo = 24345;
(* e assim por diante*)
```

Toda vez que o compilador encontrar uma referência no código-fonte a uma das constantes acima, ele substituirá a referência pelo seu respectivo valor. Uma constante deve ser sempre inicializada junto a sua declaração (atribuição de valor à constante), o que não é obrigatório para as variáveis. Pode-se imaginar que uma constante é uma variável somente leitura (do inglês *read-only variable*). Sugere-se que as variáveis sejam sempre inicializadas, pois elas o serão em sua criação com um valor aleatório chamado de lixo da memória.

Como ocorre para as palavras reservadas, existem constantes já predefinidas pela linguagem Pascal e pelo ambiente Turbo Pascal que podem ser utilizadas no desenvolvimento de programas sem a necessidade de serem declaradas. Algumas dessas constantes são:

```
PI = 3.1415926536
FALSE
TRUE
NIL                                          {Ponteiro nulo}
MAXINT = 32767
```

A declaração de uma constante na subárea CONST não explicita o tipo de dado da constante, deixando ao compilador associar automaticamente um tipo de dado à constante. Caso seja desejado criar uma constante com o seu tipo de dado explícito, pode-se usar o conceito de constante tipada, cuja sintaxe é:

```
Const variavel: tipo = valor;
```

Como exemplo, tem-se:

```
Const
  contador: integer = 100;
  c: char = 'A';
```

No exemplo acima, foram definidas duas constantes, uma chamada ***contador*** do tipo inteiro, cujo valor é 100, e outra chamada ***c*** tipo caractere de valor 'A'. A vantagem em se utilizar constantes tipadas é a explicitação de seu tipo, o que não ocorre com as constantes definidas.

2.4 Operadores

A linguagem Pascal é repleta de operadores, que são símbolos representativos da ação de uma operação sobre um ou mais argumentos (ou operandos). O sinal de adição (+), por exemplo, é um operador. Em uma expressão, como era de se esperar, esse operador soma dois valores. Um exemplo seria:

```
a := b + c;
```

Em que as variáveis A, B e C podem ser de qualquer tipo de dado numérico. O exemplo acima deve ser lido da seguinte forma: o valor da variável B será somado ao valor da variável C e o resultado, armazenado na variável A. O sinal de dois pontos e igual nesse exemplo representa um único operador, com a função de atribuir o valor à sua direita para a posição da memória representada pela variável à sua esquerda. Os operadores da linguagem Turbo Pascal são divididos em operadores aritméticos, relacionais, lógicos, de *bits* e operadores de concatenação.

2.4.1 Operadores aritméticos

Os operadores aritméticos são, como o próprio nome já indica, os operadores utilizados para a execução de operações aritméticas. Os operadores aritméticos da linguagem Pascal e sua função são apresentados na tabela abaixo.

Tabela 2.5 - Operadores aritméticos da linguagem Pascal

Operador aritmético	Função
+	Adição
–	Subtração
*	Multiplicação

(continua)

/	Divisão entre números reais
DIV	Divisão entre números inteiros
MOD	Retorna o resto da divisão

O código 2.3 mostra como utilizar os operadores aritméticos. Note que as funções utilizadas nesse programa e que ainda não foram apresentadas têm a sua função descrita na forma de comentário.

Código 2.3 - Uso de operadores aritméticos

```
Program Operadores_aritmeticos;
{ Mostra como utilizar operadores aritméticos}

Uses CRT;
Var
  x, y, z : integer;
  r1, r2  : real;
Begin
  ClrScr;                    (* limpa a tela *)
  x := 10;
  y := 20;
  z := x + y;
  writeln(z);                (*escreve o valor de z na tela do pc*)
  x := 20 DIV 3;
  y := 20 MOD 3;
  writeln(x);                (* escreve 6 na tela *)
  writeln(y);                (* escreve 2 na tela *)
  r1 := 3.24;
  r2 := r1 / 2.3;
  writeln(r2:2:3);           (* escreve 1.409 na tela *)
  readln;
(*faz com que o fluxo do programa pare até que o usuário tecle ENTER*)
End.
```

A figura a seguir apresenta o ambiente de desenvolvimento Turbo Pascal com o código 2.3. As palavras reservadas da linguagem aparecem em branco, as linhas de comentário, em cinza claro e os demais comandos, números e texto, em amarelo. As opções do menu principal do Turbo Pascal são: *File*, *Edit*, *Search*, *Run*, *Compile*, *Debug*, *Tools*, *Options*, *Windows* e *Help*.

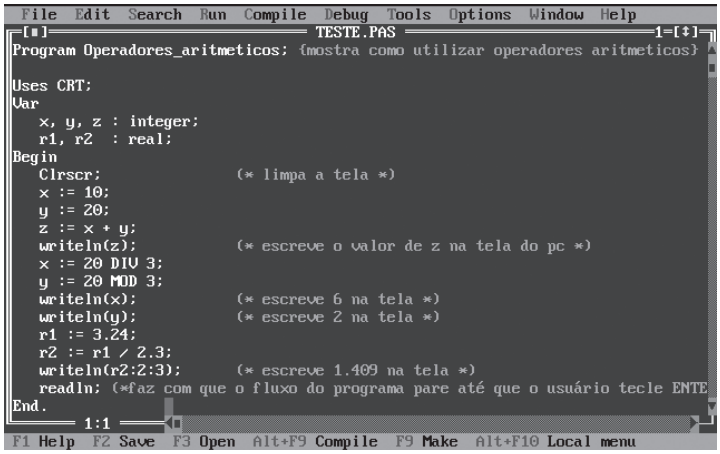


Figura 2.1 - Ambiente de desenvolvimento Turbo Pascal

O menu *File* contém funções para abrir e gravar arquivos de código-fonte, cuja extensão é .PAS. Pode-se, ainda, alterar o diretório atual, imprimir um arquivo, encerrar temporariamente a execução do Turbo Pascal e retornar para o DOS, ou encerrar definitivamente a execução do Turbo Pascal.

Já o menu *Edit* permite operações de manipulação do texto do código-fonte, tais como: desfazer alterações, refazer alterações, cortar, copiar, colar e limpar texto e exibir o conteúdo da área de trabalho. O menu *Search* contém opções de busca e troca de texto e deslocamento do cursor para um determinado local do código-fonte.

Após digitar o código-fonte de um programa, é possível compilá-lo e executá-lo a partir do menu *Run*. A opção *Run* no menu *Run* compila e executa o programa em um único comando. Se o código-fonte compilado depender de outros arquivos que tenham sido atualizados desde a última compilação, estes também serão compilados automaticamente. A opção *Trace into* executa uma única linha de código do programa. Se a linha atual for uma chamada a uma função ou procedimento, basta pressionar a tecla F7 para ir até essa função ou procedimento. A opção *Step over* funciona de forma semelhante a *Trace Into*, com a exceção de que a opção *Step over* não conduzirá às funções ou procedimentos chamados, estas serão executadas como uma única instrução.

A figura seguinte apresenta o resultado da compilação e execução do código 2.3 por meio da opção *Run* do menu de mesmo nome.

O menu *Compile* permite que o programador compile parte do seu programa ou o programa inteiro e especifique onde o programa compilado deve ser armazenado. O menu *Debug* permite diversas formas de depuração do código-fonte. O menu *Tools* facilita a utilização de ferramentas de desenvolvimento e pode ser configurado pelo próprio programador. É no menu *Options* que o programador pode especificar a forma com que se deseja trabalhar no ambiente Turbo Pascal. O menu *Window* contém os comandos de manipulação das janelas, sendo que cada janela normalmente é um arquivo .PAS contendo código-fonte. Por último, o menu *Help* é o menu de ajuda do Turbo Pascal, podendo ser acionado de qualquer parte dele pela tecla F1.

Figura 2.2 - Resultado do Código 2.3 em execução

2.4.2 Operadores relacionais

Os operadores relacionais permitem escrever instruções que comparam valores. Quando um programa executa uma expressão relacional, ele avalia seus argumentos para produzir um resultado, que pode ser verdadeiro ou

falso. Dessa forma, o resultado de um operador relacional é sempre um valor lógico (ou do tipo BOOLEAN). Pode-se usar qualquer operador da Tabela 2.6 em expressões para se comparar dois valores (numéricos ou não).

Tabela 2.6 - Operadores relacionais em Pascal

Operador	Descrição	Exemplo
<	Menor que	(a < b)
<=	Menor ou igual a	(a <= b)
>	Maior que	(a > b)
>=	Maior ou igual a	(a >= b)
=	Igual a	(a = b)
<>	Diferente de	(a <> b)
IN	Testa se um elemento está incluso em um conjunto	a IN [0..9]

A expressão (a = b) é avaliada como verdadeira se, e somente se, a e b possuírem o mesmo valor. Já a expressão (a := b) atribui o valor de b à variável a. Alguns exemplos do uso de operadores relacionais são:

i. Se A = 30 e B = 50, então:

```
( A = B )        ⇨ FALSE
( A < B )        ⇨ TRUE
```

ii. Se A = TRUE e B = FALSE

```
( A <> B )⇨ TRUE
( A = B  ) ⇨ FALSE
```

2.4.3 Operadores lógicos

Os operadores relacionais permitem escrever expressões que executam ações baseadas na veracidade ou falsidade de seus argumentos. Os operadores lógicos, por sua vez, expandem essa ideia fornecendo meios para avaliar expressões relacionais múltiplas (ou interdependentes) de modo a se obter um único resultado lógico. A Tabela 2.7 apresenta os operadores lógicos da linguagem Turbo Pascal e seu significado.

Tabela 2.7 - Operadores lógicos da linguagem Pascal

Operador lógico	Descrição
AND	E lógico
OR	OU lógico
XOR	OU EXCLUSIVO lógico
NOT	Negação

Operadores lógicos só aceitam como argumentos valores lógicos. A operação AND resulta em TRUE se, e somente se, todos os seus argumentos forem TRUE, se um deles ou ambos forem FALSE, então o resultado será FALSE. Já a operação OR resulta em TRUE quando pelo menos um dos argumentos for TRUE. Por fim, a operação XOR resulta TRUE quando os argumentos forem diferentes entre si, isto é, quando um for TRUE, o outro dever ser FALSE. O operador NOT, que possui apenas um operando, nega o valor do operando. Dessa forma, se o valor do operando for TRUE, o resultado será FALSE e se o operando for FALSE, o resultado será TRUE. O código 2.4 apresenta um exemplo do uso dos operadores lógicos.

Código 2.4 - Uso de operadores lógicos

```
Program operadores_logicos;
Uses CRT;
Var
  x,y : Boolean;
Begin
  x:= TRUE;
  y:= FALSE;
  Writeln( x OR y );       (* escreve TRUE *)
  Writeln( x AND y );      (* escreve FALSE *)
  Writeln( x XOR y );      (* escreve TRUE *);
  Writeln( NOT x );        (* escreve FALSE *);
  Writeln( NOT y );        (* escreve TRUE *);
  Readln;
End.
```

2.4.4 Operadores de *bits*

Uma razão para se trabalhar diretamente com *bits* na memória é o armazenamento de informações no menor espaço possível. Por exemplo, um *bit* em um registro de banco de dados poderia representar o *status* corrente de um fato, sendo 1 para sim e 0 para não, ou 0 para masculino e 1 para feminino. Uma série de *bits* também poderia representar um conjunto de valores: se o *bit* 1 estiver ligado, o valor de A será considerado como pertencendo ao conjunto, se o *bit* 2 estiver ligado, então o valor B pertence ao conjunto e assim por diante.[2] Os operadores de *bits* só podem ser aplicados em dados dos tipos *byte* ou inteiro, sendo o resultado sempre do tipo inteiro. Tais operadores agem *bit* a *bit* e podem ser aplicados na notação hexadecimal ou decimal. São eles SHL, SHR, NOT, AND, OR e XOR.

O operador binário SHL (do inglês *SHift Left*) desloca o seu operando para a esquerda em ***n*** *bits*. Durante o deslocamento, os *bits* à esquerda são perdidos e *bits* zeros preenchem as novas posições à direita. Exemplos do uso do operador SHL são:

1. Se X = 00010101 então:

```
X shl 2 = 01010100
X shl 5 = 10100000
```

2. 55 shl 3 = 184, pois 55 = 00110111, assim sendo, deslocando-se 3 *bits* à esquerda, tem-se 10111000, que é igual a 184.
3 $F0 shl 2 = $C0, pois $F0 = 11110000 e deslocando-se 2 *bits* à esquerda, tem-se 11000000, que é igual a $C0.

Já o operador binário SHR (do inglês *SHift Right*) desloca o seu operando para a direita em ***n*** *bits*. Durante o deslocamento, os *bits* à direita são perdidos e *bits* zeros preenchem as novas posições à esquerda. Exemplos do uso do operador SHR são:

---

2 Para mais informações sobre a base binária (base 2), que é base à qual se aplicam os operadores binários, consulte o APÊNDICE B deste livro.

1. Se X = 10101100 então:

```
X shr 3 = 00010101
X shr 6 = 00000010
```

2. 55 shr 3 = 6, pois 55 = 00110111, assim sendo, deslocando-se 3 bits à direita, tem-se 00000110, que é igual a 6.
3 $F0 shr 2 = $3C, pois $F0 = 11110000 e deslocando-se 2 bits à direita, tem-se 00111100, que é igual a $3C.

O operador binário NOT nega os *bits*, isto é, os *bits* que, antes da operação, eram iguais a 0 tornam-se 1 e os *bits* 1 tornam-se 0. Um número inteiro é representado por 2 *bytes* (16 *bits*), sendo um dos *bits* representativo do valor absoluto do número (positivo ou negativo). Portanto, ao se trabalhar com números inteiros, o seu sinal será afetado, pois o *bit* de ordem mais alta é também o *bit* do sinal. Por exemplo:

```
NOT (255) = -256
```

Para suprimir esse problema, deve-se trabalhar com variáveis do tipo *byte*. O código 2.5 apresenta um programa que ilustra a afirmação acima.

Código 2.5 - Uso do operador NOT

```
Program Exemplo;
Uses CRT;
Var
  i,j : Byte;
Begin
  ClrScr;
  i := 255;
  j := NOT(i);
  Writeln(j);    (* será escrito 0 *)
End.
```

O operador binário AND realiza a operação AND lógico *bit* a *bit*. A operação AND resulta em 1 (verdadeiro) se, e somente se, os dois operandos

forem iguais a 1, caso contrário, o resultado será igual a 0 (falso). Exemplos do uso do operador binário AND são:

1. $0F AND $F0 = $0, pois $0F = 00001111 e $F0 = 11110000. Assim sendo:

```
00001111 AND 11110000 = 00000000
```

Percebe-se que, como os dois operandos acima não possuem dois *bits* 1 na mesma posição, o resultado da operação AND binária não podia ser outro a não ser 0.

2. 255 AND 55 = 55, pois 255 = 11111111 e 55 = 00110111. Assim sendo:

```
11111111 AND 00110111 = 00110111
```

3. 34 AND 76 = 0, pois 34 = 00100010 e 76 = 01001100. Assim sendo:

```
00100010 AND 01001100 = 00000000
```

O operador binário OR realiza a operação OR lógica *bit* a *bit*. A operação OR resulta em 1 (verdadeiro) se pelo menos um dos dois operandos forem iguais a 1 (verdadeiro). Exemplos do uso do operador binário OR são:

1. $0F OR $F0 = $FF, pois $0F = 00001111 e $F0 = 11110000. Assim sendo:

```
00001111 OR 11110000 = 11111111
```

2. 255 OR 55 = 255, pois 255 = 11111111 e 55 = 00110111. Assim sendo:

```
11111111 OR 00110111 = 11111111
```

3 34 OR 76 = 110, pois 34 = 00100010 e 76 = 01001100. Assim sendo:

```
00100010 OR 01001100 = 01101110
```

O operador binário XOR realiza a operação ou exclusivo (XOR) lógico *bit* a *bit*. A operação XOR resulta em 1 (verdadeiro) se os operandos forem diferentes entre si, caso contrário, retorna 0 (falso). Exemplos do uso do operador binário XOR são:

1. $0F XOR $F0 = $FF, pois $0F = 00001111 e $F0 = 11110000. Assim sendo:

```
00001111 XOR 11110000 = 11111111
```

2. 255 XOR 55 = 200, pois 255 = 11111111 e 55 = 00110111. Assim sendo:

```
11111111 XOR 00110111 = 11001000
```

3. 34 XOR 76 = 110, pois 34 = 00100010 e 76 = 01001100. Assim sendo:

```
00100010 XOR 01001100 = 01101110
```

2.4.5 Concatenação

Essa operação é representada pelo sinal de adição (+) e ambos operandos devem ser do tipo STRING e/ou do tipo CHAR. Uma concatenação significa uma soma de duas *strings* ou de dois caracteres simples formando uma *string*. Exemplos de concatenação são:

```
a := 'Isto é uma ';
b := 'string';
c := a + b;
writeln(c);          (* escreve na tela 'Isto é uma string'*)
```

2.5 Exercícios

1. Quais dos identificadores abaixo estão corretos? O que está errado nos identificadores incorretos? Corrija-os.

a) valor	b) salário-líquido	c) _b248
d) (x2)	e) nota*do*aluno	f) a1b2c3
g) 3 x 4	h) Maria	i) km/h
j) xyz	k) nome empresa	l) sala_215
m) 'nota'	n) ah!	o) m{a}

2. Identifique o tipo de cada uma das constantes abaixo:

a) 613	b) 613,0	c) -613
d) '613'	e) -3,012 x 10	f) 17 x 1012
g) 28.3 x 10-33	h) Falso	i) 'verdadeiro'
j) 'constante'	k) 21	l) 0,21

3. Supondo que as variáveis NB, NA, NMAT e SX sejam utilizadas para armazenar a nota do aluno, o nome do aluno, o número da matrícula e o sexo, declare-as corretamente, associando o tipo adequado ao dado que será armazenado.

4. Calcule o resultado e o tipo de cada uma das expressões abaixo:

a) 5 * 2 + 3	b) 6 + 19 – 0,3	c) 3 * 5 + 1
d) 1 / 4 + 2	e) 29,0 / 7 + 4	f) 3 / 6,0 - 7
g) 16 * 6 - 3 * 2	h) 3 + 2 * (18 - 4 * 2)	i) -2 * 3
j) 2 * 2 * 3	k) (28 + 3 * 4) / 4	l) 8 - 30 / 6

5. Sendo P, Q, R e S variáveis cujos conteúdos são iguais a 2; 3; 12 e 4,5, respectivamente, quais os valores fornecidos pelas expressões aritméticas abaixo?

a) 100 * Q div P + R	b) P + R mod 5 - Q / 2
c) 1 + (P*3 + 2 * R) * (1 / 5)	d) 1 + (R + Q) div Q
e) ((20 div 3) div 3) + 2 8 / 2	f) - P 2 + (R * 10) / round(S)
g) 2 * S mod 3 – trunca(S)	h) R mod (P+1) - Q * R

NOTA: A função ROUND arredonda um número real para inteiro.

6. Considerando A, B e C variável tipo real contendo os valores 1, 4,5 e 8, respectivamente; NOME e COR variáveis do tipo caractere contendo as sequências de caracteres 'Tania' e 'Branco', e TESTE uma variável do tipo lógico contendo o valor verdadeiro, determine os resultados obtidos da avaliação das seguintes expressões lógicas:

   a) A = 1 E TESTE
   b) (NOME = 'Pedro') OU (COR = 'Branco')
   c) NÃO TESTE OU C mod 2 = 0,5
   d) (C < 10) OU TESTE E (COR = 'Preto')
   e) A * 12 + C * (1/3) = 3 E (A + (B+C) > 13) OU (NOME='Ana'))
   f) TESTE E NÃO TESTE

7. Sendo SOMA, NUM, X variáveis do tipo real; NOME, COR, DIA variáveis do tipo caractere, TESTE, CÓDIGO, TUDO variáveis do tipo lógico, assinale os comandos de atribuição considerados inválidos:

   a) NOME := 5; b) SOMA := NUM + 2 * X; c) X := NOME > CÓDIGO;
   d) TUDO := SOMA; e) COR := 'Preto' - X; f) NUM := '*ABC*';
   g) X := X + 1; h) DIA := 'Segunda'; i) SOMA := X * 2 - NUM;
   j) TESTE := CÓDIGO OU (X*2 = SOMA);

8. Quais os valores armazenados em SOMA, NOME e TUDO, supondo-se que NUM, X, COR, DIA, TESTE e CÓDIGO valem, respectivamente, 5; 2,5; 'Azul'; 'Terça'; falso e verdadeiro?

   a) NOME := DIA;
   b) SOMA := NUM * 2 / X + arredonda (X+1);
   c) TUDO := NÃO TESTE OU CÓDIGO E (SOMA < X);

9. Dê o valor da variável RESULTADO após a execução da seguinte sequência de operações (suponha que todas as variáveis sejam do tipo real):

   a) RESULTADO := 3,0 * 6;
   b) X := 2,0; Y := 3,0; RESULTADO := X * Y - X;
   c) RESULTADO := 4; X := 2; RESULTADO := RESULTADO * X;

10. Suponha que A, B, C, I, J e K sejam variáveis numéricas. Dados A = 4,0, B = 6,0 e I = 3, qual será o resultado dos seguintes comandos de atribuição?

a) C := A * B - I;
b) K := I / 4 * 6;
c) C := B / A + 1,5;
d) K := trunca (B / A + 4,7);
e) J := arredonda (A / (5 / I));
f) K := abs (A - B) * 2 + I;

11. Com base nas variáveis A, X, DIA, K e NOME que valem, respectivamente, 12,5; 5; 'terça'; verdadeiro e 'João', determine o valor resultante das expressões abaixo:

a) A + 5 * 10 - X*2 mod 3
b) DIA = 'DIA'
c) K E (A > 3) OU (NOME = DIA)
d) (X div 2 > 1) ou não (A <= 12) e K

12. Quais serão os valores obtidos das expressões a seguir, supondo que A, B e C são variáveis do tipo caractere, contendo, respectivamente, "BENS"; "!" e "PARA"?

A + B + C
A + C + B
C + A + B

3. Entrada e saída de dados (I/O)

O que seria de um programa de computador que não interagisse de forma alguma com o usuário? É admissível aceitar que a razão de um programa existir seja resolver um ou mais problemas do usuário e, assim sendo, é necessário que o programa aceite dados de entrada provenientes do usuário, que os processe e exiba os resultados. Dessa forma, a grande maioria dos códigos-fontes apresentados neste livro pode ser dividida exatamente nas partes citadas: entrada de dados, processamento dos dados e saída dos resultados. Assim sendo, é razoável admitir que todo computador deve ter ao menos um dispositivo de entrada e um de saída de dados. O dispositivo padrão de saída de dados é o monitor de vídeo e o dispositivo padrão de entrada de dados é o teclado. É possível, contudo, alterar tanto a entrada de dados padrão quanto a saída, conforme será visto.

3.1 Os comandos de saída de dados (WRITE e WRITELN)

Os comandos de saída de dados WRITE e WRITELN são os principais comandos destinados a exibir todos os tipos de dados no monitor

de vídeo. A diferença entre WRITE e WRITELN reside no fato de que o comando WRITE escreve os dados passados a este e mantém o cursor na posição posterior aos dados escritos no monitor de vídeo, ao passo que o comando WRITELN insere uma quebra de linha após o último dado impresso na tela. Esses procedimentos possuem a seguinte sintaxe:

```
Write(parâmetro_1,parâmetro_2, ...,'Texto a ser impresso');
WriteLn(parâmetro_1,parâmetro_2, ...,'Texto a ser impresso');
```

Código 3.1 - Uso dos comandos WRITE e WRITELN

```
Program Exemplo;
Uses CRT;
Var
  i : integer;
  r : real;
  c : char;
  s : string[20];
Begin
  ClrScr;   (* limpa a tela e coloca o cursor em 1,1 *)
  Writeln('Exemplos de aplicação de writeln e write');
  writeln; (* apenas pula uma linha *)
  i := 100;
  r := 3.14;
  c := 's';
  s := 'é uma string';
  writeln('Valor de i é igual a ',i);
  write('valor de r = ');
  writeln(r:3:2);
  writeln(c,' ',s);
  readln;
End.
```

A execução do código 3.1 resulta no seguinte texto no monitor de vídeo:

Exemplos de aplicação de writeln e write

```
Valor de i e igual a 100
valor de r = 3.14
s é uma string
```

Para imprimir o valor de uma variável com o comando WRITE ou WRITELN, basta escrever o nome da variável com argumento do comando, ou seja, entre parênteses. Caso se deseje imprimir mais de uma variável, elas devem ser separadas por vírgulas. Para imprimir uma *string*, basta escrever a *string* dentro de aspas simples (' ').

Quando uma variável real é impressa no monitor de vídeo, ela é impressa em notação científica. Uma maneira simples para evitar que isso aconteça é o uso de dois-pontos (:) após o nome da variável, seguido de um número inteiro dentro do comando WRITE ou WRITELN. Para imprimir uma variável real x no monitor com cinco dígitos, sendo dois dos cinco dígitos à direita da vírgula, basta escrever:

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

Caso a variável x possua mais dígitos à direita da vírgula, eles não serão exibidos. Se a variável x possuir mais que três dígitos significativos à esquerda da vírgula, a tabulação especificada pelos dois-pontos não será considerada.

3.2 Os comandos de entrada de dados (READ e READLN)

Esses comandos são utilizados para leitura de dados via teclado. O comando READ lê um dado do teclado e, quando a tecla ENTER for pressionada, o dado será armazenado em uma variável, que deve ser especificada como argumento deste. O tipo do dado será então determinado pelo tipo da variável que o receberá. Cada tecla digitada é escrita no monitor de vídeo. O comando READ realiza a entrada de dados, mas após a tecla ENTER ser pressionada, o cursor permanecerá na mesma posição. Já com o comando READLN, o cursor passará para a próxima linha. Os comandos READ e READLN são análogos aos comandos WRITE e WRITELN, tanto na sintaxe quanto na função, com a exceção de que os primeiros são para entrada de dados (*input*) e os últimos, para a saída de dados (*output*). A sintaxe geral para os comandos READ e READLN é:

```
Read(Var_1, Var_2, Var_3,...);
ReadLn(Var_1, Var_2, Var_3,...);
```

Durante a digitação dos valores das variáveis solicitadas, é possível terminar a digitação de uma e começar a digitação de outra pressionando a tecla ENTER ou inserindo um espaço em branco entre o valor das duas variáveis. O último método não é aconselhável, uma vez que este só funciona corretamente para variáveis numéricas. Uma boa prática de programação é usar um comando READ ou READLN para cada variável que se deseja ler.

O código 3.2 apresenta um programa muito simples, que tem a finalidade única de imprimir na tela do computador três números inteiros digitados pelo usuário. Outro exemplo do uso dos comandos READ e READLN é mostrado no código 3.3, que solicita ao usuário a entrada de dados de vários tipos e, após a entrada deles, imprime os dados digitados de volta para o usuário.

Código 3.2 - Uso dos comandos READ e READLN

```
Program teste;
Uses CRT;
Var
  a, b, c: integer;
Begin
  clrscr;
  writeln('Digite o valor de a, b e c');
  readln(a, b, c);
  writeln (a,' ',b,' ',c);
  readln;
End.
```

Código 3.3 - Uso dos comandos READ e READLN

```
Program teste;
Uses CRT;
Var
  i : integer;
  r : real;
  c : char;
  s : string[10];
```

```
Begin
  ClrScr;
  Write('Digite um numero inteiro ------> ');
  Readln(i);
  Write('Digite um numero real ---------> ');
  Readln(r);
  Write('Digite um caractere -----------> ');
  Readln(c);
  Write('Digite uma string -------------> ');
  Readln(s);
  Writeln;Writeln;  (* pula duas linhas *)
  Writeln(i);
  Writeln(r);
  Writeln(c);
  Writeln(s);
  Readln;
End
```

Um bom exercício é digitar um valor discordante do tipo da variável, por exemplo, digitar um valor real quando for pedido um valor inteiro. Quando isso ocorrer, o próprio programa gerará um erro, sendo o código desse e de outros erros inseridos diretamente no programa pelo ambiente Turbo Pascal. O código 3.4 mostra um programa um pouco mais útil que os mostrados anteriormente, pois solicita ao usuário digitar o valor da base e da altura de um triângulo e, de posse desses valores, calcula a sua área. Esse é o primeiro programa apresentado nesse livro em que se começa a demonstrar uma mínima preocupação com a interface do programa. Um programa com uma interface ruim é, por mais funcional que seja, susceptível a ser substituído por outro (vide a revolução dos sistemas operacionais e até mesmo os *video games* atuais). No código a seguir, é mostrado um programa que utiliza uma interface textual.

Código 3.4 - Cálculo da área de um triângulo

```
Program Area_de_Triangulos;
Uses CRT;
Var
  Base, altura: Real;
```

```
Begin
  ClrScr;
  Writeln('CALCULO DA AREA DE TRIANGULOS':55);
  Writeln;
  Write('Valor da base ------> ');
  Readln(base);
  Writeln;
  Write('Valor da altura ----> ');
  Readln(altura);
  Writeln;
  Writeln;
  Writeln('Área do triângulo = ', (base*altura/2):10:2);
  Readln;
End
```

Quando se utiliza o comando READ ou READLN e não se estabelece o nome da variável dentro dos parênteses, o valor digitado pelo usuário é perdido, pois ele não é armazenado em nenhum lugar da memória do computador. A finalidade de se utilizar o comando READ ou READLN sem nenhum identificador é apenas criar uma pausa no fluxo do programa, pausa esta que só finda quando o usuário pressiona a tecla ENTER. Contudo, o comando READKEY desempenha esse papel de forma mais elegante.

3.3 O comando de entrada de dados leia tecla (READKEY)

Outra maneira de ler um dado do teclado é utilizando o comando READKEY. Esse comando lê uma tecla sem que seja necessário pressionar a tecla ENTER para indicar o fim desse dado. O comando READKEY só pode ser utilizado conforme seu próprio nome já diz, para se ler uma única tecla cada vez que ele é executado. Dessa forma, é mais aconselhável o uso desse comando para causar uma pausa na execução de um programa que o uso dos comandos READ ou READLN sem nenhum argumento.

Código 3.5 - Uso do comando READKEY

```
Program Exemplo;
Uses CRT;
Var
  tecla: char;
1Begin
  Clrscr;
  Write('Digite uma tecla qualquer ->');
  Tecla := readkey;
  Writeln;
  writeln('Você digitou a tecla ', tecla);
  Writeln;
  Writeln;
  Writeln('Pressione qualquer tecla para sair.');
  Readkey;
End.
```

3.4 Acessando a impressora local (LPT)

É possível enviar dados para a impressora[3] pelos comandos WRITE e WRITELN. Para tanto, deve-se colocar antes dos parâmetros a serem enviados à impressora a palavra reservada LST. Para utilizar essa palavra, é necessário incluir a unidade PRINTER no código-fonte do programa na seção USES. Um exemplo do seu uso é:

```
Writeln('Esta string vai para o vídeo.');
Writeln(LST,'Esta vai para a impressora'' e esta também');
```

O código 3.6 apresenta o mesmo programa do código 3.3, apenas com a diferença de que os dados digitados pelo usuário serão enviados diretamente para a impressora local.

[3] Os procedimentos aqui mostrados são válidos apenas para impressoras conectadas na porta LPT1. Para usar uma impressora ligada em uma porta USB, pode-se criar uma porta USB pelo comando NET USE no MS DOS da seguinte forma: ***net use lpt1: \\nome_pc\nome_impressora /persistent:yes***. Em que nome_pc é nome de rede da máquina que hospeda a impressora USB e nome_impressora é o nome do compartilhamento na rede da impressora.

Código 3.6 - Uso da impressora para a saída de dados

```
Program impressora;
Uses CRT, PRINTER;
Var
  i : integer;
  r : real;
  c : char;
  s : string[10];
Begin
  ClrScr;
  Write('Digite um numero inteiro ------> ');
  Readln(i);
  Write('Digite um numero real ---------> ');
  Readln(r);
  Write('Digite um caractere -----------> ');
  Readln(c);
  Write('Digite uma string -------------> ');
  Readln(s);
  Writeln(LST,'Os valores digitados pelo usuário foram: ')
  Writeln(LST,i);
  Writeln(LST,r);
  Writeln(LST,c);
  Writeln(LST,s);
  Readln;
End.
```

3.5 Funções e procedimentos para controle de vídeo

O comando CLRSCR (do inglês *CLeaR SCReen*) tem a finalidade de limpar a tela de vídeo e colocar o cursor na primeira coluna da primeira linha. A tela de vídeo é dividida em 80 colunas e 25 linhas, sendo o canto superior esquerdo de coordenadas (1,1) e o canto inferior direito de coordenadas (80,25).

A função GOTOXY(X, Y) move o cursor para a coluna X e linha Y da tela de vídeo, sendo X um número inteiro pertencente ao intervalo [1..80] e Y, também inteiro, pertencente ao intervalo [1..25]. O código 3.7 mostra um exemplo do uso da função GOTOXY para imprimir mensagens em lugares diferentes da tela.

Código 3.7 - Uso da função do GOTOXY

```
Program Exemplo_GOTOXY;
Uses CRT;
Var
  x,y : Byte;
Begin
  ClrScr;
  Gotoxy(10,2);
  Write('Coluna 10 da linha 2');
  x := 40;
  y := 10;
  Gotoxy(x,y);
  Write('Coluna 40 da linha 10');
  Readkey;
End
```

O comando CLREOL (do inglês *CLeaR End Of Line*) limpa a linha desde a posição atual do cursor até o final da linha onde este se encontra. O comando DELLINE (do inglês *DELete LINE*), por sua vez, elimina toda a linha em que está o cursor. As linhas posteriores sobem, ocupando o lugar da linha que foi eliminada. Uma vez que a linha é eliminada, o cursor permanece no início da linha que foi apagada.

Código 3.8 - Uso do comando DELLINE

```
Program exemplo;
Uses CRT;

Begin
  ClrScr;
  Writeln('linha 1');
  Writeln('linha 2');
  Writeln('linha 3');
  Writeln('linha 4');
  Gotoxy(1,2); (*posiciona o cursor no início da linha 2*)
  Delline;
  Readkey;
End.
```

A compilação do código 3.8 imprime o seguinte texto no monitor de vídeo:

```
linha 1
linha 3
linha 4
```

Como a string 'linha 2' foi eliminada, o cursor ficará posicionado no início da string 'linha 3'.

O comando INSLINE (do inglês *INSert LINE*) faz exatamente o contrário do procedimento DELLINE, inserindo uma nova linha na posição atual do cursor. O código 3.9 apresenta um exemplo do uso do comando INSLINE.

Código 3.9 - Uso do comando INSLINE

```
Program Exemplo;
Uses CRT;
Begin
  ClrScr;
  Writeln('linha 1');
  Writeln('linha 2');
  Writeln('linha 3');
  Writeln('linha 4');
  Gotoxy(1,3);    (* cursor na 1a. coluna da 3a. linha *)
  InsLine;
  Write('teste');
  Readkey;
End
```

A compilação do código 3.9 imprime o seguinte texto no monitor de vídeo:

```
linha 1
linha 2
teste
linha 3
linha 4
```

Em se tratando de dar ênfase, ou destaque a um texto exibido na tela, o comando TEXTBACKGROUND é usado para selecionar a cor de fundo sobre o qual o texto será escrito com os comandos WRITE ou WRITELN. Sua sintaxe geral é:

```
TextBackGround(cor);
```

O argumento cor usado na função TEXTBACKGROUND acima pode ser um número inteiro ou o próprio nome da cor em inglês, conforme mostrado na Tabela 3.1. Os nomes das cores são palavras reservadas da linguagem Pascal.

Tabela 3.1 - Tabela de cores passíveis de utilização com o comando TEXTBACKGROUND

Número de referência	Cores	
	Em Inglês	Em Português
0	Black	Preto
1	Blue	Azul
2	Green	Verde
3	Cyan	Ciano
4	Red	Vermelho
5	Magenta	Magenta
6	Brown	Marrom
7	Light Gray	Cinza-claro
8	Dark Gray	Cinza-escuro
9	Light Blue	Azul-claro
10	Light Green	Verde-claro
11	Light Cyan	Ciano-claro
12	Light Red	Vermelho-claro
13	Light Magenta	Magenta-claro
14	Yellow	Amarelo
15	White	Branco
16	Blink	Pisca-pisca

Código 3.10 - Uso do comando TEXTBACKGROUND

```
Program Exemplo;
Uses CRT;
Begin
  ClrScr;
  WriteLn('teste');
  TextBackGround(7);
  Writeln('teste');
  TextBackGround(Brown);
  Writeln('teste');
  Readkey;
End
```

A compilação do código 3.10 imprime o seguinte texto no monitor de vídeo:

```
teste
teste
teste
```

Outro comando também usado para dar ênfase a um texto é o comando TEXTCOLOR que permite selecionar a cor com que o texto será impresso no monitor. Sua sintaxe geral é:

```
TextColor(cor);
```

Código 3.11 - Uso do comando TEXTCOLOR

```
Program Exemplo;
Uses CRT;
Begin
  Clrscr;
  TextBackGround(7);
  TextColor(black);
  writeln('teste');
  TextColor(black+blink);
  write('teste');
  Readkey;
End
```

A compilação do código 3.11 imprime na primeira linha do monitor de vídeo a palavra teste na cor preta. Já na segunda linha, a palavra teste aparecerá piscando (em inglês *blink*).

Em alguns programas é interessante criar uma janela, de modo a permitir ao usuário interagir com uma parte específica do programa e não com o todo. A função WINDOW tem o poder de definir uma janela de texto cujo canto superior esquerdo tem coordenadas (x_1, y_1) e o canto inferior direito tem coordenadas (x_2, y_2). Após esse comando, comandos tais como CLRSCR, WRITE e READ agem somente dentro da janela recém-definida. A instrução GOTOXY passa a utilizar como origem o ponto x_1, y_1, que passa a ser considerado como de coordenadas 1,1. Quando uma nova janela é definida, cria-se um novo sistema de coordenadas inscrito dentro do anterior, cuja origem é x_1, y_1. Sua sintaxe geral é:

```
Window(x1,y1,x2,y2);
```

Código 3.12 - Uso da função WINDOW

```
Program Exemplo;
Uses CRT;
Begin
  Writeln('Programa teste do uso do comando WINDOW':40);
  Window(10,10,70,20);
  ClrScr;            (* limpa somente a janela *);
  Writeln('teste');    (* escreve 'teste' em 10,10 *)
  Readkey;
End.
```

Em alguns casos, é importante conhecer a posição atual do cursor na tela. Para isso, existem duas funções específicas. A função WHEREX retorna à coluna onde o cursor está posicionado e a função WHEREY retorna à linha. Como a tela do computador tem dimensões 80 x 25, a função WHEREX retorna um número inteiro pertencente ao intervalo [1..80] e a função WHEREY, um número inteiro pertence ao intervalo [1..25].

Uma operação muito comum consiste em solicitar ao usuário que pressione uma tela para que o programa possa continuar a sua operação normal

ou mesmo encerrar a suas atividades. O identificador KEYPRESSED é uma função especial do Turbo Pascal que retorna um valor lógico TRUE se uma tecla foi pressionada, ou FALSE, caso contrário. Ela é utilizada para detectar teclas pressionadas no teclado.

Código 3.13 - Uso da função KEYPRESSED

```
Program Exemplo;
Uses CRT;
Begin
  ClrScr;
  Write('Pressione uma tecla -> ');
   Repeat until Keypressed; (* repita até que uma tecla seja pressionada. O comando
Repeat Until será estudado mais adiante *)
End.
```

3.6 Exercícios

1. Faça um programa que imprima na tela do computador a mensagem: "***Primeiro programa escrito no ambiente Turbo Pascal***".

2. Faça um programa que leia dois números inteiros digitados pelo usuário e calcule a sua soma.

3. Faça um programa que leia os coeficientes *a*, *b* e *c* e calcule as raízes reais de uma equação do segundo grau, sabendo-se que as raízes são dadas por:

$$x_1 = \frac{-b + \sqrt{b^2 - 4.a.c}}{2a} \quad \text{e} \quad x_2 = \frac{-b - \sqrt{b^2 - 4.a.c}}{2a}$$

Nota: A função que extrai a raiz quadrada de um número real positivo é a função SQRT.

4. Construa um programa que leia as dimensões dos lados de um retângulo. Calcule e imprima a área e perímetro desse retângulo.

5. Faça um programa, leia o raio e calcule a área de um círculo e o comprimento de uma circunferência que possa ter esse raio. Adote π = 3,141592. Dados:

$$A = \pi.r^2 \quad \text{e} \quad C = 2.\pi.r$$

6. Elabore um programa cujos dados sejam dois lados de um triângulo retângulo, calcule a respectiva hipotenusa.

7. Construa um programa que leia os valores das bases e altura de um trapézio, calcule e imprima o valor da sua área.

8. Faça um programa, leia o raio de uma esfera e calcule o seu volume e a sua área superficial. Dados:

$$V_{Esfera} = \frac{4}{3}.\pi.r^3 \quad \text{e} \quad A_{Esfera} = 4.\pi.r^2$$

9. Faça um programa que leia o raio da base e a altura de um cone e calcule o seu volume e a sua área superficial. Dados:

$$V_{Cone} = \frac{\pi}{3}.r^2.h \quad \text{e} \quad A_{Cone} = \pi.r^2 + \pi.r.\sqrt{r^2 + h^2}$$

10. Faça um programa que leia o raio da base e a altura de um cilindro e calcule o seu volume e a sua área superficial. Dados:

$$V_{Cilindro} = \pi.r^2.h \quad \text{e} \quad A_{Cone} = 2.\pi.r^2 + 2.\pi.r.h$$

11. Faça um programa que leia uma temperatura dada na escala Celsius (C) e imprima o equivalente em Fahrenheit (F). Dados:

$$F = \frac{9}{5}C + 32$$

12. Faça um programa que leia um comprimento em polegadas e imprima o equivalente em milímetros, sabendo-se que uma polegada equivale a 25,4 mm.

13. O custo ao consumidor de um carro novo é a soma do custo de fábrica com a percentagem do distribuidor e dos impostos, ambos aplicados ao custo de fábrica. Supondo que a percentagem do distribuidor seja de 12% e a dos impostos, de 45%, crie um programa para ler o custo de fábrica do carro e imprimir o custo ao consumidor.

14. Uma companhia de carros paga a seus empregados um salário de R$ 510,00 por mês mais uma comissão de R$ 50,00 para cada carro vendido e mais 5% do valor da venda.

Todo mês, a companhia prepara os seguintes dados para cada vendedor: número de carros vendidos e o valor total das vendas. Elabore um programa para calcular e imprimir o salário do vendedor num dado mês.

15. Faça um programa que leia o salário bruto de um funcionário e calcule o seu salário líquido, sabendo-se que o imposto a ser descontado é de 5% sobre o salário bruto.

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

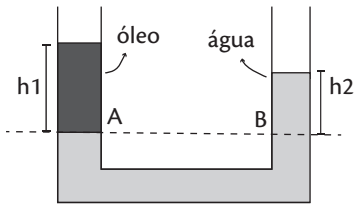
17. Elabore um programa que leia o primeiro termo de uma Progressão Aritmética (PA), sua razão, um número N e, a seguir, calcule e imprima o n-ésimo termo da PA.

18. Elabore um programa que leia o primeiro termo de uma Progressão Geométrica (PG), sua razão, um número N e, a seguir, calcule e imprima o n-ésimo termo da PG.

19. Faça um programa que leia as coordenadas x e y de dois pontos $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$, calcule e imprima a distância entre esses dois pontos, cujo valor é dado pela seguinte fórmula:

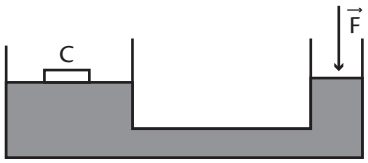
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

20. Água e óleo são colocados em um sistema de vasos comunicantes, como mostra a figura. Elabore um programa que leia a altura h1 (cm) da coluna de óleo e calcule e imprima a altura h2 da coluna de água medida acima do nível de separação entre os líquidos.



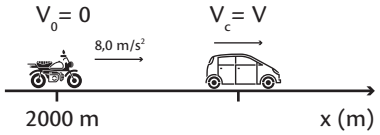
$h_1 = d_2 . h_2 / d_1$
h_1 → altura do óleo (cm)
h_2 → altura da água (cm)
d_1 → densidade do óleo (0,8 g/cm³)
d_2 → densidade da água (1,0 g/cm³)

21. A figura esquematiza uma prensa hidráulica. Uma força (F) é exercida no pistão de área A, para se erguer uma carga C no pistão maior de área 5A. Faça um programa que leia o valor da carga (C) e calcule e imprima o valor da força (F).



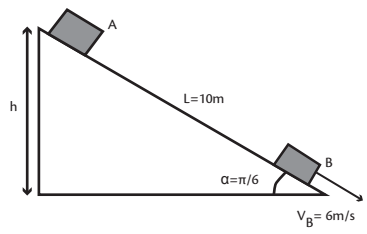
$F = C . A_2 / A_1$
A_1 → área do pistão maior (m^2)
A_2 → área do pistão menor (m^2)
C → carga do pistão maior (N)
F → força exercida no pistão menor (N)

22. Um policial rodoviário encontrava-se de moto, parado no quilômetro 2 de uma estrada retilínea, quando passou por ele um carro a uma velocidade V (km/h) maior que a permitida para o trecho. O policial levou 10 segundos para ligar a moto e iniciar a perseguição atrás do infrator. O movimento da motocicleta do policial foi retilíneo, uniformemente acelerado (com aceleração de a = 8 m/s2) e o do carro foi retilíneo uniforme durante toda perseguição. Considere t = 0 segundos o instante em que o policial iniciou seu movimento. Faça um programa que leia o valor da velocidade V (km/h) do carro e determine o tempo gasto para que o policial alcance o infrator.



$a.t^2 / 2 - V t - V t_i = 0$
a →aceleração da moto (m/s^2)
t → tempo gasto para o policial alcançar o infrator (s)
V → velocidade do carro do infrator (m/s)
t_i → tempo gasto para o policial ligar a moto e iniciar a perseguição (s)

23. Um corpo de massa M desce uma rampa de comprimento L = 10 m, inclinada de π/6 em relação à horizontal, chegando à base do plano com velocidade de 6 m/s. Sabendo-se que o corpo partiu do repouso, faça um programa que leia o valor da massa do corpo (kg) e determine o trabalho realizado pela força de atrito (de A até B).



$W_{AB} = M*(g*h - (V_B)^2/2)$
W_{AB} é o trabalho realizado pela força de atrito (de A até B)
g = $10 m/s^2$ (aceleração gravitacional)
V = 6m/s (velocidade em B)
M é a massa do corpo
h = L*sen α (seno é calculado pela função sin(x) do pascal, x em radianos)

4. Instruções de fluxo de programa

Em todos os códigos-fonte apresentados até aqui, os programas começavam no seu início e eram executados linha a linha, sequencialmente, até o seu término. A maioria dos programas de computador não funciona assim. Eles possuem estruturas de repetição (em inglês *loops*)[4] , saltam de uma seção para outra e não terminam até que o usuário lhe dê um comando específico. Tais estruturas permitem ampliar o poder de resolução de problemas de uma linguagem de programação, sendo talvez as mais importantes estruturas de uma linguagem.

Programas escritos em Pascal têm seu início na primeira instrução após a palavra reservada BEGIN, que marca o início do programa e executam instrução após instrução até atingirem o END, que termina o programa. Isso é verdade, a menos que uma ou mais instruções alterem o fluxo normal

[4] *Loop* é uma expressão em inglês, sem boas traduções, que significa uma estrutura de repetição na qual, após uma série de comandos serem executados, o fluxo do programa volta ao ponto onde este começou, como se estivesse se movendo sobre um círculo.

do programa. As instruções dessa categoria são denominadas de *instruções de fluxo de programa*, apesar de essa não ser uma expressão oficial. Tais instruções também são conhecidas como estruturas de controle.

Uma instrução de fluxo de programa controla a ordem na qual outras instruções são executadas. Essas instruções podem interromper o programa, tomar decisões, escolher elementos de um conjunto, de acordo com uma ou mais condições, e repetir blocos de comandos. Com uma instrução de fluxo de programa pode-se escrever programas que rodem até que um determinado evento planejado ocorra ou até que o usuário deseje que o programa encerre sua atividade.

4.1 A estrutura tomadora de decisões Se Então Senão (IF-THEN-ELSE)

O comando *IF* permite ao programa tomar decisões. Sua sintaxe geral é:

```
If (expressão_lógica) Then comando;
```

Em português deve ser lido como: **Se** a *expressão lógica* for verdadeira, **então** execute este *comando*. A expressão lógica utilizada pode ser simples ou complexa. Se o resultado da expressão lógica for verdadeiro (TRUE), então o comando será executado, caso contrário, não. Para os casos em que tivermos mais de um comando a ser executado, estes devem ser delimitados pelas palavras BEGIN e END, demarcando, assim, um bloco de comandos em Pascal[5]. O exemplo abaixo é semelhante ao anterior, porém um bloco de comandos foi criado e será executado somente quando a expressão lógica da estrutura SE for verdadeira. Quando isso ocorrer, todos os comandos dentro do bloco serão executados e, em caso contrário (se a expressão lógica for falsa), nenhum deles será executado.

[5] Sempre que quisermos demarcar um bloco de comandos em Pascal, devemos começar o bloco com a palavra reservada BEGIN e terminá-lo com a palavra END.

```
If (expressão_lógica) Then
 Begin
  Comando_1;
  Comando_2;
  Comando_3;
  ...
 End;
```

É possível tratar também os casos quando a expressão lógica for falsa. Para tal, pode-se complementar o comando IF com o comando ELSE, que pode ser traduzido como *senão,* em português. A estrutura IF/ELSE tem a sua tradução para o português como:

```
SE (expressão_lógica for VERDADEIRA) ENTÃO EXECUTE ...
SENÃO EXECUTE ...
```

A sintaxe geral da estrutura IF/ELSE é:

```
If (expressão_lógica) Then comando_1
Else comando_2;
```

No caso anterior, o comando_1 será executado apenas se a expressão lógica for verdadeira. Caso ela seja falsa, então o comando_2 será executado. Atenção redobrada deve ser tomada para **não** utilizar o símbolo ponto e vírgula (;) no final do comando_1. Uma vez que o ponto e vírgula é usado para marcar o final de uma instrução, ele não pode ser utilizado antes do comando ELSE em razão de esse comando pertencer à estrutura IF/ELSE. Caso seja colocado um ponto e vírgula antes do comando ELSE, o compilador acusará um erro ao compilar o programa. Os dois erros mais comuns, ao se programar em Pascal, são: a falta de pontos e vírgulas no final das instruções e o uso do mesmo símbolo antes do comando ELSE. Quando é necessário usar blocos de comandos na estrutura IF/ELSE, deve-se proceder da seguinte forma:

```
If (expressão_lógica) Then
Begin
  Comando_1;
  Comando_2;
```

```
  ...
End       (* NÃO SE USA ; ANTES DO COMANDO ELSE!!!*)
Else
Begin
  Comando_3;
  Comando_4;
  ...
End;
```

O código 4.1 apresenta um exemplo do uso da estrutura IF/ELSE para verificar se um número inteiro digitado pelo usuário é, ou não é, maior que 100.

Código 4.1 - Uso da estrutura IF/ELSE

```
Program Exemplo;
Uses CRT;
Var
  i: Integer;
Begin
  Clrscr;
  Write('Digite um inteiro maior que 100 --> ');
  Readln(i);
  Writeln;
  Writeln;
  If (i>100) Then Writeln('Você conseguiu')
  Else Writeln(i,' não é maior que 100');
  Readkey;
End.
```

O código 4.2 apresenta um programa que determina o maior entre dois números lidos do teclado. Nesse código, aparece o comando ELSE IF, que é semelhante ao IF e funciona como uma segunda opção de escolha. Dessa forma, é possível realizar várias tomadas de decisão em sequência.

Código 4.2 - Uso da estrutura IF-ELSE IF-ELSE

```
Program Exemplo;
Uses CRT;
Var
```

```
  Numero_1, Numero_2: Integer;
Begin
  ClrScr;
  Write('Primeiro número ----> ');
  Readln(Numero_1);
  Write('Segundo número -----> ');
  Readln(Numero_2);
  Writeln;
  If (Numero_1 > Numero_2) Then
    Write(Numero_1,' > ', Numero_2)
  Else If (Numero_2 > Numero_1) Then
    Writeln(Numero_2,' > ', Numero_1)
  Else
    Writeln('Os números digitados são iguais!');
  Readkey;
End.
```

O código 4.3 apresenta um programa que coloca em ordem decrescente três números inteiros lidos do teclado.

Código 4.3 - Programa que coloca três números digitados em ordem decrescente

```
Program Exemplo;
Uses CRT;
Var
  x, y, z: Integer;
Begin
  ClrScr;
  Write('Primeiro número --> ');
  Readln(x);
  Write('Segundo número ---> ');
  Readln(y);
  Write('Terceiro número --> ');
  Readln(z);
  Writeln;
  If (x >= y) Then
    If (x >= z) Then
      If (y >= z) Then Writeln(x,' ',y,' ',z)
      Else Writeln(x,' ',z,' ',y)
    Else Writeln(z,' ',x,' ',y)
  Else If (y >= z) Then
```

```
    If (x >= z) Then Writeln(y,' ',x,' ',z)
    Else Writeln(y,' ',z,' ',x)
  Else Writeln(z,' ',y,' ',x);
End.
```

4.2 Rótulos (LABELS) e o comando vá para (GOTO)

A instrução GOTO permite desviar a sequência de execução do programa para um determinado rótulo (LABEL) predefinido. Para utilizarmos um rótulo, é necessário que este tenha sido declarado na subárea LABEL, no início do programa.

Embora existam programadores que ainda insistam no uso do comando GOTO, é desaconselhável todo e qualquer tipo de uso desse comando, que, além de prejudicar a inteligibilidade do código-fonte, ainda faz com que este se torne menos eficiente. O uso do GOTO remonta a épocas em que as linguagens de programação (normalmente de baixo nível) não possuíam outras instruções para realizar desvios no fluxo do programa.

Nas linguagens de programação de alto nível, existem estruturas que suplantam o uso do GOTO sem as suas debilidades. Essa seção deve ser lida mais como uma curiosidade do que como uma técnica a ser aprendida.

O código 4.4 mostra um programa que coloca três números digitados pelo usuário em ordem decrescente e utiliza o comando GOTO para evitar o seu encerramento, enquanto o usuário assim o desejar. Nesse código, o comando GOTO é usado em uma estrutura semelhante a um *loop* para manter o programa em execução enquanto o usuário desejar. Contudo, o comando GOTO não é uma estrutura de repetição, por definição.

Código 4.4 - Uso do comando GOTO

```
Program Exemplo;
Uses CRT;
Label
  Inicio;
Var
  x, y, z: Integer;
  tecla: Char;
```

```
Begin
Inicio:
  ClrScr;
  Write('Primeiro número --> ');
  Readln(x);
  Write('Segundo número ---> ');
  Readln(y);
  Write('Terceiro número --> ');
  Readln(z);
  Writeln;
  If (x >= y) Then
    If (x>=z) Then
      If (y >= z) Then Writeln(x,' ',y,' ',z)
      Else Writeln(x,' ',z,' ',y)
    Else Writeln(z,' ',x,' ',y)
  Else If (y >= z) Then
    If (x >= z) Then Writeln(y,' ',x,' ',z)
    Else Writeln(y,' ',z,' ',x)
  Else Writeln(z,' ',y,' ',x)b;
  Writeln;
  Write('Deseja Continuar --> ');
  Tecla := Readkey;
  If ((Tecla = 'S') OR (Tecla = 's')) Then Goto Inicio;
End.
```

O código 4.5 determina se três números digitados pelo usuário formam os lados de um triângulo. Para verificar se três valores quaisquer (x, y e z) formam os lados de um triângulo, basta verificar se (x < y + z) e (y < x + z) e (z < x + y).

Se os valores x, y e z formarem um triângulo, pode-se verificar qual o tipo dele, seguindo-se os passos:

1. Se x = y = z, então o triângulo é equilátero.
2. Se x = y ou x = z ou y = z, então o triângulo é isósceles.
3. Se x <> y <> z, então o triângulo é escaleno.

Código 4.5 - Programa para averiguação se três números reais são lados de triângulo

```
Program Exemplo;
Uses CRT;
```

```
Label
  INICIO;
Var
  x, y, z: Real;
  Tecla: Char;
Begin
INICIO:
  ClrScr;
  Write('X = ');
  Readln(x);
  Write('Y = ');
  Readln(y);
  Write('Z = ');
  Readln(z);
  Writeln;
  If ((x < y + z) and ( y < x + z) and (z < x + y)) Then
    If ((x = y) and (x = z)) Then
      Writeln('TRIANGULO EQUILATERO')
    Else If ((x = y) Or (x = z) Or (y = z)) Then
      Writeln('TRIANGULO ISOCELES')
    Else Writeln('TRIANGULO ESCALENO')
  Else Writeln('X,Y,Z NAO SAO LADOS DE UM TRIANGULO');
  Writeln;
  Write('Deseja Continuar ? --> ');
  Tecla := ReadKey;
  If (Tecla='s') Or (Tecla='S') Then Goto INICIO;
End.
```

4.3 O laço de repetição Para (FOR)

Quando se sabe, ou pode-se calcular previamente, o número de vezes que um bloco de instruções deve ser executado, o uso do laço (também chamado de estrutura de repetição ou, em inglês, *loop*) FOR é normalmente a melhor escolha. A sintaxe geral do laço FOR é:

```
For (variável := valor_inicial) to/downto (valor_final) do
  comando;
```

A variável de controle do laço deve ser, obrigatoriamente, do tipo ordinal (inteiro, caractere ou lógica). À variável de controle será adicionada, ou

subtraída, uma unidade a cada iteração do laço até que a variável de controle atinja o valor final preestabelecido para o término do laço. A palavra reservada TO, após a atribuição do valor inicial à variável de controle, indica que o laço FOR será crescente, ao passo que a palavra DOWNTO indica que o laço será decrescente. O laço FOR termina quando a variável de controle atinge o valor final especificado na declaração do laço. O código 4.6 apresenta um programa que usa o laço FOR para imprimir no monitor de vídeo uma contagem progressiva de 0 até 15.

Código 4.6 - Uso do loop FOR para contar de 0 a 15

```
Program Exemplo_FOR_1;
Uses CRT;
Var
  i: Integer;
Begin
  ClrScr;
  For i := 0 to 15 do Writeln(i);
(* para i igual a 0 até 15 faça escreva i *)
  Readkey;
End.
```

O código 4.7 apresenta a contagem inversa à apresentada no código 4.6, ou seja, um programa que usa um laço FOR para imprimir na tela uma contagem regressiva de 15 até 0.

Código 4.7 - Uso do loop FOR para contar de 15 a 0

```
Program Exemplo_FOR_2;
Uses CRT;
Var
  i: Integer;
Begin
  ClrScr;
  For i := 15 downto 0 do Writeln(i);
  Readkey;
End.
```

O código 4.8 apresenta um programa que usa um laço FOR para escrever os quadrados dos números inteiros de 1 até 20.

Código 4.8 - Uso do loop FOR para calcular os quadrados de 1 a 200

```
Program Exemplo_FOR_3;
Uses CRT;
Var
  i: Integer;
Begin
  ClrScr;
  For i:=1 to 20 do
   Begin
    Write('Valor de i --> ');
    Write(i:3);
    Write('..... quadrado de i = ');
    Writeln(SQR(i):5);
   End;
  Readkey;
End.
```

Note que no código 4.8 é apresentada a função SQR (do inglês *SQuaRe*), que retorna o quadrado de um número inteiro ou real. A função que extrai a raiz quadrada de um número qualquer positivo é a função SQRT (do inglês *SQuare RooT*).

O código 4.9 apresenta um programa que usa o laço FOR para calcular a soma de todos os números inteiros compreendidos em um intervalo, cujos limites foram informados pelo usuário. Quando se deseja calcular um somatório (Σ), deve-se recorrer ao laço FOR.

Código 4.9 - Programa para cálculo da soma dos números inteiros de um intervalo

```
Program Exemplo_FOR_4;
Uses CRT;
Var
  i, a, b, soma: Integer;
Begin
  ClrScr;
```

```
  Write('Primeiro Numero --> ');
  Readln(a);
  Write('Segundo Numero ---> ');
  Readln(b);
  Writeln;
  Soma:=0;
  For i := a to b do Soma := Soma + i;
  Writeln('A Soma dos números inteiros entre ',a,' e ',b,' = ',soma);
  ReadKey;
End.
```

O código 4.10 apresenta um programa que calcula o fatorial de um número inteiro informado pelo usuário. Sabe-se que fatorial de um número é definido como:

$$n! = n.(n-1).(n-2)...2.1 \quad \text{sendo que} \quad 1! = 1 \text{ e } 0! = 1$$

Código 4.10 - Programa para cálculo do fatorial de um número

```
Program Exemplo_FOR_5;
Uses CRT;
Var
  n, fat, i : Integer;
Begin
  Clrscr;
  Write('N = ( menor que 0 = fim) --> ');
  Readln(n);
  If n < 0 then Exit;
  fat := 1;
  If (n > 0) Then
    For i := n downto 1 do
      fat := fat * i;
   Writeln('Fatorial de ':30,n,' = ',fat);
   Readkey;
End.
```

No código 4.10 é apresentado o procedimento EXIT, que provoca o término do programa. Se o usuário digitar um valor menor que zero para o cálculo do fatorial, o programa encerrará a sua execução. Um programa útil

é mostrado no código 4.11, que utiliza o loop FOR para exibir no monitor de vídeo a tabela ASCII do computador.

Código 4.11 - Programa para a exibição da tabela ASCII do computador

```
Program ascii;
Uses CRT;
Var
  i: byte;
Begin
  clrscr;
  Writeln;Writeln;
  for i := 0 to 254 do
   Begin
    TextColor(Red);
    Write(i:7);
    TextColor(White);
    Write(' = ', chr(i));
    if ((i MOD 7) = 0) then Writeln;
    if (i = 133) then
     Begin
      Writeln;Writeln;
      Write('Digite qualquer tecla para continuar...');
      Readkey;
      clrscr;
      Writeln;Writeln;
     End;
   End;
  readkey;
End.
```

4.4 O laço de repetição Repita Enquanto (REPEAT-UNTIL)

Esse laço repete um bloco de instruções até que uma condição seja satisfeita. Sua sintaxe geral é:

```
Repeat
  Comando_1;
  Comando_2;
  Comando_3;
  ...
Until (expressão_lógica);
```

Todos os comandos entre as palavras reservadas REPEAT e UNTIL serão executados até que a expressão lógica seja avaliada como verdadeira (TRUE) e, quando isso ocorrer, o laço encerrará a sua atividade. Deve-se ter o cuidado para que a expressão lógica torne-se verdadeira em algum momento do laço, pois, caso contrário, teremos um *laço infinito*. E quando um loop infinito ocorre, o programa fica preso dentro de uma estrutura para *sempre*. O código 4.12 apresenta um programa, escreve na tela uma contagem de 1 a 10 usando o laço de repetição REPEAT-UNTIL.

Código 4.12 - Uso do laço REPEAT-UNTIL para contar de 0 a 10

```
Program Exemplo_REPEAT_UNTIL_1;
Uses CRT;
Var
  i: Integer;
Begin
  ClrScr;
  i := 1;
  Repeat
    Writeln(i);
    i := i + 1;
  Until i = 10;
  Readkey;
End.
```

Como o teste da expressão lógica que controla o laço REPEAT-UNTIL é feito no final do laço, este sempre será executado ao menos uma vez. Essa característica do laço REPEAT-UNTIL é útil em diversos casos, mas pode levar a erros em outros.

O código 4.5 apresenta um programa que utiliza o comando GOTO com a finalidade única de criar um artifício que permita ao usuário encerrar, ou não, o programa. Uma alternativa ao uso do GOTO para essa finalidade é a utilização do laço REPEAT-UNTIL, como pode ser visto no código 4.13, que mostra o mesmo programa do código 4.5, apenas com a substituição do comando GOTO pelo laço REPEAT-UNTIL.

Código 4.13 - Alternativa ao uso do comando GOTO como loop principal

```
Program Exemplo_REPEAT_UNTIL_2;
Uses CRT;
Var
  x, y, z: Real;
  Tecla: Char;
Begin
 Repeat
   ClrScr;
   Write('X = ');
   Readln(x);
   Write('Y = ');
   Readln(y);
   Write('Z = ');
   Readln(z);
   Writeln;
   If ((x < y + z) and ( y < x + z) and (z < x + y)) Then
     If ((x = y) and (x = z)) Then
       Writeln('TRIANGULO EQUILATERO')
     Else If ((x = y) Or (x = z) Or (y = z)) Then
       Writeln('TRIANGULO ISOCELES')
     Else Writeln('TRIANGULO ESCALENO')
   Else Writeln('X,Y,Z NAO SAO LADOS DE UM TRIANGULO');
   Writeln;
   Write('Deseja Continuar ? --> ');
   Tecla := ReadKey;
 Until ((Tecla <> 's') and (Tecla <> 'S'));
End.
```

4.5 O laço de repetição Enquanto Faça (WHILE-DO)

O laço WHILE-DO não permite controlar o número de vezes que uma instrução (ou um bloco de instruções) será executada, como acontece no laço FOR. Esse laço assemelha-se ao laço REPEAT-UNTIL, executando um bloco de instruções enquanto uma determinada condição for verdadeira.

A diferença entre o laço WHILE-DO e do laço REPEAT-UNTIL deve-se ao fato de a avaliação da expressão lógica no laço WHILE-DO ocorrer em seu início, ao passo que, no laço REPEAT-UNTIL, o teste é realizado no seu fim.

Dessa forma, se a expressão lógica do laço WHILE-DO for falsa, os seus comandos não serão executados, o que não ocorre com o laço REPEAT-UNTIL, uma vez que, necessariamente, este será executado, mesmo quando inicialmente a sua expressão lógica for falsa. A sintaxe geral do laço WHILE-DO é:

```
While (expressão_lógica) Do
Begin
  comando_1;
  comando_2;
  ...
End;
```

Outra diferença entre os laços WHILE-DO e REPEAT-UNTIL é o fato de que o laço REPEAT-UNTIL não precisa ter o seu bloco de instruções delimitado pelas palavras reservadas BEGIN/END, o que não acontece com os laços WHILE-DO e FOR.

O código 4.14 apresenta um programa que escreve na tela uma contagem de 0 a 10, usando a estrutura WHILE-DO, semelhante ao exibido no código 4.12.

Código 4.14 - Uso do laço WHILE-DO para contar de 0 a 10

```
Program Exemplo_WHILE_DO_1;
Uses CRT;
Var
  i: Integer;
Begin
  ClrScr;
  i := 0;
  While (i < 11) Do
   Begin
    Writeln(i);
    i := i + 1;
   End;
  Readkey;
End.
```

4.6 A estrutura de seleção condicional Escolha (CASE)

A estrutura CASE permite selecionar uma opção de uma lista de opções, baseado no valor de uma variável ou expressão. Sua sintaxe geral é:

```
Case (expressão ou variável) of
valor 1) :
Begin
  comando_1;
  comando_2;
  . . .
End;
(valor 2) :
Begin
  comando_3;
  comando_4;
  . . .
End;
. . .
(valor n) :
Begin
  comando_m;
  comando_n;
  . . .
End;
Else
Begin
  comandos;
  . . .
End;
End;
```

A expressão ou variável de controle da estrutura CASE deve ser do tipo caractere inteiro ou um intervalo de valores (conjunto) desses dois tipos de dados. Após a avaliação da expressão, seu valor é comparado com os valores discriminados como rótulos (cada valor discriminado na estrutura CASE é um rótulo válido apenas para a estrutura CASE onde o rótulo foi declarado). Se o resultado da expressão for igual a algum rótulo, serão executados os comandos pertencentes àquele rótulo. Quando o resultado da expressão

for diferente dos rótulos declarados, então os comandos discriminados na cláusula ELSE serão executados (de forma análoga à estrutura IF/ELSE IF/ELSE).

O código 4.15 apresenta um programa que calcula a soma, subtração, multiplicação ou divisão de dois números digitados pelo usuário. Para realizar a escolha de qual operação será realizada, é usada a estrutura CASE.

Código 4.15 - Uso da estrutura CASE

```
Program Exemplo_CASE_1;
Uses CRT;
Var
  oper: Char;
  x, y: Real;
Begin
  ClrScr;
  Write('Valor de X = ');
  Readln(x);
  Write('Valor de Y = ');
  Readln(y);
  Writeln;
  Write('Operacao --> ');
  oper := ReadKey;
  Writeln(oper);Writeln;
  Case oper of
   '+' : Write('X + Y = ':10,x+y:6:2);
   '-' : Write('X – Y = ':10,x-y:6:2);
   '*' : Write('X * Y = ':10,x*y:6:2);
   '/' : Write('X / Y = ':10,x/y:6:2);
   Else  Writeln(oper,'  não é uma operação válida!');
  End;                    (* End do CASE *)
  Readkey;
End.                                (* End do programa *)
```

O código 4.16 apresenta um programa para o cálculo da área do triângulo, quadrado, retângulo e círculo. Apesar de muito simples, esse programa ilustra a aplicação mais usada da estrutura CASE, que consiste na geração de menus de escolha.

Código 4.16 - Programa para o cálculo da área de figuras planas

```
Program Exemplo_CASE_2;
Uses CRT;
Var
  escolha, continua: Char;
  x, y: real;
Begin
  Repeat
    ClrScr;
    Write('Calculo de área de figuras':53);
    Gotoxy(25, 5);Write('1 – Sair do programa');
    Gotoxy(25, 7);Write('2 – Triangulo');
    Gotoxy(25, 9);Write('3 – Quadrado');
    Gotoxy(25,11);Write('4 – Retângulo');
    Gotoxy(25,13);Write('5 – Circulo');
    TextBackGround(7);
    TextColor(0+16);
    Gotoxy(10,17);Write('Sua escolha ---> ');
    escolha := ReadKey;
    Textbackground(0);
    Textcolor(14);
    Case escolha of
    '2' :
    Begin
      ClrScr;
      Writeln('Calculo da área de triangulos':55);
      continua := 's';
      While continua = 's' Do
      Begin
        Writeln;
        Write('Base   = ');
        Readln(x);
        Write('Altura = ');
        Readln(y);
        Writeln;
        Writeln('Área = ',x*y/2:8:2);
        Writeln;
        Writeln;
        Write('Mais cálculos (s/n) --> ');
        continua := ReadKey;
        Writeln;Writeln;
    End;
```

```
'3' :
Begin
  ClrScr;
  Writeln('Calculo da área de quadrados':55);
  continua := 's';
  While continua = 's' Do
  Begin
    Writeln;
    Write('lado = ');
    Readln(x);
    Writeln;
    Writeln('Área = ',x*x:8:2);
    Writeln;
    Writeln;
    Write('Mais cálculos (s/n) --> ');
    continua := Readkey;
    Writeln;Writeln;
  End;
End;
'4' :
Begin
  ClrScr;
  Writeln('Calculo da área de retangulos':55);
  continua := 's';
  While Continua='s' Do
  Begin
    Writeln;
    Write('comprimento = ');
    Readln(x);
    Write('largura     = ');
    Readln(y);
    Writeln;
    Writeln('Área = ',x*y:8:2);
    Writeln;
    Writeln;
    Write('Mais cálculos (s/n) --> ');
    continua := readkey;
    Writeln;Writeln;
  End;
End;
'5' :
Begin
  ClrScr;
```

```
      Writeln('Calculo da área de circulos':55);
      continua := 's';
      While Continua='s' Do
      Begin
        Writeln;
        Write('raio = ');
        Readln(x);
        Writeln;
        Writeln('Área = ',PI*x*x:8:2);
        Writeln;
        Writeln;
        Write('Mais cálculos (s/n) --> ');
        continua := readkey;
        Writeln;Writeln;
      End;
    End;
  End;
  Until escolha = '1';
End.
```

A estrutura CASE pode ser substituída por uma estrutura IF/ELSE IF/ELSE, pois a finalidade de ambas é a mesma. Contudo, o contrário não é verdadeiro, uma vez que a estrutura IF/ELSE IF/ELSE é capaz de realizar comparações mais complexas que a estrutura CASE, que opera apenas com números inteiros ou caracteres. Dessa forma, quando possível, é aconselhável o uso da estrutura CASE, pois o código-fonte com tal estrutura é de mais fácil compreensão e manutenção, se necessário.

4.7 Exercícios

1. Faça um programa para ler um valor numérico e determinar se ele é par ou ímpar.
2. Faça um programa para ler dois valores numéricos e determinar o menor deles.
3. Faça um programa para ler três valores numéricos e determinar o menor deles.
4. Faça um programa que leia o cargo e o salário de um funcionário e calcule seu novo salário reajustado. Se o cargo for *operador*, o funcionário deverá receber um reajuste de 20%, se o cargo for *programador*, o funcionário deverá receber um reajuste de 18%. O algoritmo deve escrever o novo salário do funcionário, já reajustado.

5. Faça um programa que leia o nome, o sexo e a idade de uma pessoa e determine se a pessoa já atingiu a maioridade, sabendo-se que: as pessoas do sexo masculino atingem a maioridade aos 18 anos e as pessoas do sexo feminino atingem a maioridade aos 21 anos. O programa deve escrever o resultado encontrado.

6. Faça um programa que leia o nome, nível, salário bruto e número de dependentes de um funcionário. Calcule e escreva o salário líquido, juntamente com o nome e nível do funcionário. Considere que: para o nível "A", o desconto é de 3% + 5% (se houver dependentes); para o nível "B", o desconto é de 5% + 5% (se houver dependentes); para o nível "C", o desconto é de 8% + 7% (se houver dependentes); e para o nível "D", o desconto é de 10% + 7% (se houver dependentes).

7. O Departamento do Meio Ambiente mantém três listas de indústrias conhecidas, por serem altamente poluentes da atmosfera. Os resultados de várias medidas são combinados para formar o que é chamado de "índice de poluição". Isso é controlado regularmente. Normalmente, os valores caem entre 0.05 e 0.25. Se o valor atingir 0.30, as indústrias da lista A serão chamadas a suspender as operações até que os valores retornem ao intervalo normal. Se o índice atingir 0.40, as indústrias da lista B serão notificadas também. Se o índice exceder 0.50, indústrias de todas as três listas serão avisadas para suspenderem as atividades. Prepare um programa para ler o índice de poluição e escrever as notificações apropriadas.

8. Faça um programa para ler a base e a altura de um triângulo e imprimir a área desse triângulo (área = 1/2 * base * altura). Durante a preparação dos dados para esse algoritmo, é possível que se cometa um erro e valores negativos sejam inseridos para a base ou para a altura. Isso é indesejável, pois a área impressa será negativa. O algoritmo deve prever a possibilidade de verificar valores negativos na entrada. Se um valor negativo é encontrado, o algoritmo deve imprimir uma mensagem identificando esse valor como base ou altura (isso permite corrigir o erro mais facilmente).

9. Tendo como dados de entrada a altura e o sexo de uma pessoa, construa um programa que calcule seu peso ideal, utilizando as seguintes fórmulas:
   - para homens: (72.7 * altura) – 58;
   - para mulheres (62.1 * altura) – 44.7.

10. Escreva um programa que leia uma quantidade indeterminada de números inteiros, encontre o maior e o menor dentre esses números e escreva o maior e o menor valor obtido. Defina uma maneira de interromper a entrada de dados.

11. A prefeitura de uma cidade fez uma pesquisa entre seus habitantes, coletando dados sobre o salário e número de filhos. A prefeitura deseja saber:

    a) Média do salário da população.

    b) Média do número de filhos.

    c) Maior salário.

    d) Percentual de pessoas com salário até R$510,00.

    O final da leitura de dados deve acontecer quando o salário for negativo.

12. Luís Felipe tem 1,50 m de altura e cresce 2 cm por ano. João Guilherme tem 1,10 m e cresce 3 cm por ano. Faça um programa que calcule e escreva quantos anos serão necessários para que João Guilherme seja mais alto que Luís Felipe.

13. Faça um programa que leia uma quantidade indeterminada de números. O programa deverá calcular e escrever a média aritmética dos números pares apenas. A leitura dos dados deve ser encerrada quando o número digitado for igual a 0 (zero).

14. Faça um programa que leia 20 valores numéricos para uma variável n e, para cada um deles, calcule a tabuada de 1 até n. O algoritmo deve apresentar a tabuada na forma:

    1 x n = n

    2 x n = 2n

    3 x n = 3n

    ...

    n x n = n2

15. O DETRAN do Estado de Goiás compilou dados de acidentes de tráfego no estado no último ano. Para cada motorista envolvido num acidente, um registro foi preparado com as seguintes informações: ano de nascimento do motorista (numérico), sexo ("M" ou "F") e código de registro (1 para Goiás e 0 para qualquer outro estado). Prepare um programa para ler um conjunto de dados e imprimir a seguinte estatística de motoristas envolvidos em acidentes:

    a) Percentagem de motoristas com menos de 18 anos.

    b) Percentagem de mulheres.

    c) Percentagem de motoristas com registro feito fora de Goiás.

16. Escreva um programa que gere os números de 1000 a 1999 e escreva aqueles cuja divisão por 11 resulta em resto igual a 5.

17. Escreva um programa que leia uma quantidade desconhecida de números e conte quantos deles estão nos seguintes intervalos: [0..25], [26..50], [51..75] e [76..100]. A entrada de dados deve terminar quando for lido um número negativo.

18. Uma empresa deseja aumentar o salário de seus 25 funcionários. O ajuste salarial deve obedecer à seguinte tabela:

Salário Atual (R$)	Ação
0 a 510,00	Aumento de 20%
510,01 a 800,00	Aumento de 10%
800,01 a 1.200,00	Aumento de 5%
Acima de 1.200,00	Nenhum aumento

Faça um algoritmo que leia o nome e o salário atual de cada funcionário e imprima seu nome, o salário atual e o salário ajustado. No final da impressão, o programa deve dar o total da folha de pagamento sem o aumento e o total da folha de pagamento com o aumento.

19. Foi feita uma pesquisa de audiência de canal de TV em várias casas de uma cidade, num determinado dia. Para cada casa visitada, foi registrado o número do canal (2, 4, 5, 7, 12) e o número de pessoas que estavam assistindo a ele naquele momento. Se a televisão estivesse desligada, a casa não entrava na pesquisa. Faça um programa que:
    a) Leia um número indeterminado de dados. A entrada de dados termina quando o usuário digitar o número do canal igual a zero.
    b) Calcule e escreva a percentagem de audiência para cada canal.

20. Uma firma fez uma pesquisa para saber se as pessoas gostaram ou não de um novo produto lançado no mercado. Para isso, registrou o sexo do entrevistado e sua resposta (sim ou não). Considerando um número indeterminado de pessoas entrevistadas, faça um programa que calcule e escreva:
    a) O número de pessoas que responderam sim.
    b) O número de pessoas que responderam não.
    c) A percentagem de pessoas do sexo feminino que responderam sim.
    d) A percentagem de pessoas do sexo masculino que responderam não.

21. Uma pesquisa realizada perguntou a um grupo de pessoas qual produto, dentre quatro, que elas gostariam de receber como demonstração. Os produtos eram xampu, desodorante, detergente e sabonete. Faça um programa para calcular e escrever os seguintes dados:
    a) Número de mulheres que preferem xampu.
    b) Média de idade dos homens que preferem desodorante.
    c) Percentagem de pessoas que preferem sabonete, entre o total de pessoas.
    d) Número total de mulheres entrevistadas.

22. Uma loja deseja mandar uma correspondência a cada um dos seus clientes anunciando um bônus especial. Escreva um programa que:
    a) Leia o nome de cada cliente e o valor das suas compras no ano passado.
    b) Calcule um bônus de 10% se o valor das compras for menor que R$ 500.000,00 e de 15 %, caso contrário.
    c) Escreva o bônus cedido a cada cliente.

23. Faça um programa que leia um número indeterminado de valores numéricos e calcule a média aritmética dos valores lidos, a quantidade de valores positivos, a quantidade de valores negativos e o percentual de valores negativos e positivos.

24. Existem 4 candidatos a governador. Feita a eleição, os votos são digitados, via teclado, um a um. O voto de cada eleitor foi codificado da seguinte forma:

    1 2 3 4 => votos para os candidatos 1, 2, 3 e 4, respectivamente.
    0 => voto branco.
    9 => voto nulo.

    Faça um programa para determinar:
    a) O número do candidato vencedor.
    b) O número de votos em branco.
    c) O número de votos nulos.
    d)O número de eleitores que compareceram às urnas.

25. Faça um programa que pede para o usuário digitar vogais. Caso o usuário digite qualquer outro caractere, o programa informa o erro e pede para o usuário entrar com outra vogal. O programa deve terminar quando o usuário informar que quer parar. Nesse

momento, o programa informa quantos caracteres válidos (vogais) e quantos inválidos (qualquer outro) foram digitados.

26. Faça um programa que leia dez números e imprime apenas os que forem divisíveis por dois.

27. Faça um programa que leia diversos números e imprima o somatório total, a média e a quantidade de números lidos. A totalização ocorre quando o usuário digitar um número negativo.

28. Faça um programa que leia caracteres e depois imprima a quantidade de cada uma das vogais que foram digitadas.

29. Faça um programa que, para cada inteiro digitado (este é valido se estiver na faixa entre 10 e 60), gera-se uma linha com um valor correspondente de asteriscos.

5. Tipos estruturados de dados

Existem outros tipos de dados além daqueles do tipo simples (tais como os tipos BYTE, INTEGER, REAL, CHAR e BOOLEAN), chamados *tipos complexos* ou *estruturados*. O tipo STRING é um exemplo de um tipo de dado estruturado. Os tipos de estruturas de dados presentes na linguagem Pascal são os vetores (ARRAY), os registros (RECORD), os conjuntos (SET) e os arquivos (FILE).[6]

5.1 Vetores (ARRAY)

Caso fosse necessário declarar dez variáveis do tipo inteiro, bastaria escrever:

```
Var
  i1, i2, i3, i4, i5, i6, i7, i8, i9, i10: Integer;
```

Se o código anterior precisasse ser alterado para contemplar agora cem variáveis inteiras, e não somente dez, seria possível declará-las da mesma

[6] O tipo FILE refere-se a arquivos de discos e será estudado no capítulo 7. Os demais tipos serão vistos neste capítulo.

forma, como foi feito anteriormente. Se, além dessas cem variáveis do tipo inteiro, fossem necessárias outras mil variáveis do tipo caractere, o código-fonte necessário para a declaração de tais variáveis seria extenso, e o trabalho realizado para declarar tais variáveis, altamente tedioso. Uma forma de resolver o problema de declarar várias variáveis do mesmo tipo é o uso de vetores (ou em inglês ARRAY). Com esse tipo de dado estruturado, é possível criar um grande número de variáveis do mesmo tipo sem os inconvenientes anteriores.

Todos os vetores possuem duas características principais. Em primeiro lugar, são declarados para armazenar duas ou mais variáveis do mesmo tipo de dados (um vetor pode armazenar variáveis inteiras ou variáveis reais, mas não pode armazenar uma variável inteira e uma variável real ao mesmo tempo). A segunda característica típica dos vetores é o seu tamanho. Quando se declara um vetor, devem-se planejar antecipadamente, além do tipo dos dados que serão armazenados nesse vetor, quantos elementos serão armazenados nele. Esse é um importante passo, porque o compilador só pode alocar memória suficiente para o vetor se lhe for informado quantos elementos comporão o vetor. O compilador multiplica o número de itens pelo tamanho de cada um e reserva o montante de memória para a alocação do vetor. A declaração geral de um vetor é:

```
Type
  Vetor = Array[1..100] of Integer;
Var
  i: Vetor;
```

A declaração acima pode ser dividida em duas partes. Na primeira parte, foi criado um novo tipo de dado na subárea TYPE, chamado `Vetor`, que é composto por um vetor cujo índice inicial é 1 e o último índice 100, de forma que possui cem posições de números inteiros. Na segunda parte, foi criada uma variável do tipo `Vetor` denominada `i`. Esse tipo de declaração, em que o vetor é um novo tipo de dado, é interessante quando se deseja criar outros tipos de dados que dependam do vetor. Se este não for o caso, pode-se usar uma forma reduzida da declaração anterior, que consiste em se declarar diretamente o vetor como uma variável, da seguinte forma:

```
Var
  i: Array[1..100] of Integer;
```

Após a declaração acima, teríamos definidas cem variáveis do tipo inteiro, cujos identificadores seriam o próprio identificador usado na declaração do vetor (no caso, a letra i), indexado de acordo com os limites pré-estabelecidos na declaração do vetor (ou seja, de 1 a 100). Para acessar a décima variável inteira do vetor `i` e atribuir-lhe um valor (por exemplo, cinco), procede-se da seguinte maneira:

```
i[10] := 5;
```

Se uma variável armazenada na memória do computador pode ser imaginada como uma caixa que possui nome e tamanho predefinidos, um vetor pode ser imaginado como uma fila dessas caixas, em que cada caixa possui um número que indica a sua ordem na fila, uma vez que o nome agora designa a fila em si, e não cada caixa. Cada variável pode armazenar um único valor de cada vez, sendo esse princípio válido também para cada uma das variáveis que compõem o vetor. A figura 5.1 apresenta uma representação gráfica de um vetor de cem posições de números inteiros, semelhante ao vetor `i` declarado anteriormente, em que as variáveis foram representadas como caixas na memória do computador. Pode-se notar que cada caixa é representada por um número, ou *índice*, e armazena um único valor inteiro.

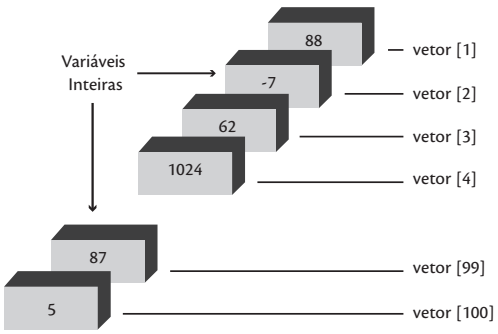


Figura 5.1 - Representação gráfica de um vetor de 100 posições de inteiros

O código 5.1 apresenta um programa que armazena dez números inteiros digitados pelo usuário em um vetor e, após a leitura deste, escreve-os no monitor de vídeo na ordem inversa a que eles foram digitados.

Código 5.1 - Uso de vetores em Pascal

```
Program Exemplo_ARRAY_1;
Uses Crt;
Type
  vetor = Array[1..10] Of Integer;
Var
  a: vetor;
  i: Integer;
Begin
  ClrScr;
  For i:=1 to 10 do
  Begin
    Write('a[',i:2,'] = ');
    Readln(a[i]);
  End;
  ClrScr;
  For i:=10 downto 1 do writeln('a[',i:2,'] = ', a[i]);
  Readkey;
End.
```

O código 5.2 apresenta um programa que lê cem números reais e, de posse deles, ordena-os em ordem crescente. O método de ordenação apresentado é conhecido como método da bolha (ou, em inglês, *bubble sort*). Existem outros métodos de ordenação de vetores, porém esse é considerado um dos mais simples.

Código 5.2 - Programa que ordena crescentemente 100 números reais lidos do teclado

```
Program Exemplo_ARRAY_2;
Uses CRT;
Const
  Num_Max = 100;
Type
  vetor = Array[1..Num_Max] of Real;
```

```
Var
  i, j, n: Integer;
  a: vetor;
  z: Real;
Begin
  ClrScr;
  Writeln('Ordenação de números lidos do teclado':40+19);
  Writeln;Writeln;
  n := 0;
  Writeln('Digite –1 para sair':40+19);
  Writeln;Writeln;
  Repeat
    n := n + 1;
    Write('a[',n:3,'] = ');
    Readln(a[n]);
  Until ((n = Num_max) Or (a[n] < 0));
  if (a[n] < 0) then n := n – 1;
  ClrScr;
  For i := 1 to n Do
    For j:=i+1 to n Do
      If (a[i] >= a[j]) Then
      Begin
        z := a[i];
        a[i] := a[j];
        a[j] := z;
      End;
  For i:=1 to n Do Writeln(i:3, '  ->  ', a[i]:10:2);
  Readkey;
End.
```

O código 5.3 mostra a funcionalidade dos vetores em relação a caracteres, e não somente a números, uma vez que este lê trinta nomes do teclado e, após a leitura, ordena-os em ordem alfabética.

Código 5.3 - Programa que ordena strings em ordem alfabética

```
Program Exemplo_ARRAY_3;
Uses CRT;
Const
  Num_Max = 100;
```

```
Type
  nomes = String[30];
  vetor = Array[1..Num_max] of nomes;

Var
  i, j, n: Integer;
  a: vetor;
  z: nomes;
Begin
  ClrScr;
  Writeln('Ordenação de nomes lidos do teclado':40+19);
  Writeln;Writeln;
  n := 0;
  Writeln('Digite 0 para sair':50);
  Writeln;Writeln;
  Repeat
    n := n + 1;
    Write('a[',n:3,'] = ');
    Readln(a[n]);
  Until ((n = Num_Max) Or (a[n]='0'));
  if (a[n] = '0') then n := n – 1;
  ClrScr;
  For i := 1 to n Do
    For j := i+1 to n Do
      If (a[i] >= a[j]) Then
      Begin
        z:=a[i];
        a[i]:=a[j];
        a[j]:=z;
      End;
  For i := 1 to n Do Writeln(i:3, '  ->  ', a[i]:30);
  Readkey;
End.
```

5.2 Matrizes (vetores multidimensionais)

Uma matriz nada mais é que um vetor multidimensional. Se um vetor necessita de um número apenas como índice de seus elementos, uma matriz necessitará de dois ou mais números. A representação mais comum de uma matriz é uma matriz bidimensional, composta por linhas e colunas. A figura 5.2 apresenta a representação gráfica de uma matriz bidimensional denominada `multi`, composta por sete linhas e seis colunas.

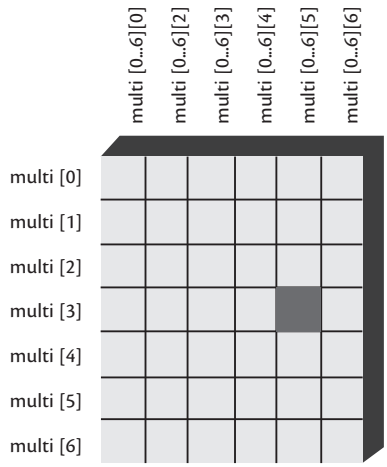


Figura 5.2 - Representação gráfica de uma matriz

Matrizes bidimensionais são constantemente utilizadas (vide o próprio monitor de vídeo, que, para o Turbo Pascal, é descrito como uma matriz de 80 colunas e 25 linhas) e sua sintaxe típica é:

```
Var
  a: array[1..10,1..5] of Integer;
```

A declaração acima define uma matriz de cinquenta elementos denominada `a`, composta por dez linhas indexadas de 1 a 10 e de cinco colunas indexadas de 1 a 5. O acesso a cada elemento da matriz é feito da seguinte forma:

```
a[1,1]   a[1,2]   a[1,3]   a[1,4]   a[1,5]
a[2,1]   a[2,2]   a[2,3]   a[2,4]   a[2,5]
a[3,1]   a[3,2]   a[3,3]   a[3,4]   a[3,5]
a[4,1]   a[4,2]   a[4,3]   a[4,4]   a[4,5]
a[5,1]   a[5,2]   a[5,3]   a[5,4]   a[5,5]
a[6,1]   a[6,2]   a[6,3]   a[6,4]   a[6,5]
a[7,1]   a[7,2]   a[7,3]   a[7,4]   a[7,5]
a[8,1]   a[8,2]   a[8,3]   a[8,4]   a[8,5]
a[9,1]   a[9,2]   a[9,3]   a[9,4]   a[9,5]
a[10,1]  a[10,2]  a[10,3]  a[10,4]  a[10,5]
```

Para se declarar a matriz a anterior, procede-se da seguinte forma:

```
Var
  a: array[1..10] of array[1..5] of Integer;
```

Uma matriz é declarada com dois pares de colchetes, sendo indicado, em cada um deles, o tamanho desejado para cada dimensão da matriz. Tecnicamente, uma matriz é um *vetor de vetores*. Contudo, é mais fácil visualizar uma matriz como um conjunto de linhas e colunas. O código 5.4 apresenta um programa que lê uma matriz 10 x 20 (dez linhas e vinte colunas) de inteiros digitada pelo usuário e, de posse desta, multiplica uma coluna ou uma linha por uma constante informada pelo usuário.

Código 5.4 - Exemplo de uma matriz em Pascal

```
Program Exemplo;
Uses CRT;
Const
  NUM_MAX_COL = 20; (* número máximo de colunas *)
  NUM_MAX_LIN = 10; (* número máximo de linhas  *)
Var
  a: array[1..NUM_MAX_LIN,1..NUM_MAX_COL] of Integer;
  i, j, k, p, nl, nc: Integer;
  lc: char;
Begin
  ClrScr;
(* lê o número de linhas da matriz *)
  Repeat
    Write('Numero de linhas da matriz ----------> ');
    Readln(nl);
  Until (nl <= NUM_MAX_LIN);
(* lê o número de colunas da matriz *)
  Repeat
    Write('Numero de colunas da matriz ---------> ');
    Readln(nc);
  Until (nc <= NUM_MAX_COL);
(* lê a constante de multiplicação *)
  Write('Constante para multiplicacao --------> ');
  Readln(k);
(* coluna ou uma linha a ser multiplicada *)
```

```
  Repeat
    Write('Coluna ou linha a ser multuplicada (c/l)? ');
    Readln(lc);
  Until ((lc='c') Or (lc='l'));
(* Qual índice da coluna ou da linha a ser multiplicada *)
  If (lc ='c') Then
    Repeat
      Write('Numero da coluna a ser multipiplicada? ');
      Readln(p);
    Until (p <= nc)
  Else
    Repeat
      Write('Numero da linha a ser multiplicada? ');
      Readln(p);
    Until (p <= nl);
  Writeln;
  TextBackGround(7);
  TextColor(15+16);
  Gotoxy(24,7);
  Write('Entre com os elementos da matriz');
  Textcolor(8);
  For i:=1 to nl do
    For j:=1 to nc do
      Begin
        Gotoxy(8*j,i+8);
        Write('+');
      End;
  TextBackGround(0);
  Textcolor(13);
(* lê os elementos da matriz *)
  For i:=1 to nl do
    For j:=1 to nc do
    Begin
      Gotoxy(8*j,i+8);
      Read(a[i,j]);
    End;
(* faz a multiplicação da coluna ou da linha *)
  If (lc ='c') Then
    For i:=1 to nl do a[i,p]:=a[i,p]*k
  Else
    For j:=1 to nc do a[p,j]:=a[p,j]*k;
  TextBackGround(0);
  TextColor(15+16);
```

```
  Gotoxy(24,7);
(* apresenta o resultado final na tela *)
  Write('.....Resultado final.....');
  Textcolor(13);
  For i:=1 to nl do
    For j:=1 to nc do
    Begin
      Gotoxy(8*j,i+8);
      Write(a[i,j]);
    End;
End.
```

5.3 Registros (RECORD)

Vetores e matrizes são estruturas de dados em que todos os seus elementos possuem o mesmo tipo de dado. Dessa forma, são denominadas de estruturas *homogêneas de dados*. O tipo RECORD, chamado estrutura *heterogênea de dados*, permite a criação de uma estrutura composta por campos de diferentes tipos de dados. Supondo que fosse necessário armazenar o nome, idade, sexo e altura relativos ao cadastro de um cliente, seria preciso declarar quatro variáveis, uma para cada dado, da seguinte forma:

```
Var
  Nome: String[40];
  Idade: Byte;
  Sexo: Char;
  Altura: Single;
```

Se fosse necessário armazenar as informações acima citadas para não só uma, mas mil pessoas, seria muito difícil para um programador criar um programa que declarasse e manipulasse essas quatro mil variáveis. Uma forma mais fácil e menos trabalhosa de se declarar e manipular essas variáveis seria encapsulá-las em uma única variável, o que é possível realizar criando um registro de dados na subárea RECORD dentro da subárea TYPE. Para criar mil variáveis desse novo tipo de dado, bastaria criar um vetor desse mesmo tipo de dado, resolvendo o problema da criação das quatro mil variáveis com poucas linhas de código.

A definição de um registro de dados na subárea RECORD começa pelo nome do registro que se deseja definir, seguido da palavra reservada RECORD. Na sequência, declaram-se os nomes e tipos de dados dos campos que comporão o registro. A palavra reservada END, seguida de ponto e vírgula, termina a definição do registro de dados. A sintaxe geral de um registro é vista abaixo, aproveitando o exemplo anterior:

```
Type
  Registro = Record
    Nome: String[40];
    Idade: Byte;
    Sexo: Char;
    Altura: Single;
  End;
Var Nome_Do_Registro : Registro;
```

Uma vez que o tipo RECORD é visto como um novo tipo de dados, a forma acima de declaração é a mais utilizada, pois evidencia a criação desse novo tipo. Contudo, pode-se criar diretamente uma variável do tipo registro, conforme mostrado abaixo:

```
Var
  Nome_Do_Registro : Record
    Nome: String[40];
    Idade: Byte;
    Sexo: Char;
    Altura: Single;
  End;
```

Para acessar os campos do registro, utiliza-se o nome da variável do tipo registro, seguido de um ponto final e do nome do campo que se deseja acessar. No jargão da linguagem de programação, o uso do ponto final separando o nome do registro e de seus campos é chamado de *notação de ponto*.[7]

[7] A linguagem Delphi, evolução do Turbo Pascal, desenvolvido também pela Borland International, utiliza amplamente a notação de ponto. Dessa forma, é fortemente recomendado ao leitor dominar essa técnica, caso seja seu interesse o aprendizado futuro da linguagem.

Para acessar os campos altura e sexo da variável Nome_Do_Registro declarada anteriormente, basta escrever:

```
Nome_Do_Registro.Altura := 1.78;
Nome_Do_Registro.Sexo   := 'M';
```

O código 5.5 apresenta um programa embrião de um banco de dados. Apesar de muito simples, o programa estrutura a ideia de um banco de dados e de sua arquitetura, de como criar um tipo de variável única para o banco de dados e como acessar os seus campos de dados. Durante a sua execução, o programa lê uma série de dados sobre uma pessoa e, logo em seguida, mostra ao usuário os dados que ele acabou de digitar.

Código 5.5 - Uso de variáveis do tipo RECORD como um banco de dados simples

```
Program Exemplo_RECORD;
Uses CRT;
Type
  Pessoas = Record
    Nome: String[40];
    Idade: Byte;
    Sexo: Char;
    Altura: Single;
  End;
Var p: Pessoas;
Begin
  ClrScr;
  Write('Nome ------> ');
  Readln(p.Nome);
  Write('Idade -----> ');
  Readln(p.Idade);
  Write('Sexo ------> ');
  Readln(p.Sexo);
  Write('Altura ----> ');
  Readln(p.Altura);
  Writeln;
  Writeln('Você digitou os seguintes dados :');
  Writeln;Writeln;
  Writeln(p.nome);
  Writeln(p.idade);
```

```
  Writeln(p.sexo);
  Writeln(p.altura:1:2);
  Readkey;
End.
```

É possível declarar um vetor de qualquer tipo de dados, inclusive do tipo registro. Pensando no desenvolvimento de um banco de dados, mesmo que simplório, o código 5.6 mostra uma atualização do programa anterior, sendo o programa agora capaz de ler dados de, no máximo, 20 pessoas, com a utilização de um vetor de registros. Em seguida, o programa faz uma listagem dos dados de modo a colocá-los em ordem alfabética, de acordo com os nomes digitados.

Código 5.6 - Um simplório banco de dados

```
Program Exemplo;
Uses CRT;
Type
  Pessoas = Record
    Nome: String[40];
    Idade: Byte;
    Sexo: Char;
    Altura: Single;
  End;
Var
  p: array[1..20] of Pessoas;
  i, x, y: Integer;
  s: Pessoas;
Begin
  ClrScr;
  i := 0;
  Repeat
    i := i + 1;
    Write('Nome (0 para sair): ');
    Readln(p[i].Nome);
    if p[i].Nome='0' then
    Begin
      Write('Idade --------> ');
      Readln(p[i].Idade);
```

```
      Write('Sexo ---------> ');
      Readln(p[i].Sexo);
      Write('Altura -------> ');
      Readln(p[i].Altura);
      Writeln;
    End;
  Until ((p[i].Nome='0') or (i=20));
  If (i < 20) then i := i – 1;
  For x:=1 to i do
    For y:=x+1 to i do
        If ((p[x].nome) >= (p[y].nome)) then
        Begin
          s := p[x];
          p[x] := p[y];
          p[y] := s;
        End;
  ClrScr;
  Writeln('NOME':40,'IDADE':10,'SEXO':10,'ALTURA':10);
  For x:=1 to i do
  Writeln(p[x].nome:40, p[x].idade:10, p[x].sexo:10, p[x].altura:10:2);
  Readkey;
End.
```

Quando existe uma série de campos de uma mesma variável do tipo RECORD que será acessada repetidamente, torna-se cansativo escrever diversas vezes o nome da variável na frente do campo. Para resolver esse problema, pode-se utilizar o comando WITH, cuja sintaxe geral é:

```
WITH Variável_do_tipo_record DO
Begin
  comando_1;
  comando_2;
  ...
End;
```

Os comandos citados dentro do bloco definido pelo BEGIN-END do comando WITH entendem os campos do registro sem a necessidade da menção explicita ao nome do registro. O código 5.7 mostra o código 5.5 utilizando o comando WITH para simplificar o acesso aos campos da variável do tipo registro.

Código 5.7 - Uso do comando WITH para acessar campos de variáveis

```
Program Exemplo;
Uses CRT;
Type
  Pessoas = Record
    Nome: String[40];
    Idade: Byte;
    Sexo: Char;
    Altura: Single;
  End;
Var p: Pessoas;
Begin
  ClrScr;
  With p do
  Begin
    Write('Nome ------> ');
    Readln(Nome);
    Write('Idade -----> ');
    Readln(Idade);
    Write('Sexo ------> ');
    Readln(Sexo);
    Write('Altura ----> ');
    Readln(Altura);
    Writeln;
    Writeln('Você digitou os seguintes dados :');
    Writeln;Writeln;
    Writeln(nome);
    Writeln(idade);
    Writeln(sexo);
    Writeln(altura:1:2);
  End;
  Readkey;
End.
```

5.4 Conjuntos de dados (SET)

Matematicamente, define-se um conjunto como uma coleção de objetos, nomes, números, etc. Chamamos de elementos os objetos, nomes ou números que pertencem a um conjunto. Na linguagem Pascal, também é possível utilizar o conceito de conjuntos, o que não ocorre em outras linguagens

de programação, tais como o C e o C++. Em Pascal, um conjunto é uma coleção de elementos de mesmo tipo de dado, sendo o seu tamanho máximo 256 elementos. Um conjunto pode ser constituído de nenhum (conjunto vazio), um, dois ou mais elementos do mesmo tipo base que, obrigatoriamente, devem ser de um tipo simples, podendo ser qualquer escalar com exceção do tipo REAL. Os conjuntos têm seus elementos inclusos em colchetes e separados por vírgulas. Pode-se usar também a representação de uma subfaixa (*range*) de valores. Exemplos de conjuntos são:

```
[1,3,5,8,9,11,13]     – alguns inteiros
[3..7]                – inteiros entre 3 e 7
[3,4,5,6,7]           – equivalente  ao anterior
['A'..'Z']            – caracteres alfabéticos maiúsculos
[gol,passat,fusca]    – marcas de carro
[]                    – conjunto vazio
```

A forma geral para definição de um conjunto é:

```
Type identificador = SET OF tipo_base;
```

Abaixo, seguem exemplos da declaração de quatro tipos de dados do tipo conjunto, bem como a declaração de uma variável para cada um dos conjuntos criados. Nota-se que uma variável do tipo conjunto é iniciada como um conjunto vazio, não importa a qual conjunto ela pertence. Os valores usados na declaração do conjunto são os possíveis valores que uma variável do tipo conjunto poderá assumir.

```
Type
  caracteres       = set of Char;
  letras_maiúsculas = set of 'A'..'Z';
  digitos          = set of 0..9;
  carros           = set of (fusca,gol,escort,opala);
Var
  c      : caracteres;
  letras  : letras_maiúsculas;
  numeros : dígitos;
  marca   : carros;
```

As operações com conjuntos são similares às operações realizadas com conjuntos na matemática, sendo possível realizar dois tipos de operações com conjuntos: as operações aritméticas e as operações relacionais entre conjuntos. As operações relacionais entre conjuntos são:

- a = b, que retorna verdadeiro (TRUE) se todos os elementos do conjunto a pertencem ao conjunto b.
- a <> b, que retorna verdadeiro se um ou mais elementos do conjunto a **não** pertencem ao conjunto b.
- a >= b, que retorna verdadeiro se todos os elementos de b pertencem ao conjunto a, podendo a possuir mais elementos que b.
- a <= b, que retorna verdadeiro se todos os elementos do conjunto a pertencem ao conjunto b, podendo b possuir mais elementos que a.
- 'Z' IN b, que retorna verdadeiro se o elemento 'Z' for um elemento do conjunto b, sendo 'Z' um elemento do mesmo tipo base do conjunto b.

As operações aritméticas com conjuntos na linguagem Pascal são: atribuição, união, diferença e interseção de conjuntos. O operador de atribuição é o mesmo operador utilizado para tipos simples (:=). Nos exemplos abaixo, elementos são atribuídos aos conjuntos c, letras e números:

```
c := ['a','e','i','o','u'];
letras := ['B'..'H'];
numeros := [0,3,5];
```

O operador união de conjuntos, representado pelo sinal de adição (+), resulta em um terceiro conjunto, constituído pelos elementos de ambos os conjuntos passados como argumentos para o operador união. Elementos em comum aos dois conjuntos não aparecem duplicados no conjunto resposta, como pode ser visto no exemplo abaixo:

```
a := [1,2,3];
b := [2,3,4,5];
c := a + b;              {que resulta em c = [1,2,3,4,5]}
```

O operador diferença de dois conjuntos, representado pelo sinal subtração (–), retorna um conjunto cujos elementos estão no primeiro conjunto e não estão no segundo conjunto. O exemplo abaixo ilustra o uso desse operador:

```
a := [1,2,3,6];
b := [2,3,4,5];
c := a – b;          {que resulta em c = [1,6]}
c := b – a;          {que resulta em c = [4,5]}
```

O operador interseção entre dois conjuntos é representado pelo sinal de multiplicação (*) e resulta num terceiro conjunto, constituído pelos elementos que fazem parte de ambos os conjuntos passados como argumento ao operador, simultaneamente. Dessa forma:

```
a := [1,2,3];
b := [2,3,4,5];
c := a * b;          {que resulta em c = [2,3]}
```

O código 5.8 apresenta um programa que lê uma tecla qualquer pressionada pelo usuário e informa se essa tecla é um número, uma letra maiúscula, uma letra minúscula ou um caractere especial. O programa encerra a sua atividade quando o usuário digita um ponto de interrogação (?).

Código 5.8 - Uso de conjuntos (SETS) em Pascal

```
Program Exemplo_SET_1;
Uses CRT;
Type simbolos = Set of Char;
Var
  Maiusc, Minusc, Numeros: simbolos;
  tecla: char;
Begin
  ClrScr;
  Maiusc  := ['A'..'Z'];
  Minusc  := ['a'..'z'];
  Numeros := ['0'..'9'];
  Repeat
    Tecla := Readkey;
    If tecla IN Maiusc Then
```

```
      Writeln('LETRA MAIÚSCULA')
    Else if tecla IN minusc Then
      Writeln('letra minúscula')
    Else if tecla IN numeros Then
      Writeln('Caractere especial')
    Else
      Writeln('Não dei o que e');
  Until (tecla = '?');
  Writeln('Fim do programa... ');
  Readkey;
End.
```

O código 5.9 apresenta um programa que conta o número de vogais, de consoantes e de espaços em branco existentes em uma frase qualquer digitada pelo usuário.

Código 5.9 - Contagem do número de vogais, consoantes e espaços em branco de uma frase

```
Program Exemplo_SET_2;
Uses CRT;
Type simbolos = set of char;
Var
  Alfabeto,vogais,consoantes: simbolos;
  frase: string[50];
  v, c, b, x : integer;
Begin
  Vogais:=['a','e','i','o','u','A','E','I','O','U'];
  alfabeto:=['a'..'z']+['A'..'Z'];
  consoantes:=alfabeto-vogais;
  Clrscr;
  Write('Digite uma frase --> ');
  Readln(frase);
  b := 0;
  c := 0;
  v := 0;
  For x:=1 to length(frase) do
    if frase[x] in vogais then v := v + 1
    else if frase[x] in consoantes then c := c + 1
    else if frase[x] = ' ' then b := b + 1;
  Writeln;
  writeln('A frase digitada possui:');
```

```
  writeln(b:3,' brancos;');
  Writeln(c:3,' consoantes e');
  Writeln(v:3,' vogais.');
  Readkey;
End.
```

No código anterior, foi utilizada a função LENGTH, que retorna o tamanho de seu argumento. A função LENGTH retorna um número inteiro e seu argumento deverá ser uma string. Sua sintaxe geral é:

```
LENGTH(string);
```

As bases numéricas mais utilizadas em programação são a base binária (base dois), a base octal (base 8), a base decimal (base 10) e a base hexadecimal (base 16). A base hexadecimal é composta pelos algarismos de zero a nove e das letras A até F. O código 5.10 apresenta um programa que faz a conversão de um número da base hexadecimal para a base decimal.

Código 5.10 - Programa para a conversão de hexadecimal para decimal

```
Program Exemplo;
Uses CRT;

Function pow(a, n: integer): integer;
Var
  i: integer;
  result: integer;
Begin
  result := 1;
  For i := 1 to n do result := a * result;
  pow := result;
End;

Const MAX = 3;
Type h = set of char;
Var
  letras, digitos, hex: h;
  ch: char;
  s : string[MAX];
```

```
  w: word;
  i: integer;
Begin
  letras := ['A'..'F'];
  digitos := ['0'..'9'];
  hex := letras + digitos;
  Repeat
    Clrscr;
    s := '';
    Write('Digite um valor Hexadecimal: ');
    Repeat
      Repeat
        ch := UPCASE(Readkey);
      Until (ch in hex) or (ch = #13);
      If ch in hex then
      Begin
        s := s + ch;
        Write(ch);
      End;
    Until (ch = #13) or (length(s) = MAX);
    While length(s) < MAX do
      s := '0' + s;
    w := 0;
    For i := 1 to MAX do
    Begin
      If s[i] in digitos then
        w := w + (ORD(s[i]) – 48) * pow(16, MAX – i)
      Else
        w := w + (ORD(s[i]) – 55) * pow(10, MAX – i);
    End;
    Writeln;
    Writeln('O valor ',s,' corresponde ao decimal ', w);
    Writeln;
    Write('Deseja realizar outra conversão (S/N)? ');
    ch := upcase(Readkey);
  Until ch = 'N';
End.
```

No início do Código 5.10 foi declarada uma função chamada POW. Por enquanto, basta entender que essa função retorna a potência de uma base inteira elevada a um expoente também inteiro, pois o capítulo 6 deste livro abordará

os conceitos necessários para a criação e manipulação de procedimentos e funções. A função ORD, apresentada no código 5.10, retorna o valor numérico relacionado ao tipo ordinal de seu argumento. Caso esse argumento seja do tipo CHAR, a função ORD retorna o código ASCII do argumento. Já a função UPCASE converte um caractere, ou uma string, para maiúsculas. Se o argumento passado a essa função já estiver em maiúsculas, ele é retornado sem que nenhuma alteração seja realizada. A função UPCASE ignora caracteres numéricos e/ou caracteres especiais, tais como sinais de pontuação, etc.

5.5 Exercícios

1. Qual é o número de elementos de cada um dos vetores a seguir?
   a) var vet : array[-5 .. 5] de real
   b) var nome : array [0 .. 10] de char
   c) var oc : array [1 .. 10] de integer
   d) var arr : array [0 .. 12] de boolean

2. Faça um programa para ler um vetor de 80 elementos do tipo numérico e verificar se existem elementos iguais a 120. Se existirem, o programa deverá escrever quantas vezes aparecem e quais as posições em que estão armazenados.

3. Faça um programa que:
   a) Leia um vetor C de 100 elementos do tipo real.
   b) Construa e imprima outro vetor F, formado da seguinte maneira:
      - Os elementos cuja posição é par devem ser os correspondentes de C divididos por 2.
      - Os elementos cuja posição é ímpar devem ser os correspondentes de C multiplicados por 3.

4. Faça um programa que:
   a) Leia um vetor chamado NOMES com 50 elementos do tipo caractere.
   b) Leia o valor de uma variável N também do tipo caractere.
   c) Verifique a existência desse caractere no vetor. Caso exista, dê a seguinte mensagem: "*O caractere encontra-se na posição*": e informe a posição. Caso não encontre, escreva uma mensagem negativa.

5. Faça um programa que:
   a) Leia uma variável composta A com 30 valores do tipo real.
   b) Leia outra variável composta B com 30 valores do tipo real.
   c) Leia o valor de uma variável X do tipo real.
   d) Verifique qual o elemento de A que é igual a X.
   e) Imprima o elemento de B de posição correspondente à do elemento de A igual a X.

6. Faça um programa que:
   a) Leia dois vetores contendo, cada um, 25 elementos do tipo numérico.
   b) Intercale os elementos desses dois vetores, formando um novo vetor de 50 elementos, também do tipo numérico.
   c) Escreva o novo vetor obtido.

7. Analise o trecho de programa na linguagem Pascal abaixo:

```
...
i := 1;
x := 3;
y := 11;
while x <> y do
begin
  if x > y then
  begin
    y := y + 11;
    vet[i] := 1;
  end
  else
  begin
    x := x + 2;
    vet[i] := 0;
  end;
  i := i + 1;
end;
...
```

Pergunta-se:

   a) Com que valor da variável "x" a estrutura de repetição do programa encerra a sua execução?
   b) Quantas posições do vetor "vet" valerão "1" e quantas posições valerão "0" ao final da execução da estrutura de repetição?

8. Dado um conjunto de 100 valores numéricos, faça um programa para armazená-los em um vetor B, calcule e escreva o valor do somatório dado a seguir:

$$S = (B_1 - B_{100})^2 + (B_2 - B_{99})^2 + (B_3 - B_{98})^2 + \ldots + (B_{50} - B_{51})^2$$

9. Faça um programa que:

   a) Leia um conjunto de valores inteiros correspondentes a 80 notas dos alunos de uma turma, notas estas que variam de 0 a 10.
   b) Calcule a frequência absoluta e relativa de cada nota.
   c) Imprima os valores das notas (de 0 a 10) e sua respectiva frequência absoluta e relativa.

   Observações:

   - Frequência absoluta de uma nota é o número de vezes que ela aparece no conjunto de dados.
   - Frequência relativa é a frequência absoluta dividida pelo número total de dados.
   - Utilizar como variável composta somente aquelas que forem necessárias.

10. Em uma cidade do interior, sabe-se que, de janeiro a abril de 2010 (121 dias), não ocorreu temperatura inferior a 15ºC nem superior a 40ºC. As temperaturas verificadas em cada dia estão disponíveis para entrada de dados. Faça um programa que calcule e imprima:
    a) A menor temperatura ocorrida.
    b) A maior temperatura ocorrida.
    c) A temperatura média.
    d) A quantidade de dias nos quais a temperatura foi inferior à temperatura média.

11. Está disponível para entrada de dados o estoque de mercadorias de uma loja. São dadas as mercadorias e as respectivas quantidades existentes. A seguir, estão disponíveis para entrada os pedidos dos clientes. Faça um programa para atualização do estoque, tal que:
    a) Seja lido e listado o estoque inicial (máximo de 100 mercadorias).
    b) Sejam lidos os pedidos dos clientes, constituídos, cada um, do número do cliente, código da mercadoria e quantidade desejada.
    c) Seja verificado, para cada pedido, se ele pode ser integralmente atendido. Em caso negativo, imprima o número do cliente e a mensagem: "*Não temos mercadoria suficiente em estoque*.".

d) Seja atualizado o estoque após cada operação.

e) Seja listado o estoque final.

Observação: crie uma maneira para encerrar a entrada dos pedidos dos clientes.

12. O curso de Engenharia de Minas da UFG deseja saber se existem alunos cursando, simultaneamente, as disciplinas "Introdução à Ciência da Computação" e "Fundamentos de Simulação Computacional I". Existem disponíveis na unidade de entrada os números de matrícula dos alunos de "Introdução à Ciência da Computação" (no máximo 50 alunos) e de "Fundamentos de Simulação Computacional I" (no máximo 30 alunos). Formule um programa que imprima o número de matrícula dos alunos que estão cursando essas disciplinas simultaneamente.

13. Escreva um programa que faça reserva de passagens aéreas de uma companhia. Além da leitura do número dos voos e da quantidade de lugares disponíveis, o programa deverá ler vários pedidos de reserva, constituídos do número da carteira de identidade do cliente e do número do vôo desejado. Para cada cliente, verificar se há disponibilidade no voo desejado. Em caso afirmativo, imprimir o número da identidade do cliente, e o número do voo, atualizando o número de lugares disponíveis. Caso contrário, avisar ao cliente da inexistência de lugares. Para indicar o fim dos pedidos de reserva, existe um passageiro cujo número da carteira de identidade é 999. Considere fixo e igual a 40 o número de voos da companhia.

    Sugestão: Crie dois vetores. O primeiro para armazenar os números dos voos e o segundo para armazenar a quantidade de lugares disponíveis para cada voo.

14. Faça um programa para corrigir provas de múltipla escolha. Cada prova tem 10 questões, cada questão vale um ponto. O primeiro conjunto de dados a ser lido será o gabarito para a correção da prova. Os outros dados serão os números de inscrição dos alunos e suas respectivas respostas. O algoritmo deverá calcular e imprimir, para cada aluno, o seu número de inscrição e sua nota.

    Sugestão: Crie um vetor para armazenar o gabarito da prova e um vetor para armazenar as respostas para cada aluno.

15. Faça um programa que leia uma matriz 4 x 65 de números reais. As posições carregadas serão apenas aquelas cuja soma dos índices for par. Ex. (2,4) é carregado. (5,2) não.

16. Faça um programa que leia uma matriz 4 x 50 de notas de alunos de quatro turmas. Por turma, calcule a média dos alunos e a quantidade de alunos com nota superior a 6.

17. Faça um programa que leia uma matriz 50 x 50 de números inteiros e depois calcule qual a coluna (ou linha) que apresenta o maior somatório.

18. Faça um programa que carrega duas matrizes com 50 nomes de pessoas cada uma. Depois, o algoritmo verifica quais nomes são iguais, em posições correspondentes.

19. Faça um programa que copia uma matriz 10 x 10 de caracteres para um vetor de 100 posições.

20. Sejam dois vetores V1 e V2, de tamanho 20 cada um, do tipo caractere. Escreva um programa que, além de carregar os dois vetores, gere:
    a) Um vetor V3 correspondente à união de V1 e V2.
    b) Um vetor V4 com os elementos comuns de V1 e V2.
    c) Um vetor V5 com os elementos de V2 que não estão em V1.
    d) Um vetor V6 que é o resultado da comparação de V1 com V2, posição a posição. Quando forem iguais, o valor é 1. Quando forem diferentes, o valor é 0.

21. Sejam M1, M2 e M3 matrizes 4 x 4 de números inteiros. Faça um programa que calcule a soma dessas matrizes.

6. Modularização de programas

Neste ponto, já foram examinados grande parte dos operadores e das estruturas de dados da linguagem Pascal. Com tais elementos e com a ajuda das instruções de fluxo de programa, é possível escrever programas complexos que repetem determinadas ações e que tomam decisões. Todavia, na medida em que os programas crescem, o programa principal (delimitado pelo BEGIN e END, que marcam o início e o fim do programa) tende a se expandir, como um balão de ar quente com sua válvula de entrada aberta. A menos que se faça algo para aliviar a pressão interna, cedo ou tarde o balão explodirá. Assim sendo, a menos que se faça algo para impedir o abarrotamento do programa principal com instruções após instruções, o programa pode entrar em pane na medida em que ele se torna mais difícil de ser desenvolvido e mesmo mantido pelo programador.

Um programa que se estenda por várias páginas de código-fonte é, além de deselegante, susceptível a erros de programação denominados de *bugs*. Um *bug* é um erro de programação que o compilador não consegue tratar, pois se trata de um erro de lógica de programação e não de sintaxe da linguagem.

Um exemplo seria digitar a palavra "faça" em um editor de texto e, por um erro não intencional de digitação, ser escrita a palavra "faca". Como ambas as palavras são válidas na língua portuguesa, o corretor ortográfico não acusará o erro e, dessa forma, tem-se um *bug* no código-fonte. Bons programadores evitam escrever extensos códigos-fonte. Em vez disso, eles dividem os seus programas em pequenas partes, que são mais fáceis de administrar.

A técnica mais utilizada e tida como a mais vantajosa na confecção de programas de grande porte é a modularização do programa, que consiste na divisão dele em diversos módulos, ou subprogramas, de forma independente uns dos outros. O módulo principal, a partir do qual são chamados os outros módulos, recebe o nome de programa principal, ao passo que os outros módulos são chamados de sub-rotinas. Na linguagem Pascal existem dois tipos de sub-rotinas: os procedimentos (do inglês *procedures*) e as funções (do inglês *functions*).

6.1 Procedimentos (PROCEDURES)

Um procedimento define um conjunto de comandos que serão executados uma ou mais vezes no decorrer de um programa. Um procedimento tem a mesma estrutura básica de um programa, com declarações de variáveis e de constantes, e deve ser definido antes de ser utilizado em um programa. Dessa forma, um procedimento age como se fosse um subprograma dentro de um programa. Um procedimento pode ser ativado (ou chamado) pelo programa principal, por outro procedimento, por uma função, ou até por ele mesmo. A diferença entre um procedimento e uma função é que a função retorna um valor, ao passo que um procedimento não retorna valor algum. O procedimento CLRSCR não retorna valor algum, ele simplesmente apaga a tela e encerra a sua execução. O mesmo não acontece com a função READKEY, que retorna a tecla digitada pelo usuário.

Um procedimento tem praticamente a mesma estrutura de um programa, possuindo um cabeçalho, uma área de declarações e o corpo do procedimento. Na área de declarações, pode-se ter as seguintes subáreas:

```
Label – Const – Type – Var – Procedure – Function
```

Tudo que for declarado dentro de um procedimento só será reconhecido dentro dele próprio. Esse conceito é denominado de escopo, e será abordado em detalhes ainda neste capítulo.

O código 6.1 apresenta um programa que utiliza um procedimento chamado `linha`, para traçar uma linha horizontal na tela do computador.

Código 6.1 - Uso de procedimentos em Pascal

```
Program Exemplo_PROCEDURE_1;  (* cabeçalho do programa *)
Uses CRT;

Procedure linha;   (* cabeçalho da procedimento linha *)
Var i: integer;        (* subárea Var do procedimento linha *)
Begin                          (* corpo da procedimento linha *)
  For i:=1 to 80 do write('-');
End;

Begin                          (* corpo do programa principal *)
  ClrScr;
  linha;                       (* chamada ao procedimento linha *)
  writeln('teste');
  linha;                       (*outra chamada ao procedimento linha*)
  Readkey;
End.
```

O programa anterior realiza os seguintes comandos:

i. Apaga a tela e coloca o cursor em 1,1.
ii. Ativa o procedimento linha.
iii. Escreve a palavra teste.
iv. Ativa novamente o procedimento linha.
v. Termina o programa.

O procedimento `linha` traça uma linha horizontal composta por oitenta sinais de subtração a partir da posição corrente do cursor na tela. Como a variável inteira *i*, definida dentro do procedimento linha, só existe dentro desse procedimento, isso significa que toda vez que o procedimento `linha` é

ativado, a variável *i* é criada na memória; e toda vez que esse procedimento termina, a variável é destruída da memória do computador.

No exemplo acima não houve passagem de nenhum parâmetro para o procedimento linha na sua ativação. Para criar um procedimento também chamado linha com a mesma função do procedimento apresentado no código 6.1, mas que permita ao usuário escolher qual o caractere a ser usado para a criação da linha e também o número de repetições desse caractere, é necessária a passagem de dois parâmetros para o procedimento. O código 6.2 apresenta o procedimento linha com a implementação dos parâmetros desejados.

Código 6.2 - Procedimento LINHA com passagem de parâmetros

```
Program Exemplo_PROCEDURE_2;
Uses CRT;

Procedure linha(ch: char; n: integer);
Var i: integer;
Begin
  For i:=1 to n do write(ch);
  Writeln;
End;

Var
  ch: char;
  n: integer;
Begin
  ClrScr;
  Write('Caractere a ser utilizado para criar a linha: ');
  ch := Readkey;
  writeln(ch);
  Write('Digite o tamanho da linha: ');
  Readln(n);
  linha(ch, n);        (* chamada ao procedimento linha *)
  writeln('teste');
  linha(ch, n);        (*outra chamada ao procedimento linha*)
  Readkey;
End.
```

A sintaxe geral de um procedimento é:

```
PROCEDURE identificador (parâmetro_1, parâmetro_2: Tipo_1; parâmetro_3, parâmetro_4: Tipo_2; ...; parâmetro_m, parâmetro_n: Tipo_n); [EXTERNAL;] [INTERRUPT;] [FORWARD;]
[TYPE]
[VAR]
[CONST]
BEGIN
  [comandos;]
END;
```

Os termos que estão entre colchetes são de uso opcional.

Embora a existência de parâmetros em um procedimento também seja opcional, os colchetes antes da lista de parâmetros foram omitidos. O código 6.3 apresenta um procedimento com passagem de parâmetros cuja finalidade é somar dois números inteiros quaisquer.

Código 6.3 - Programa que utiliza passagem de parâmetros para um procedimento

```
Program Exemplo_PROCEDURE_3;
Uses CRT;
Var i, j: integer;

Procedure soma(x,y: integer);
Begin
  writeln(x + y);
End;

Begin
  ClrScr;
  soma(3,4);
  i := 45;
  j:=34;
  soma(i, j);
  Readkey;
End.
```

Como o procedimento soma depende de dois parâmetros inteiros e na sua chamada deverão ser fornecidos os dois parâmetros, estes podem ser dois

números ou duas variáveis do tipo inteiro. Caso não haja compatibilidade entre os parâmetros passados ao procedimento e os parâmetros declarados no cabeçalho deste, o compilador gerará um erro na compilação do programa.

A passagem de parâmetros dos exemplos anteriores é chamada de *passagem de parâmetros por valor*, pois é feita a atribuição do valor da variável que foi passada como argumento ao procedimento à variável declarada no cabeçalho do procedimento. Apesar de, por definição, os procedimentos não retornarem nenhum valor, existe uma maneira para que eles também possam retornar dados, utilizando o conceito de passagem de *parâmetros por referência*. O código 6.4 apresenta um exemplo desse tipo de passagem de parâmetro.

Código 6.4 - Passagem de parâmetros por referência para um procedimento

```
Program Exemplo_PROCEDURE_4;
Uses CRT;
Var i: Integer;

Procedure Soma(x, y:Integer; Var z:Integer);
Begin
  z := x + y;
End;

Begin
  ClrScr;
  Soma(3, 4, i);
  Writeln(i);
  Readkey;
End.
```

Quando o procedimento `soma` apresentado no código 6.4 for chamado com o comando `Soma(3, 4, i),` ocorrerão as seguintes passagens:

i. O parâmetro x da função `soma` receberá o valor 3.
ii. O parâmetro y da função `soma` receberá o valor 4.
iii. O parâmetro z da função `soma` receberá o valor 7 (z := x + y).
iv. A variável i do programa principal receberá o valor retornado pelo parâmetro z.

Nota-se que houve uma passagem de dados do programa para o procedimento e, ao término do procedimento, dele de volta para o programa. A declaração das variáveis que serão utilizadas como parâmetro de referência vem precedida pela declaração VAR. O uso dessa declaração indica ao compilador que o local do endereço de memória usado para armazenar a variável local (declarada dentro de um procedimento ou função) deverá ser o mesmo endereço que o da variável global (declarada no programa principal) passada como parâmetro. Assim sendo, quando alteramos o valor da variável local, alteramos também o valor da variável global. O código 6.5 apresenta um programa com dois procedimentos, um chamado soma; outro, linha.

Código 6.5 - Programa que possui dois procedimentos, um chamando o outro

```
Program Exemplo_PROCEDURE_5;
Uses CRT;

Procedure Soma(x,y:Integer);
Begin
  linha;
  Writeln(x + y);
  Readkey;
End;

Procedure Linha;
Var i: integer;
Begin
  For i:=1 to 80 do Write('-');
End;

Begin
  ClrScr;
  Soma(3,4);
End.
```

O procedimento SOMA invoca em seus comandos o procedimento LINHA. No entanto, o procedimento LINHA está declarado mais à frente no código e, portanto, o compilador gerará um erro, pois ele não reconhecerá o

identificador LINHA.[8] Isso se deve ao fato de a compilação ser feita de cima para baixo e da esquerda para a direita. O uso da declaração FORWARD, cuja finalidade é a de indicar ao compilador que determinado procedimento está definido mais à frente, resolve problemas desse tipo, pois o compilador procurará nas próximas linhas de código a declaração do procedimento ou função. O código 6.5 mostra um programa que não compilará, a não ser que a declaração FORWARD seja utilizada, conforme mostrado no código 6.6.

Código 6.6 - Uso da declaração FORWARD

```
Program Exemplo_PROCEDURE_6;
Uses CRT;

Procedure Linha; Forward;

Procedure Soma(x,y:Integer);
Begin
  linha;
  Writeln(x + y);
  Readkey;
End;

Procedure Linha;
Var i: integer;
Begin
  For i:=1 to 80 do Write('-');
End;

Begin
  ClrScr;
  Soma(3,4);
End.
```

Instruções podem se referir a variáveis apenas quando essas variáveis se encontram dentro do mesmo *escopo*, ou nível, das instruções. O escopo de uma variável se estende até as fronteiras do bloco que lhe define.

[8] O compilador emitirá a seguinte mensagem: "Error 3: Unknown identifier".

O código 6.7 apresenta um programa que aborda o escopo de diferentes objetos da linguagem Pascal.

Código 6.7 - Escopo de objetos em um programa na linguagem Pascal

```
Program Exemplo_PROCEDURE_7;
Uses CRT;
Const a = 100;                                    (* constante global *)
Label  fim;                                       (* Label global *)
Var  i, x, y: Integer;              (* variáveis globais *)

Procedure Linha;
Var i: Integer;          (* i é local ao procedimento linha *)
Begin
  For i:=1 to 80 do Write('-');
End;

Procedure Teste;
  Procedure Sub_teste;
(* O procedimento Sub_teste é local ao procedimento Teste*)
  Begin
    Write('Estive em sub_teste');
    Readkey;
  End;
Begin
  Sub_teste;
  Writeln('Estive em Teste');
  Readkey;
End;

Begin
  ClrScr;
  i := 100;
  Linha;
  x := 20;
  y := 30;
  Teste;
  Linha;
  Writeln('i = ',i,', y = ',y,' e x = ',x);
  Readkey;
End.
```

Todos os elementos (constantes, variáveis, rótulos, etc.) que forem definidos no programa principal (após a declaração PROGRAM) são considerados globais (ou de *escopo global*) e podem ser utilizados por todos os procedimentos, funções e pelo próprio programa.

O espaço da memória que armazenará tais elementos é alocado durante a compilação do programa. Já os elementos declarados dentro de um procedimento ou de uma função só existem dentro da sub-rotina que o declarou. Tais elementos são considerados locais ao procedimento (ou de *escopo local*). Como exemplo, pode-se notar que existe uma variável *i* inteira declarada antes do início do programa, portanto global, e outra dentro do procedimento linha, portanto local a esse procedimento. Não há nenhum problema nesse tipo de declaração, uma vez que o Pascal considerará as variáveis como diferentes uma da outra. O código escrito dentro do procedimento acessará a variável i local e o código fora do procedimento acessará a variável i global.

Ainda que variáveis de dois escopos diferentes se apresentem de maneira muito semelhante nos programas, elas são armazenadas na memória do computador de maneiras diferentes. O compilador reserva um espaço fixo na memória do computador para cada variável global, denominado *segmento de dados*. Ao fazer isso, o compilador inicializa todas as variáveis globais com o valor zero (no caso das variáveis do tipo caractere, o seu conteúdo é nulo ou vazio, do inglês *void*). Já para as variáveis locais, o compilador cria um espaço temporário denominado de *pilha* (do inglês *stack*) e elas são inicializadas com lixo da memória do computador.

Um programa útil em qualquer linguagem de programação é o programa que testa se o compilador está ou não inicializando corretamente as variáveis declaradas em seu código-fonte. Seguindo esse raciocínio, o código 6.8 mostra um programa que declara duas variáveis de cada tipo simples da linguagem Pascal, sendo uma delas global e outra local. Em seguida, o programa imprime na tela o valor de cada variável. Como era de se esperar, as variáveis globais são inicializadas com zero (ou nulo, no caso das variáveis do tipo caractere), mas com as variáveis locais o resultado é surpreendente.

Código 6.8 - Programa para o teste da inicialização de variáveis globais e locais

```
Program Teste_Var;
Uses CRT;
Var                                   (* variáveis globais *)
  a: Shortint;
  b: Integer;
  c: Longint;
  d: Byte;
  e: Word;
  f: Real;
  g: Single;
  h: Double;
  i: Extended;
  j: Comp;
  k: Char;
  l: String[10];

Procedure Local;
Var                                   (* Variáveis locais ao procedimento Local *)
  a: Shortint;
  b: Integer;
  c: Longint;
  d: Byte;
  e: Word;
  f: Real;
  g: Single;
  h: Double;
  i: Extended;
  j: Comp;
  k: Char;
  l: String[10];

Begin
  Writeln('VARIÁVEIS LOCAIS':30);
  Writeln;Writeln;
  Writeln('a = ', a);
  Writeln('b = ', b);
  Writeln('c = ', c);
  Writeln('d = ', d);
  Writeln('e = ', e);
  Writeln('f = ', f);
  Writeln('g = ', g);
```

```
  Writeln('h = ', h);
  Writeln('i = ', i);
  Writeln('j = ', j);
  Writeln('k = ', k);
  Writeln('l = ', l);
  Readkey;
End;

Begin
  ClrScr;
  Writeln('VARIÁVEIS GLOBAIS':30);
  Writeln;Writeln;
  Writeln('a = ', a);
  Writeln('b = ', b);
  Writeln('c = ', c);
  Writeln('d = ', d);
  Writeln('e = ', e);
  Writeln('f = ', f);
  Writeln('g = ', g);
  Writeln('h = ', h);
  Writeln('i = ', i);
  Writeln('j = ', j);
  Writeln('k = ', k);
  Writeln('l = ', l);
  Readkey;
  Local;
End.
```

6.2 Funções (FUNCTIONS)

Tal como os procedimentos, as funções são pequenas partes de um programa. Elas reúnem várias instruções sob um único nome, que, dessa forma, pode ser chamado uma ou mais vezes pelo programa, para a execução das instruções. Funções economizam espaço, reduzindo as repetições, e tornam a programação mais fácil, fornecendo um meio de dividir um grande projeto em pequenos módulos de fácil concepção e administração.

Diferentemente dos procedimentos, as funções são rotinas que retornam um determinado valor. A sintaxe de uma função é muito parecida com a sintaxe de um procedimento, sendo geralmente dada por:

```
FUNCTION identificador (parâmetro_1, parâmetro_2: Tipo_1; parâmetro_3, parâmetro_4: Tipo_2; ...; parâmetro_m, parâmetro_n: Tipo_n): tipo_da_função; [EXTERNAL;] [FORWARD;]

[TYPE]
[VAR]
[CONST]
BEGIN
  [comandos;]
END;
```

Uma função retorna apenas um resultado e o tipo da função é definido como o tipo do resultado retornado por ela. Caso se deseje que uma função retorne mais que um valor, basta utilizar passagem de parâmetros por referência. Na área de declarações podemos declarar rótulos, constantes, variáveis e até mesmo procedimentos e funções. Deve-se lembrar que tais elementos só poderão ser utilizados dentro do corpo da função, pois são locais a ela. O código 6.9 apresenta um exemplo de uma função em Pascal.

Código 6.9 - Uso de funções na linguagem Pascal

```
Program Exemplo_FUNCTION_1;
Uses CRT;
Var x, y: Real;                          (* variáveis globais *)

Function Soma(a, b:real):real;
(* Soma é uma função que depende de dois parâmetros reais e retorna um valor real *)
Begin
  Soma := a + b;
(* reparem que o valor da função é retornado pelo seu próprio nome*)
End;

Begin
  ClrScr;
  x := Soma(4, 5);
  y := Soma(3, 6) – Soma(45.5, 5.6);
  Writeln(x:10:2,y:10:2);
  Writeln;
```

```
  Write('Valor de x --> ');
  Readln(x);
  Write('Valor de y --> ');
  Readln(y);
  Writeln;
  Writeln(Soma(x,y):10:2);
  Readkey;
End.
```

O código 6.9 mostra que, após uma função executar os devidos cálculos necessários, deve-se atribuir o valor que será retornado ao programa principal ao próprio nome da função. Isso pode representar um problema quando se trabalha com funções *recursivas*, que são funções que chamam elas próprias (assunto do próximo tópico deste livro). É sugerida, então, a criação de uma variável do mesmo tipo da função denominada *Result* (ou resultado). Ao término da função, o valor da variável *Result* é atribuído à função. Os programas que serão apresentados neste livro terão a declaração da variável adicional *Result* em suas funções, como mostra o código 6.10, que apresenta um programa que calcula o fatorial de um número inteiro *n* lido do teclado.

Código 6.10 - Cálculo do fatorial de um número inteiro

```
Program Fat;
Uses CRT;
Var
  n: Integer;
  ch: char;

Function Fatorial(n: integer): Real;
Var
  i: Integer;
  Result: Real;
Begin                    (* Begin da função Fatorial *)
  Result := 1;
  If (n > 1) Then
    For i := n downto 1 do
```

```
    Result := Result * i;
  Fatorial := Result;
End;                          (* End da função Fatorial *)

Begin                  (* Begin do programa *)
  Repeat
    ClrScr;
    Write('Valor de n (0 para sair) --> ');
    Readln(n);
    Writeln;
    If (n > 0) Then
    Begin
      Writeln('Fatorial de n = ',fatorial(n):10:0);
      Writeln;Writeln;
      Write('Deseja calcular outro fatorial (S/N)?');
      ch := UPCASE(Readkey);
    End;
  Until (ch = 'N') or (n <= 0);
End.                          (* End do programa *)
```

O argumento de uma função ou de um procedimento pode ser outra função, desde que o argumento da função mais externa seja do mesmo tipo da função mais interna (ou função argumento), como no trecho abaixo:

```
ch := UPCASE(Readkey);
```

A função UPCASE tem como parâmetro a própria função READKEY. Como o resultado da função READKEY é um caractere, que é justamente o argumento da função UPCASE, a condição mencionada acima foi respeitada. O código 6.11 apresenta um programa para calcular um determinado elemento da série de Fibonacci, que é definida como:

$$Fib(0) = 0$$
$$Fib(1) = 1$$
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

O elemento atual da série de Fibonacci, dado por Fib(n), é determinado pela soma dos dois elementos anteriores da sequência.

Código 6.11 - Programa para cálculo do valor da série de Fibonacci de um número *n*

```
Program Fibonacci;
Uses CRT;
Var
  n:integer;
  ch: char;

Function Fib(n:integer):integer;
Var
  a1, a2, i, Result: Integer;
Begin
  If (n = 0) Then
    Result := 0
  Else If (n = 1) Then
    Result := 1
  Else
  Begin
    a1 := 0;
    a2 := 1;
    For i :=1 to n-1 do
    Begin
      Result := a1 + a2;
      a1 := a2;
      a2 := Result;
    End;
  End;
  Fib := Result;
End;

Begin
  ClrScr;
  Repeat
    Write('Fib(');
    Read(n);
    Writeln(') = ',fib(n));
    Writeln;
    Write('Deseja calcular outro número (S/N)? ');
    ch := UPCASE(Readkey);
    writeln;
  Until (ch = 'N');
End.
```

6.3 Recursividade

Uma função é dita recursiva quando ela chama a si mesma. Deve-se tomar cuidado ao lidar com esse tipo de função, pois o uso incorreto da recursividade pode levar à criação de *loops* infinitos. A recursividade permite criar funções elegantes e tornar os programas mais fáceis de serem entendidos. Os códigos 6.12 e 6.13 apresentam os programas anteriores, códigos 6.10 e 6.11, respectivamente, reescritos utilizando funções recursivas.

Código 6.12 - Cálculo do fatorial de um número usando recursividade

```
Program Fatorial;
Uses
  CRT;
Var
  n: Integer;
  ch: char;
Function Fat(n:integer):real;
Var
  Result: Real;
Begin
  if (n = 0) Then Result := 1
  Else Result := n * Fat(n-1);
(* repare que estamos chamando novamente a função Fat *)
  Fat := Result;
End;

Begin
  ClrScr;
  Repeat
    Write('Valor de n (0 para sair) --> ');
    Readln(n);
    Writeln;
    If (n > 0) Then
    Begin
      Writeln('Fatorial de n = ',Fat(n):10:0);
      Writeln;Writeln;
      Write('Deseja calcular outro fatorial (S/N)?');
      ch := UPCASE(Readkey);
    End;
  Until (ch = 'N') or (n <= 0);
End.
```

Código 6.13 - Cálculo da série de Fibonacci usando recursividade

```
Program Fibonacci;
Uses
  CRT;
Var
  n: integer;
  ch: char;
Function Fib(n:integer):integer;
Var
  Result: integer;
Begin
  If (n = 0) Then Result := 0
  Else If (n = 1) Then Result := 1
  Else Result := Fib(n-1) + Fib(n-2);
  Fib := Result;
End;

Begin
  ClrScr;
  Repeat
    Write('Fib(');
    Read(n);
    Writeln(') = ',fib(n));
    Writeln;
    Write(' Deseja calcular outro número (S/N)? ');
    ch := UPCASE(Readkey);
    writeln;
  Until (ch = 'N');
End.
```

6.4 Criando unidades de programa (UNIT)

O desenvolvimento de uma UNIT é semelhante ao desenvolvimento de um programa na linguagem Pascal. Para criar uma UNIT contendo um conjunto de funções, basta criar um arquivo no ambiente Turbo Pascal, seguindo a estrutura apresentada abaixo:

```
UNIT nome_da_unit;
INTERFACE
{nesta subseção são definidos os cabeçalho das sub-rotinas}
```

```
  Function inttobin(x: word): double;
  Function bintoint(y:double): word;

IMPLEMENTATION
USES CRT; {lista de UNITs a serem usadas na unidade atual}
VAR     {declarações das variáveis globais da UNIT}
  i : Integer;
  c: char;
{Nesta área são implementadas as sub-rotinas}
Function inttobin(x: word): double;
Begin
...
End;

...
End.
```

A necessidade da criação de uma base numérica remonta ao tempo em que o ser humano começava a caminhar pelas árduas trilhas da matemática. Obviamente, a base adotada foi a base decimal, uma vez que o primeiro instrumento de cálculo que o ser humano adotou foram os dedos das suas mãos. Apesar de extremamente funcional para a matemática, a base decimal não é a base numérica utilizada pelos computares, uma vez que estes utilizam a base binária, que, como o próprio nome já diz, utiliza a base dois.

Quando os primeiros computadores foram construídos, eles trabalharam com o conceito de 0 ou 1. Por exemplo, lâmpada apagada (0) ou acessa (1), resultado de uma conta TRUE (1) ou FALSE (0). Esse conceito ainda não mudou, mesmo nos PCs mais modernos. Internamente, um computador faz contas utilizando apenas os números 0 e/ou 1. Uma vez que a base binária é formada apenas com os algarismos 0 e 1, pode-se estabelecer a seguinte correlação entre as duas bases:

BASE 10	BASE 2
0	0
1	1
2	10
3	11

(continua)

(fim)

BASE 10	BASE 2
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
e assim por diante...	

Para converter um número da base decimal para a base binária, basta dividir o número que se deseja converter por dois, sucessivamente, até que o resto seja 0. Uma vez realizada essa operação, monta-se o binário, utilizando os restos da divisão ordenados da última para a primeira divisão. Abaixo, segue a conversão do decimal vinte e três no seu respectivo binário:

23 / 2 = 11 e resto **1**
11 / 2 = 5 e resto **1**
5 / 2 = 2 e resto **1**
2 / 2 = 1 e resto **0**
1 / 2 = 0 e resto **1**

Portanto, o decimal 23 é igual ao binário 10111. Para a realização da operação inversa, conversão da base 2 para a base 10, utiliza-se um polinômio de potências de dois, em que os coeficientes do polinômio são os algarismos do binário. O exemplo abaixo mostra a conversão do binário 1011 para o decimal 23.

$1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 =$
$16 + 0 + 4 + 2 + 1 = 23$

O polinômio para a conversão de um número na base binária para a base decimal é dado por:

$$a_n.2^n + ... + a_3.2^3 + a_2.2^2 + a_1.2 + a_0 = \sum_{i=0}^{n} a_i.2^i$$

a_n é o n-ésimo dígito do número binário e n é a ordem do n-ésimo dígito do binário. Abaixo seguem duas funções na linguagem Pascal destinadas à conversão de um número da base decimal para a base binária (INTTOBIN) e da base binária para a base decimal (BINTOINT). O código 6.14 apresenta uma unidade denominada CONVERSAO_BINARIO composta por duas funções para conversão entre a base binária e a base decimal.

Código 6.14 - Estrutura interna de uma UNIT.

```
UNIT CONVERSAO_BINARIO;
INTERFACE
{cabeçalho das funções da UNIT}
  Function inttobin(x: word): double;
  Function bintoint(y:double): word;

IMPLEMENTATION
USES CRT; {lista de UNITs a serem usadas}

Function inttobin(x: word): double;
Var
 resto: word;
 y, i: double;
Begin
  y := 0;
  if (x = 0) then y := 0
  else if (x = 1) then y := 1
  else
   begin
    i := 1;
    while (x > 1) do
     begin
      resto := x MOD 2;
      x := ROUND((x – resto)/2);
      y := y + (i * resto);
      i := i * 10;
      if (x = 1) then y := y + i;
```

```
    end;
   end;
  inttobin := y;
End;

Function bintoint(y:double): word;
Var
  x, bin: word;
  i, j, k: integer;
Begin
  bin := ROUND(y);
  if (bin = 0) then x := 0
  else if (bin = 1) then x := 1
  else
   begin
    i := 0;
    x := 0;
    while (bin > 1) do
     begin
      if (bin MOD 2 <> 0) then
       begin
        k := 1;
        for j := 1 to i do k := k * 2;
        x := x + k;
        bin := ROUND((bin – 1)/10);
       end
      else bin := ROUND(bin / 10);
      i := i + 1;
     end;
    k := 1;
    for j := 1 to i do k := k * 2;
    x := x + k;
   end;
  bintoint := x;
End;
END.
```

Para utilizar a unidade CONVERSAO_BINARIO em um programa qualquer, basta adicionar o nome da unidade (no caso CONVERSAO_BINARIO) na subárea USES do programa. Quando alguma das funções da unidade for chamada, ela será executada como se estivesse declarada no próprio programa principal.

6.5 Exercícios

1. Mostre o que será impresso pelo programa a seguir.

```
program calculo;
var a, b, c : integer;
procedure p(x, y : integer; var z: integer);
begin
  z := x + y + z;
  writeln (x, y, z);
end;
begin
  a:=5; b:=8; c:=3;
  p(a, b, c);
  p(7, a+b+c, a);
  p(a*b, a div b, c);
end.
```

2. Faça uma função que recebe um valor inteiro e verifica se o valor é positivo ou negativo. A função deve retornar um valor booliano.

3. Faça uma função que recebe um valor inteiro e verifica se o valor é par ou ímpar. A função deve retornar um valor booliano.

4. Construa uma função chamada SIGMA para calcular a somatória dos 50 elementos de um vetor (o vetor é parâmetro da função).

5. Construa uma função chamada SISTEMA com dois parâmetros, x e n, que retorne o seguinte:

$$x + \frac{x^n}{n} - \frac{x^{n+2}}{n+2} \quad se\ x \geq 0 \qquad x + \frac{x^{n-1}}{n-1} - \frac{x^{n+1}}{n+1} \quad se\ x < 0$$

   Observação: a linguagem Pascal não possui uma função embutida para cálculo de potência. Construa uma função chamada POTENCIA para realizar esse cálculo (vide Código 5.10). O escopo dessa função deve estar restrito à função SISTEMA.

6. Faça um procedimento que recebe a idade de um nadador por parâmetro e retorna, também por parâmetro, a categoria desse nadador de acordo com a tabela abaixo:

Idade	Categoria
5 a 7 anos	Infantil A
8 a 10 anos	Infantil B
11 a 13 anos	Juvenil A
14 a 17 anos	Juvenil B
Maiores de 18 anos (inclusive)	Adulto

7. Foi realizada uma pesquisa entre 500 habitantes de certa região. De cada habitante foram coletados os dados: idade, sexo, salário e número de filhos. Faça um procedimento que leia esses dados em um vetor de registros. O vetor de registros deve ser enviado por referência.

8. Faça um procedimento que receba o vetor de registro definido no exercício anterior, por parâmetro, e retorna também por parâmetro: a média de salário entre os habitantes, a menor e a maior idade do grupo e a quantidade de mulheres com 3 filhos que recebe até R$ 510,00.

9. Faça um procedimento que recebe um vetor X de 30 elementos inteiros, por parâmetro, e retorna, também por parâmetro, dois vetores A e B. O vetor A deve conter os elementos pares de X e o vetor B, os elementos ímpares.

10. Faça um procedimento que recebe 2 vetores A e B de tamanho 10 de inteiros, por parâmetro. Ao final do procedimento, B deve conter o fatorial de cada elemento de A. O vetor B deve retornar alterado.

A	4	1	0	3	....
B	24	1	1	6	...

11. Uma locadora de vídeos tem guardada, em um vetor A de 500 posições, a quantidade de filmes retirados por seus clientes durante um ano. Agora, essa locadora está fazendo uma promoção e, para cada 15 filmes retirados, o cliente tem direito a uma locação grátis. Faça um procedimento que receba o vetor A por parâmetro e retorna, também por parâmetro, um vetor contendo a quantidade de locações gratuitas a que cada cliente tem direito.

12. Faça um procedimento que recebe, por parâmetro, um vetor A(25) de inteiros e substitui todos os valores negativos de A por zero. O vetor A deve retornar alterado.

13. Existem 4 candidatos a governador. Feita a eleição, os votos são digitados, via teclado, um a um. O voto de cada eleitor foi codificado da seguinte forma:

```
1 2 3 4 => votos para os candidatos 1, 2, 3 e 4, respectivamente.
      0 => voto branco.
      9 => voto nulo.
```

Faça um programa com módulos (procedimentos e/ou funções) para determinar: o número do candidato vencedor; o número de votos em branco; o número de votos nulos e o número de eleitores que compareceram às urnas. Admite-se que não são possíveis empates.

14. Faça um programa com módulos (procedimentos e/ou funções) que mostre os conceitos finais dos 75 alunos de uma classe, considerando:

a) Os dados de cada aluno (número de matrícula e nota numérica final) serão fornecidos pelo usuário.

b) A tabela de conceitos segue abaixo.

Nota	Conceito
De 0,0 a 4,9	D
De 5,0 a 6,9	C
De 7,0 a 8,9	B
De 9,0 a 10,0	A

15. Desenvolva um programa com módulos (procedimentos e/ou funções) que determinem o salário bruto de cada um de três empregados. A empresa paga "hora normal" pelas primeiras 40 horas trabalhadas de cada empregado e "horas extras" com 50% de gratificação para todas as horas trabalhadas além de 40 horas. Você recebe uma lista de empregados da empresa, o número de horas trabalhadas de cada empregado na última semana e o salário-hora de cada empregado. Seu programa deve ler essas informações para cada empregado e deve determinar e exibir o salário bruto do empregado.

16. Desenvolva um programa com módulos (procedimentos e/ou funções) que determinem se um cliente de uma loja excedeu o limite de crédito em uma conta-corrente. Para cada cliente, os seguintes dados estão disponíveis:

a) Número da conta.

b) Saldo no início do mês.
c) Total de todos os itens cobrados desse cliente no mês.
d) Total de pagamentos feitos pelo cliente no mês.
e) Limite autorizado de crédito.

O programa deve ler cada um desses dados como inteiros, calcular o novo saldo (novo saldo = saldo inicial + cobrança – pagamentos), exibir o novo saldo e determinar se o mesmo excede o limite de crédito do cliente. Para aqueles clientes cujo limite de crédito for excedido, o programa deve exibir a mensagem "*limite de crédito excedido*".

17. Faça um programa que calcula a área de um quadrado, um triângulo e um círculo. Cada uma é calculada por meio de uma função específica.

18. Incremente o algoritmo anterior de forma que seja apresentado um menu de opções com as três áreas possíveis, além da opção Sair. Esse menu deve ficar dentro de uma função específica, que retorna o valor selecionado.

19. Faça um programa que leia o preço e a quantidade de um produto e calcula o valor da compra. Uma função deve ser definida para, a partir do código do produto, fornecer seu preço. Outra função deve permitir ao usuário selecionar o produto e indicar a quantidade. Uma terceira exibe o resultado. Utilize apenas nomes globais.

20. Faça um programa que possua uma função denominada ImpAsterisco. Ela recebe por parâmetro o número de asteriscos a serem impressos na tela e os imprime.

21. Faça um programa que possua a função ImpCaracteres. Ela recebe por parâmetro um caractere a ser exibido e o imprime.

22. Utilizando a função do exercício anterior, como você geraria na tela o texto "asd***klo"?

23. Faça um programa que possua a função Imp10Caracteres. Ela recebe um caractere e o imprime 10 vezes.

24. Como você faria para gerar um quadrado de 10 por 10 caracteres no centro da tela, utilizando as funções ImpCaractere e Imp10Caracteres?

25. Faça uma função que recebe por valor dois inteiros, e retorna, por referência, a soma de ambos. Caso ambos os valores sejam menores que 100, a função retorna –1 e a função principal que a chamou informa que os valores passados eram inválidos.

26. O fatorial de um número inteiro n é definido como:

$$n! = 1 \text{ se } n = 0 \quad \text{e} \quad n! = n(n-1)! \text{ se } n > 0$$

a) Escreva um programa não recursivo (interativo) que calcule o fatorial de n.

b) Escreva um programa recursivo que calcule o fatorial de n.

27. Faça uma função recursiva que calcule o valor de *S* dado *N*, em que:

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N}$$

28. Dada a função X definida abaixo:

```
function X(n,m : integer):integer;
Begin
  if (n=m) or (m=0) then x:=1
  else x := x(n-1,m) + x(n-1,m-1);
End;
```

Pergunta-se:

a) Qual o valor de X (5,3)?

b) Quantas chamadas serão feitas na letra (a)?

29. Dada a função abaixo:

```
Function X(n: integer):integer;
Begin
  if n=0 then x:=0
  else if n=1 then x:=1
  else if n=2 then x:=2
  else x := x(n-1) + x(n-2) + x(n-3);
End;
```

Pergunta-se: Quantas chamadas serão executadas para avaliar X(4)?

30. Mostre passo a passo o que será impresso pelo programa abaixo:

```
Program prog;
var i,j : integer;

function C : integer;
  function f(x:integer):integer;
```

```
  begin (* f  *)
    if x=0 then f := 1
    else f := x * f(x-1);
  end; (* f *)
begin (* C *)
  c := i * f(i) div f(i-1);
end;

begin (* prog *)
  for i:=1 to 4 do
  begin
    for j:=1 to i do
      write(C);
    writeln;
  end;
end.
```

31. Os polinômios de Legendre podem ser calculados da seguinte forma:

$$P(i) = \frac{2i-1}{i}.x.P(i-1).\frac{i-1}{i}.P(i-2), \text{ se i} > 1$$

$$P(i) = x, \text{ se i} = 1$$

$$P(i) = 1, \text{ se i} = 0$$

x é qualquer número real entre –1 e 1 e i = 2, 3, 4,... Escreva uma função recursiva que retorne o i-ésimo valor do polinômio. Faça os valores de i e x parâmetros de entrada da função.

7. Manipulação de arquivos em disco

Quando se pensa em um arquivo, é possível imaginar um arquivo físico, daqueles de aço encontrados normalmente em escritórios e empresas. De forma geral, um arquivo se presta ao armazenamento de informações por meio de fichas. Em se tratando de programação de computadores, esses conceitos são perfeitamente válidos, pois, para o armazenamento de informações em um arquivo, é necessário dar forma ao arquivo para, então, adicionar informações a ele. O tipo de dado arquivo (ou FILE, em inglês) é uma estrutura constituída de elementos do mesmo tipo dispostos sequencialmente. Essa estrutura é utilizada para comunicação com o meio externo, principalmente com discos magnéticos. A sintaxe geral para definir uma variável do tipo arquivo é:

```
Type
  Arquivo = File of Tipo_do_arquivo;
Var
  a: Arquivo;
```

É possível declarar uma variável diretamente como um arquivo, da seguinte forma:

```
Var
  a: File of Tipo_do_arquivo;
```

Após as declarações acima, a variável `a` passa a representar um arquivo de elementos do tipo `Tipo_do_arquivo`. Abaixo, seguem alguns exemplos de arquivos de diferentes tipos de dados:

```
Type
  Arq = File Of Integer;
  Pessoa = Record
    Nome: String[30];
    Idade: Integer;
    Sexo: Char;
    Altura: Real;
  End;

Var
  Arquivo_Int: Arq;
  Arquivo_Real: File Of Real;
  Arquivo: File Of Pessoa;
```

O acesso a um arquivo sempre segue uma mesma sequência lógica:

i. Associação de um identificador a um arquivo.
ii. Abertura do arquivo.
iii. Leitura e/ou escrita de dados no arquivo.
iv. Fechamento do arquivo.

O comando ASSIGN (em português, associar) permite que associemos o nome externo de um arquivo a uma variável do tipo arquivo. O nome externo (ou caminho do arquivo) é o identificador utilizado pelo sistema operacional para localizar um arquivo no disco, portanto esse identificador deve ser válido para ele. Após o uso do comando ASSIGN, o arquivo ficará associado à variável até que ela seja associada a outro arquivo ou até que a variável seja destruída. O tamanho máximo do nome de um arquivo é 79 bytes. O comando ASSIGN nunca deve ser usado em um arquivo já aberto, pois, caso o nome do arquivo tenha tamanho zero ou ocorra algum erro na

sua associação, o arquivo será associado ao dispositivo padrão de saída de dados (monitor de vídeo). Sua sintaxe geral é:

```
Assign (Variável_do_tipo_FILE, Nome_do_arquivo);
```

Abaixo, segue o trecho de um programa em que um arquivo de números inteiros é criado e associado ao arquivo EXEMPLO.DAT localizado na raiz do disco rígido (diretório C:).

```
Program Exemplo;
Uses
  CRT;
Type
  Arquivo = File Of Integer;
Var
  Arq: Arquivo;
Begin
  Assign(Arq,'C:\EXEMPLO.DAT');
(* a partir desse instante, todas as operações de escrita e/ou leitura que forem realizadas
com a variável Arq serão automaticamente feitas no arquivo EXEMPLO.DAT gravado no
drive C *)
  ...
  ...
End.
```

Para abrir um arquivo, existem dois procedimentos distintos: RESET e REWRITE. O procedimento RESET permite abrir um arquivo em disco já existente. No caso do uso desse comando para a abertura de um arquivo inexistente, ocorrerá um erro de execução do programa. Para que o procedimento seja executado com sucesso, é necessário que, antes de executá-lo, tenha ocorrido a associação do arquivo a uma variável do tipo arquivo com o procedimento ASSIGN. Sua sintaxe geral é:

```
Reset(Arq);
```

Abaixo, segue o trecho de um programa em que um arquivo de números inteiros é criado, associado ao arquivo EXEMPLO.DAT localizado na raiz do disco rígido (diretório C:) e aberto com o procedimento RESET.

```
Program Exemplo;
Uses
  CRT;
Type
  Arquivo = File Of Integer;
Var
  Arq: Arquivo;
Begin
  Assign(Arq,'C:\EXEMPLO.DAT');
  Reset(Arq);
(* Após estas declarações, o arquivo no drive C com o nome 'EXEMPLO.DAT' está aberto
e pronto para as operações de entrada e saída de dados. *)
  . . .
  . . .
End.
```

O procedimento REWRITE permite criar um novo arquivo em disco. Caso o arquivo já exista, ele terá o seu conteúdo escrito. Antes de abrir um arquivo com o comando REWRITE, é necessário associar o arquivo a uma variável do tipo arquivo com o procedimento ASSIGN. A sintaxe geral do procedimento REWRITE é:

```
Rewrite(Arq);
```

Abaixo, segue o trecho de um programa em que um arquivo de números inteiros é criado, associado ao arquivo EXEMPLO.DAT localizado na raiz do disco rígido (diretório C:) e criado/aberto com o procedimento REWRITE.

```
Program Exemplo;
Uses
  CRT;
Type
  Arquivo = File Of Integer;
Var
  Arq: Arquivo;
Begin
  Assign(Arq,'C:\EXEMPLO.DAT');
  Rewrite(Arq);
```

```
(* Após estas declarações teremos um novo arquivo no drive  C com o nome 'EXEMPLO.
DAT' *)
  ...
  ...
End.
```

Os procedimentos WRITE e WRITELN podem ser utilizados para escrever dados dispositivos, tais como o monitor de vídeo, uma impressora local e arquivos em discos. A sintaxe geral do procedimento WRITE, quando usado para escrever dados em um arquivo, é:

```
Write(Arquivo, variavel);
```

O procedimento WRITE gravará os dados sequencialmente no arquivo, ou seja, um após o outro. Abaixo, segue o trecho de um programa em que um arquivo de números inteiros é criado, associado ao arquivo EXEMPLO.DAT localizado na raiz do disco rígido (diretório C:), criado e/ou reiniciado com o procedimento REWRITE e, usando o comando WRITELN, é escrita uma contagem de 1 a 100.

```
Program Exemplo;
Uses
  CRT;
Type
  Arquivo = File Of Integer;
Var
  Arq: Arquivo;
  i: Integer;
Begin
  Assign(Arq,'C:\EXEMPLO.DAT');
  Rewrite(Arq);
  For i:=1 to 100 do Writeln(Arq,i);
(* com a instrução acima teríamos escrito sequencialmente no arquivo C:\EXEMPLO.DAT
os números inteiros de 1 a 100 *)
  ...
  ...
End.
```

O ambiente Turbo Pascal mantém uma variável de registro (chamada de ponteiro do arquivo) que indica o próximo dado que será lido ou escrito; e toda vez que uma leitura ou escrita é feita em um registro, tal variável é atualizada automaticamente. O procedimento SEEK permite alterar o valor dessa variável e, portanto, permite acessar qualquer registro de um arquivo. A sintaxe geral do procedimento SEEK é:

```
Seek(Arquivo, número_do_registro);
```

O procedimento SEEK pode ser usado em arquivos de qualquer tipo, exceto os arquivos do tipo texto. O número do registro é um número inteiro indexado em zero, ou seja, o número do primeiro registro é zero.

A função FILEPOS retorna a posição atual do ponteiro do arquivo. Quando um arquivo é aberto, seja pelo procedimento RESET ou REWRITE, a posição do ponteiro é zero. Sua sintaxe geral é:

```
FILEPOS(nome_arquivo): longint;
```

Para ler dados de um arquivo, utiliza-se o procedimento READ ou READLN. A sintaxe geral do procedimento READ utilizado para a manipulação de arquivos é:

```
Read(Arquivo, variavel);
```

Se o procedimento READ for executado após o fim do arquivo ter sido alcançado, será gerado um erro de execução. Abaixo, segue o trecho de um programa em que um arquivo de números inteiros é criado, associado ao arquivo EXEMPLO.DAT localizado na raiz do disco rígido (diretório C:), criado e/ou reiniciado com o procedimento REWRITE e usando o comando WRITELN é escrita uma contagem de 1 a 100. Após tal operação, o ponteiro do arquivo é retornado para o início do arquivo e as duas primeiras linhas do arquivo são lidas para a variável i.

```
Program Exemplo;
Uses
  CRT;
```

```
Type
  Arquivo = File Of Integer;
Var
  Arq: Arquivo;
  i: Integer;
Begin
  Assign(Arq,'C:\EXEMPLO.DAT');
  Rewrite(Arq);
  For i:=1 to 100 do Write(Arq,i);
  Seek(Arq,0);
(*Posiciona o ponteiro de registro no registro de número 0 *)
  Read(Arq,i);
(* a variável i fica igual ao conteúdo do registro de número 0, que no presente exemplo
valeria 1 *)
  Read(Arq,i);  (* i agora está valendo 2 *)
  . . .
  . . .
End.
```

Na maioria das vezes em que um arquivo é manipulado, não se sabe ao certo a quantidade de registros contidos nele. Para saber quando o final do arquivo foi atingido, pode-se utilizar a função EOF, que retorna TRUE quando o ponteiro do arquivo estiver no fim do arquivo. A sintaxe geral do comando EOF é:

```
EOF(nome_arquivo): Boolean;
```

A função FILESIZE retorna o tamanho de um arquivo em número de registros, não sendo essa função utilizável em arquivos do tipo texto. Caso o arquivo esteja vazio, a função retornará zero. Sua sintaxe geral é:

```
FILESIZE(nome_arquivo): longint;
```

A função FILESIZE, em conjunto com o procedimento SEEK, permite colocar o ponteiro do arquivo no final dele. Tal operação é útil quando se deseja adicionar mais registros num arquivo já existente. Para tanto, basta utilizar a seguinte instrução:

```
Seek(Arquivo, FileSize(Arquivo));
```

Quando o arquivo em questão tiver mais de 32 KB (quilo bytes), a função LONGFILEPOS deve ser utilizada em lugar da função FILEPOS, pois essa função só pode ser utilizada para arquivos menores que 32 KB. O mesmo ocorre com a função FILESIZE, que deve ser substituída pela função LONGFILESIZE e pela função SEEK, que deve ser substituída pela função LONGSEEK.

Quando um arquivo permanece aberto ao fim do programa ou quando se tenta abrir o mesmo arquivo mais de uma vez, gera-se um erro de execução. Em alguns casos, o conteúdo do arquivo pode vir a ser danificado, e o arquivo, perdido. O uso adequado de um arquivo segue a sequência: abrir, usar (seja para leitura e/ou escrita) e fechar o arquivo. Para fechar um arquivo utiliza-se o procedimento CLOSE, cuja sintaxe é:

```
Close(Arquivo);
```

O código 7.1 apresenta um programa que lê números inteiros do teclado e, em seguida, grava tais números em um arquivo do tipo inteiro chamado 701.DAT na raiz do disco rígido. Quando se deseja que um arquivo seja criado dentro de um subdiretório, este deve existir previamente, caso contrário será gerado um erro de execução do programa.

Código 7.1 - Programa que grava números inteiros lidos do teclado em um arquivo

```
Program grava_inteiros;
Uses
  CRT;
Var
  arquivo: File Of Integer;
  i: Integer;
Begin
  ClrScr;
 Assign(arquivo,'C:\701.dat');
  ReWrite(arquivo);
  Repeat
    Write('Número (0 para sair)--> ');
    Readln(i);
    Write(arquivo,i);
```

```
  Until (i = 0);
  Close(arquivo);
End.
```

O código 7.2 apresenta um programa que lê os números inteiros gravados no arquivo 701.DAT criado pelo programa mostrado no código 7.1 e escreve-os na tela do computador na ordem em que estes foram lidos do arquivo.

Código 7.2 - Programa que lê o arquivo criado pelo Código 7.1

```
Program le_inteiros;
Uses
  CRT;
Var
  arquivo: File Of Integer;
  i: Integer;
Begin
  ClrScr;
  Assign(arquivo,'C:\701.dat');
  Reset(arquivo);
  Repeat
    Read(arquivo,i);
    Writeln(i);
  Until (i = 0);
  Close(arquivo);
  Readkey;
End.
```

O código anterior usa um artifício para encontrar o seu fim do último registro do arquivo 701.DAT, pois se sabe que o seu valor é o número zero. A maneira correta de encontrar o fim de um arquivo é mostrada no código 7.3, que consiste na alteração do código 7.2 para o uso do comando EOF.

Código 7.3 - Uso da função EOF (*End of File*) para encontrar o fim de um arquivo

```
Program uso_de_EOF;
Uses
  CRT;
Var
```

```
  arquivo: File Of Integer;
  i: Integer;
Begin
  ClrScr;
  Assign(arquivo,'C:\701.dat');
  Reset(arquivo);
  Repeat
    Read(arquivo,i);
    Writeln(i);
  Until (EOF(arquivo));
  Close(arquivo);
  Readkey;
End.
```

O código 7.4 apresenta um programa que grava em um arquivo do tipo real chamado 704.DAT, localizado na raiz do disco rígido, o quadrado dos números inteiros de 0 a 100. Após essa etapa, é possível consultar o resultado das contas realizadas por meio do procedimento SEEK.

Código 7.4 - Uso do comando SEEK para percorrer um arquivo em disco

```
Program Exemplo;
Uses
  CRT;
Var
  Arq: File of Real;
  i: Integer;
  s: real;
Begin
  Assign(Arq,'C:\704.dat');
  Rewrite(Arq);
  For i:=0 to 100 do
  Begin
    s := i * i;
    Write(Arq, s);
  End;
  Close(Arq);
  Reset(Arq);
  ClrScr;
  While i>=0 do
```

```
  Begin
    Write('Consulatar o quadrado de qual número (-1 para sair)? ');
    Readln(i);
    if (i >= 0) And (i <= 100) Then
    Begin
      Seek(Arq,i);
      Read(Arq,s);
      Writeln;
      Writeln(s:10:0);
      Writeln;
    End
    Else if (i > 100) Then
      Writeln('O valor a consular deve estar entre 0 e 100!');
    Else
      Writeln('Fim do programa.');
  End;
  Close(Arq);
  Readkey;
End.
```

O código 7.5 apresenta um programa cuja finalidade é gerenciar um arquivo em disco do tipo registro. Tal registro é composto por dois campos: NOME do tipo String[20] e IDADE do tipo inteiro.

Código 7.5 - O programa ARQUIVO.PAS

```
{$I-}
Program ARQUIVO_PAS;
Uses CRT;
Type
  Pessoa = Record
    Nome: String[20];
    Idade: Integer;
  End;
  Frase  = String[80];
Var
  Arquivo: File Of Pessoa;
  P: Pessoa;
  escolha: Integer;
Procedure Linha;
(* traça uma linha na posição atual do cursor *)
```

```
Var
  i: Integer;
Begin
  For i:=1 to 80 Do Write('-');
End;

Procedure Centro(S:Frase);
(* centraliza uma string S na tela *)
Var
  x: integer;
Begin
  x := 40 + (Length(S)) DIV 2;
(* lenght retorna o número de  caracteres do parâmetro *)
  Writeln(S:x);
End;

Procedure InReg;
(* procedimento para incluir novos registros *)
Var
  resposta: char;
Begin
  ClrScr;
  Linha;
  Centro('ROTINA PARA ENTRADA DE REGISTROS');
  Reset(arquivo);
(* Neste trecho do programa iremos utilizar a função FILESIZE(arq), que retorna quantos
registros possui o arquivo"arq" *)
  Seek(arquivo,FileSize(arquivo));
(* posiciona o ponteiro de registros no final do arquivo *)
  resposta:='S';
  Linha;
  While resposta = 'S' Do
  Begin
    gotoxy(1,5);
    clreol; (* limpa até final da linha *)
    gotoxy(1,6);
    clreol;
    gotoxy(1,5);
    Write('Nome da pessoa ---> ');
    Readln(P.Nome);
    clreol;
    Write('Idade da pessoa --> ');
    Readln(P.Idade);
```

```
    Linha;
    Write(arquivo,P);
    Write('Deseja continuar (S/N)? -->':50);
    resposta := UPCASE(Readkey);
  end;
  close(arquivo);
End;

Procedure LiReg;
(* procedimento para listar os registros na tela *)
Begin
  Reset(arquivo);
  Clrscr;
  Linha;
  Writeln('NOME':15,'IDADE':18);
  linha;
  While not EOF(arquivo) do
  Begin
    read(arquivo,P);
    Writeln(P.nome:21,' – - – ',P.idade);
  End;
  Linha;
  Close(arquivo);
  Write('Pressione qualquer tecla para sair...');
  Readkey;
End;

Procedure PeNo;
(* pesquisa por nome *)
Var
  nome: string[20];
Begin
  Reset(arquivo);
  nome := '1';
  While nome <> 'SAIR' Do
  Begin
    Clrscr;
    Linha;
    Centro('PESQUISA POR NOME');
    linha;
    Write('Nome (SAIR = fim) --> ');
    Readln(nome);
    if nome <> 'SAIR' Then
```

```
    Begin
      linha;
      seek(arquivo,0);
      While not EOF(arquivo) do
      Begin
        Read(arquivo,P);
        If Pos(nome,P.nome) <> 0 Then
          Writeln(P.nome:21,' – - – ',P.idade);
      End;
      Linha;
      Write('Pressione qualquer tecla para sair...');
      Readkey;
    End;
  End;
  close(arquivo);
End;
(* Aqui começa o nosso programa. Inicialmente devemos verificar se o arquivo "arquivo.
dat" existe, se não existir, então ele deverá ser criado *)
Begin
  Assign(arquivo, 'C:\teste\arquivo.dat');
  Reset(arquivo);
  If IOresult <> 0 Then ReWrite(arquivo);
  Close(arquivo);
  Repeat
    ClrScr;
    Linha;
    Writeln(' Programa para gerenciar um arquivo contendo nomes e idades de pessoas');
    Linha;
    Gotoxy(24,12);
    Writeln('1 – Sair do programa');
    Gotoxy(24,14);
    Writeln('2 – Entrar com registros');
    Gotoxy(24,16);
    Writeln('3 – Listar todos os registros');
    Gotoxy(24,18);
    Writeln('4 – Pesquisar por nome');
    Gotoxy(33,10);
    LowVideo;
    Writeln('SUA ESCOLHA :');
    NormVideo;
    Repeat
      Gotoxy(47,10);
      read(escolha);
```

```
    Until (escolha > 0 ) and (escolha < 5);
    Case escolha of
    2 : InReg;
    3 : LiReg;
    4 : PeNo;
    End;
  Until escolha = 1;
  ClrScr;
  Gotoxy(13,12);
  Writeln('Fim do programa. Pressione qualquer tecla para sair...');
  Readkey;
End.
```

A diretiva de compilação `{$I-}`[9] tem por finalidade indicar ao compilador que os erros de I/O (entrada/saída de dados) serão verificados pelo programador. Assim, se houver algum erro de I/O durante a execução do programa, este não irá terminar as suas atividades e nem gerar um erro de execução. Para que o programador saiba se uma determinada operação de I/O funcionou corretamente, deve-se verificar o valor da variável IORESULT. Se o valor for diferente de zero, então ocorreu algum erro na execução do programa. Como IORESULT é uma variável predefinida no ambiente Turbo Pascal, ela assume determinados valores inteiros quando algum erro de Entrada/Saída de dados (I/O) ocorre. Os valores que a variável IORESULT pode assumir e seus respectivos significados podem ser vistos na Tabela 7.1.

Tabela 7.1 - Valores assumidos pela variável IORESULT

Valor	Significado do erro de entrada e saída de dados
01	Arquivo não existe.
02	***Arquivo não foi aberto para entrada***. Provavelmente, você está tentando ler de um arquivo que ainda não foi aberto.
03	***Arquivo não foi aberto para saída***. Provavelmente, você está tentando escrever num arquivo que ainda não foi aberto.

(continua)

[9] Para mais informações sobre diretivas de compilação, vide o apêndice A.3.

(fim)

Valor	Significado do erro de entrada e saída de dados
04	***Arquivo não aberto***. Esse tipo de erro costuma ocorrer quando se tenta utilizar os procedimentos BLOCKREAD ou BLOCKWRITE sem antes usar os comandos RESET ou REWRITE.
16	***Erro no formato numérico***. Quando se tenta ler uma string de um arquivo texto para uma variável numérica que não está de acordo com o formato numérico.
32	***Operação não permitida para um dispositivo lógico***. Por exemplo, você tenta ler de um arquivo que foi assinalado para a impressora.
33	Não permitido no modo direto.
34	Não permitida assinalação para arquivos standard.
144	***Erro de comprimento de registros***. Por exemplo, você tenta ler um registro que contém um número inteiro e uma string para uma variável de estrutura diferente.
145	SEEK dado para uma posição posterior ao final do arquivo.
153	Fim de arquivo foi alcançado antes de se encontrar o CTRL-Z.
240	Erro de escrita no disco.
241	Diretório cheio.
242	Overflow do comprimento do arquivo.
243	***Arquivo desapareceu***. Imagine que antes de se executar o comando CLOSE você troque o disco.

O procedimento ERASE permite apagar um arquivo em disco e sua sintaxe geral é:

```
Erase(Arquivo);
```

Um exemplo da utilização do comando ERASE pode ser visto no trecho de programa abaixo:

```
Program Exemplo_ERASE;
Uses
  CRT;
Var
  arq: file;
Begin
  Assign(arq,'c:\teste.001');
  . . .
  Erase(arq);
(* após a execução deste trecho de programa o arquivo 'teste.001' será eliminado do disco
*)
  Close(arq);
End.
```

Quando se deseja renomear um arquivo, pode-se utilizar o procedimento RENAME, cuja sintaxe geral é:

```
Rename(Arquivo, Novo_Nome);
```

Um exemplo da utilização do comando RENAME pode ser visto no trecho de programa abaixo:

```
Program Exemplo_RENAME;
Uses
  CRT;
Var
  Arq: File;
Begin
  Assign(Arq,'teste.001');
  Rename(Arq,'teste.002');
  ...
(* após a execução deste trecho de programa o arquivo 'teste.001' terá o seu nome trocado para 'teste.002' *)
End.
```

O procedimento BLOCKREAD lê um número especificado de blocos de 128 bytes cada de um arquivo sem tipo para uma variável. O número de registros lidos é retornado em uma variável inteira opcional. Já o procedimento BLOCKWRITE faz o inverso do procedimento BLOCKREAD, escrevendo um bloco em vez de lê-lo. Suas sintaxes gerais são:

```
BlockWrite(Arquivo, Variável, No_de_Registros, Resultado);
BlockRead(Arquivo, Variável, No_de_Registros, Resultado);
```

O código 7.6 mostra um programa usado para copiar o conteúdo de um arquivo para outro. Ao final da cópia, é feita uma verificação para ver se ela foi bem-sucedida ou não.

Código 7.6 - Uso dos comandos BLOCKREAD e BLOCKWRITE

```
Program Exemplo_BLOCKREAD_BLOCKWRITE;
Uses
  CRT;
```

```
Const
  Buf_Regs = 100;
(* Número de blocos de 128 bytes  que serão transferidos pelo BlockRead ou pelo
BlockWrite *)
Var
  Fonte, Destino: String[33];
  F, D: File;  (* Arquivos não tipados  *)
  No_Regs_restantes, Regs_para_ler, i, r, r1: Integer;
  Buffer, Buffer1: Array[1..12800] Of Byte;
Procedure Erro(x:integer);
Begin
  Writeln('. . . . . Problemas com a copia');
  If x = 1 Then
    Writeln('. . . . . Arquivos de tamanhos diferentes')
  Else
    Writeln('. . . . . Arquivos diferentes');
  Writeln('Tente novamente. Pressione qualquer tecla para sair.');
  Readkey;
  Exit;
End;
Begin
  ClrScr;
  Lowvideo;
  Writeln('Copiador de arquivos':50);
  NormVideo;
  Write('Fonte ----> ');
  Readln(Fonte);
  Assign(F,Fonte);
{$I-}
  Reset(F);
{$I+}
  If IORESULT = 0 Then
  Begin
    Write('Destino --> ');
    Readln(Destino);
    Assign(D,Destino);
    Rewrite(D);
    No_Regs_Restantes := Filesize(F);
(* FileSize retorna o número de registros que contém o arquivo *)
    While No_Regs_Restantes > 0 do
    Begin
      If Buf_Regs < No_Regs_Restantes Then
        Regs_para_ler := Buf_regs
```

```
      Else
        Regs_para_ler := No_Regs_Restantes;
      BlockRead(F,Buffer,Regs_para_ler);
      BlockWrite(D,Buffer,Regs_para_ler);
    No_Regs_restantes := No_regs_restantes-Regs_para_ler;
    End;
    Close(F);
    Close(D);
    Reset(F);
    Reset(D);
    No_Regs_Restantes := Filesize(F);
    While No_Regs_Restantes > 0 do
    Begin
      If Buf_Regs < No_Regs_Restantes Then
        Regs_para_ler := Buf_regs
      Else
        Regs_para_ler := No_Regs_Restantes;
      BlockRead(F,Buffer,Regs_para_ler,r);
      BlockRead(D,Buffer1,Regs_para_ler,r1);
    No_Regs_restantes := No_regs_restantes-Regs_para_ler;
      If r<>r1 Then Erro(1);
      For i:=1 to 128*r do
        If buffer[i]<>buffer1[i] Then Erro(2);
    End;
  End
  Else
  Begin
    Writeln('..... Este arquivo não existe');
    Writeln('..... Operacao nao realizada');
    Writeln('Pressione qualquer tecla para sair.');
    Readkey;
  End;
  Writeln('..... Operacao realizada com sucesso!!!');
  Writeln('Pressione qualquer tecla para sair.');
  Readkey;
End.
```

Em alguns casos, é necessário descartar parte de um arquivo, e não o arquivo todo. Por exemplo, apagar as últimas 58 páginas de um arquivo de texto que possua 200 páginas, preservando as primeiras 142 páginas. O comando TRUNCATE exclui os registros finais do arquivo a partir do registro corrente, preservando os registros iniciais. Sua sintaxe geral é:

```
Truncate(Arq);
```

O código 7.7 apresenta um programa que escreve uma contagem progressiva de um a cem em um arquivo de números inteiros e usa o procedimento TRUNCATE para truncar o arquivo a partir do décimo registro.

Código 7.7 - Uso do comando TRUNCATE

```
Program Exemplo;
Uses
  CRT;
Var
  a: file of integer;
  i: integer;
Begin
  Assign(a,'c:\707.dat');
  Rewrite(a);
  For i:=1 to 100 do write(a, i);
  Close(a);
(* O arquivo '807.dat' contém 100 números inteiros de 1 até 100 *)
  Reset(a);
  Seek(a,10);
  truncate(a);
(* o arquivo foi truncado a partir do registro 10 *)
  Seek(a,0);
  while not EOF(a) do
  Begin
    Read(a,i);
    Writeln(i); (* ser escrito de 1 até 10 no vídeo *)
  End;
  Readkey;
End.
```

7.1 Manipulação de arquivos de texto

Os arquivos de texto são utilizados por várias aplicações (Wordstar, MS Word, Dbase III, etc) para armazenar texto digitado pelo usuário. Para declarar um arquivo do tipo texto procede-se da seguinte forma:

```
Var
  Arquivo: Text;
```

Após essa declaração, o Turbo Pascal aloca um *buffer* de 128 bytes para as operações de entrada e saída de dados do arquivo de texto. É possível modificar o tamanho desse *buffer* da seguinte forma:

```
Var
  Arquivo: Text[tamanho_desejado_para_o_buffer];
```

Da mesma forma que nos arquivos binários, para manipularmos arquivos texto devemos assinalar o nome lógico, abrir e só então ler ou escrever dados no arquivo. Quando se deseja abrir um arquivo de texto para a inclusão de texto no final dele, deve-se usar o comando APPEND. A sintaxe geral do comando APPEND é:

```
Append(Arquivo);
```

Esse procedimento não permite que se faça a abertura de um arquivo que ainda inexiste. O seu uso correto se faz apenas quando o arquivo já existir. Se em um computador existir um arquivo chamado AUTOEXEC.BAT (localizado normalmente em C:\AUTOEXEC.BAT), pode-se fazer um teste executando o programa apresentado no código 7.8. Após a execução do programa, pode-se visualizar o arquivo AUTOEXEC.BAT e verificar a linha extra inclusa pelo programa no final dele. Ao fim do teste, a linha inserida pode ser removida sem nenhum problema.

Código 7.8 - Uso do comando APPEND

```
Program teste_append;
Var
  Arq: Text;

Begin
  Assign(Arq,'C:\autoexec.bat');
  Append(arq);
  Writeln(arq,'Echo INCLUíDO PELO TURBO PASCAL');
  Close(arq);
End.
```

Para ler uma linha de dados de um arquivo de texto, isto é, ler dados do arquivo até que se encontre um caractere indicativo de fim de linha, utiliza-se o comando READLN de modo análogo ao visto para entrada de dados via teclado. Sua sintaxe geral para esse caso é:

```
Readln(Arquivo, Variavel);
```

O arquivo a ser lido deverá ser do tipo texto e variável do tipo String. De modo também análogo à saída de dados padrão no monitor de vídeo, usa-se o comando WRITELN para escrever uma linha de dados em um arquivo de texto, isto é, escrever uma linha de dados e inserir o caractere fim de linha. Sua sintaxe geral para esse caso é:

```
Writeln(Arquivo, Variavel);
```

O código 7.9 apresenta um exemplo do uso dos comandos WRITELN e READLN para escrever um texto digitado pelo usuário em um arquivo de texto. A entrada de dados termina quando o usuário digitar a palavra Fim.

Código 7.9 - Uso dos comandos WRITELN e READLN em arquivos de texto

```
Program Exemplo;
Uses
  CRT;
Var
  Frase: String[200];
  a: Text;

Begin
  ClrScr;
  Assign(a,'709.txt');
  ReWrite(a);
  Frase := ' ';
  While Frase <> 'Fim' do
  Begin
    Writeln('Digite uma frase (Fim para sair):');
    Readln(Frase);
    Writeln(a, Frase);
  End;
  Close(a);
```

```
  Reset(a);
  ClrScr;
  While not EOF(a) do
  Begin
    Readln(a,Frase);
    Writeln(Frase);
  End;
  Close(a);
  Readkey;
End.
```

Um comando importante quando se lida com arquivos de texto é a função EOLN (do inglês *End of Line*), que retorna TRUE, quando for alcançado um fim de linha no arquivo texto, e falso, caso contrário. Sua sintaxe geral é:

```
Eoln(arquivo);
```

Quando se deseja saber se foi atingido o final do arquivo, e não de uma dada linha, usa-se a função EOF (do inglês *End Of File*). Essa função retorna TRUE quando for alcançado o fim de um arquivo do tipo texto e falso em caso contrário. A diferença entre o comando EOF para arquivos de texto e comando EOF para arquivos tipados é que, para arquivos de texto, a referência ao arquivo é opcional. Sua sintaxe geral para arquivos de texto é:

```
EOF[(arq: Text)]: Boolean;
```

Quando se deseja percorrer um arquivo de texto de maneira rápida, por exemplo buscando o final de uma linha ou mesmo o final do arquivo, pode-se usar as funções SEEKEOF (do inglês *SEEK End Of File*), que retorna TRUE se encontrado o *status* de final de arquivo, ou a função SEEKEOLN (do inglês *SEEK End Of Line*), que retorna TRUE se encontrada a marca de fim de linha. As funções SEEKEOF e SEEKEOLN são similares às funções EOF e EOL, exceto que a função SEEKEOF salta todos os brancos e tabulações quando da leitura do arquivo e a função SEEKEOLN salta todos os caracteres em branco e tabulações encontradas durante a leitura da linha. A sintaxe geral das duas funções é:

```
SEEKEOF(arquivo): Boolean;
SEEKEOLN(arquivo): Boolean;
```

Quando o comando WRITELN é aplicado a arquivos de texto, ele não é executado no mesmo instante de sua chamada no código-fonte. Isso quer dizer que o sistema operacional não gravará as novas informações no arquivo de pronto, mas as armazenará em um espaço de memória denominado *buffer*. Dessa forma, é importante esvaziar periodicamente o *buffer* quando o volume de informações a serem gravadas é alto.

O procedimento FLUSH faz com que a área de *buffer* de um arquivo do tipo texto seja descarregada, e as informações, salvas de fato. Em sua utilização, pode ser usada a função IORESULT, que retornará zero, caso a operação tenha sido bem-sucedida. Sua sintaxe geral é:

```
FLUSH(arq: Text);
```

Quando as informações a serem gravadas não descarregadas pelo comando FLUSH, elas serão descarregadas quando o arquivo de texto for fechado pelo comando CLOSE. Desse modo, o uso do comando FLUSH torna-se opcional quando o tamanho do texto é menor que 128 bytes. Contudo, se desejarmos armazenar mais texto no *buffer* de memória, sem ter que descarregá-lo, pode-se configurar o tamanho do *buffer* de memória associando um arquivo do tipo texto a uma área de *buffer* na área de dados do programa usando o comando SETTEXTBUF. Sugere-se, sempre que possível, utilizar múltiplos de 128 bytes para o novo tamanho do *buffer*, uma vez que essa área já é alocada automaticamente pelo sistema operacional com tamanho de 128 bytes. O comando SETTEXTBUF deve ser usado antes de o arquivo ser aberto, podendo ser usado tanto para a leitura quanto para a escrita. Esse comando acelera os procedimentos de leitura e gravação de blocos de texto maiores que 128 bytes. Sua sintaxe geral é:

```
SETTEXTBUF(arq: Text, buffer, tamanho: Word);
```

O código 7.10 apresenta um exemplo do uso do comando SETTEXTBUF para leitura e escrita de arquivos do tipo texto.

Código 7.10 - Uso do comando SETTEXTBUF

```
Program Exemplo_SETTEXTBUF;
Uses
  CRT;
Var
  nomearq: string[79];
  arq: text;
  ch: char;
  buf: array[1..10240] of char; {10 kbytes de buffer}
Begin
  Clrscr;
  Write('Entre com o nome do arquivo: ');
  Readln(nomearq);
  Assign(arq, nomearq);
  SetTextBuf(arq, buf);
{$I-}
  Reset(arq);
{$I+}
  If IOResult <> 0 then
  Begin
    Writeln('Falha na abertura do arquivo! ');
    Readkey;
  End
  Else
    While not EOF(arq) do
    Begin
      Read(arq, ch);
      Write(ch);
    End;
  Readkey;
End.
```

7.2 Manipulação de diretórios

Inicialmente, é importante saber que todo programa, ao ser executado, mantém uma variável interna que indica qual o diretório atual do programa. É possível alterar o valor dessa variável navegando-se entre as unidades e os diretórios existentes no computador. Contudo, a maioria dos comandos a serem vistos neste capítulo aplicam-se ao chamado diretório corrente, que indica o caminho armazenado na variável de caminho (ou *path*) do Turbo

Pascal. Para conhecer o nome do diretório corrente, pode-se utilizar o comando GETDIR, devendo ser passada como parâmetro um número inteiro representando a letra da unidade. Caso a unidade seja igual a 0, será exibido o diretório da unidade corrente, 1 para a unidade A, 2 para a unidade B, 3 para a unidade C e assim sucessivamente. A sintaxe geral do comando GETDIR é:

```
GETDIR(drive: Byte; VAR s: String);
```

O comando MKDIR (do inglês *MaKe DIRectory*) cria um subdiretório de acordo com o parâmetro informado ao comando. Esse parâmetro deve ser compatível com o padrão de nomenclatura de diretórios do DOS[10] e não é possível a criação de um subdiretório já existente. A sintaxe geral do comando MKDIR é:

```
MKDIR(s: String);
```

A string s passada ao comando MKDIR é o nome do novo diretório a ser criado.

O comando CHDIR (do inglês *CHange DIREctory*) muda o diretório corrente do programa para o diretório especificado no parâmetro informado ao procedimento. Sua sintaxe geral é:

```
CHDIR(s: String);
```

A string s passada ao comando CHDIR é o nome do diretório para onde se deseja mover.

O procedimento RMDIR (do inglês *ReMove DIRectory*) é usado para remover um subdiretório existente passado como parâmetro para a função. O subdiretório a ser removido deve estar necessariamente vazio e não pode ser o subdiretório corrente. A sintaxe geral do procedimento RMDIR é:

---

[10] Os caracteres proibidos em nomes de diretórios são: asterisco (*), barra vertical (|), barra (\), contrabarra (/), aspas duplas (“), menor que (<), maior que (>), dois-pontos (:) e o sinal de interrogação (?).

```
RMDIR(s: String);
```

O código 7.11 mostra um exemplo que engloba todos os procedimentos utilizados para a manipulação de subdiretórios.

Código 7.11 - Exemplo de manipulação de diretórios em Pascal

```
Program Exemplo_diretorios;
Uses
  CRT;
Var
  dir, dir_atual: String[67];

Begin
  Clrscr;
  Gotoxy(15, 1);
  Write('Manutenção de diretórios');
  Gotoxy(10, 10);
  Write('Entre com o nome do diretório a ser criado: ');
  Readln(dir);
{$I-}
  MKDIR(dir);
{$I+}
  If IOResult <> 0 then
  Begin
    Gotoxy(15, 15);
    Write('O diretório não pode ser criado!!!');
    Readkey;
  End
  Else
  Begin
    Gotoxy(10, 12);
    GETDIR(0, dir_atual);
    Write('O diretório atual é: ', dir_atual);
    Gotoxy(10, 14);
    CHDIR(dir);
    GETDIR(0, dir);
    Write('Diretório trocado para: ', dir);
    CHDIR(dir_atual);
    GETDIR(0, dir_atual);
    RMDIR(dir);
```

```
    Gotoxy(10, 16);
    Write('O diretório ', dir,' foi removido com sucesso! ');
    Readkey;
  End;
End.
```

7.3 Exercícios

1. Faça um programa para ler registros com os campos nome, endereço, telefone, *e-mail*, salário e armazená-los em um arquivo. ***Observação***: o último registro possui o campo nome igual a "fim".

2. Faça um programa para ler os dados do arquivo criado no exercício anterior e para guardar em outro arquivo (do tipo texto) o nome e o salário de cada um e o valor total dos salários.

3. Crie um programa que leia um vetor de nomes de pessoas e grave-o em um arquivo de texto. Depois, ordene os nomes lidos em ordem alfabética e grave o arquivo novamente com um novo nome.

4. Escreva um programa para ler o nome completo de várias pessoas (até encontrar o final do arquivo) e determinar as pessoas pertencentes às várias famílias do grupo. Lembre-se de que uma mesma pessoa geralmente pertence a mais de uma família.

5. Escreva um programa para ler as relações de até 100 nomes de pessoas que compareceram a duas festas diferentes. A segunda relação começa com "FESTA 2" e termina após a leitura dos 100 nomes ou com o final do arquivo de entrada. O programa deve imprimir:
   a) Uma listagem das pessoas festeiras (compareceram a uma das festas).
   b) Uma listagem das pessoas muito festeiras (compareceram a ambas as festas).

6. Uma empresa dispõe de uma listagem com os dados de seus empregados contendo as seguintes informações: Matrícula, Nome, Categoria ("S" para nível superior ou "M" para nível médio), Departamento e Salário. Faça um programa para ler de teclado os dados referentes aos funcionários da empresa e gravar 2 arquivos, "CADASTRO.SUP" e "CADASTRO.MED", com os dados dos funcionários de nível superior (categoria = "S") e de nível médio (categoria = "M"), respectivamente, conforme a estrutura de registros descrita abaixo. O número total de funcionários deve ser informado inicialmente. Estrutura dos registros para os dois arquivos:

```
Matricula: Integer;
Nome: String[40];
Departamento: String[15];
Salario: Real;
```

7. Faça um programa para criar dois arquivos no disco rígido com os dados dos produtos vendidos na empresa "Compre Mais" com as seguintes informações:
   a) O código do produto é um número inteiro.
   b) A descrição do produto é um texto de 25 caracteres.
   c) O código do setor de vendas é um número inteiro (1 para gênero alimentício e 2 para eletrodoméstico).
   d) A unidade do produto é um texto com três caracteres.
   e) O valor unitário do produto é um número real.

   O programa deve executar, em processo repetitivo, os seguintes passos:
   a) Ler o código do produto.
   b) Ler os demais dados referentes ao produto.
   c) Armazenar os dados no arquivo "CM_ALIM.CAD" ou "CM_ELD.CAD", se o código do setor de vendas for, respectivamente, 1 ou 2 (permitir somente um desses dois valores).

   O processo repetitivo de leitura dos dados e gravação dos arquivos termina quando for digitado o valor 9999 para o código do produto (esse código 9999, por não corresponder a nenhum produto, não deve ser gravado em nenhum dos dois arquivos).

8. Faça um programa para:
   a) Ler, do teclado, o número de alunos de uma turma.
   b) Ler, de teclado e em processo repetitivo, o nome, a altura, o peso e o sexo dos alunos da turma, gravando os valores em dois arquivos denominados ALUNOS.MC, com os dados dos alunos do sexo masculino, e ALUNOS.FM, com os dados dos alunos do sexo feminino.

   Os dois arquivos têm a mesma estrutura de dados, descrita abaixo:

Nome do aluno	Altura	Peso
String [35]	Real	Real

9. O arquivo "CARROS.AUX" contém os dados referentes aos carros de uma empresa que comercializa carros usados cujos registros têm a seguinte estrutura:

Modelo	Preço Venda	Ano de Fabricação
String[45]	Real	Integer

A empresa resolveu atualizar os preços de venda para todos os automóveis que tenham seu ano de fabricação a partir de 1999, independentemente do modelo, conforme regra abaixo:

a) Aumento de 2,75% para os veículos com ano de fabricação entre 1999 e 2000, inclusive.
b) Aumento de 3,33% para os veículos com ano de fabricação entre 2001 e 2002, inclusive.
c) Aumento de 6,47% para os veículos com ano de fabricação maior que 2002.

Faça um programa que leia todos os registros do arquivo e imprima uma listagem com o modelo e o novo preço de venda para os veículos que tiveram seus preços alterados.

Observações:

- O aumento do preço deve ser calculado utilizando uma função que receba, por meio de parâmetros, o ano de fabricação e o preço de venda atual; e retorne com o novo valor, conforme regra descrita acima.
- Carros com ano de fabricação até 1998 não tiveram aumento algum, portanto não devem aparecer na listagem.
- Não é necessário atualizar o valor alterado no arquivo.

8. Alocação dinâmica de memória

Até o momento, aprendemos a trabalhar com variáveis que foram declaradas antes da execução do programa, são variáveis denominadas *variáveis estáticas*, e a alocação de memória para esse tipo de variável é feita antes da execução do programa. A desvantagem desse tipo de variável é o fato de que, uma vez criado o espaço de memória que a variável ocupa, ele não pode mais ser alterado. As *variáveis dinâmicas,* por sua vez, podem ser criadas e/ou destruídas durante a execução de um programa, sendo essa a grande vantagem das variáveis dinâmicas sobre as estáticas. As variáveis dinâmicas, também chamadas de ponteiros, podem ser declaradas por meio de um tipo predefinido de dados na linguagem Pascal, chamado *Pointer* (ou ponteiro, em português).

O tipo ponteiro, como o próprio nome já diz, aponta para o local da memória do computador onde está armazenada uma variável e não para o conteúdo da variável, como ocorre com as variáveis estáticas. O procedimento para se declarar uma variável do tipo pointer é simples, sendo sua sintaxe dada por:

```
Var
  p: ^Integer;
```

Após essa declaração, é criada uma variável do tipo ponteiro que ocupa quatro bytes (lembre-se de que o ponteiro aponta para um endereço físico de memória, e no IBM/PC um endereço de memória é formado por dois números: segmento e *offset*, cada um com tamanho de dois bytes) e que irá apontar para uma variável do tipo inteiro. O tipo inteiro foi usado apenas como exemplo, pois é possível a criação de ponteiros de quaisquer outros tipos de dados, até mesmo de vetores, matrizes e registros.

Até agora não foi criada nenhuma variável dinâmica, mas, sim, uma variável do tipo ponteiro, que irá apontar para o endereço físico de memória de uma variável dinâmica do tipo inteiro. A princípio, isso pode parecer complicado, mas aos poucos o funcionamento desse novo tipo de variável será melhor explicado. Uma boa pergunta a ser feita é: Para onde está apontando a variável recém-criada chamada p? Simplesmente para nenhum lugar, pois, como nenhum valor foi atribuído ao ponteiro, ele é nulo, ou, na linguagem Pascal, *NIL*. Quando a atribuição abaixo é realizada no decorrer de um programa:

```
p := NIL;
```

A variável p do tipo ponteiro deixa de apontar para qualquer variável outra variável (ou endereço físico de memória). Sempre que uma variável do tipo ponteiro é criada, esta é inicializada como o valor inicial NIL.

O procedimento NEW permite a alocação de uma área da memória do computador de tamanho correspondente ao tamanho da variável associada ao ponteiro passado como argumento ao procedimento. Essa área da memória fica num local chamado HEAP. No caso do IBM/PC, o HEAP é toda a memória não utilizada pelo sistema, ou seja, a memória RAM livre do computador. Caso não exista espaço suficiente no HEAP de memória, ocorrerá um erro de execução do programa. A sintaxe geral do procedimento NEW é:

```
New(ponteiro);
```

A declaração NEW (p) aloca um espaço de memória no HEAP suficiente para armazenar uma variável do tipo inteiro e retorna o endereço físico dessa região de memória para a variável p do tipo ponteiro. A variável p assim definida é conhecida como variável dinâmica. Para acessar o conteúdo de uma variável dinâmica, utiliza-se a seguinte simbologia:

```
p^
```

O código 8.1 ilustra como utilizar um ponteiro, bem como o uso do comando NEW.

Código 8.1 - Uso de comando NEW

```
Program Exemplo;
Uses
  CRT;
Type
  Ponteiro = ^Integer;
Var
  p: Ponteiro;
  i: Integer;
(* p é uma variável do tipo pointer que aponta para variáveis dinâmicas do tipo integer *)

Begin
  ClrScr;
  If p = NIL Then Writeln('Sim');
(* como p acabou de ser criada ela não deve estar apontando para nenhum endereço, ou
seja, seu valor inicial deve ser NIL. Para descobrirmos se isso é verdade basta compará-la
com NIL *)
  New(p);
(* acabamos de criar uma variável dinâmica do tipo Integer, e seu endereço foi colocado
no pointer p *)
  p^ := 100;
(* estamos atribuindo o valor 100 à variável dinâmica recém criada *)
  Writeln(p^);
  i := 200;
  p^ := i;
```

```
  Writeln(p^);                    (* será escrito 200 *)
(* A função addr(var) retorna o endereço da variável var *)
  p := addr(i);
(* o pointer contém agora o endereço da variável i *)
  p^:=1000;
(* indiretamente estou atribuindo o valor 1000 à variável i *)
  Writeln(i);                              (* será escrito 1000 *)
  Readkey;
End.
```

O procedimento DISPOSE permite que se libere a memória do computador alocada previamente pelo procedimento NEW. Após a execução desse procedimento, o conteúdo do ponteiro é destruído e qualquer referência a ele causará um erro de execução no programa, assim também ocorrerá se for solicitada a liberação de uma área da memória que estiver fora da região do HEAP. O procedimento DISPOSE nunca poderá ser utilizado em conjunto com os procedimentos MARK e RELEASE, descritos a seguir. Sua sintaxe geral é:

```
DISPOSE(ponteiro);
```

Quando se deseja que uma variável fique com todo o valor do HEAP, o qual, por sua vez, é todo o espaço da memória RAM do computador disponível, usa-se o comando MARK. O espaço da memória ficará alocado até que o comando RELEASE seja usado, o qual permite liberar toda a memória alocada pelo comando MARK. Tanto o comando MARK quanto o comando RELEASE não podem ser utilizados em conjunto com os comandos DISPOSE e FREEMEM. Quando o comando RELEASE é utilizado, todos os processos de alocação de memória dinâmica, tais como NEW ou GETMEM, utilizados após o comando MARK, serão liberados. A sintaxe geral dos comandos MARK e RELEASE é:

```
MARK(ponteiro);
RELEASE(ponteiro);
```

O comando GETMEM cria uma nova variável dinâmica com o tamanho especificado no parâmetro e coloca o endereço físico da memória em

uma variável do tipo ponteiro. Caso não haja espaço suficiente para a alocação da variável na memória, ocorrerá um erro de execução do programa. O tamanho da variável não pode ultrapassar o tamanho do HEAP. A sintaxe geral do comando GETMEM é:

```
GETMEM(ponteiro, tamanho);
```

Para liberar uma área de memória, usa-se a função FREEMEM, sendo a quantidade de memória a ser liberada passada à função com parâmetro. O sistema operacional irá, então, destruir toda referência associada à memória liberada. A função FREEMEM não pode ser utilizada com os comandos MARK ou RELEASE. A sintaxe geral da função FREEMEM é:

```
FreeMem(ponteiro, tamanho);
```

Duas outras funções úteis para manipulação de ponteiros são MAXAVAIL, que retorna o tamanho do maior bloco contínuo de memória do HEAP (correspondente à maior variável dinâmica que pode ser utilizada pelo programa) e MEMAVAIL, que retorna o valor de toda a memória disponível, ou seja, o somatório de todos os blocos de memória livre existentes. Como as duas funções não possuem argumentos, basta chamá-las diretamente no código para que sejam executadas.

8.1 Estruturas de dados dinâmicas

Suponha que fosse necessário desenvolver um programa para ler uma quantidade indeterminada de registros do teclado. Não se sabe de antemão se serão dez, cem ou mil registros a serem lidos. A princípio, poder-se-ia superdimensionar um vetor. Se o computador que fosse executar o programa tivesse memória suficiente, não haveria nenhum problema nessa estratégia. Contudo, corre-se o risco de, no futuro, ser necessário redimensionar o vetor para armazenar mais elementos.

Em um caso como esse, deve-se utilizar o conceito das variáveis dinâmicas aplicadas às estruturas de dado. Para tanto, deve-se declarar um ponteiro para uma variável cuja estrutura seja constituída de dois campos:

um contendo o valor que se quer armazenar e o outro apontando para a próxima variável dinâmica. Esse tipo de estrutura é conhecido como estrutura *encadeada de dados*, pois forma uma cadeia de dados.

O código 8.2 apresenta um programa que lê registros informados pelo usuário até que esse digite a palavra fim como nome da pessoa. O programa tem a capacidade de ler um número ilimitado de registros, sem a preocupação de se definir um vetor e sua respectiva dimensão. Os registros lidos pelo programa têm a seguinte estrutura: nome da pessoa do tipo string de tamanho trinta caracteres, sexo do tipo caractere, idade do tipo inteiro e altura do tipo real.

Código 8.2 - Uso de estruturas de dados com ponteiro

```
Program Exemplo;
Uses
  CRT;
Type
  Pessoa   = Record
    Nome: String[30];
    Sexo: Char;
    Idade: Integer;
    Altura: Real;
  End;

  ponteiro = ^Pessoas;
  Pessoas  = Record
    Valor: Pessoa;
    Prox: Ponteiro;
  End;
Var
  p, prim : Ponteiro;

Procedure Linha;
Var
  i: integer;
Begin
  For i:=1 to 80 do write('-')
End;

Begin
  Prim := nil;
```

```
  ClrScr;
  Repeat
    Linha;
    New(p);
    Write('Nome da pessoa -----> ');
    Readln(p^.valor.Nome);
    If (p^.valor.Nome<>'fim') Then
    Begin
      Write('Sexo ----------> ');
      Readln(p^.valor.Sexo);
      Write('Idade ----------> ');
      Readln(p^.valor.Idade);
      Write('Altura ----------> ');
      Readln(p^.valor.altura);
      p^.Prox:=Prim;
      Prim:=p;
    End;
  Until p^.valor.nome='fim';
  ClrScr;
  Linha;
  p := prim;
  Writeln('Nome:':30,'Sexo:':10,'Idade:':10,'Altura:':10);
  While p<>nil do
  Begin
    With p^.valor do
      Writeln(nome:30,sexo:10,idade:10,altura:8:2);
      p := p^.prox;
  End;
End.
```

8.2 Exercícios

1. Crie um programa que, utilizando uma lista encadeada de dados, implemente três listas denominadas L1, L2 e L3 e:
   a) Verifique se L1 é uma lista ordenada.
   b) Faça uma cópia da lista L1 na lista L2.
   c) Faça uma cópia da lista L1 na lista L2 eliminando os elementos repetidos de L1.
   d) Intercale a lista L1 com a lista L2, gerando a lista L3, eliminando os elementos repetidos de L3.

2. Supondo a seguinte declaração:

```
Type
  pont_no = ^no;
  no = record
    Info:tipo_base;
    Lig:pont_no
  end;
Var
  p,q:pont_no;
```

Tendo sido utilizadas duas chamadas new(p) e new(q) com preenchimento adequado dos campos info e lig de cada ponteiro, explique o que acontece nas atribuições abaixo. ***Dica***: utilize desenhos na resposta.

a) p^.lig := q
b) p^.lig:=q^.lig
c) p^.info:=q^.info
d) p:=q
e) p^.lig:=q^.lig
f) p^=q^
g) p:=p^.lig
h) p:=p^.lig^.lig

3 Crie um programa que implemente uma lista encadeada de clientes de uma loja, com as seguintes operações:

a) Adicionar um novo cliente no fim da lista.
b) Buscar os dados de um cliente, dado seu nome.
c) Contar o número de clientes cadastrados.
d) Mostrar os dados do primeiro cliente da lista.

9. Outros ambientes de desenvolvimento Pascal

Existem outros ambientes integrados de desenvolvimento (IDE) que utilizam a linguagem Pascal para codificar programas. Citam-se: o Dev-Pascal, Lazarus e Irie Pascal. Tais IDEs apresentam uma alternativa ao uso da IDE Turbo Pascal 7.0, que foi lançada em 1992, e é executada em ambiente DOS. Segue-se uma pequena introdução sobre tais IDEs.

9.1 O ambiente integrado de desenvolvimento Bloodshed Dev-Pascal

O Bloodshed Dev-Pascal é um ambiente integrado de desenvolvimento para aplicações em Pascal gratuito e de código-fonte aberto, desenvolvido em Delphi. Com ele, é possível a criação de programas na linguagem Pascal baseados no Windows ou console (DOS) e também de DLLs (do inglês *Dynamic-Link Library*) por meio do compilador Free Pascal, que é distribuído junto com o Bloodshed Dev-Pascal. O Free Pascal é um compilador de Object Pascal que roda em vários sistemas operacionais (citam-se: Linux, Windows, OS/2, Mac OS tradicional, Mac OS X, ARM, BSD, BeOS e DOS). Ele foi desenhado para compilar código com a sintaxe do Delphi ou

dos dialetos Pascal do Macintosh e gerar executáveis para diferentes plataformas a partir de um mesmo código-fonte. Algumas funcionalidades do Bloodshed Dev-Pascal são:

- Suporte a compiladores baseados em GCC (GNU *Compiler Collection*).
- Depuração integrada utilizando GDB (GNU *Project Debugger*).
- Gerenciador de projetos.
- Editor com coloração de sintaxe personalizável.
- Função de autocompletar o código-fonte que está sendo digitado.
- Edição e compilação dos arquivos contidos nos projetos criados.
- Gerenciador de ferramentas.
- Suporte a CVS (do inglês *Concurrent Version System* ou Sistema de Versões Concorrentes). CVS é um sistema de controle de versão que permite que se trabalhe com diversas versões de arquivos organizados em um diretório e localizados local ou remotamente, mantendo-se suas versões antigas e os registros de quem e quando manipulou tais arquivos.

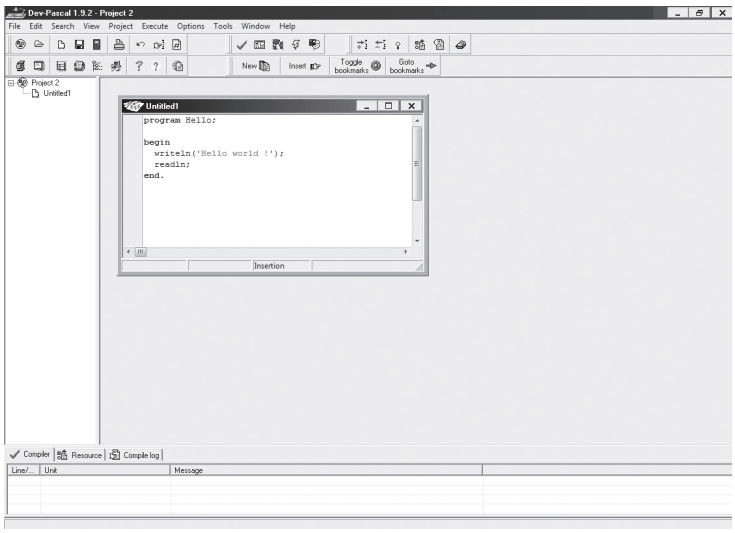


Figura 9.1 - Ambiente Bloodshed Dev-Pascal em execução

A figura 9.1 apresenta o ambiente Bloodshed Dev-Pascal sendo utilizado para a geração de um programa do tipo *Hello World*. Esse tipo de programa não passa de um teste muito simples, que consiste apenas em escrever a mensagem *Hello World* na tela do computador.

O Bloodshed Dev-Pascal pode ser baixado diretamente do *site* da Bloodshed no seguinte endereço: http://www.bloodshed.net/devpascal.html. Ao final da instalação do Bloodshed Dev-Pascal, é necessário alterar o nome da pasta "Icons" criada durante a instalação para "Icon" (por padrão localizada em C:\Dev-Pas\Icons). Se essa operação não for realizada, o Dev-Pascal não conseguirá associar um ícone padrão para os projetos criados.

9.2 O ambiente integrado de desenvolvimento Lazarus

O Lazarus é um ambiente de desenvolvimento gratuito e de código livre para o compilador Free Pascal. Esse ambiente permite o desenvolvimento de aplicações gráficas, console e para a internet, podendo ser executado em sistemas operacionais Linux, MS Windows (95 ou superior), Mac OS X, BSD e Solaris. A figura 9.2 apresenta o ambiente Lazarus sendo utilizado para a geração de um programa do tipo Hello World.

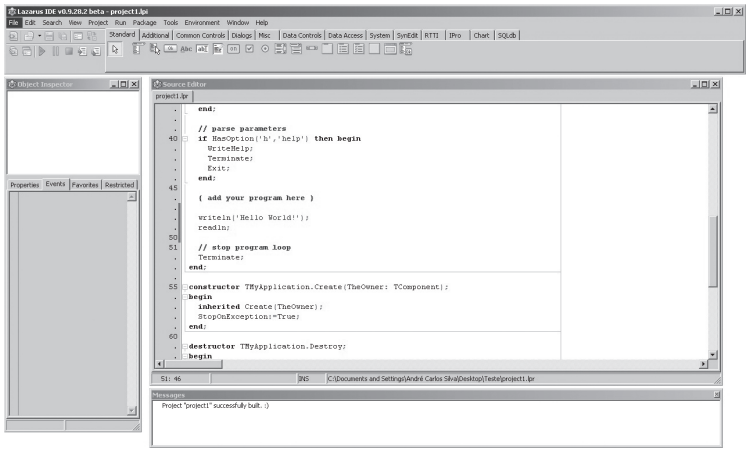


Figura 9.2 - Ambiente Lazarus em execução

A maior vantagem em se utilizar o ambiente Lazarus reside no fato de que ele pode ser usado para a geração de programas com interface gráfica usando o conceito RAD (do inglês *Rapid Application Development*, ou Desenvolvimento Rápido de Aplicações). Segundo tal conceito, pode-se desenvolver aplicações gráficas em um tempo de desenvolvimento consideravelmente menor. O Lazarus pode ser baixado diretamente no seguinte endereço: <http://www.lazarus.freepascal.org>.

9.3 O ambiente integrado de desenvolvimento Irie Pascal

O Irie Pascal é uma IDE proprietária desenvolvida e comercializada pela Irie Tools, que executa em sistema operacional Windows (95 ou superior), Linux, Solaris/Sparc, Solaris/x86 e FreeBSD. Dentre as suas principais vantagens destaca-se a interface simples e funcional. É uma boa opção para quem não quer usar a IDE Turbo Pascal por ser executada em DOS. A figura 9.3 apresenta o ambiente Irie Pascal e o programa *Hello World*.

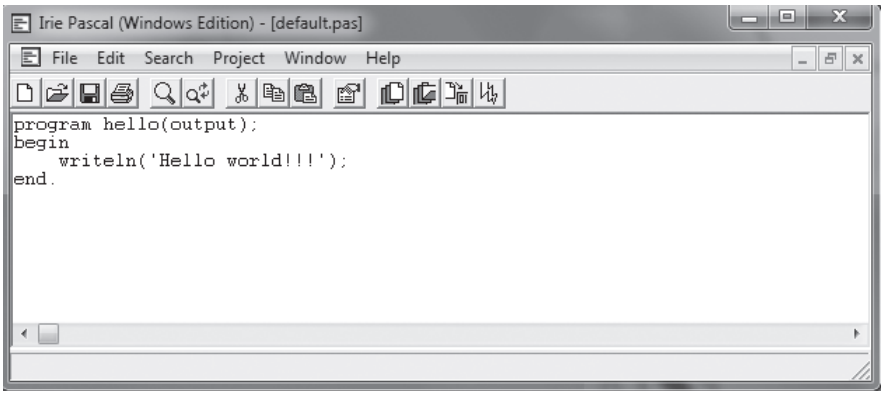


Figura 9.3 - Ambiente Irie Pascal em execução

A versão de demonstração do Irie Pascal pode ser obtida diretamente do *site* da Irie Tools no seguinte endereço: <http://www.irietools.com>; ela expira em trinta dias após a sua instalação.

Anexos

A.1 - Algoritmo estruturado

O que caracteriza a programação estruturada é a possibilidade da divisão das tarefas a serem cumpridas em sub-rotinas ou em blocos independentes entre si. Antes de começar o desenvolvimento de um programa estruturado, deve-se seguir determinadas regras, que são:

A decomposição lógica do problema em partes independentes.

i. Fazer um esboço do fluxograma do programa também chamado de algoritmo do programa.
ii. Desenvolver o programa, rotina a rotina, sempre colocando linhas de comentário em cada uma delas, para que sejam facilitadas manutenções futuras.
iii. Testar cada uma das rotinas logo após a sua conclusão. Tal teste deve ser realizado individualmente, sempre que possível.
iv. Evitar o uso de artimanhas de programação que dificultem a manutenção ou que prejudiquem a inteligibilidade do programa.

v. Os *enfeites* de um programa devem ser deixados para o final dele. Deve-se, primeiramente, ter certeza de que o fundamental foi feito.

A programação é estruturada independentemente da linguagem que será usada, porém algumas linguagens têm estruturas de controle predefinidas, como é o caso das linguagens Pascal, C, Basic e Fortran. No Pascal, essas estruturas são criadas por *blocos de comando*, que são conjuntos de um ou mais comandos colocados entre dois delimitares BEGIN e END. Com o uso dos blocos, pode-se construir dois tipos de estruturas básicas: as estruturas condicionais (ou tomadoras de decisões) e as estruturas de repetição.

Nas estruturas condicionais podem-se ter dois tipos básicos: o primeiro composto por um desvio no fluxo do programa, sendo a execução de tal desvio dependente da veracidade de uma determinada condição. A figura A.1 mostra o algoritmo de uma estrutura condicional típica. Esse tipo pode também assumir a forma da execução ou não de um determinado bloco, dependendo da veracidade de uma condição, conforme mostrado na figura A.2.

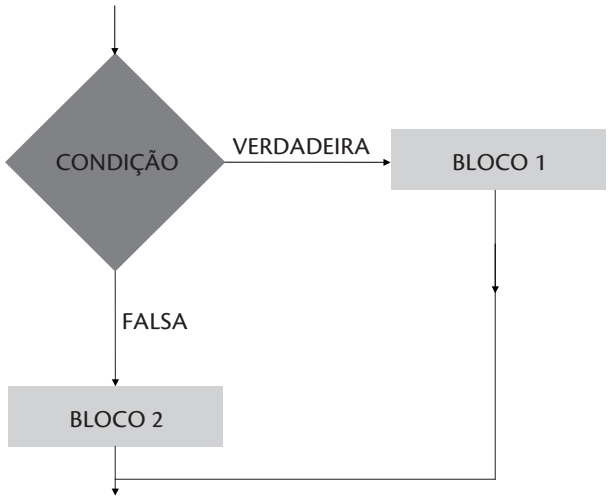


Figura A.1 - Estrutura condicional onde são realizados dois blocos de controle, um para cada resultado possível da condição, que pode ser verdadeira ou falsa

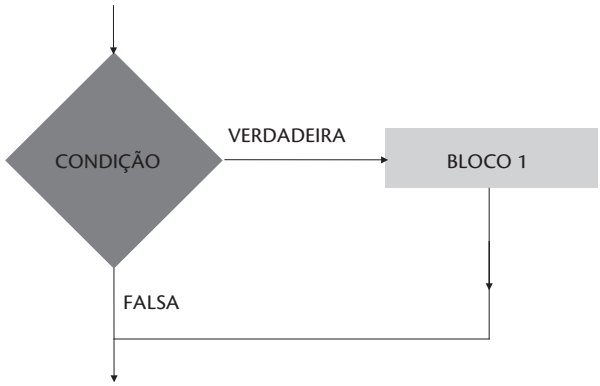


Figura A.2 - Estrutura condicional onde um bloco de instruções será realizado, ou não, dependendo da veracidade da condição

O segundo tipo de estrutura condicional é a estrutura condicional seletiva, em que, dentre vários grupos de blocos diferentes, apenas um é executado, dependendo do resultado de uma expressão que é avaliada no início da estrutura condicional. O resultado da expressão é, então, comparado a uma série de constantes e é executado o bloco de comandos associado à constante cujo valor for igual ao resultado da expressão (figura A.3).

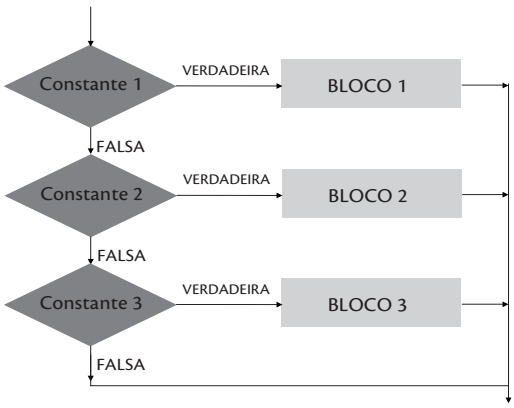


Figura A.3 - Estrutura condicional seletiva

Nas estruturas de repetição existem três formas básicas de formulação de um algoritmo: a primeira permite que um bloco de instruções seja repetido até que uma condição seja satisfeita. Esse tipo de estrutura possui duas características importantes: a primeira é que o teste da condição é realizado no final do bloco, a segunda, que é uma consequência direta da primeira, é que o bloco de instruções será sempre executado pelo menos uma vez. A figura A.4 apresenta o algoritmo dessa estrutura.

O segundo tipo de estrutura de repetição será executado enquanto uma determinada condição for verdadeira. Caso essa condição seja falsa já no início da estrutura, o bloco de instruções não será executado. A figura A.5 mostra o algoritmo dessa estrutura de repetição.

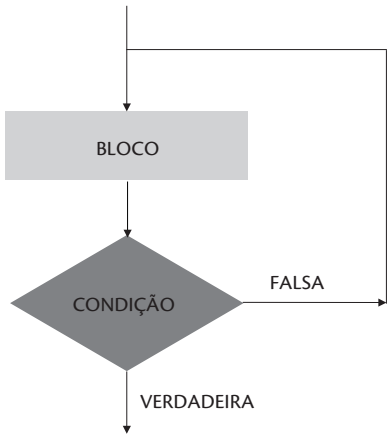


Figura A.4 - Estrutura de repetição na qual um bloco de instruções é repetido até que uma determinada condição seja verdadeira

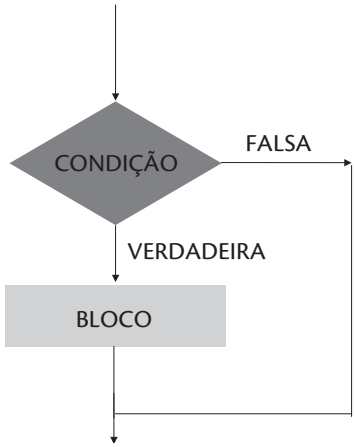


Figura A.5 - Estrutura de repetição que é executada enquanto determinada condição for verdadeira

O terceiro tipo de estrutura permite executar um determinado bloco de instruções *n* vezes, de acordo com uma variável de controle que é incrementada, ou decrementada, automaticamente pelo programa (Figura A.6).

Uma vez conhecidas as estruturas acima, pode-se criar quase todos os programas desejados, apenas tendo o bom senso e a criatividade de usar as estruturas condicionais e de repetição da maneira correta.

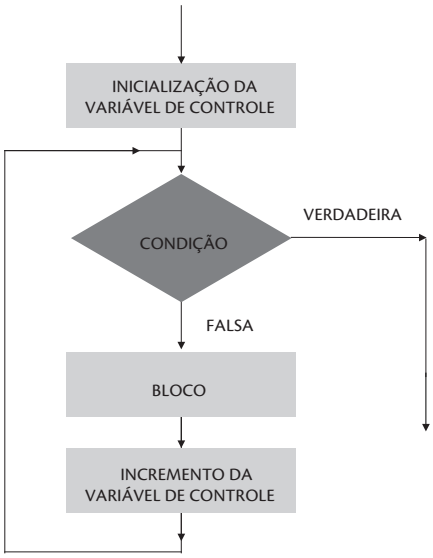


Figura A.6 - Estrutura de repetição na qual um bloco de instruções é executado certo número de vezes pré estabelecido

Sugiro ao leitor que está iniciando os seus estudos de programação por este livro que faça o algoritmo estruturado de **todos** os programas nele contidos neste, uma vez que foram apresentados apenas os códigos-fontes destes. O domínio das técnicas de elaboração de um algoritmo estruturado servirá para nada menos que nos guiar durante todo o desenvolvimento de um programa.

A.2 - Diretivas de compilação

As diretivas de compilação são comentários com sintaxe especial que podem ser usados em qualquer parte do programa, começando pelo caractere $ precedido por um delimitador de abertura de comentário, normalmente "{" e a respectiva diretiva. Podem vir acompanhadas por especificações + ou – após a diretiva. Essas diretivas têm efeitos apenas quando ocorre a compilação do programa e podem ser globais ou locais. Pode-se colocar um grupo de diretivas a serem executadas separando-as por vírgula. As possíveis diretivas de compilação são:

i. **$A** – Alinhamento de dados
SINTAXE: {$A+} ou {$A-}
Default: {$A+}
Tipo: Global
Menu equivalente: O/C/A

As chaves + ou – permitem o alinhamento por byte ou por word, para variáveis ou constantes tipadas. O alinhamento por word não tem efeito no processador 8088. Entretanto, nos grupos de processadores 80x86, faz com que a execução de comandos seja mais rápida, pois essa é a maneira com que são comparados os itens e endereços na memória. No estado {$A+}, todas as variáveis e constantes tipadas que têm como tamanho um número ímpar de bytes são *alargadas* de um byte para que o processador trabalhe no formato word. No estado {$A-}, não existe alinhamento; o endereço de cada variável ou constante tipada é colocado sequencialmente, respeitando-se apenas o próprio tamanho.

ii. **$B** – Avaliação booliana
SINTAXE: {$B+} ou {$B-}
Default: {$B+}
Tipo: Local
Menu equivalente: O/C/B

A avaliação de expressões lógicas boolianas pode ter gerações de códigos diferentes para os operadores boolianos *and* e *or*. No estado {$B+}, o compilador gera o código completo para a avaliação da expressão booliana. Essa forma, apesar de tornar a compilação mais demorada, garante que o valor da operação booliana, que contenha os operadores *and* e *or*, tenha um resultado lógico. No estado {$B-}, o compilador gera um código para uma curta análise da expressão e esta poderá ter um resultado que *trave* o programa, quando do momento da execução, cabendo então ao programador cuidar para que a expressão booliana seja lógica. Em contrapartida, o processador diminui consideravelmente o tempo para se determinar se uma expressão é verdadeira ou falsa.

iii. **$D** – Informações para *Debug*
SINTAXE: {$D+} ou {$D-}

Default: {$D+}
Tipo: Global
Menu equivalente: O/C/D

As informações para *debug* de um programa podem estar disponíveis, ou não. Essas informações consistem do número da linha para a tabela de procedimentos e o mapa do endereço do código-objeto em relação ao programa-fonte. Quando do uso da informação de *debug*, esta irá para o programa ou para a unidade que estiver integrada ao *debugger*, podendo-se marcar pontos de parada (*breakpoints*) em locais estratégicos do programa, executá-lo passo a passo, ou mesmo rotina a rotina. Quando ocorrer um erro de execução em um programa, ou unidade, compilado com a diretiva {$D+}, o ambiente Turbo Pascal poderá localizar o causador do erro no programa. O D/S e o O/L/M são chaves que nos permitem criar um mapa completo de informações do arquivo, porém só estarão disponíveis se utilizada a diretiva {$D+}.

iv. **$E** – Emulação
SINTAXE: {$E+} ou {$E-}
Default: {$E+}
Tipo: Global
Menu equivalente: O/C/E

A diretiva de emulação permite habilitar, ou não, o uso de uma simulação por *software* do coprocessador aritmético 8087, quando este não estiver instalado fisicamente. Quando compilamos um programa utilizando as diretivas {$N+, E+}, o Turbo Pascal conecta todo o emulador 8087. O resultado do arquivo .EXE criado pode ser, então, utilizado em máquinas que tenham ou não um coprocessador aritmético 8087 instalado. O próprio Turbo Pascal fará uso do emulador quando for necessário. No caso de usarmos as diretivas {$N+, E-}, o Turbo Pascal unirá uma pequena rotina de pontos flutuantes, podendo apenas ser utilizada na presença do 8087. A emulação do 8087 não terá efeito se utilizada em uma unidade. Somente tem efeito quando utilizada em programas. Também, quando do uso da diretiva {$N-}, o emulador não terá efeito. Simplesmente será ignorado.

v. **$F** – Forçar FAR CALLS
SINTAXE: {$F+} ou {$F-}
Default: {$F+}
Tipo: Local
Menu equivalente: O/C/F

Essa diretiva controla o tipo de chamadas (*calls*) das rotinas do programa, podendo ser locais (*near*) ou distantes (*far*). Procedimentos e funções compiladas com a diretiva {$F+} usam chamadas distantes e as compiladas com a diretiva {$F-} usam a chamada local. O Turbo Pascal efetua automaticamente a seleção mais apropriada ou a que achar mais conveniente. Em programas que utilizam os conceitos de *overlays* é conveniente que se utilize a diretiva {$F+}.

vi. **$I** – Checagem de entrada/saída ou inclusão de arquivos
SINTAXE: {$I+} ou {$I-} ou {$I nome_arquivo}
Default: {$I+}
Tipo: Local
Menu equivalente: O/C/I ou O/D/I

A diretiva de inclusão {$I nome_arquivo} permite que seja incluído no programa, quando da compilação, o arquivo especificado em "nome_arquivo". É uma solução quando o programa tem tamanho que excede a memória de edição. Caso não seja especificada a extensão no nome do arquivo, esta será assumida como .PAS. Podemos usar essa diretiva em até 15 níveis. As diretivas de checagem de entrada e saída (I/O) podem habilitar ou inibir erros de execução de um programa a cada entrada e saída, seja qual for o dispositivo utilizado. Caso a diretiva {$I+} esteja habilitada e houver um erro na rotina de entrada e/ou saída, o programa será interrompido com a respectiva mensagem de erro. Ao contrário, se esta estiver desabilitada {$I-}, o retorno do erro será enviado a uma função chamada IORESULT, e poderá ser tratado pelo programador.

*vii.***$L** – Símbolos locais
SINTAXE: {$L+} ou {$L-} ou {$L nome_arquivo}
Default: {$L+}

Tipo: {$L+} ou {$L-} Global, {$L nome_arquivo} Local
Menu equivalente: O/C/L e O/D/O

Essa diretiva habilita, ou não, a geração de informações referentes aos símbolos locais, que consistem em nomes e tipos de todas as variáveis e constantes do módulo em questão. A opção O/L/M e D/S somente gerará as informações para símbolos locais se for utilizada a diretiva de compilação {$L+}. Se a diretiva de debug estiver desligada {$D-}, a diretiva {$L} não terá efeito algum. A diretiva {$L nome_arquivo} permite que o objeto de nome_arquivo seja conectado quando da compilação de um programa ou de uma unidade. Quando usada para juntar programas escritos em linguagem Assembly, deverá ser utilizada a declaração EXTERNAL. Quando não utilizarmos a extensão, será assumida como .OBJ.

viii. **$M** – Alocação do tamanho da memória
SINTAXE: {$M ESPAÇO_DO_STACK, HEAPMIN, HEAPMAX}
Default: {$M 16384, 0, 655360}
Tipo: Local

Essa diretiva permite que se escolha o espaço disponível para o Stack e para o Heap. No estado *default* temos 16 kbytes para o Stack e 0 kbytes de memória mínima e 640 kbytes de memória máxima para o Heap. O espaço para o Stack pode variar de 1024 a 65520, sendo este, de preferência, um múltiplo de 1024 (1 kbyte). O Heap mínimo pode variar entre 0 e 655360 e o Heap máximo varia entre o Heap mínimo e 655360. Essa diretiva não tem efeito quando usada em uma unidade.

ix. **$N** – Processador numérico
SINTAXE: {$N+} ou {$N-}
Default: {$N-}
Tipo: Global
Menu equivalente: O/C/N

A diretiva de processador numérico permite dois modos diferentes de geração de código de ponto flutuante para Turbo Pascal, quando da utilização de tipos próprios, tais como: SINGLE, DOUBLE, EXTENDED e

COMP. Na opção {$N-}, o Turbo Pascal utiliza um *software* em substituição ao coprocessador numérico quando da execução do programa. Na opção {$N+} é utilizado o coprocessador aritmético 8087.

x. **$O** – Geração de código de Overplay
SINTAXE: {$O+} ou {$O-} ou {$O nome_unit}
Default: {$O-}
Tipo: {$O+} ou {$O-} Global, {$O nome_unit} Local
Menu equivalente: O/C/O

Essa diretiva habilita, ou não, a geração do código de overplay. As unidades (*units*) no Turbo Pascal só poderão ser utilizadas em overplay quando essa diretiva estiver habilitada {$O+}. Quando a diretiva $O é utilizada para o carregamento de uma unidade, só terá efeito quando da compilação de uma unidade como .OVR de um arquivo .EXE.

xi. **$R** – Checagem de *range*
SINTAXE: {$R+} ou {$R-}
Default: {$R-}
Tipo: Local
Menu equivalente: O/C/R

Quando da utilização da diretiva {$R+}, todas as tabelas de strings indexadas serão verificadas em toda a sua faixa de índice, assim como também todas as variáveis escalares. Essa diretiva faz com que o programa fique mais longo e mais lento, porém os possíveis erros terão a sua mensagem correspondente.

xii. **$S** – Checagem do estouro do Stack (ou pilha)
SINTAXE: {$S+} ou {$S-}
Default: {$S+}
Tipo: Local
Menu equivalente: O/C/S

Essa diretiva permite, ou não, a geração de código de estouro de STACK. Quando estiver ligada {$S+}, o compilador gerará código no começo de um procedimento, ou função, e verificará se há espaço suficiente na pilha para o

armazenamento das variáveis locais. No caso de não haver espaço suficiente, o programa será interrompido com o código de erro correspondente. Se essa diretiva estiver desligada e não existir espaço suficiente no STACK, poderão ocorrer erros no sistema operacional.

xiii. **$V** – Checagem de String
SINTAXE: {$V+} ou {$V-}
Default: {$V+}
Tipo: Local
Menu equivalente: O/C/V

A checagem de variáveis do tipo String é utilizada quando elas são usadas como parâmetros. No estado {$V+}, é requerido que os tipos sejam exatamente iguais, tanto o parâmetro quanto a variável local. Já no estado {$V-}, o parâmetro passado pode ter qualquer tamanho menor ou igual à variável local.

A.3 - Mensagens de erros do Turbo Pascal

A versão 7.0 do Pascal apresenta o mesmo sistema das versões anteriores no que diz respeito às mensagens de erros, pois também divide os erros em categorias: Erros de Compilação, Erros em tempo de Execução provenientes do DOS ou I/O e Erros Críticos. Também na versão 7.0 mantém-se a característica de se apontar o erro com o cursor se posicionando sobre o erro, com a mensagem sendo destacada na linha de comandos.

A.3 .1 - Erros de compilação

1 **Out of memory –** Esse erro ocorre quando a quantidade de memória disponível no equipamento não é suficiente para compilar o programa. Algumas soluções podem ser tentadas para solucionar esse problema:
- Compilar o Programa com destino para o disco e não para memória, para isso utilizamos a opção DESTINATION DISK.
- Utilizar a opção LINK *BUFFER* DISK, de modo a utilizar o disco e não a RAM como *buffer* de enlace.
- Retirar todos os utilitários instalados na memória.

- Compilar por meio do TPC.EXE, que é um arquivo menor que o TURBO.EXE.
- Dividir o programa em UNITS, compilando-as separadamente.

2 **Identifier expected –** Identificador esperado. Nesse local era esperado um identificador, ou houve a tentativa de redefinir uma palavra reservada.

3 **Unknown identifier –** Identificador não declarado no programa.

4 **Duplicate identifier –** Identificador repetido no mesmo bloco.

5 **Syntax error –** Erro de sintaxe. Existe algum caractere ilegal na declaração ou os "abre-fecha" aspas em torno de uma string não estão corretos.

6 **Error in real constant –** Constante real inválida.

7 **Error in integer constant** – Constante inteira inválida.

8 **String constant exceeds line** – Constante string maior que 255 caracteres.
Verificar os "abre-fecha" aspas ou se o tamanho da string de fato excede 255 caracteres.

9 **Unexpected end of file** – Fim de Arquivos inesperado. Pode-se verificar:
- O casamento dos BEGIN-END.
- O casamento de "abre-fecha" chaves de comentários.
- Se algum arquivo de inclusão está finalizando de forma inválida.
- O compilador não encontrou o END de final de arquivo.

10 **Line too long** – Linha muito longa, ultrapassando o limite de 126 caracteres.

11 **Type identifier expected** – Identificador de tipo era esperado.

12 **Too many open files** – Muitos arquivos abertos ao mesmo tempo. Esse erro ocorre quando não existe o arquivo CONFIG.SYS no sistema operacional, ou foram abertos mais arquivos do que o especificado no arquivo de configuração, indica-se FILES=20.

13 **Invalid file name** – Arquivo não encontrado. Verificar se os nomes estão corretos ou se existe a especificação do caminho.

14 **File not found** – Arquivo não encontrado no diretório. Verificar nome do arquivo e o caminho.

15 **Disk full** – Disco cheio. Delete alguns arquivos desnecessários ou utilize outro disco.

16 **Invalid compiler directive** – Diretiva de compilação incorreta.

17 **Too many files** – Existem muitos arquivos envolvidos na compilação do programa.

18 **Undefined type in pointer definition** – Tipo pointer não definido.

19 **Variable identifier expected** – Identificador de variável esperado.

20 **Error in type** – Erro na definição do tipo, o caractere encontrado é inválido.

21 **Structure too large** – Tipo estruturado com mais de 65.520 bytes.

22 **Set base type out of range** – Tipo base fora do intervalo de 0 a 255 ou tipos enumerados maiores ou iguais a 256.

23 **File components may not be files** – Arquivos de arquivo não são uma estrutura permitida.

24 **Invalid string length** – String com mais de 255 caracteres.

25 **Type mismatch** – Tipo trocado. Pode-se verificar o seguinte:
- Incompatibilidade de tipos em um comando de atribuição.
- Incompatibilidade de parâmetros e argumentos num procedimento ou função.
- Incompatibilidade entre índices de vetores em uma declaração.
- Incompatibilidade entre tipos de operandos em uma expressão.

26 **Invalid subrange base type** – Intervalo de variação do tipo base está inválido.

27 **Lower bound greather than upper bound** – Limite superior do intervalo é maior que o do tipo.

28 **Ordinal type expected** – Tipo ordinal esperado, outros tipos como string, real e pointer não são aceitos.

29 **Integer constant expected** – Constante inteira esperada.

30 **Constant expected** – Constante esperada.

31 **Integer ou real constant expected** – Constante real ou inteira esperada.

32 **Type identifier expected** – Identificador de tipo esperado.

33 **Invalid function result type** – Como resultado de uma função, somente é esperado um tipo simples, string ou pointer.

34 **Label identifier expected** – Identificador de rótulo esperado.

35 **BEGIN expected** – BEGIN esperado.

36 **END expected** – END esperado.

37 **Integer expression expected** – Expressão inteira esperada.

38 **Ordinal expression expected** – Expressão ordinal esperada.

39 **Boolean expression expected** – Expressão booliana esperada.

40 **Operand types do not match operator** – Tipo do operando incompatível com o tipo do operador.

41 **Error in expression** – Verifique a validade de sua expressão.

42 **Illegal assignment** – Não é permitido atribuir valores a variáveis sem tipo, nem a arquivos. Outra verificação é que um identificador de função somente poderá receber valores dentro da própria função (escopo da variável).

43 **Field identifier expected** – Identificador de campo esperado.

44 **Object file too large** – Arquivo .OBJ com mais de 64 kb.

45 **Undefined external** – External indefinido.

46 **Invalid object file record** – Inválido registro de arquivo objeto.

47 **Code segment too large** – Programa ou unidade ultrapassou o limite de 65 Kb.

48 **Data segment too large** – Arquivo de dados ultrapassou limite de 65 Kb.

49 **DO expected** – Comando DO esperado.

50 **Invalid PUBLIC definition** – Definição PUBLIC inválida.

51 **Invalid EXTRN definition** – Definição EXTRN inválida.

52 **Too many EXTRN definitions** – Número de definições EXTRN superior a 256.

53 **OF expected** – Cláusula OF esperada.

54 **Interface expected** – Cláusula INTERFACE esperada.

55 **Invalid relocatable reference** – Referência realocável inválida.

56 **THEN expected** – THEN esperado.

57 **TO or DOWNTO expected** – TO ou DOWNTO esperado.

58 **Undefined FORWARD** – FORWARD indefinido:

- O procedimento ou função foi declarado na INTERFACE, mas sua declaração nunca ocorreu na IMPLEMENTATION.

- O procedimento ou função foi declarado como FORWARD, mas essa definição não foi encontrada.

59 **Invalid typecast** – Molde de tipos inválidos.

60 **Division by zero** – Divisão por zero. Esse erro ocorre também quando a versão 7.0 do Turbo Pascal não foi atualizada. Existe um *patch* (remendo) para corrigir esse problema disponível na internet.

61 **Invalid file type** – Tipo de arquivo inválido.

62 **Cannot Read or Write variables of this type** – Não é possível ler ou escrever em uma variável desse tipo com os procedimentos READ ou WRITE.

63 **Pointer variable expected** – Variável do tipo POINTER esperada.

64 **String variable expected** – Variável do tipo STRING esperada.

65 **String expression expected** – Expressão do tipo STRING esperada.

66 **Circular unit reference** – Uma UNIT foi incluída na cláusula USES dela própria.

67 **Unit name mismatch** – UNIT com nome trocado. UNIT declarada, porém não existe o correspondente arquivo .TPU.

68 **Unit version mismatch** – Versão da UNIT incorreta, deve-se recompilar as UNITS.

69 **Duplicate unit name** – UNIT com nome duplicado.

70 **Unit file format error** – Arquivo .TPU de alguma UNIT com problema.

71 **Implementation expected** – IMPLEMENTATION esperado.

72 **Constant and case types do not match** – Tipos trocados entre as constantes do CASE e a variável seletor.

73 **Record variable expected** – Variável do tipo registro esperada.

74 **Constant out of range** – Valor da constante fora da faixa de limites.
75 **File variable expected** – Variável do tipo arquivo esperada.
76 **Pointer expression expected** – Expressão do tipo POINTER esperada.
77 **Integer or real expression expected** – Expressão do tipo inteiro ou real esperada.
78 **Label not within current block** – LABEL não está presente no bloco corrente.
79 **Label already defined** – LABEL já definido.
80 **Undefined label in preceding statement part** – LABEL não definido na área anterior ao comando.
81 **Invalid @ argument** – Inválido argumento @.
82 **UNIT expected** – UNIT esperada.
83 **";" expected** – ; esperado. Verificar linha de comando superior à indicada pelo compilador.
84 **":" expected** – : esperado.
85 **"," expected** – , esperada.
86 **"(" expected** – (esperado.
87 **")" expected** –) esperado.
88 **"=" expected** – = esperado.
89 **":=" expected** – := esperado.
90 **"[" or "(." expected** – [ou (esperado.
91 **"]" or ".)" expected** –] ou) esperado.
92 **"." expected** – . esperado.
93 **".." expected** – .. esperado.
94 **Too many variables** – Muitas variáveis. O total de variáveis não pode ultrapassar o limite dos 64 KB, considerando: procedimentos, funções, units e programas.
95 **Invalid FOR control variable** – Variável do FOR inválida. Verificar o tipo da variável.
96 **Integer variable expected** – Variável inteira esperada.
97 **Files are not allowed here** – Arquivos não podem ser usados aqui.
98 **String length mistmatch** – String com comprimento incompatível.
99 **Invalid ordering of fields** – Ordenação inválida dos campos.
100 **String constant expected** – Constante do tipo string esperada.
101 **Integer or real variable expected** – Variável inteira ou real esperada.
102 **Ordinal variable expected** – Variável escalar esperada.
103 **INLINE error** – Erro no comando INLINE.
104 **Character expression expected** – Expressão do tipo caractere esperada.
105 **Too many relocations items** – Muitos itens a serem realocados. Seu programa ultrapassou o limite de tamanho e deve ser dividido em partes.

106 **Overflow in arithmetic operation** – Estouro de variável. O resultado da operação aritmética anterior não está na faixa LONGINT. Corrija a operação ou utilize um valor do tipo real em vez de valores do tipo inteiro.

107 **No enclosing FOR, WHILE, or REPEAT statement** – Os procedimentos padrões BREAK e CONTINUE não podem ser utilizados fora dos comandos FOR, WHILE ou REPEAT.

108 **CASE constant out of range** – Constante do CASE maior que 32.676 ou menor que -32.768.

109 **Error in statement** – Erro no comando. Verifique se o nome da função ou procedimento está correto.

110 **Cannot call an interrupt procedure** – Não se pode fazer uma chamada a um procedimento interrompido.

111 **Must be in 8087 mode to compile this** – As operações que utilizam números reais do tipo SINGLE, DOUBLE, EXTENDED E COMP exigem um coprocessador 8087 ou 80287.

112 **Target address not found** – Houve erro com a opção FIND ERROR do menu de compilação e não foi encontrado o local do erro procurado.

113 **Include files are not allowed here** – Não é aceita a inclusão de arquivos nesse local.

114 **No inherited methods are accessible here** – Você está utilizando a palavra chave INHERITED fora do método ou em um método do tipo objeto que não tem ancestral.

115 **Invalid qualifier** – Qualificador inválido. Verificar:
- Quando usar conjuntos, indexar apenas uma variável.
- Quando especificar campos, definir antes qual o registro.
- A referência somente pode se retirada das variáveis tipo pointer.

116 **Invalid variable reference** – Para chamar uma função do tipo pointer deve-se tirar a referência ao resultado.

117 **Too many symbols** – Os símbolos utilizados por seu programa ou UNIT ultrapassam o limite de 64 Kb.

118 **Statement part too large** – Verifique se a parte de execução de seu programa ultrapassa o limite de 24 KB.

119 **Files must be var parameters** – É esperado pelo Turbo Pascal que seus arquivos tenham parâmetros do tipo VAR.

120 **Too many conditional symbols** – Símbolos condicionais em excesso.

121 **Misplaced condicional directive** – Diretiva de compilação condicional incompleta ou em local errado.

122 **ENDIF directive missing** – Falta diretiva {$ENDIF} da diretiva {$IFYYY}.

123 **Error in initial conditional defines** – Erro nas definições das condições iniciais.

124 **Header does not match previous definition** – Cabeçalho não corresponde ao que foi definido na parte de INTERFACE ou FORWARD do procedimento ou função.

125 **Canot evaluate this expression** – Não é possível avaliar essa expressão.

126 **Expression incorrectly terminated** – Expressão terminada incorretamente.

127 **Invalid format specifier** – Especificador de formato inválido.

128 **Invalid indirect reference** – Referência indireta inválida.

129 **Structured variables are not allowed here** – Variáveis estruturadas não são permitidas aqui.

130 **Cannot evaluate without system unit** – Avaliação não é possível sem a UNIT SYSTEM.

131 **Cannot acess this symbol** – Não é permitido acessar esse símbolo.

132 **Invalid floating-point operation** – Operação de ponto flutuante inválida ou divisão por zero.

133 **Cannot compile overlays to memory** – Não é possível compilar overlays na memória.

134 **Procedural or function variable expected** – Variável de procedimento ou função esperada.

135 **nvalid procedure or function reference** – Referência inválida a um procedimento ou função.

136 **Cannot overlay this unit** – Não é possível tornar esta UNIT num overlay.

137 **File access denied** – O arquivo não pode ser aberto ou criado. O compilador está tentando escrever em um arquivo de somente leitura.

138 **Object type expected** – O identificador não reconhece o tipo do objeto.

139 **Local object types are not allowed** – O tipo objeto só pode ser definido dentro do escopo de um programa ou unidade (global), não podendo ser definido dentro de procedimentos e funções (local).

140 **VIRTUAL expected** – A palavra reservada VIRTUAL não foi encontrada.

141 **Method identifier expected** – Identificador de método esperado.

142 **Virtual constructors are not allowed** – Construtores virtuais não são permitidos.

143 **Constructor identifier expected** – Identificador CONSTRUCTOR esperado ou não reconhecido.

144 **Destructor identifier expected** – Identificador DESTRUCTOR esperado ou não reconhecido.

145 **Fail only allowed within constructors** – O procedimento padrão FAIL só pode ser utilizado dentro de CONSTRUCTORS.

146 **Invalid combination of opcode and operands** – Operação de código Assembler não aceita essa combinação de operandos. Pode ocorrer excesso ou falta de operandos ou o tipo e ordem não estão de acordo com o código Assembler.

147 **Memory reference expected** – O operador Assembler não é uma referência de memória, a qual é requerida aqui. Muito provavelmente você esqueceu de colocar colchetes em torno do registrador.

148 **Cannot add or subtract relocatable symbols** – A única operação aritmética que pode ser realizada com símbolo realocável em um operando Assembler é adição ou subtração de constantes. Variáveis, procedimentos, funções e rótulos são símbolos realocáveis.

149 **Invalid register combination** – Combinação de registrador inválida.

150 **Instructions are not enabled** – Use a diretiva de compilação {$G+} para permitir operações de código no 286/287, mas esteja consciente de que o código resultante não pode rodar nas máquinas 8086 e 8088.

151 **Invalid symbol reference** – Esse símbolo não pode ser acessado em um operando Assembler.

152 **Code generation error** – Parte do comando precedente contém uma instrução que não pode atingir o seu término. Verifique os loops do programa.

153 **ASM expected** – Você está tentando compilar uma função ou procedimento em Assembler que contém o comando BEGIN...END em vez de ASM...END.

A.3.2. Erros em tempo de execução

Os erros em tempo de execução (ou *Runtime errors*) podem ser classificados em:

- DOS ERRORS – Erros do DOS (1 a 99).
- I/O ERRORS – Erros de Entrada e Saída (100 a 149).
- CRITICAL ERRORS – Erros críticos (150 a 199).
- FATAL ERRORS – Erros fatais (200 a 255).

Um erro em tempo de execução interrompe o processamento e envia para o vídeo a seguinte mensagem:

```
RUNTIME ERROR nnn AT xxxx:yyyy
onde: nnn – representa o número do erro.
xxxx:yyyy – representa o endereço do erro.
```

A.3.2.1. Erros de DOS

1 **Invalid function number** – Chamada de função inexistente no DOS.

2 **File not found** – Arquivo solicitado pelo RESET, APPEND, RENAME ou ERASE não foi encontrado.

3 **Path not found** – Não foi encontrado o caminho indicado, verificar a estrutura de subdiretórios, bem como as linhas de chamadas de arquivos nesse subdiretório.

4 **Too many open files** – Verifique a existência de um arquivo CONFIG.SYS, que deverá indicar o número máximo de arquivos que podem ser abertos ao mesmo tempo.

5 **File access denied** – Não foi possível acessar os arquivos através do RESET, APPEND, FILEMODE, REWRITE, RENAME, ERASE, MKDIR, RMDIR, READ, BLOCKREAD, WRITE OU BLOCKWRITE.

6 **Invalid file handle** – Erro no manuseio do arquivo, verificar se a variável referente ao nome do arquivo não foi alterada.

12 **Invalid file access code** – Arquivo indicado pelo RESET ou APPEND tem um valor de FILEMODE inválido, causando código inválido de acesso ao arquivo.

15 **Invalid drive number** – Driver inválido, erro acusado pelo GETDIR.

16 **Cannot remove current directory** – Impossível remover o diretório atual, erro acusado pelo RMDIR.

17 **Cannot rename across drives** – Impossível renomear o nome do arquivo, pois os dois se encontram no mesmo driver. Erro acusado pelo RENAME.

18 **No more files** – Relatado pela variável DOSERROR na unidade DOS e WINDOS, quando FINDFIRST ou FINDNEXT não encontra arquivo que combine com o nome de arquivo especificado e atributos.

A.3.2.2 - Erros de entrada e saída de dados (I/O ERRORS)

Os erros de entrada e saída podem ser tratados através da diretiva {$I}. O valor default dessa diretiva é {$I+}, isso provoca uma interrupção na execução do programa, sempre que ocorrer erros. A outra opção para essa diretiva é {$I-}, o que não acarretará a interrupção do programa, porém um código de erro é emitido, podendo ser recolhido pela função IORESULT e com isso tomarmos as providências necessárias.

100 **Disk read error** – Tentativa de ler uma variável após o final do arquivo, ocasionando erro de leitura.

101 **Disk write error** – Não há espaço no disco para gravação.

102 **File not assigned** – Não foi feita a associação correta do arquivo através do comando ASSIGN.

103 **File not open** – Arquivo não foi aberto; ocorre num CLOSE, READ, WRITE, SEEK, EOF, FILEPOS, FILESIZE, FLUSH, BLOCKWRITE, BLOCKREAD.

104 **File not open for input** – Arquivo não aberto para entrada de dados; refere-se a um arquivo tipo texto.

105 **File not open for output** – Arquivo não foi aberto para saída de dados; refere-se também a um arquivo tipo texto.

106 **Invalid numeric format** – Valor numérico incompatível com o arquivo do tipo texto.

A.3.2.3 - Erros críticos

150 **Disk is write-protected** – Disco Protegido contra gravação.

151 **Unknown unit** – UNIT não encontrada no disco ou no diretório informado.

152 **Drive not ready** – Disco não formatado ou com defeito.

153 **Unknown command** – Comando não conhecido pelo Pascal.

154 **CRC error in data** – Do inglês *Cyclic Redundancy Check* ou verificação de redundância cíclica é um código detector de erros. Esse erro indica um ou mais setores danificados no disco.

155 **Bad drive request structure length** – Acesso ao disco danificado.

156 **Disk seek error** – Erro de procura em disco.

157 **Unknown media type** – Tipo de meio de comunicação não encontrado.

158 **Sector not found** – Setor não encontrado, disco com problemas.

159 **Printer out of paper** – Impressora sem papel.

160 **Device write fault** – Direcionamento de saída de dados com defeito.

161 **Device read fault** – Direcionamento de leitura de dados com defeito.

162 **Hardware failure** – Hardware falhando.

A.3.2.4 - Erros fatais

200 **Division by zero** – Divisão por zero.

201 **Range check error** – Intervalo de variação do índice de um vetor está fora da faixa.

202 **Stack overflow error** – Falta de espaço na pilha de alocação de variáveis locais de um subprograma; pode-se solucionar pela opção MEMORY SIZES do menu de compilação.

203 **Heap overflow error** – Espaço insuficiente na memória para o HEAP.

204 **Invalid pointer operation** – Operação com variáveis do tipo POINTER inválida.

205 **Floating point overflow** – Operação com ponto flutuante causando transbordo de memória.

206 **Floating point underflow** – Operação causando transbordo negativo de memória, ocorre geralmente quando se trabalha com coprocessador 8087.

207 **Invalid floating point operation** – Operação com ponto flutuante inválida, por exemplo SRQT ou LN com argumentos de valores negativos ou estouro de pilha com 8087.

208 **Overlay manager not installed** – Gerenciador de overlay não foi instalado.

209 **Overlay file read error** – Erro na leitura do arquivo overlay.

210 **Object not initialized** – Com uma faixa de verificação em operação você fez um chamado para um método de objeto virtual antes que o objeto tenha sido inicializado via chamada do construtor.

211 **Call to abstract method** – Esse erro é gerado por um procedimento abstrato numa unidade objeto; isso indica que o seu programa tenta executar um método abstrato virtual.

212 **Stream registration error** – Esse erro é gerado pelo procedimento REGISTER TYPE na unidade objeto.

213 **Collection index out of range** – O índice passado para o método de uma TCollection está fora de faixa.

214 **Collection overflow error** – O erro é relatado por uma TCollection se uma tentativa é feita para adicionar um elemento quando a coleção não pode ser expandida.

215 **Arithmetic overflow error** – Esse erro é relatado por um comando compilado na diretiva {$Q+} quando uma operação aritmética inteira provoca um estouro, tal qual quando o resultado de uma operação está fora da faixa suportada.

Referências

DIJKSTRA, E. W. *A short introduction to the art of programming*. Netherlands: Technological University Eindhoven, 1971.

ZIVIANI, N. *Projeto de algoritmos* – com implementações em Pascal e C. 4. ed. São Paulo: Pioneira, 1999. 552p.

Bibliografia sugerida

ALBANO, R. S. *Turbo Delphi Explorer*. Rio de Janeiro: Ciência Moderna, 2008. 368p.

ASCENCIO, A. F. G. *Lógica de programação com Pascal*. São Paulo: Makron Books, 1999. 120p.

_____; CAMPOS, E. A. V. *Fundamentos da programação de computadores* – algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson/Prentice Hall, 2007. 448p.

AVILLANO, I. C. *Algoritmos e Pascal:* manual de apoio. Rio de Janeiro: Ciência Moderna, 2006. 560p.

_____. *Object Pascal para Delphi*. Rio de Janeiro: Ciência Moderna, 2009. 192p.

BOENTE, A. *Aprendendo a programar em Pascal*. Rio de Janeiro: Brasport, 2003. 288p.

DAGHLIAN, J. *Lógica e álgebra de Boole*. 4. ed. São Paulo: Atlas, 1995. 168p.

EVARISTO, J. *Programando com Pascal*. 2. ed. Maceió: Book Express, 2004.

FARRER, H.; BECKER, C. G.; FARIA, E. C. *Programação estruturada de computadores* – Pascal estruturado. Rio de Janeiro: Guanabara, 1985. 141p.

FORBELLONE, A. L.; EBERSPACHER, H. F. *Lógica de programação*. 3. ed. Rio de Janeiro: Prentice Hall Brasil, 2005. 232p.

GUIMARÃES, A. M.; LAGES, N. A. C. *Algoritmos e estruturas de dados*. Rio de Janeiro: LTC, 1994. 216p.

LAUREANO, M. *Lógica de programação* – uma abordagem em Pascal. Rio de Janeiro: Ciência Moderna, 2010. 376p.

LOPES, A.; GARCIA, G. *Introdução à programação* – 500 algoritmos resolvidos. Rio de Janeiro: Campus, 2002. 488p.

MANZANO, J. A. N. G.; OLIVEIRA, J. F. *Estudo dirigido de algoritmos*. 13. ed. São Paulo: Érica, 2010. 240p.

_____; YAMATUMI, W. Y. *Free Pascal* – Programação de Computadores. São Paulo: Érica, 2007. 392p.

NAYAK, A.; STOJMENOVIC, I. *Handbook of applied algorithms:* solving scientific, engineering, and practical problem. New Jersey: John Wiley & Sons, 2008. 544p.

PAIVA, S. *Introdução à programação:* do algoritmo às linguagens atuais. Rio de Janeiro: Ciência Moderna, 2008. 208p.

SALVETTI, D. D.; BARBOSA, L. M. *Algoritmos*. São Paulo: Makron Books, 1997. 274p.

SCHLIEVE, P. L. *Turbo Pascal Ilustrado*. Rio de Janeiro: Lutécia, 1987. 357p.

Revisão
Cânone Editoração

Projeto gráfico da coleção
Alanna Oliva

Editoração eletrônica
Alanna Oliva

Dados internacionais de catalogação-na-publicação (CIP)
(Cássia Oliveira)

S381t Silva, André Carlos.
Turbo Pascal para Engenheiros / André Carlos Silva. – Goiânia : DEPECAC-UFG/FUNAPE, 2013.

208p. (Coleção Lavor)
ISBN: 978-85-8083-067-5

1. Informática 2. Engenharia da computação 3. Programação de computadores 4. Turbo Pascal I. Título.

CDU 004.43

Projeto gráfico, impressão e acabamento
Campus Samambaia, Caixa Postal 131
CEP: 74001-970 - Goiânia - Goiás - Brasil
Fone: (62) 3521-1107 - Fax: (62) 3521-1814
editora@editora.ufg.br
www.editora.ufg.br

Cegraf
UFG