APRESENTAÇÃO

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

O aprendizado de algoritmos não se consegue a não ser através de muitos exercícios. Não se aprende algoritmos apenas copiando e estudando. Algoritmos só se aprendem construindo e testando.

"A rocha é imensa e dura. O cortador bate uma, duas, três, dez vezes, nenhuma rachadura. Ele dá 100 marteladas, só tirando lascas. Na centésima primeira batida, a rocha imensa e dura se parte em duas. O cortador de pedras sabe que não foi somente aquela martelada a que conseguiu, mas também todas as que vieram antes".

"E na próxima pedra, o cortador pode pegar uma ferramenta mais apropriada e cortar a pedra mais rapidamente".

Comentários, críticas e sugestões serão muito bem-vindos e podem ser enviados para o email giselle@ucam-capos.br.

Home page da disciplina:
http://www.ucam-campos.br/graduacao/computacao/disciplinas/matriz2/2periodo/tec_prog1/tec_prog1.htm

Giselle Teixeira de Almeida

Campos dos Goytacazes, março de 2003.

CAPÍTULO I – INTRODUÇÃO

Este capítulo tem por objetivo dar ao leitor os principais conhecimentos necessários ao aprendizado da Linguagem Turbo Pascal.

---

1. A Linguagem Pascal

A Linguagem Pascal destina-se à programação de computadores. Seu nome trata-se de uma homenagem ao matemático e filósofo francês, Blaise Pascal (1623-1662), criador da primeira calculadora mecânica. Esta linguagem foi desenvolvida por volta de 1968, por Niklaus Wirth, no Technical University, em Zurique, na Suíça, com o objetivo de ensinar aos seus alunos a programar em ALGOL e PLI. Desde então, as qualidades da Linguagem Pascal vêm sendo universalmente reconhecidas.

A criação da Linguagem Pascal teve por objetivo a obtenção de uma linguagem simples, capaz de incentivar a confecção de programas claros e facilmente legíveis, favorecendo a utilização de boas técnicas de programação. Suas características foram criteriosamente escolhidas visando a permitir a obtenção de programas confiáveis, modularizados e bem estruturados, processáveis por compiladores compactos, eficientes e econômicos.

Sob outros aspectos, a Linguagem Pascal inovou, em relação às linguagens existentes até então, ao apresentar e utilizar outros conceitos: criação de novos tipos e subtipos dentro de um programa; manipulação de tipos enumeráveis; a estruturação de dados, que permite criar e manipular, além de arranjos, também registros, conjuntos e arquivos; a alocação dinâmica de variáveis, com o auxílio de ponteiros; declaração de identificadores para constantes: utilização de procedimentos que lêem e escrevem em campos individuais em vez de registros completos, uso de procedimentos e função recursivos, etc.

2. Turbo Pascal

Em 1970 foi disponibilizado o primeiro compilador para a Linguagem Pascal, ainda um produto acadêmico. Foi em novembro de 1983 que surgiu a primeira versão do Turbo Pascal, criada pela Borland International, logo atualizada para a versão 2.0. Naquela época, a Borland era uma pequena e desconhecida empresa de Scotts Valley. Ainda na década de 80, foram lançadas outras versões: 3.0, 4.0, 5.0, 5.5, 6.0, e o 7.0. O nosso objetivo de estudo é a versão 7.0. A mesma apresenta várias características importantes:

- É uma linguagem extremamente rápida, tanto no tempo de compilação quanto no tempo de execução dos programas.

- Embora o compilador possa usar arquivos criados por muito editores de texto, o editor do Turbo Pascal é extremamente eficiente e está diretamente ligado às rotinas de manipulação de erros do compilador. Quando o compilador detecta um erro, o controle é automaticamente transferido para o editor e o cursor indica a localização do erro juntamente com uma mensagem de descrição.

- O Turbo Pascal permite utilizar com eficiência os recursos de hardware do seu computador assim, por exemplo, utilizando declarações do Pascal, você pode desenhar na tela e compor músicas. Os programadores mais experientes podem combinar programas em Pascal com subrotinas em linguagem de máquina e se comunicarem diretamente com os terminais de entrada e de saída (I/O) e com o sistema operacional do computador.

3. Compilação e Linkedição

Quando escrevemos um programa em qualquer linguagem de alto nível (como C, Pascal, Delphi, Java, VisualBasic, etc), utilizamos um editor de textos para escrever uma série de comandos e códigos que desejamos que o computador execute. Este arquivo contendo estes códigos e comandos em linguagem de alto nível é chamado programa fonte.

Entretanto, o computador não é capaz de compreender os comandos contidos neste programa (ou arquivo) fonte, pois a única linguagem que o computador entende é a linguagem de baixo nível, conhecida como linguagem de máquina, extremamente complexa e desagradável para os padrões humanos. Assim, deve haver um processo de "tradução" que transforma o nosso programa fonte em um programa equivalente escrito em linguagem de máquina. As duas principais maneiras de realizar este processo de tradução são chamadas de interpretação e compilação.

No processo de interpretação, o programa interpretador analisa cada linha de seu programa fonte separadamente, verifica se esta linha está correta e, caso esteja, gera uma linha equivalente em linguagem de máquina e realiza a execução. Este processo se repete até que a última linha do seu programa tenha sido executada. No final, o código em linguagem de máquina resultante da tradução das linhas de seu programa fonte não é guardado. Todas as linhas são descartadas e todo o processo de tradução deve ser repetido em uma nova execução.

Já no processo de compilação, o programa compilador da linguagem analisa todo o programa fonte gerado. Caso nenhum erro seja encontrado, todo o programa fonte será traduzido para uma linguagem de baixo nível e armazenado em um arquivo separado, chamado de arquivo objeto. Um processo adicional chamado de linkedição, transforma este arquivo objeto em um arquivo executável, também capaz de ser compreendido pela máquina. O arquivo executável, como o próprio nome indica, está pronto para ser executado pelo computador.

Uma das principais vantagens da compilação está no fato de que, uma vez gerado o arquivo executável, ele pode ser utilizado inúmeras vezes sem a necessidade da presença do compilador ou qualquer outro utilitário, ao passo que os interpretadores são sempre necessários para a execução de seus programas.

Outra vantagem está na velocidade de execução. Um programa compilado possui uma execução muito mais rápida que um equivalente interpretado, pois a interpretação sempre tem de realizar a tradução de cada linha do seu programa fonte.

Por último, os compiladores garantem segurança em relação ao seu código-fonte, já que ele não é necessário para a execução de um programa já compilado.

4. Ambiente de Desenvolvimento

O ambiente de desenvolvimento do Turbo Pascal é composto de um editor de texto, um compilador, um programa ligador (link) e um depurador (debugger), harmonicamente integrados.

O número de janelas que podem ser abertas simultaneamente na área de trabalho está limitado pelo espaço na memória RAM do equipamento. Cada janela pode conter um arquivo. Um mesmo arquivo pode ser colocado em várias janelas diferentes, caso haja necessidade de visualizar simultaneamente áreas diferentes. Apenas a janela ativa é completamente visível.

TECLAS	EFEITO
F5	Zoom da janela ativa
F6	Próxima janela
F3	Abrir arquivo
F2	Salvar arquivo
Alt+x	Sair do Turbo Pascal
Alt+n	Ativar a janela n
Alt+F3	Fechar arquivo
Alt+F5	Janela do Usuário
Alt+F9	Compilar o programa
Ctrl+Ins	Copiar conteúdo marcado para a memória intermediária (clipboard)
Ctrl+F9	Rodar o programa
Shift+Del	Mover conteúdo marcado para o clipboard
Shift+F6	Janela anterior
Shift+Ins	Colar o conteúdo copiado para o clipboard

Tabela 1.1 - Tabela de teclas de atalho para janelas e operações.

5. Execução de um Programa

Após editar um arquivo é natural compilá-lo e rodá-lo, para ver os resultados do programa. Durante esse processo, o Pascal irá procurar erros em tempo de compilação. Senão houver erros, o programa rodará normalmente. Caso contrário, o processo de compilação será interrompido e o compilador indicará o erro e a sua posição.

Para executar um programa, ative a opção RUN do Menu RUN ou pressione Ctrl+F9.

Ao terminar a execução, o Pascal volta para a janela de edição. Para ver os resultados do programa, basta pressionar Alt+F5.

A compilação normalmente é feita na memória. Se desejar compilar o programa para um disco em um arquivo executável, mude a opção DESTINATION do menu COMPILE para DISK.

CAPÍTULO II – DADOS NO PASCAL

Este capítulo tem por objetivo demonstrar ao leitor a utilização de dados nos seus programas escritos em Pascal.

---

1. Manipulando Dados

Quando escrevemos nossos programas, trabalhamos com dados que nós fornecemos ao programa (literais) e dados são fornecidos ao programa através do usuário (variáveis). No Turbo Pascal, esses dados são divididos em diversos tipos, cada qual com a sua função específica.

2. Variáveis

O objetivo das variáveis é armazenar dados na memória do computador. Podemos utilizar as literais em nossos programas, mas sem as variáveis não seria possível, por exemplo, solicitar dados ao usuário para alimentar o programa.

Uma variável, no Pascal, é referenciada por um identificador e, por isso, sua criação segue as regras da formação dos identificadores: (os identificadores servem para nomear procedimentos, funções, tipos de dados, etc).

- Os identificadores devem começar com uma letra (A..Z) ou por um sinal de sublinhado/underscore (_).
- Todos os outros caracteres devem ser uma letra (A..Z), um número (0..9) ou um sinal de sublinhado (_). Todos os caracteres são significativos, inclusive o sinal de sublinhado.
- Um identificador não pode ter mais de 127 caracteres, o comprimento máximo de uma linha em Pascal (é aconselhável utilizar no máximo 10 caracteres).
- O Turbo Pascal não faz diferença entre letras maiúsculas e minúsculas.
- Os identificadores não podem ser palavras reservadas (nomes de comandos, procedimentos do Pascal, etc) – Ver Apêndice C.

DICA: É aconselhável que se defina o nome da variável de forma que este lembre a função assumida pela variável no contexto do programa.

3. Tipos de Dados

Em Pascal, quando definimos uma variável, devemos especificar o seu tipo, isto é, o conjunto de valores que podem ser atribuídos a estas variáveis.

Os tipos de dados podem ser divididos em:
- Tipos predefinidos, isto é, que já fazem parte da própria linguagem;
- Tipos definidos pelo usuário, isto é, que são criados pelo próprio usuário.

Os tipos predefinidos estão divididos em escalares ou simples e estruturados ou compostos.

ESCALARES	ESTRUTURADOS
Integer	Array
Byte	String
Real	Record
Char	File
Boolean	Text
String	Set
Word	Pointer
Shortint	-
Longint	-
Single	-
Double	-
Comp	-
Extended	-

Tabela 2.3 – Tipos escalares e estruturados.

3.1. O Tipo Integer

O tipo integer está representado pelo conjunto dos números inteiros no intervá-lo –32768 a 32767. Um número inteiro ocupa 2 bytes na memória.

Você não poderá atribuir um valor a uma variável inteira fora da faixa definida. Caso isso aconteça, o sistema detecta o erro em tempo de compilação. Entretanto, numa operação matemática do tipo 20000 + 20000 não será detectado nenhum erro em tempo de compilação, mas o resultado da operação 40000 ultrapassa a faixa e o mesmo não será o esperado e sim –25536 (40000 - 65536).

O Pascal dispõe de uma constante predefinida, maxint, que é sempre igual a 32767. Use a mesma em seus programas no lugar de 32767 sempre que desejar referir-se ao maior número inteiro.

Os números inteiros podem ser expressos em notação decimal ou em hexadecimal. Exemplos: 10=$A, 42=$2^{A}, etc.

3.2. O Tipo Byte

O tipo byte é simplesmente um subconjunto dos números inteiros. A faixa coberta pelo tipo byte é de 0 a 255. O tipo byte ocupa apenas 1 byte na memória.

Se o valor de uma expressão ultrapassar 255, o resultado que será atribuído à variável tipo byte será o resto da divisão do número por 256, somado com 256. Exemplos: 200 + 200 + 112 = 0; 200 – 200 = 112.

3.3. O Tipo Real

Número real em Pascal é qualquer número que possua uma parte inteira e uma parte decimal.

Os números reais podem variar de -+ 2.9 x 10^{-39} a -+ 1.7 x 10^{38}, mantendo uma precisão de 11 algarismos significativos. Um número real ocupa 6 bytes na memória.

Um número real pode ser escrito em notação decimal (20.3, 40.34) ou em notação exponencial (2.9E+01, -1.22E-02 = mantissa e expoente; 2.4E03=2,4 x 10^{3}).

Se uma variável receber um valor que ultrapasse 1.7 x 10^{38}, o compilador acusará o erro. Mas se o resultado de uma expressão ultrapassar o referido valor, o programa será abortado. Se o resultado de uma operação for menor ou igual a -+2.9 x 10^{-39}, o sistema assumirá o valor zero.

3.4. O Tipo Char

O computador não dispõe de nenhum meio para guardar letras e outros caracteres não numéricos na memória; ele só pode guardar números. Assim, foram inventados códigos que associam um número diferente a cada caracter. Esses números são guardados na memória do computador no lugar de letras e símbolos. Quase todos os computadores pessoais utilizam um tipo particular de código, o código ASCII ampliado (ASCII trata-se da abreviação de American Standard Code for Information Interchange, ou seja, Código Padrão Americano para Intercâmbio de Informações). Assim, por exemplo, o código ASCII da letra A é 65, para a letra B é 66, e assim por diante. O código da letra a é 97, a letra b é 98, e assim por diante. Pode-se observar que o código da letra minúscula é o código da letra maiúscula correspondente somada ao número 32.

Até o número 127, o código ASCII é padronizado. A partir desse número são representados caracteres gráficos, acentuados, letras gregas, etc. Estes caracteres não são padronizados e podem variar de equipamento para equipamento.

Os primeiros 32 lugares na tabela ASCII, os códigos de 0 a 31, têm um uso especial que nada tem a ver com a aparência dos caracteres mostrados. São usados para passar informações especiais para o computador, ou para outro computador através de linha telefônica ou cabos de rede, comunicação com impressoras, etc.

Código Decimal	Código Hexadecimal	Teclas de Controle	Nome	Descrição	Significado
0	00	^@	NUL	Null character	Caracter nulo
1	01	^A	SOH	Start of header	Início de cabeçalho
2	02	^B	STX	Start of text	Início de texto
3	03	^C	ETX	End of text	Fim de texto
4	04	^D	EOT	End of transmission	Fim de transmissão
5	05	^E	ENQ	Enquire	Caracter de consulta
6	06	^F	ACK	Acknowledge	Confirmação
7	07	^G	BEL	Bell	Alarme ou chamada
8	08	^H	BS	Backspace	Retrocesso
9	09	^I	HT	Horizontal tab	Tabulação horizontal
10	0^A	^J	LF	Line feed	Alimentação de linha
11	0B	^K	VT	Vertical tab	Tabulação vertical
12	0C	^L	FF	Form feed	Alimentação de página
13	0D	^M	CR	Carriage return	Retorno de carro
14	0E	^N	SO	Shift out	Mudança para números
15	0F	^O	SI	Shift in	Mudança para letras
16	10	^P	DLE	Delete	Caracter de supressão
17	11	^Q	DC1	Device control 1	Controle de dispositivo 1
18	12	^R	DC2	Device control 2	Controle de dispositivo 2
19	13	^S	DC3	Device control 3	Controle de dispositivo 3
20	14	^T	DC4	Device control 4	Controle de dispositivo 4
21	15	^U	NAK	Negative acknowledge	Confirmação negada
22	16	^V	SYN	Synchronize	Sincronismo
23	17	^W	ETB	End of text block	Fim de bloco de texto
24	18	^X	CAN	Cancel	Cancelamento
25	19	^Y	EM	End of medium	Fim de meio de dados
26	1^A	^Z	SUB	Substitute	Substituição
27	1B	^[	ESC	Escape	Diferenciação
28	1C	^\	FS	File separator	Separador de arquivo
29	1D	^]	GS	Group separator	Separador de grupo
30	1E	^^	RS	Record separator	Separador de registro
31	1F	^	US	Unit separator	Separador de unidade

Tabela 2.4 - Caracteres de Controle.

Os dados do tipo char fazem parte do conjunto de caracteres ASCII. São usados quando o programa envolve manipulação de caracteres isolados. Ocupam 1 byte na memória.

Os caracteres devem ser colocados entre apóstrofos (‘ ‘), e qualquer caracter poderá ser representado pelo seu código ASCII em decimal, precedido por #, ou em hexadecimal, precedidos por # e $.

Exemplos:

#65 = ‘A’	#27=ESC	#32= espaço em branco
#$20= espaço em branco	#$41=’A’	#$D = #13 = ^M = enter

3.5. O Tipo Boolean

Os dados do tipo boolean só podem assumir um dos dois valores: true (verdadeiro) ou false (falso). O Pascal dispõe de duas constantes pré-declaradas, true e false, para facilitar as operações com dados booleanos. Esses dois valores são definidos de tal forma que false < true. Um dado boolean ocupa 1 byte na memória.

3.6. O Tipo String

Uma string é uma seqüência de caracteres entre dois apóstrofos. Exemplo: ‘eu sou uma string’.

Para representar um sinal de apóstrofo dentro de uma string, basta usar dois apóstrofos seguidos: ‘O nome do cliente é”:’ – ‘O nome do cliente e’: ‘.

A string mais curta é a string vazia (‘’) e seu comprimento é zero. O comprimento máximo de uma string em Pascal é 255.

Ao definir uma variável como string é preciso informar ao computador o número máximo de caracteres que a string pode guardar como, por exemplo, nome: string[30];. O espaço alocado por uma variável string é um byte a mais que o comprimento máximo de caracteres indicado em sua definição de tipo. O byte adicional será usado para guardar o tamanho atual da seqüência de caracteres, na forma de caracter ASCII. Este byte de comprimento está contido no elemento zero da matriz de caracteres. Utilizando a função ORD do Pascal, que retorna a posição de um elemento numa seqüência, podemos obter o comprimento da string:

```
Nome:=’TÉCNICAS DE PROGRAMAÇÃO I’;
Write(ord(nome[S]));     {vai escrever 8}
```

As strings podem ser concatenadas ou unidas. Observe os exemplos:

‘20’+ ‘20’ = ‘2020’	20 + 20 = 40	’20 + 20’ = ’20 + 20’

3.7. O Tipo Word

O tipo Word trata-se de um tipo numérico inteiro, representando números de 0 a 65535. Ocupa 2 bytes na memória. Disponível apenas a partir da versão 4.0 do Turbo Pascal.

3.8. O Tipo Shortint

O tipo Shortint trata-se de um tipo numérico inteiro, representando os números entre –128 até 127. Ocupa 1 byte na memória. Utiliza o bit mais à direita como sinalizador, se 0 positivo, se 1 é somado ao valor e torna-se negativo. Disponível apenas a partir da versão 4.0 do Turbo Pascal.

3.9. O Tipo Longint

O tipo Longint trata-se de um tipo numérico inteiro, representando os números entre –2.147.483.648 até 2.147.483.648. Ocupa 4 bytes na memória. Utiliza o bit mais à direita como sinalizador, se 0 positivo, se 1 é somado ao valor e torna-se negativo. Disponível apenas a partir da versão 4.0 do Turbo Pascal.

3.10. O Tipo Single

O tipo Single trata-se de um tipo numérico real, representando os números entre 1.5 X 10^{-45} até 3.4 X 10^{38}. Ocupa 4 bytes na memória e tem de 7 a 8 dígitos significativos. Disponível apenas a partir da versão 4.0 do Turbo Pascal e seu uso só é permitido com o co-processador presente.

3.11 O Tipo Double

O tipo Double trata-se de um tipo numérico real, representando os números entre 5 X 10^{-324} até 1.7 X 10^{308}. Ocupa 8 bytes na memória e tem entre 15 e 16 dígitos significativos. Seu uso só é permitido com o co-processador 8087 presente.

3.12. O Tipo Comp

O tipo Comp trata-se de um tipo numérico, representado uma faixa de valores que variam entre –9.2 X 10^{18} a 9.2 X 10^{18}. Ocupa 8 bytes de memória e tem entre 19 a 20 dígitos significativos. Só pode ser usado com o co-processador 8087 presente. Apesar de ser do tipo real, só armazena números inteiros e só está disponível a partir da versão 4.0 do Turbo Pascal.

3.13. O Tipo Extended

O tipo Extended trata-se de um tipo numérico real, representando uma faixa de valores entre 3.4 X 10-4932 a 1.1 X 104932. Ocupa 10 bytes na memória e tem entre 19 a 20 dígitos significativos. Somente disponível a partir da versão 4.0 do Turbo Pascal e seu uso só é permitido com o co-processador presente.

4. Comando de Atribuição

Quando definimos uma variável é natural atribuirmos a ela uma informação. Uma das formas de colocar um valor dentro de uma variável, consequentemente colocado este dado na memória do computador, é através da atribuição direta, do valor desejado que a variável armazena. Para isto utilizaremos o símbolo (:= (Pascal) , <- (Algoritmo)), que significa: recebe, ou seja, a posição, de memória que uma variável representa, receberá uma informação, a qual será armazenada no interior desta variável.

Exemplo:

```
Program Teste;
var
   numero: integer;
begin
   numero:=10;
end.
```

O Exemplo acima informa que:

- foi definida uma variável com o nome de “Número” que só pode aceitar dados numéricos entre -32768 a +32767 (tipo INTEGER);
- atribuímos à variável “Número” o valor 10.

A memória se comportaria da seguinte forma, de acordo com os itens acima:

- Variável Conteúdo - Número indefinido;
- Variável Conteúdo - Número 10.

5. Operadores Aritméticos

Muito da manipulação de dados que ocorre na criação dos programas é feita com operadores. Três são as categorias de operadores: aritméticos, lógicos e relacionais.

Chamamos de expressão a qualquer combinação de literais, constantes, identificadores de variáveis, com um ou mais operadores. Se uma expressão só contém operadores aritméticos a mesma é dita expressão aritmética.

Quando mais de um operador aparece numa expressão, a seqüência de cálculo depende da precedência. O operador de mais alta precedência será calculado primeiro. Se dois ou mais operadores tiverem o mesmo nível de precedência, o cálculo processa da esquerda para a direita.

PRECEDÊNCIA	SÍMBOLO	NOME	EXEMPLO
1	-	Menos unário	-50; -n=10, se n=10
2	*	Produto	14 * 2 = 28
2	/	Divisão real	25/5=5.00000000E00
2	Div	Divisão inteira	5 div 2 = 2
2	Mod	Resto da divisão inteira	5 mod 2 = 1
3	+	Adição	5 + 3 = 8
3	-	Subtração	5 – 3 = 2

Tabela 2.5 – Tabela de precedência dos operadores aritméticos.

OBS.: em muitos casos é necessária a utilização dos parênteses para fugir da lei da precedência dos operadores. Exemplo: 5 + 3 * 2 = 11, porém (5 + 3) * 2 = 16.

6. Operadores Lógicos

O Pascal possui quatro operadores lógicos. Três dos quais – AND, OR e XOR – são binários, usados para combinar pares de valores lógicos. O quarto operador – NOT – é unário, pois modifica o operando lógico para o seu valor oposto.

A	B	A and B	A or B	not A	A xor B
False	False	False	False	True	False
False	True	False	True	True	True
True	False	False	True	False	True
True	True	True	True	False	False

PRECEDENCIA	OPERADOR
1	- (menos unário)
2	Not
3	*, /, div, mod, and
4	+, -, or, xor
5	=, <>, <, >, <=, >=, in

Tabela 2.6 - Precedência dos Operadores Lógicos

OBS.: ao utilizar os operadores lógicos juntamente com os relacionais numa mesma expressão, pode ser necessário utilizar parênteses, já que a ordem de precedência de ambos é diferente.

REGRA: sempre que utilizar operadores lógicos em comparações, separe os mesmos com parênteses. Veja o exemplo na página 15.

7. Operadores Relacionais

Uma relação é uma comparação realizada entre valores de mesmo tipo ou tipos compatíveis. Estes valores podem ser constantes, identificadores de variáveis, expressões, etc.

Uma operação relacional compara dois itens de dados e fornece um valor booleano como resultado da comparação.

Logo a seguir encontramos os seis operadores relacionais:

OPERADOR	SIGNIFICADO
=	Igual a, mesma quantidade
>	Maior que, acima de
<	Menor que, abaixo de
>=	Maior ou igual a, a partir de
<=	Menor ou igual a, até
<>	Diferente de

Tabela 2.7 – Operadores relacionais.

Existe um operador relacional (IN) que resulta em um valor booleano. O operador IN trabalha com o tipo set (conjunto) e retorna true se o conteúdo de uma variável pertencer ao conjunto descrito.

Exemplo: Letra in [‘a’, ‘e’, ‘i‘, ‘o’, ‘u’] {retornará true se a letra estiver contida no conjunto}
Em expressões mais elaboradas envolvendo os operadores aritméticos, relacionais e lógicos; a avaliação observa a seguinte precedência:

1. Expressões dentro de parênteses;
2. Operador unário menos ou negação;
3. Operador NOT;
4. Operadores multiplicativos *, /, DIV, MOD e AND;
5. Operadores aditivos +, -, OR e XOR;
6. Operadores relacionais =, <, >, <>, <=, >= e IN.

CAPÍTULO III – CRIANDO OS PRIMEIROS PROGRAMAS

Este capítulo tem por objetivo instruir o leitor na tarefa de criação dos primeiros programas em Pascal.

---

1. Estrutura de um Programa

Um programa em Pascal é dividido em três áreas distintas:
- Área de cabeçalho;
- Área de definições e declarações;
- Área do programa principal - instruções.

De uma forma geral, a área de cabeçalho e a área de definições e declarações são opcionais. A única área realmente necessária é a área de instruções que é feita dentro do bloco principal do programa.

2. Área de Cabeçalho

No Turbo Pascal esta área é opcional. Serve apenas para nomear o programa, como o título em um livro. O cabeçalho deve ser iniciado com a palavra reservada PROGRAM, a seguir um identificador e terminar com ponto e vírgula (;).

Exemplo: *Program teste;*

3. Área de Definições e Declarações

A área das definições e declarações está dividida em seis partes e também é opcional:
- Definição de UNITS;
- Declaração de LABELS;
- Declaração de CONSTANTES;
- Declaração de TIPOS;
- Declaração de VARIÁVEIS;
- Declaração de PROCEDIMENTOS E FUNÇÕES.

3.1. Definição de Units

O objetivo desta área é definir quais são as units utilizadas no seu programa. A UNIT nada mais é que um conjunto de procedimentos e funções que trabalham sob um mesmo objetivo. Por exemplo, a unit Graph possui comandos para criação de desenhos, figuras, barras, entre outros; a unit Crt possui os procedimentos básicos de entrada e saída de dados. Esta área se inicia com a palavra reservada USES.

Exemplo: *uses crt;*

3.2. Declaração de um Label

O objetivo de um label (rótulo) é marcar o destino de uma instrução goto. Na verdade, um programa planejado adequadamente não deverá precisar de labels. No entanto, se você escrever um programa que necessite usar um label, ele deve ser declarado nesta área. Para isso, use a palavra reservada LABEL seguida da lista de rótulos válidos, separados por vírgulas (,) e terminados por um ponto e vírgula (;). Para identificar outros rótulos podem ser usados letras e números.

Exemplo: *label inicio,fim,erro;*

3.3. Declaração de Constantes

Em Turbo Pascal, constantes são utilizadas para associar um valor fixo a um identificador. A área das constantes deve começar com a palavra reservada CONST, logo após um identificador, o sinal de igual, o valor da constante e terminar com ponto e vírgula.

Exemplo: *const mínimo = 130;*

O Turbo Pascal possui algumas constantes pré-declaradas:

CONSTANTE	VALOR
PI	Contém o valor do π: 3.1415926536
MAXINT	Contém o maior valor inteiro: 32767
TRUE, FALSE	Constantes do tipo boolean

3.4. Declaração de Variáveis

Quando se declara uma variável em Turbo Pascal, aloca-se um espaço na memória do computador, fornecendo um lugar para se colocar este dado específico. A declaração não guarda nenhum valor no espaço que foi reservado. O valor da variável fica indefinido até que a mesma receba um valor através da atribuição.

Todas as variáveis devem ser declaradas na área de declaração de variáveis, que começa com a palavra reservada VAR.

Exemplo:
```
var
  salario, inss, liquido: real;
  nome: string;
  idade: byte;
```

Como já foi mencionado, até que lhes atribua um valor, o espaço na memória representado por uma variável conterá "lixo" da memória. Essa atribuição pode ser:
- Feita pelo operador de atribuição :=
- Feita com um procedimento do Pascal para leitura de dados de dispositivos.

Exemplo:
```
taxa_inss := 11/100;  {foi atribuído um valor diretamente}
read(salario);    {observe que aqui o valor está sendo lido, ou seja, solicitado ao usuário}
inss := salario * taxa_inss;  {foi atribuído o resultado de uma expressão}
```

4. Área do Programa Principal

Contém os comandos que definem a ação do programa: atribuição, controle de fluxo, procedures, functions, etc. Começa com a palavra reservada BEGIN e termina com outra palavra reservada END, seguida de um ponto.

Exemplo:
```
Program exemplo;
uses crt;
const desc = 9/100;  {ou seja, 9% -também poderíamos ter escrito 0.09}
var
  prod: string;
  preco: real;
begin
  clrscr; {procedimento para limpar a tela, está na unit CRT}
  write ('Entre com o produto: ');    readln (prod);
  write ('Seu preço unitário: ');     readln (preco);
  writeln;
```

```
   write ('Preço de venda: R$ ', preco*(1-desc):10:2);
   readkey;  {lê um caracter do teclado, um efeito pausa}
end.
```

5. Procedimentos de Entrada e Saída

Os procedimentos de entrada e saída permitem uma maior interação com o usuário, permitindo não só a visualização de informações na tela, bem como a entrada de dados pelo teclado.

Os procedimentos de entrada do Pascal são:READ e READLN, ao passo que os procedimentos de saída são WRITE e WRITELN.

5.1. Procedimento READ

Procedimento para entrada de dados. Seu formato é:
read (dispositivo, lista de variáveis);

COMANDO	EFEITO
read (input,var);	Leitura feita através do dispositivo de entrada padrão (keyboard). Equivale a read(var);. Após o ENTER, o cursor permanece na mesma linha.
read (arq,var);	Sendo arq uma variável do tipo arquivo, a leitura dos dados será feita em disco no arquivo assinalado por **arq**.
readln (var);	Lê a variável **var** e após o ENTER o cursor se desloca para o início da linha seguinte.

Tabela 3.1 - Tabela de exemplos.

Para ler apenas um caracter do teclado, sem pressionar a tecla ENTER, use a função READKEY.

Exemplo:
```
uses crt;
var
   tecla: char;
begin
   tecla := readkey;
   write(tecla);
end.
```

5.2. Procedimento WRITE

Procedimento para saída de dados. Seu formato geral é:
write (dispositivo, lista de variáveis);

COMANDO	EFEITO
write(output,var);	Escrita feita através do dispositivo de saída padrão (vídeo). Equivale a write(var);. Após escrever o conteúdo de var, o cursor permanece na mesma linha.
writeln(var);	Escreve o conteúdo de var e avança o cursor para o início da linha seguinte.
write(lst,var);	Direciona o conteúdo de var para a impressora.
write(arq,var);	Se arq é uma variável do tipo arquivo, a saída é direcionada para o arquivo em disco assinalado por arq.

Tabela 3.2 - Tabela de Exemplos.

O procedimento write aceita parâmetros para a formatação da saída. O formato completo é:
write(var:m,d) onde:

- M é a máscara que determina o tamanho do campo (incluindo ponto e casas decimais). Máximo de 255.
- D é o número de casas decimais (para variáveis do tipo real). Máximo de 24.

*write (1200*10/100);* 1.200000000000E+02 (saída não formatada)
*write (1200*10/100:7:2)* 120.00 (saída formatada)

	1	2	0	.	0	0

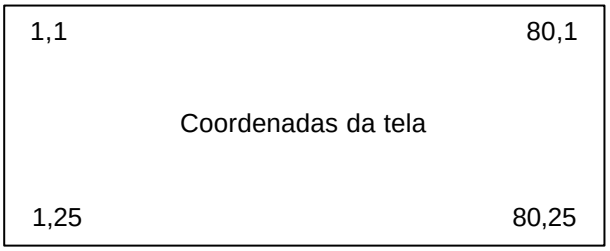
Observações:

- O comando writeln sem parâmetros apenas imprime uma linha em branco e salta para a linha seguinte.
- O comando readln sem parâmetros funciona como um comando de espera dando uma pausa até que o usuário pressione ENTER.

6. Procedimentos de Controle de Vídeo

6.1. Textmode

No modo texto, temos 25 linhas disponíveis com 40 ou 80 colunas.



A coordenada x cresce para a direita e a coordenada y para baixo.

O procedimento que controla o modo texto é o textmode e pode ser chamado dos seguintes modos:

COMANDO	EFEITO
textmode	Limpa o vídeo, mas não muda a seleção atual.
textmode(bw40);	Coloca o vídeo em modo 40x25 em preto e branco. **Bw40** é uma constante inteira predefinida que tem o valor 0.
textmode(c40);	Coloca o vídeo em modo 40x25 a cores. **C40** é uma constante inteira predefinida que tem valor 1.
textmode(bw80);	Coloca o vídeo em modo 80x25 em preto e branco. **Bw80** é uma constante inteira predefinida que tem o valor 2.
textmode(c80);	Coloca o vídeo em modo 80x25 a cores. **C80** é uma constante inteira predefinida que tem o valor 3.

Nos modos colorido de texto, cada caracter pode ter uma das 16 cores possíveis, controladas pelo procedimento TextColor, e também uma das possíveis cores de fundo, controladas pelo procedimento TextBackground.

As cores são numeradas com valores de 0 a 15, porém o Turbo Pascal possui 16 constantes predefinidas correspondentes às cores. Observe a tabela a seguir:

CONSTANTE	COR	CONSTANTE	COR	CONSTANTE	COR
0 – black	Preto	6 - brown	Marrom	12 – lightred	Vermelho claro
1 – blue	Azul	7 - lightgray	Cinza claro	13 – lightmagenta	Magenta claro
2 – green	Verde	8 - darkgray	Cinza escuro	14 – yellow	Amarelo
3 – cyan	Ciano	9 - lightblue	Azul claro	15 – white	Branco
4 – red	Vermelho	10 - lightgreen	Verde claro		
5 – magenta	Magenta	11 – lightcyan	Ciano claro		

Tabela 3.3 – Tabela de cores.

6.2. TextColor

Esse procedimento seleciona a cor que deverá ser usada na representação dos próximos caracteres no vídeo. A sintaxe é:
textcolor(cor);

A cor é um inteiro na faixa [0..15] podendo ser uma das constantes da tabela anterior. Caso se deseje que o caracter fique "piscando", deve-se somar 16 ao valor da cor, ou somar à constante outra constante chamada "blink".

textcolor (2+16); {verde piscando}
textcolor (red + blink); {vermelho piscando}

6.3. Textbackground

Esse procedimento ajusta a cor de fundo a ser utilizada no modo texto colorido. Sua sintaxe:
textbackground(cor);

A cor é um inteiro na faixa [0..7].
textcolor(15); textbackground(0); {letra branca com fundo preto}

6.4. Clrscr

O procedimento CLRSCR (clear screen) limpa o vídeo e coloca o cursor no canto esquerdo do mesmo. Sintaxe:
Clrscr;

6.5. Gotoxy(x,y)

O procedimento gotoxy permite colocar caracteres de texto em qualquer lugar da tela. Seu objetivo é única e exclusivamente posicionar o cursor numa coordenada informada através dos parâmetros X e Y.

Sintaxe: *gotoxy(x,y);*

COMANDO	EFEITO
gotoxy(1,1);	Coloca o cursor no canto superior esquerdo da tela.
gotoxy(40,25);	Coloca o cursor no meio da última linha do vídeo.
gotoxy(80,12);	Coloca o cursor no final da linha do meio do vídeo.

Exemplo:

```
Program calcula_resistência_elétrica;
uses crt;
const titulo = 'CÁLCULO DA RESISTÊNCIA ELÉTRICA';
unidade = #32#234; {espaço junto com o símbolo da resistência, o ohm - W}
var
  corrente, ddp, R: real;
begin
  textmode(c40);
  textcolor(0);
  textbackground(7);
  gotoxy(4,1);
  write(titulo);
  textcolor(15);
  textbackground(0);
  gotoxy(6,7);
  write('U');
  gotoxy(1,8);
  write('R=___=___=');
  gotoxy(6,9);
  write('I');
```

```
   gotoxy(12,7);
   read(ddp);
   gotoxy(12,9);
   read(corrente);
   r := ddp/corrente;
   gotoxy(20,8);
   write(R:7:1, ' '+unidade);
   readkey;
end.
```

CAPÍTULO IV – ESTRUTURAS DE CONDIÇÃO

Este capítulo tem por objetivo orientar o leitor no que diz respeito à tomada de decisão com base nas estruturas de condição do Turbo Pascal.

---

1. Decisão

Uma característica peculiar do computador é a capacidade de tomar decisões. Num programa interativo freqüentemente temos que tomar decisões, escolher entre os percursos de ação disponíveis. O Turbo Pascal fornece duas estruturas de controle para a tomada de decisões: a estrutura IF e a estrutura CASE.

2. A Estrutura IF

Uma estrutura de decisão IF seleciona um entre dois comandos (simples ou compostos) para a execução. A estrutura completa consiste em:

COMANDO SIMPLES	COMANDO COMPOSTO

IF <condição>
THEN
Comando1
ELSE
Comando2;
	IF <condição>
	THEN
	Begin
	Comando1;
	Comando2;
	End
	ELSE
	Begin
	Comando3;
	Comando4;
	End;

Observe que não colocamos um ponto e vírgula após o comando 1 (antes do ELSE). O Pascal interpretaria o ponto e vírgula como o fim do comando IF (uma vez que o ELSE é opcional na estrutura) e a cláusula ELSE não seria reconhecida, resultando um erro em tempo de compilação.

Exemplo:

```
Program e_par_ou_impar;
uses crt;
var
  n: integer;
begin
  clrscr;
  read(n);
  if odd(n) then
        write ('é ímpar')
  else
        write ('é par');
  readkey;
end.
```

A cláusula ELSE da estrutura IF é opcional, então a forma simples é a seguinte:

Exemplo:

```
Program e_par_ou_impar; {o mesmo programa anterior, porém sem ELSE}
uses crt;
var
  n:integer;
  msg: string;
```

```
begin
   clrscr;
   read(n);
   msg := 'é par';
   if odd(n) then
         msg := 'é ímpar';
   write (msg);
   readkey;
end.
```

2.1. Aninhamento de Estruturas IF

As estruturas IF podem estar aninhadas, ou seja, uma dentro da outra. O aninhamento de estruturas pode resultar em seqüências de decisão complexas e poderosas. Veja o exemplo:

Exemplo:

```
Program maior_numero;
uses crt;
var
   a,b,c: integer;
begin
   clrscr;
   write ('Entre com o primeiro número: ') ;     readln(a);
   write ('Entre com o segundo número: ') ;    readln(b);
   write ('Entre com o terceiro número: ') ;     readln(c);
   if (a>b) and (a>c) then
         write ('O maior é: ', a)
   else
      if (b>c) then
         write ('O maior é: ', b)
      else
         write ('O maior é: ', c);
   readkey;
end.
```

Observe que ao utilizar operadores lógicos e relacionais, separamos as comparações com parênteses. Isto evitará erro e tipos misturados (boolean com integer).

> Sem dúvida alguma a endentação é um fator importante no aninhamento, pois facilita a compreensão do que está escrito.
>
> Antes do ELSE não se use ponto e vírgula, lembre-se disso.
>
> Caso tenhamos vários IF's aninhados, a cláusula ELSE, independente de onde estiver, pertence ao último comando IF aberto sem ELSE.

3. A Estrutura CASE

Como a estrutura IF, a CASE divide uma seqüência de possíveis ações em seções de código individual. Para a execução de um determinado comando CASE, somente uma dessas seções será selecionada. A seleção está baseada numa série de testes de comparação, sendo todos executados sobre um valor desejado. A estrutura CASE também é chamada de seleção múltipla.

COMANDO SIMPLES	COMANDO COMPOSTO

CASE valor OF
  Opção1: comando1;
  Opção2: comando2;
  …
  OpçãoN: comando n;
ELSE
  comandos;
END;
	CASE valor OF
	  Opção1:
	  Begin
	  comando1;
	  comando2;
	  End;
	ELSE
	  Begin
	  comandos;
	  End;
	END;

Para executar a decisão nesta estrutura, o Pascal começa por avaliar a expressão de seletor na cláusula CASE OF. Esse valor deve pertencer a um tipo ordinal (real e string não são permitidos) predefinido ou definido pelo usuário.

O seletor se torna alvo das comparações com cada valor subseqüente. Numa execução, o Pascal avalia cada comparação, seleciona o primeiro valor que corresponde com o seletor e executa o(s) comando(s) especificado(s) após os dois pontos. Não será executado mais nenhum outro comando do CASE.

Você pode expressar valor em qualquer uma das formas a seguir:

Uma constante literal ou nomeada:

22 comando;
F comando;

Uma lista de constantes separadas por vírgulas:

'a', 'e', 'i ', 'o', 'u', comando;
2, 4, 6, comando;

uma sub-faixa (subrange) de constantes:

1..10 comando;
'a'..'z' comando;

Exemplo:

```
Program exemplo_case;
uses crt;
var
  tecla: char;
begin
  clrscr;
  write('Entre com um caracter: ');      tecla := readkey;
  case tecla of
       'a'..'z': writeln('O caracter é alfabético minúsculo');
       'A'..'Z': writeln('O caracter é alfabético maiúsculo');
       '0'..'9': writeln('O caracter é numérico');
       #0..#31: writeln('O caracter é um caracter de controle ASCII');
  else
       writeln('O caracter é: ', tecla);
  end;
  readkey;
end.
```

Exemplo:

```
Program meses;
uses crt;
var
  mes:byte;
begin
  clrscr;
  write('Entre com o número do mês: ');    readln(mes);
```

```
  case mes of
        01: writeln('Janeiro');
        02: writeln('Fevereiro');
        03: writeln('Março');
        04: writeln('Abril');
        05: writeln('Maio');
        06: writeln('Junho');
        07: writeln('Julho');
        08: writeln('Agosto');
        09: writeln('Setembro');
        10: writeln('Outubro');
        12: writeln('Novembro');
        12: writeln('Dezembro');
  else
    writeln('Número de mês inválido !');
  end;
  readkey;
end.
```

CAPÍTULO 5 – ESTRUTURAS DE REPETIÇÃO

Este capítulo tem por objetivo instruir o leitor na utilização de estruturas de repetição nos programas feitos em Pascal.

---

1. Introdução

A repetição é a essência de muitas aplicações em computadores. Um loop é uma estrutura de controle que governa os processos de repetição num programa. Como as estruturas de decisão, o loop delimita um bloco de comandos para processamento especial.

O Turbo Pascal fornece três estruturas de repetição diferentes, cada qual com um esquema diferente para o controle do processo de iteração (repetição):

1. O loop FOR que especifica a priori o número de iterações;
2. O loop WHILE que expressa uma condição sobre a qual o loop continua. As iterações terminam quando a condição se torna falsa;
3. O loop REPEAT também contém uma expressão condicional de controle, mas neste caso o loop continua até que a condição se torne verdadeira (inverso do WHILE).

2. Estrutura de Repetição FOR

O loop FOR é, talvez, a estrutura de repetição mais familiar e comumente usada. Pode ser usado com TO, onde a variável de controle assume valores crescentes e com DOWNTO, onde a variável assume valores decrescentes.

COMANDO SIMPLES	COMANDO COMPOSTO

FOR var:=inicio TO fim DO
comando;
 	
FOR var:=inicio DOWNTO fim DO
comando;
	FOR var:=inicio TO fim DO
	begin
	comando1;
	comando2;
	end;

No loop FOR que usa a cláusula TO o valor de início deve ser menor que o valor de fim. Se isso não acontecer o loop não produzirá nenhum efeito. O mesmo é válido para DOWNTO, só que invertido.

Exemplo:

```
Program tabela_raízes_quadrados_e_quadrados;
uses crt;
var
  num: byte;
begin
   clrscr;
   writeln('Número Quadrado Raiz Quadrada');
   for num := 0 to 20 do
         writeln(num:6,' ',sqrt(num):8,' ',sqrt(num):13:2);
   readkey;
end.
```

Exemplo:

```
Program notas_de_alunos;
uses crt;
var
  x, y: byte;
  soma, nota, media: real;
  nome: string;
begin
```

```
    for x := 1 to 10 do
    begin
        write('Entre com o nome do aluno: ');
        readln (nome);
        soma := 0;
        for y:= 1 to 4 do
        begin
              write('Sua nota',y,':');
              readln(nota);
              soma := soma + nota;
        end;  {do for y}
        media := soma / 4;
        writeln ('Sua media é: ',media:6:1);
        readkey;
    end;  {do for x}
end.      {do programa principal}
```

Exemplo:

```
Program alfabeto;
uses crt;
var
   tecla: char;
begin
   clrscr;
   writeln('Alfabeto Completo');
   for tecla:='a' to 'z' do
        write(tecla:8);
   readkey;
end.
```

Ao terminar a execução do loop FOR, a variável de controle permanece com o último valor da seqüência.

2.1. Aninhamento de Estruturas FOR

Em programação de computadores, podemos colocar um loop dentro do outro, para formar loops aninhados. O resultado é um padrão complexo e poderoso de ações repetitivas.

Exemplo:

```
Program tabuada;
uses crt;
var
   a,b: byte;
begin
   clrscr;
   writeln('Tabela de Tabuada de Adição');
   for a:= 1 to 10 do
       write(a:8);
   for a:= 1 to 10 do
           for b := 1 to 10 do          {loops FOR aninhados}
                write('b:2,'+'',a:2,'=',a+b:2);
   readkey;
end.
```

3. Estrutura de Repetição WHILE

Diferentemente do loop FOR, WHILE depende de uma condição expressa para determinar a duração do processo de repetição.

Num loop WHILE a condição vem no início do mesmo e as iterações continuam enquanto a condição é verdadeira. Dizemos que o loop WHILE testa a condição no início se ela for falsa no primeiro teste, não resultará nenhuma iteração.

COMANDO SIMPLES	COMANDO COMPOSTO

WHILE condição DO
comando;
	WHILE condição DO
	Begin
	comandos;
	End;

A condição é uma expressão que o Pascal avalia como true ou false. A repetição continua enquanto a condição for true. Normalmente, em algum ponto, a ação dentro do loop comuta a condição para false e o loop termina. Se a condição não for alterada para false teremos um loop infinito.

Exemplo:

```
Program exemplo_while;
uses crt;
var
  n: byte;
begin
  clrscr;
  n := 0;          {valor inicial de n}
  while n<= 20 do
  begin
        write(n:8);
        n := n + 1;        {contador}
   end;
   readkey;
end.
```

Exemplo:

```
Program notas_alunos_escola;
uses crt;
var
  n: integer;
  soma, nota: real;
begin
  clrscr;
  writeln('Para encerrar a entrada de notas, digite - 1');
  writeln;
  n := 1;
  soma := 0;
  write ('Nota',n,':');    readln(nota);
  while nota <> -1 do
  begin
     soma := soma   + nota;
     n:= n + 1;
     write('Nota',n,':');      readln(nota);
  end;
  if n>1 then
    write('A media de notas é: ',soma/(n-1):8:2)
  else
    write('Nehuma nota digitada');
  readkey;
end.
```

4. Estrutura de Repetição REPEAT UNTIL

A declaração REPEAT faz com que comandos sejam executados repetidas vezes, até que certa condição seja satisfeita. Observe que na forma composta da declaração, as palavras BEGIN e END não são usadas

porque as palavras reservadas REPEAT e UNTIL servem como delimitadores para os comandos dentro do loop.

COMANDO SIMPLES E COMPOSTO

REPEAT
  comando1;
  comando2;
  comandoN;
UNTIL condição;

Como a condição é verificada no final do loop, todo loop REPEAT UNTIL é executado pelo menos uma vez. Se a condição nunca assumir o valor true, o loop só para de ser executado quando o programa é interrompido manualmente (Ctrl+C ou Ctrl+Break). Por esse motivo, dentro do loop deve ter um comando que tome a condição verdadeira pelo menos uma vez.

Exemplo:

```
Program notas_alunos_escola;
uses crt;
var
  n: integer;
  soma, nota: real;
begin
  clrscr;
  n := 1;
  soma := 0;
  writeln('Para encerrar a entrada de notas, digite - 1');
  writeln;
  repeat
       write ('Nota',n,':');          readln(nota);
       soma := soma   + nota;
       n:= n + 1;
   until nota = -1;
   if n>2 then
      write('A media de notas é: ',(soma+1)/(n-2):8:2)
   else
     write('Nehuma nota digitada');
   readkey;
end.
```

Caso você entre num loop sem fim devido a algum erro de programação, pressione Ctrl+Break para abortar a execução do seu programa. O loop sem fim ocorre quando você usa um WHILE ou REPEAT com condições cujo seu valor booleano não se altera, ou quando no FOR, o valor da variável de controle é constantemente alterado e não consegue chegar no seu valor final.

CAPÍTULO VI – FUNÇÕES E PROCEDIMENTOS PREDEFINIDOS

Este capítulo tem por objetivo demonstrar ao leitor a utilização de funções e procedimentos predefinidos da linguagem Pascal.

---

1. Funções e Procedimentos Matemáticos

1.1. EXP(num)

A função EXP devolverá o exponencial de um num, que será e^{num}, onde e é a base dos logaritmos neperianos e vale aproximadamente 2.718282. A função EXP sempre devolverá um valor real. Exemplo: w*rite(exp(2));* {7.3890560989E00, o mesmo que e^2}

1.2. LN(num)

A função logarítmica LN devolverá o logaritmo natural ou neperiano do parâmetro entre parênteses, na forma de um número real. O parâmetro deverá ser maior que zero.
Exemplos: *write(ln(2));* {6.9314718056E-01, o mesmo que $\log_e 2$}
*write(exp()2*ln(5)));* {2.50000000000E+01, o mesmo que 5^2}

1.3. SQR(num)

A função SQR devolverá o quadrado do seu parâmetro, que poderá ser um número inteiro ou real. O tipo do resultado será o mesmo tipo de parâmetro. Exemplo: *write(sqr(9));* {81, o mesmo que 9^2}

1.4. SQRT(num)

A função SQRT devolverá a raiz quadrada do seu parâmetro, que poderá ser integer ou real. Exemplo: *write(sqrt(16));* {4.0000000000E00, o mesmo que $\sqrt{16}$ }

1.5. ABS(num)

A função ABS devolverá o módulo do parâmetro informado, que pode ser inteiro ou real. O tipo do resultado será o mesmo tipo do parâmetro. Exemplo: *write(abs(-150));* {150}

1.6. INT(num)

A função INT devolverá a parte inteira do número informado. O tipo do resultado é sempre real. Exemplo: *write(int(-94.34));* {-9.40000000000E+C1, o mesmo que –94.00}

1.7. TRUNC(num)

A função TRUNC devolverá a parte inteira do número informado, sendo o parâmetro do tipo real, mas o resultado será inteiro. Exemplo: *write(trunc(-94.34));* {-94}

1.8. ROUND(num)

A função ROUND devolverá a parte inteira do número informado, porém fazendo o arredondamento. Exemplo: *write(round(-94.84));* {95} *write(round(1.5);* {2}

1.9. FRAC(num)

A função FRAC devolverá a parte fracionária do número informado. O resultado será sempre real. Exemplo:
write(frac(-2.34)); {-3.40000000000E-01, o mesmo que –0.34}

1.10. SIN(num)

A função SIN devolverá o valor do seno do ângulo em radianos informado. O resultado será real. Exemplo:
write(sin(pi):5:2); {seno π rad = 0.00}
*write(sin(30*pi/180):5:2);* {seno 30^0 = 0.50}

1.11. COS(num)

A função COS devolverá o valor do coseno do ângulo em radianos informado. O resultado será real.
Exemplo: *write(cos(pi):5:2);* {cos π rad = -1.00}
*write(cos(60*pi/180):5:2);* {cos 60^0 = 0.50}

Sen 30^0 = cos 60^0, pois são ângulos complementares (somados resulta em 90^0)

O Pascal não possui a função tan para calcular a tangente, porém podemos chegar neste resultado através de sin(ang)/cos(ang).

Sin e cos trabalham, com ângulos em radianos, porém para trabalhar com ângulos em graus basta multiplicar o valor em graus pela constante PI do Pascal e dividir por 180.

1.12. VAL(atr,num,code)

O procedimento val tentará converter a string informada em NUM valor numérico, seja real ou integer. Caso não seja possível, a variável inteira CODE retornará a posição de erro encontrada.
Exemplo:
val(str_idade,val_idade,code); {exemplo, se str_idade tiver "12^a", code terá 3}
*write('Número de dias: ',val_idade*365);* {se não houver erro, code terá 0}

1.13. STR(num,str)

O procedimento STR converterá o valor numérico informado em uma string.
Exemplo: *str(val_idade,str_idade);*
write('Sua idade: '+str_idade);

2. Funções e Procedimentos Booleanos

2.1. Keypressed

A função Keypressed retornará true caso haja alguma informação no buffer do teclado. Caso contra'rio, retornará false. Exemplo:
repeat until keypressed; {dará uma pausa no programa até que se pressione algo}

2.2. ODD(num)

A função ODD retornará true se o valor informado for ímpar (inteiro). Caso contrário, retornará false.
Exemplo: *if odd(num) then write('é ímpar');*

3. Funções e Procedimentos Ordinais

3.1. INC(num, val)

O procedimento INC irá incrementar o número informado em 1 (um), caso o segundo parâmetro não seja informado. Se o mesmo for informado, o incremento será de seu valor.
Exemplo: *inc(x);* {o mesmo que x:=x+1;}
inc(y,5); {o mesmo que y:=y+5;}

3.2. DEC(num,val)

O procedimento DEC irá decrementar o número informado em 1 (um), caso o segundo parâmetro não seja informado. Se o mesmo for informado, o decremento será de seu valor.
Exemplo: *dec(x);* {o mesmo que x:=x-1;}
dec(y,5); {o mesmo que y:=y-5;}

3.3. SUCC(num)

A função SUCC retornará o sucessor do número informado.
Exemplo: *write(succ(10));* {mostrará 11}

3.4. PRED(num)

A função PRED retornará o antecessor do número informado. Exemplo: *write(pred(10));* {mostrará 9}

3.5. ORD(par)

A função ORD retornará a ordem ou posição do parâmetro informado na lista ou conjunto que ele faz parte. Esse parâmetro pode ser um elemento de um tipo definido pelo usuário, ou um número, ou um caracter. Neste caso, mostrar a ordem de um caracter é mostrar seu código ASCII em decimal.
Exemplo:

```
write(ord('a'),#32,ord('A'));        {mostrará 97 65, os respectivos códigos ASCII}
uses crt;
var
   i: char;
begin
  clrscr;
  for I:=#0 to #255 do
        write('Cord: ',ord(i):3,'Carac: ',I,' ');
  readkey;
end.
```

4. Funções e Procedimentos de Caracter

4.1. UPCASE(char)

A função UPCASE retornará o caracter informado em formato maiúsculo. Caso o caracter informado não seja alfabético, o mesmo será retornado como foi informado. Exemplo: *write(upcase('a'));* {mostrará A}

{Obs.: para converter um carácter em maiúsculo, basta diminuir 32 de seu código ASCII, veja:}

```
write(letra);      {'a'}
dec(letra,32);
write(letra);      {'A'}
```

4.2. CHR(code)

A função CHR retornará o caracter correspondente ao código ASCII informado.
Exemplo: *write(chr(97));* {mostrará 'a'}

4.3. READKEY

A função READKEY retornará o caracter (tipo char) digitado pelo usuário, sem mostrá-lo na tela. Esta função provoca uma parada na execução do programa, esperando até que o usuário pressione uma tecla.
Exemplo: *write(readkey);* {após o usuário digitar, a informação irá aparecer na tela}

5. Funções e Procedimentos de String

5.1. INSERT(str,str_destino,pos_inicio)

O procedimento INSERT irá inserir a string de primeiro parâmetro na string de segundo parâmetro, a partir da posição informada no terceiro parâmetro. Somente o segundo parâmetro será alterado. Exemplo:

```
msg := 'O Brasil foi penta!';
adic := 'não'#32;
insert(adic,msg,10);
write(msg);      {'O Brasil não foi penta!'}
```

5.2. DELETE(str,pos_inicio,quant)

O procedimento DELETE irá eliminar uma string interna a outra string. O primeiro parâmetro é a string, o segundo é a posição inicial e o terceiro e último a quantidade de caracteres a remover. Somente o primeiro parâmetro será alterado. Exemplo:

```
msg := 'O Brasil não foi penta!';
delete(msg,10,4);
write(msg);      {'O Brasil foi penta!'}
```

5.3. CONCAT(str1,str2,...,strN)

A função CONCAT retornará a concatenação ou união de todas as strings informadas como parâmetros. O resultado da concatenação não pode ultrapassar os 255 caracteres permitidos. Utilize o operador "+" ao invés de CONCAT, pois "+" é mais veloz. Exemplo:

```
msg1:='Linguagem';
msg2:=#32;
msg3:='Pascal';
write(concat(msg1,msg2,msg3));      {'Linguagem Pascal'}
```

5.4. COPY(str,pos_inicio,quant)

A função COPY retornará uma substring a partir de uma string original. A string original é o primeiro parâmetro, a posição inicial de cópia é o segundo e a quantidade de caracteres a serem copiados é o terceiro. Exemplo:

```
msg := 'A Linguagem Pascal';
write(copy(msg,13,6));      {Pascal}
```

5.5. LENGTH(str)

A função LENGTH retornará o comprimento da string, ou seja, seu número de caracteres. Exemplo:

```
msg := 'A Linguagem Pascal';
write(length(msg),#32,pos(msg[0]));      {1919 = são commandos equivalentes}
```

5.6. POS(str_proc,str)

A função POS retornará a posição da string de primeiro parâmetro na string de segundo parâmetro. Caso a mesma não exista, será retornado o valor 0 (zero). Exemplo:

```
msg := 'A Linguagem Pascal';
a := 'Delphi';
b := 'Pascal';
write(pos(a,msg),#32,pos(b,msg));        {0 13}
```

6. Funções e Procedimentos Diversos

6.1. CLREOL

O procedimento CLREOL limpará parte da tela apenas da posição do cursor até o final da mesma linha. Exemplo:

```
for i := 1 to 10 do
  begin
    write('Digite a nota: ');
    clreol;
    read(nota);
  end;
```

6.2. DELLINE

O procedimento DELLINE limpará toda a linha onde estiver o cursor, sem movê-lo. Exemplo:

```
gotoxy();
delline;
```

6.3. INSLINE

O procedimento INSLINE irá inserir uma linha onde estiver o cursor, sem movê-lo. Todos os dados que estiverem abaixo da posição do cursor serão deslocados para baixo. Caso se queira inserir mais de uma linha, basta colocar a procedure num loop for. Exemplo:

```
for x := 1 to 10 do insline;        {irá inserir10 linhas a partir da posição do cursor}
```

6.4. SOUND (freq)

O procedimento SOUND irá ativar o auto-falante do micro, bastando apenas informar no parâmetro a freqüência do som desejado. Normalmente utiliza-se esse procedimento com os procedimentos delay e nosound, já que sound sozinho irá ativar o som e o mesmo ficará tocando o tempo todo. Exemplo:

```
sound(200);
```

6.5. NOSOUND

O procedimento NOSOUND irá desligar o som do auto-falante do micro. Exemplo:

```
nosound;
```

6.6. DELAY(ms)

O procedimento DELAY irá dar uma pausa na execução do programa, só que diferentemente dos procedimentos já citados, essa pausa se dará por um tempo informado no parâmetro em milisegundos (10^{-3}s). Exemplo:

```
sound(240);
delay(1000);     {equivale a 1s}
nosound;
```

OBS.: O tempo em milisegundos não é exato e sim aproximado.

6.7. RANDOM(num)

A função RANDOM retornará um valor randômico ou aleatório a partir do número informado. Esse número randômico será um número qualquer na faixa de 0<= α < num. Caso não seja informado um número, o valor randômico estará na faixa 0<= α < 1. Normalmente esta função é utilizada com o procedimento randomize. Exemplo:

```
random(300);    {através de fórmulas internas, será gerado um número entre 0 e 300}
```

6.8. RANDOMIZE

O procedimento RANDOMIZE irá fazer apenas um pequeno diferencial no uso da função RANDOM. Toda vez que se utiliza RANDOM, o valor inicial das fórmulas é o mesmo. Portanto, os valores randômicos sempre serão os mesmos. Para que isto não aconteça, basta utilizar o procedimento RANDOMIZE, o mesmo irá fazer com que o valor inicial das fórmulas de randomização seja baseado na hora do sistema, o que sabemos que muda constantemente. Exemplo:

```
randomize;
repeat
   write(random(500):8);
   delay(50);
until keypressed;
```

6.9. WHEREX

A função WHEREX retornará a posição horizontal (abscissa) do cursor na tela (valor coluna). Exemplo:

```
x := wherex;
y := wherey;
gotoxy(x+10,y);
write('teste');
```

6.10. WHEREY

A função WHEREY retornará a posição vertical (ordenada) do cursor na tela (valor linha). Exemplo:

```
x := wherex;
y := wherey;
gotoxy(x,y+1);
write('teste');
```

6.11. EXIT

O procedimento EXIT irá fazer com que a execução do programa saia do procedimento atua e vá para o seu chamador. Caso o procedimento atual seja o programa principal, exit terminará a execução do programa.
Exemplo 1: `if tecla=#27 then exit;` { Pressionou ESC, fim }
Exemplo 2:

```
uses crt,graph;
var
  i,driver,modo,X1,X2,Y1,Y2: integer;
begin
    detectGraph(Driver,Modo);
    initGraph(Driver,Modo,'d:\tpascal7\bgi');
    repeat
        x1 := random(getMaxX);
        y1 := random(getMaxY);
        x2 :=  random(getMaxX-X1)+X1;
        y2 := random(getMaxY-Y1)+Y1;
        setColor(random(15));
        rectangle(X1,Y1,X2,Y2);
    until keypressed;
    closeGraph;
end.
```

CAPÍTULO VII – UNITS

Este capítulo tem por objetivo orientar o leitor sobre as units disponíveis no Turbo Pascal.

---

1. Definição

As UNITS são rotinas compiladas separadamente do programa principal. O conceito de UNIT foi incorporado ao Turbo Pascal a aprtir da versão 4.0 e as que se encontram disponíveis atualmente são:

1.1. System

Esta é a única unidade que pode ser utilizada sem ser citada. Nela está contida a maior parte das rotinas padrão do Pascal.

1.2. Crt

Nesta unidade está contida a maioria das rotinas e variáveis de vídeo e de geração de som, porém, além destas rotinas estão disponíveis algumas constantes e variáveis.

1.3. Dos

Esta unidade é responsável pela execução das rotinas que envolvem o sistema operacional - DOS. Com ela torna-se fácil a utilização da maioria dos procedimentosa e controles de baixo nível.

1.4. Graph

Esta unidade é responsável pelo suporte gráfico do Turbo. É uma biblioteca gráfica com mais de 50 rotinas, para a utilização de alguns arquivos externos, sendo estes arquivos gráficos (.BGI) ou fontes (.CHR). No modo gráfico deixamos de ter linhas e colunas e passamos a ter pontos nas coordenadas X e Y, que podem ter diversas combinações de acordo com o hardaware. Possui uma grande quantidade de rotinas gráficas bastante poderosas, tem suporte para diversos tipos de vídeo e é bastante rápida.

1.5. Printer

Esta unidade tem apenas a função de definir LST como sendo um arquivo texto, já direcionado para a impressora. Operação que pode ser feita pelo usuário sem muito esforço.

1.6. Overlay

Nesta unidade, podemos gerenciar as atividades de overlays de um programa. O processo de overlay nada mais é do que se aproveitar uma mesma área de memória para várias rotinas diferentes. Com este procedimento podemos diminuir consideravelmente a memória requerida para execução de um programa. As partes geradas pelo Turbo Pascal como overlay têm extensão .OVR. Em suma, esta unidade permite a geração de unidades para ocuparem um mesmo espaço em memória. Só está disponível a partir da versão 5.0 do Turbo.

Para a utilização de uma ou mais unidades, é necessário o uso da declaração USES <nome_unidade>, com exceção apenas para a Unit SYSTEM.

2. Criando Units

As units foram criadas para facilitar a construção de programas, pois eliminam a necessidade de duplicação de código, uma vez que blocos de comandos usados repetidamente podem ser transformados em sub-rotinas.

Este conceito se aplica muito bem para apenas um algoritmo / programa, mas imagine que você necessite elaborar dois sistemas: Um para cadastro de clientes de uma loja qualquer e outro para o cadastro de alunos de um colégio. Os dois sistemas serão totalmente diferentes na função que realizam, mas poderão ter sub-rotinas idênticas (genéricas) como, por exemplo, sub-rotinas que manipulam a tela, sub-rotinas para programar a impressora, sub-rotinas para armazenar e recuperar informações no disco, sub-rotinas para gerenciar memória do computador etc.

Pelo conhecimento visto até agora, quando da construção destes sistemas, ou outros no futuro, seria necessário repetir a digitação destas mesmas sub-rotinas tantas vezes quantos forem os sistemas a serem construídos.

Através do uso de sub-rotinas dentro dos programas, podemos compartilhar blocos de comandos, facilitando assim a construção de um sistema. Mas quando se trata de elaborar vários sistemas, o uso de sub-rotinas não é o bastante, pois precisam também compartilhar sub-rotinas genéricas entre sistemas diferentes.

Pensando nisto, foi criado um novo conceito de programação, onde podemos construir um tipo especial de programa onde são definidos não apenas sub-rotinas, mas também variáveis, constantes e tipos de dados que podem ser usados não apenas por um programa, mas sim por diversos programas diferentes. A este exemplo de programação deu-se o nome de programação modular e a este programa especial deu-se o nome de módulo.

O Pascal dá a este modo o nome de UNIT e a sintaxe para a sua construção é a seguinte:

UNIT <Nome da Unit>;
INTERFACE
USES <lista de UNITs importadas>
< definição de variáveis, constantes e tipos exportados>
<cabeçalho das sub-Rotinas exportadas>
IMPLEMENTATION
USES <lista de UNITs importadas privativas ao módulo>
<definição de variáveis, constantes e tipos internos a UNIT>
<sub-Rotinas internas a UNITs>
<corpo das sub-Rotinas exportadas>
BEGIN
<Comandos a serem executados na ativação da Unit>
END.

Observações:

1 - A seção conhecida por INTERFACE define todas as sub-rotinas, variáveis, constantes e tipos de dados que são exportados, ou sejam, são visíveis em outros programas.

2 - Os tipos de dados, variáveis e constantes definidos na seção IMPLEMENTATION serão visíveis somente dentro da UNIT, portanto não sendo exportados.

3 - As sub-rotinas definidas na seção IMPLEMENTATION e que não tenham o seu cabeçalho definido na seção INTERFACE serão internas a UNIT, não sendo desta forma exportadas.

4 - Para usar as sub-rotinas, variáveis, constantes e tipos de dados definidos em outras UNITs basta utilizar a palavra reservada USES seguido da relação de nomes das UNITs desejadas.

Exemplo: Construir uma UNIT que contenha uma sub-rotina para escrever uma STRING qualquer em uma determinada linha e coluna na tela do computador.

```
UNIT tela;
INTERFACE
PROCEDURE Escreve_Str( linha, coluna : BYTE; Texto : STRING);
IMPLEMENTATION
USES CRT;
PROCEDURE Escreve_Str( linha, coluna : BYTE; texto : STRING);
BEGIN
GOTOXY(coluna, linha);
WRITE(texto);
END;
END.
```

Como complementação do exemplo anterior, vamos construir um pequeno programa que use a sub-r*otina* definida acima:

```
Program Testa_Unit;
uses Tela;
begin
   Escreve_Str(10, 10, 'Teste de Unit');
end;
```

CAPÍTULO VIII – A UNIT CRT

Este capítulo tem por objetivo orientar o leitor na utilização da unit CRT do Turbo Pascal.

---

1. Unit CRT

A unit CRT como já foi dito em capítulos anteriores possui rotinas que favorecem as tarefas de manipulação de vídeo e de geração de som, além de disponibilizar algumas constantes e variáveis. Algumas das principais rotinas desta unit já foram apresentadas no Capítulo VI. Vejamos então algumas outras.

1.1. ASSIGNCRT

Este procedimento permite a associação de um arquivo do tipo texto com a console de saída. No caso de vídeo, esta forma de saída ou eventualmente de entrada, possibilita uma maior velocidade em relação às operações de entrada e/ou saída standards de vídeo. Este procedimento é bastante similar ao ASSIGN. Sintaxe: *assigncrt(var <arq>: text);*

1.2. WINDOW

Este procedimento permite que se defina a área útil do vídeo que será utilizada. O default é a partir da coluna 1 e linha 1 até a coluna 80 e linha 25, para modo de 80 colunas, e coluna 1 e linha 1 a coluna 40 e linha 25, para o modo de 40 colunas. As funções e procedimentos utilizados após este comando, terão como coordenadas as posições definidas pela nova janela. Sintaxe: *window(<x1>,<y1>,<x2>,<y2>: byte);* onde x1,y1 representam as coordenadas coluna e linha inicial e x2,y2, as coordenadas coluna e linha final. No caso das variáveis *x* e *y* estarem fora dos parâmetros válidos, o procedimento simplesmente não será executado e nenhuma mensagem de erro será exibida.

1.3. HIGHVIDEO

Este procedimento seleciona a cor de texto para alta intensidade. Sintaxe: *highvideo;*

1.4. LOWVIDEO

Este procedimento seleciona a cor de texto para baixa intensidade. Sintaxe: low*video;*

1.5. NORMVIDEO

Este procedimento permite que se retorne a mesma cor para o texto, da posição do cursor no momento em que foi carregado o programa. Sintaxe: *normvideo;*
Exemplo:

```
Program vídeo;
uses crt;
begin
  textcolor(15);
  textbackground(0);
  writeln('Texto em alta intensidade pelo TEXTCOLOR');
  lowvideo;
  writeln('Texto em baixa intensidade pelo LOWVIDEO');
  highvideo;
  writeln('Texto em alta intensidade pelo HIGHVIDEO');
  normvideo;
  writeln('Texto em modo normal NORMVIDEO');
  readkey;
end.
```

CAPÍTULO IX – A UNIT DOS

Este capítulo tem por objetivo orientar o leitor na utilização da unit DOS do Turbo Pascal.

---

1. Unit DOS

No capítulo anterior abordamos a unit CRT, que é a unit que possui os principais procedimentos de entrada e saída de dados em seus programas. Porém, conheceremos agora alguns procedimentos da unit DOS, a qual se dedica a operações relacionadas com o sistema operacional do computador hora, data, arquivos, disco, etc.

2. Funções e Procedimentos de Data e Hora

2.1. GETDATE (ano,mês,dia,semana)

O procedimento GETDATE irá armazenar nas variáveis-parâmetros informadas os respectivos dados da data atual do sistema. É importante ressaltar que todos os parâmetros devem ser declarados como do tipo Word, pois o procedimento exige essa declaração para seu perfeito funcionamento (ver tabela no final da página).

2.2. GETTIME(hora,min,s,cent_s)

O procedimento GETTIME irá armazenar nas variáveis-parâmetros informadas, os respectivos dados da hora atual do sistema. É importante ressaltar que todos os parâmetros devem ser declarados como do tipo Word, pois o procedimento exige essa declaração para seu perfeito funcionamento (ver tabela no final da página).

Exemplo:

```
uses crt, dos;
const dias_semana: array[0..6] of string = ('Domingo','Segunda','Terça'
                                            'Quarta','Quinta','Sexta','Sábado');
meses: array[0..11] os string = ('Janeiro','Fevereiro','Março','Ábril','Maio','Junho','Julho',
                                 'Agosto','Setembro','Outubro','Novembro','Dezembro');
var
   ano, mes, dia, semana, hora, min, s, s100: word;
begin
  clrscr;
  getdate(ano,mes,dia,semana);
  writeln(dias_semana[semana],',',dia,','de',meses[mes-1],'de',ano);
  repeat
    gettime(hora,min,s,s100);    {hora, minuto, Segundo e centésimos de segundo}
    gotoxy(1,20);
    write(hora,'h',min,'min',s,'s',s100);
  until keypressed;
end.
```

TIPO	FAIXA	SINAL	TAMANHO
Shortint	-128..127 $-(2^7) <= n < (2^7)$	com	8 bits
Integer	-32768..32767 $(2^{15}) <= n < (2^{15})$	com	16 bits
Longint	-2147483648..2147483647 $(2^{31}) <= n < (2^{31})$	com	32 bits
Byte	0..255 $0 <= n < (2^8)$	sem	8 bits
Word	0..65535 $0 <= n < (2^{16})$	sem	16 bits

Tabela 8.1 - Tipos inteiros.

É importante ressaltar que as faixas cabíveis de getdate são: ano (1980..2099), mês (1..12), dia (1..31) e dia da semana (0..6). E as faixas de gettime: hora (0..23), minuto (0..59), segundo (0..59) e centésimos de segundo (0..9).

2.3. SETDATE (ano,mes,dia)

O procedimento SETDATE irá determinar qual será a nova data atual do sistema. Basta informar os valores desejados. Exemplo: *setdate(1999,03,01);* {data atual é 01/03/1999}

2.4. SETTIME (hora,min,s,cent_s)

O procedimento SETTIME irá determinar qual será a nova hora atual do sistema. Basta informar os valores desejados. Exemplo: *settime(13,30,0,0);* {a hora atual é 13:30}

2.5. PACKTIME (VAR <dt>: DATETIME; VAR <ftime>: LONGINT)

O procedimento PACKTIME permite compactar data e hora em 4 bytes. A data e a hora deverão estar em um registro especial, DATETIME predefinido na UNIT DOS da seguinte forma:

```
datetime = record
   year,month,day,hour,min,sec: word;
end;
```

Este procedimento normalmente é utilizado quando da necessidade de atualizar a data e a hora de criação de um arquivo, podendo também ser utilizado para outros fins.

Exemplo:

```
Program teste_packtime;
uses dos;
var
  ftime: longint;
  dt: datetime;
begin
   with dt do
      begin
         write(Digite o dia: ); readln(day);
         write(Digite o mês: ); readln(month);
         write(Digite o ano: ); readln(year);
         write(Digite a hora: ); readln(hour);
         write(Digite o minuto: ); readln(min);
         write(Digite o segundo: ); readln(sec);
          packtime(dt,ftime;)
          writeln('Data compactada: ',ftime:0);
     end;
readkey;
end.
```

2.6. UNPACKTIME (<ftime>: LONGINT; VAR <dt>: DATETIME)

Este procedimento permite descompactar um valor em 4 bytes para o registro DATETIME do procedimento anterior. É utilizado normalmente para exibir a data e hora de última gravação de um arquivo, podendo ter putros fins também.

Exemplo:

```
Program teste_unpacktime;
uses dos;
var
   ftime: longint;
   dt: datetime;
begin
```

```
  with dt do
    begin
      write('Digite o dia: '); readln(day);
      write('Digite o mês: '); readln(month);
      write('Digite o ano: '); readln(year);
      write('Digite a hora: '); readln(hour);
      write('Digite o minuto: '); readln(min);
      write('Digite o segundo: '); readln(sec);
      packtime(dt,ftime;)
      ftime:=ftme+10000000
      unpacktime(ftime,dt);
      write('O novo dia: ',day);
      write('O novo mês: ',month);
      write('O novo ano: ',year);
      write('Digite a hora: ',hour);
      write('Digite o minuto: ',min);
      write('Digite o segundo: ',sec);
    end;
readkey;
end.
```

2.7. GETFTIME (VAR <arquivo>; VAR <dh>:LONGINT)

Este procedimento retorna ahora e a data da última atualização de um arquivo, que por sua vez, deverá ser associado a seu nome externo e estar aberto. O argumento retornado está compactado, ou seja,. Hora e data em uma única variável. Exemplo: *getftime(f,ftime);*

2.8. SETFTIME (VAR <arquivo>; VAR <ftime>:LONGINT)

Este procedimento permite estabelecer uma nova data de última gravação que deverá estar previamente compactada em 4 bytes. Exemplo: *setftime(arq,nftime);*

Observação: é importante frisar que as funções F+GETFTIME e SETFTIME não fazem a consistência das datas e horas atribuídas a eles, portanto cabe ao programador fazer a verificação da validade das mesmas.

3. Funções e Procedimentos de Disco

3.1. DISKFREE (drive)

A função DISKFREE retornará a quantidade de bytes que se encontra livre no drive-parâmetro especificado. Especifique como parâmetro 0 (zero) para o drive padrão, 1 (um) para o drive A, 2 (dois) para o drive B, 3 (três) para o drive C, e assim sucessivamente. Exemplo:

```
writeln('Espaço livre no disco C: ',diskfree(3) div (1024*1024),'Mbytes');
writeln('Espaço livre no disco D: ',diskfree(4) div (1024*1024),'Mbytes');
```

3.2. DISKSIZE(drive)

A função DISKSIZE retornará a quantidade de bytes total que o drive-parâmetro pode armazenar. Segue as mesmas regras do diskfree. Exemplo:

```
writeln('Espaço total no disco default ou padrão: ',disksize(0) div 1024,'Kbytes');
```

3.3. GETVERIFY(bol)

O procedimento GETVERIFY irá armazenar no parâmetro booleano o valor true caso a verificação de escrita no disco esteja ativa. Caso contrário, armazenará false. Exemplo:

```
getverify(grava);
write('Verificação de escrita está: ',grava);
```

3.4. SETVERIFY(bol)

O procedimento setverify irá determinar a verificação de gravação no disco. Caso o parâmetro booleano seja true a verificação de escrita no disco será ativa. Caso contrário (false), será desligada. Exemplo:

```
grava := true;
setverify(grava);          {verificação ligada}
```

4. Funções e Procedimentos Diversos

4.1. DOSVERSION

A função DOSVERSION retornará o número da versão do sistema operacional em uso. O valor de retorno é do tipo Word, porém o mesmo valor é composto de duas partes: uma antes do ponto e outra depois. Para evitar cálculos, usamos as funções lo e hi para obtermos ambos os valores separadamente, já que os mesmos estão no formato binário. Lo obtém a parte baixa do Word (os últimos oito bits) e hi obtém a parte alta (os primeiros oito bits). Exemplo:

```
ver := dosversion;
writeln('Versão do Sistema Operacional: ',lo(ver),'.',hi(ver));
```

4.2. GETCBREAK(bol)

O procedimento GETCBREAK irá armazenar no parâmetro booleano, o valor true, caso a verificação de Ctrl+Break esteja ativa. Caso contrário, armazenará false. Estando true significa que o Ctrl+Break será verificado em todas as chamadas do sistema. Estando false significa que o mesmo só será verificado nas operações de entrada/saída, impressora e comunicação entre dispositivos. Exemplo:

```
getcbreak(grava);
write('Verificação Ctrl + Break está: ',grava);
```

4.3. SETCBREAK(bol)

O procedimento SETCBREAK irá determinar a verificação de Ctrl+Break. Caso o parâmetro booleano seja true, a verificação de Ctrl+Break será ativa. Caso contrário (false), será dsligada. Exemplo:

```
grava := false;
setcbreak(grava);          {verificação desligada}
```

4.4. ENVCOUNT

A função ENVCOUNT retornará o número de variáveis de ambiente existentes no sistema operacional.

4.5. ENVSTR(ind)

A função ENVSTR retornará a string da variável ambiente e seu conteúdo, utilizando o índice-parâmetro que será informado. Exemplo:

```
for i:= 1 to envcount do
   writeln(envstr(i));     {isto irá escrever na tela todas as variáveis ambiente}
```

4.6. GETENV(str)

A função getenv retornará o conteúdo da variável ambiente, utilizando a string-parâmetro que será informado (que é o nome da variável ambiente). Exemplo:

```
writeln(getenv('COMSPEC'));          {mostrará C:\WINDOWS\COMMAND.COM}
```

4.7. EXEC

A função EXEC executa um comando passado como parâmetro. O programa deve ter seu caminho completo inclusive a extensão do arquivo. Caso seja necessário passar parâmetros, estes devem ser informados como segundo argumento. Sintaxe: *exec(caminho,linhadecomando: string);*

4.8. DOSEXITCODE

Esta função retorna o código de saída de um subprocesso. Se o resultado retornado for 0, indica um término regular, se for 1, indica terminado por ^C, se for 2, houve umerro e caso tenha sido 3, indica término de um programa residente. Sintaxe: *dosexitcode: word;*

4.9. FEXPAND

Esta função retorna uma string, contendo o nome do programa passado como parâmetro em sua totalidade.
Sintaxe: *fexpand(path:PATHSTR): PATHSTR;*
Observação: o tipo PATHSTR corresponde ao mesmo que string[79].

4.10. FSEARCH

Esta função permite procurar um determinado arquivo em uma lista de diretórios. A lista de diretórios pode ser separada por ponto-e-vírgula (;). Se o arquivo especificado não for encontrado, a função retornará uma string vazia. A pesquisa é feita de qualquer forma, a partir do diretório corrente.
Sintaxe: fsearch(path:PATHSTR;lista:STRING):PATHSTR;

4.11. FSPLIT

Este procedimento explode o nome do arquivo, passando como parâmetro, em três partes, o diretório, o nome do arquivo e a extensão.
Sintaxe: *fsplit (path: PATHSTR; VAR dir: DIRSTR; VAR nome:NAMESTR; VAR ext: EXTSTR);*

Observação: Os tipos PATHSTR, DIRSTR, NAMESTR e EXTSTR estão definidos na unit DOS da seguinte forma: *TYPE*

PATHSTR: string[79];
DIRSTR: string[67];
NAMESTR: string[08];
EXTSTR: string[04];

4.12. FINDFIRST

Este procedimento procura um determinado arquivo, especificando seu nome e seu atributo, em um diretório específico ou no diretório corrente. São passados como parâmetros três argumentos, o path, o atributo do arquivo e uma variável predefinida do tipo Searchrec. Os possíveis erros são reportados na variável DOSERROR, que retornará 2 para diretório não existente e 18 quando não existirem mais arquivos.
Sintaxe: *findfirst (<caminho>: string; <atributo>: word; var <s>:searchrec);*

4.13. FINDNEXT

Este procedimento devolve a próxima entrada, de acordo com o nome de arquivo e também o atributo utilizado previamente na chamada do procedimento. Os possíveis erros são reportados na variável DOSERROR e esta retornará 18 indicando que não existem mais arquivos.
Sintaxe: *findnext (var <s>: searchrec);*

4.14. GETFATTR

Este procedimento retorna o atributo de um determinado arquivo. Estes atributos são estabelecidos pelo sistema operacional. Sintaxe: *getfattr (<arq>; VAR <attr>: word);*

4.15. SETFATTR

Este procedimento permite estabelecer um novo atributo para um arquivo.Os erros possíveis são reportados na função DOSERROR e podem apresentar os seguintes códigos: 3 (path inválido) e 5 (arquivo protegido). Este procedimento não pode ser utilizado em um arquivo aberto. Sintaxe: *setfattr (VAR <arq>; <attr>: word);*

4.16. GETINTVEC

Este procedimento retorna o endereço de memória de um vetor de interrupção específico, com o intuito de salvá-lo visando uma futura utilização. Sintaxe: *getintvec (<nuint>: byte; VAR <vetor>: POINTER);*

4.17. SETINTVEC

Este procedimento determina um novo endereço para um vetor de interrupção específico. O número da interrupção pode ser de 0 a 255. Sintaxe: *setintvec (<nuint>:BYTE; VAR <vetor>:POINTER);*

4.18. SWAPVECTORS

Este procedimento permite que sejam salvos todos os vetores de interrupção do sistema. Normalmente é utilizado em conjunto com o procedimento EXEC e salva os ponteiros dos vetores de interrupção, evitando assim que uma interrupção tenha seu endereço trocado durante a chamada de um outro programa, não interfirindo assim no programa principal e vice-versa. Sintaxe: *swapvectors;*

4.19. INTR

Este procedimento executa uma interrupção do sistema operacional. Para que possa ser utilizado corretamente, deve-se passar como parâmetro o número da interrupção e um registrador.
Sintaxe: *INTR (<intnro>: BYTE; VAR <reg>: REGISTERS);*

4.20. MSDOS

Este procedimento executa uma chamada na interrupção 21H do sistema operacional. Para tanto, deve ser passado como parâmetro o registrador. Sintaxe: *MSDOS(VAR <reg>: REGISTERS);*

4.21. KEEP

Este procedimento permite terminar um programa e mantê-lo residente em memória. Para que seja executado com sucesso, deve ser reservada uma área de memória com o uso da diretiva $M.
Sintaxe: *KEEP (<saida>: WORD);*

CAPÍTULO X – A UNIT GRAPH

Este capítulo tem por objetivo orientar o leitor na utilização da unit GRAPH do Turbo Pascal.

---

1. Unit GRAPH

Esta unidade é reponsável pelo suporte gráfico do Turbo Pascal. É uma biblioteca gráfica com mais de 50 rotinas para a utilização de alguns arquivos externos como drivers gráficos (.BGI) ou fontes de caracteres (.CHR). No modo gráfico, deixam de existir linhas e colunas e passam a existir pontos de coordenadas x e y, que por sua vez, podem ter diversas combinações de acordo com o hardware.

2. Procedimentos Diversos

2.1. DETECTGRAPH

Este procedimento permite detectar o hardware e o modo gráfico atuais. Estas informações são retornadas em duas variáveis do tipo integer. Tal procedimento é muito útil quando não se conhece o ambiente gráfico. Sintaxe: *detectgraph(var <grafdrv>,<grafmod>:integer);*

2.2. INITGRAPH

Este procedimento faz a inicialização gráfica para que sejam possíveis todas as operações gráficas. São passados como parâmetros o driver, o modo gráfico e o path do driver.
Sintaxe: *initgraph(var <grafdrv>:integer; var <grafmod>: integer; <pathdodrv>:string);*

2.3. GETDRIVERNAME

Esta função retorna uma string contendo o nome do driver gráfico corrente. Sintaxe: *getdrivername;*

2.4. GETMODENAME

Esta função retorna uma string referente ao modo gráfico passado como parâmetro. Sintaxe: *getmodename(<nummod>:word):string;*

2.5. GETMODERANGE

Este procedimento retorna a faixa de valores que podem ser utilizados no modo gráfico passado como parâmetro. Sintaxe: *getmoderange(<grafmod>:integer; var <menor>,<maior>:integer);*

2.6. GRAPHRESULT

Esta função retorna o código de erro da última operação gráfica realizada. Os erros estão predefinidos. Sintaxe: *graphresult: integer;*

2.7. GETGRAPHMODE

Esta função retorna o valor do modo gráfico corrente, o qual pode ser de 0 a 5, dependendo do driver gráfico corrente. Sintaxe: *getgraphmode: integer;*

2.8. SETGRAPHMODE

Este perocedimento permite estabelecer um novo modod gráfico e também faz com que a tela seja limpa. Sintaxe: *setgraphmode(<mode>:integer);*

2.9. GETMAXMODE

Esta função retorna o maior valor que pode ser utilizado para estabelecer o modo gráfico. Sintaxe: *getmaxmode: integer;*

2.10. GRAPHERRORMSG

Esta função retorna a mensagem de erro em string, de acordo com o número do erro passado como parâmetro. Sintaxe: *grapherrormsg(<erro>:integer):string;*

2.11. CLOSEGRAPH

Este procedimento permite voltar ao modo anterior, ao uso de INITGRAPH, e deve ser utilizado ao término de uma sessão gráfica, permitindo também a liberação de memória alocada para as rotinas gráficas. Sintaxe: *closegraph*;

2.12. ARC

Este procedimento permite desenhar um arco. Devem ser passados como parâmetros os pontos centrais, neste caso, coordenadas x e y em relação à tela como um todo, o raio, o ângulo de início e o ângulo fim. Caso o ângulo de início seja zero e o final 360, o gráfico será uma circunferência.
Sintaxe: *arc(<x,y>:integer;<anginicio>,<angfim>,<raio>:word);*

2.13. GETARCCOORDS

Este procedimento permite recuperar informações a respeito das coordenadas passadas como parâmetros no procedimento ARC. Para tanto, deve-se utilizar como parâmtero uma variável do tipo ARCOORDSTYPE. Sintaxe: *getarccoords(var <arccoord>: ARCOORDSTYPE);*

2.14. BAR

Este procedimento permite que seja desenhada uma barra, desde que se passem como parâmteros, as coordenadas dos cantos superior esquerdo (x1,y1) e inferior direito (x2,y2). Esta barra será preenchida com o estilo de preenchimento (fillstyle) e cor correntes. Sintaxe: *bar(<x1>,<y1>,<x2>,<y2>:integer);*

2.15. BAR3D

Este proedimento permite que seja desenhada uma barra em três dimensões, desde que se passem como parâmetros as coordenadas dos cantos superio esquerdo (x1,y1), inferior direito (x2,y2), profundidade e uma variável booleana, indicando se será preenchido o topo ou não. Esta barra será preenchida com o estilo de preenchimento (fill style) e cor correntes.
Sintaxe: *bar3D(<x1>,<y1>,<x2>,<y2>:integer; prof:word; topo:boolean);*

2.16. CIRCLE

Este procedimento permite que seja desenhado um círculo com centro nas coordenadas x,y e de acordo com o raio passado como parâmetro. Sintaxe: *circle(<x>,<y>:integer;<raio>:word);*

2.17. ELLIPSE

Este procedimento permite desenhar uma elipse com centro nas coordenadas x e y; ângulo inicial; ângulo final; raio da coordenada x (horizontal) e y (vertical).
Sintaxe: *ellipse (<x>,<y>:integer; <angini>,<angfinal>:word;<raiox>,<raioy>:word);*

2.18. LINE

Este procedimento desenha uma reta que parte de um par de coordenadas x e y iniciais e outro par final. Sintaxe: *line(<x1>,<y1>,<x2>,<y2>:integer);*

2.19. LINEREL

Este procedimento desenha uma reta de acordo com o deslocamento relativo.
Sintaxe: *linerel(<x>,<y>:integer);*

2.20. LINETO

Este procedimento permite desenhada uma linha do ponto corrente até as coordenadass passadas como parâmetro no procedimento. Sintaxe: *lineto(<x>,<y>:integer);*

2.21. MOVETO

Este procedimento move o ponteiro corrente para as coordenadas indicadas pelo parâmetro x.
Sintaxe: *moveto(<x>,<y>:integer);*

2.22. MOVEREL

Este procedimento permite a alteração do ponteiro corrente relativamente às coordenadas x e y passadas como parâmetro. Sintaxe: *moverel(<x>,<y>:integer);*

2.23. GETY

Esta função retorna o valor da coordenada y do ponto corrente. Sintaxe: *gety:integer;*

2.24. GETX

Esta função retorna o valor da coordenada x do ponto corrente. Sintaxe: *getx:integer;*

2.25. GETMAXX

Esta função retorna a posição mais à direita possível no modo de vídeo e driver correntes, ou seja, resolução(x). Sintaxe: *getmaxx:integer;*

2.26. GETMAXY

Esta função retorna a posição mais abaixo possível no modo de vídeo e driver correntes, ou seja, resolução(y). Sintaxe: *getmaxyx:integer;*

2.27. RECTANGLE

Este procedimento desenha um retângulo nas posições passadas como parâmetros. Estes parâmetros são o canto superior esquerdo (x1,y1) e o canto inferior direito (x2,y2). O canto superior esquerdo não pode ser

menor que (0,0) e o canto inferior direito não pode ultrapassar os valores máximos dos eixos x e y obtidos com GETMAXX e GETMAXY, respectivamente. Sintaxe: *rectangle(<x1>,<y1>,<x2>,<y2>:integer);*

2.28. DRAWPOLY

Este procedimento permite que se desenhe um polígono, utilizando a linha e a cor corrente. São passados como parâmetros os números de pontos e uma variável do tipo POINTTYPE. O procedimento cuida de se posicionar e ligar os pontos através de linhas na ordem em que é definida e passada como parâmetro. Sintaxe: *drawpolly(<numpontos>, var <ptdopoligono>: array of pointtype);*

2.29. SECTOR

Este procedimento desenha e preenche um setor elíptico. São passados como parâmetros as coordenadas centrais (x,y), ao ângulos inicial e final e os raios dos eixos x e y.
Sintaxe: *sector(<x>,<y>:integer;<anginicial>,<angfinal>,<raiox>,<raioy>:word);*

2.30. FILLPOLY

Este procedimento desenha um polígono predefinido, já sendo preenchido com o estlo corrente de fillpattern. Sintaxe: *fillpoly(<numpontos>:Word; var <polypontos>: array pointtype);*

2.31. SETGRAPHBUFSIZE

Este procedimento permite estabelecer uma nova área de buffer para ser usada na varredura e preenchimento de uma determinada região. O buffer default é de 4Kbytes, suficiente para um desenho de cerca de 650 vértices. Sintaxe: *setgraphbufsize(<buffer>:word);*

2.32. FILLELLIPSE

Este procedimento desenha uma elipse cheia com o fillstyle corente.
Sintaxe: *fillellipse(<x>,<y>:integer; <raiox>,<raioy>:word);*

2.33. FLOODFILL

Este procedimento permite preencher uma área delimitada com o fillstyle corrente, sendo interrompido quando for encontrada uma linha em branco. Sintaxe: *floodfill(<x>,<y>:integer;<borda>:word);*

2.34. GETASPECTRATIO

Este procedimento retorna a efetiva resolução da tela gráfica para que possa ser efetuada a propoerção entre yasp e xasp. Normalmente, é utilizado para fazer círculos, arcos, etc.
Sintaxe: *getaspectratio(var <xasp>,<yasp>:word);*

2.35. SETASPECTRATIO

Este procedimento permite a troca do rateio do aspect. É utilizado para fazer desenhos que use formas circulares. Sintaxe: *setaspectratio(<xasp>,<yasp>:word);*

2.36. GETCOLOR

Esta função retorna a cor corrente utilizada para desenhos, traços, círculos, etc. Sintaxe: *getcolor: word;*

2.37. GETMAXCOLOR

Esta função retorna o maior valor válido para se determinar uma cor. Se por acaso o valor retornado for 15, indica que pode-se ter 16 cores de 0 até 15, se retornar 1, pode-se ter as cores 0 e 1, e assim sucessivamente. Esta função é utilizada em conjunto com o procedimento setcolor. Sintaxe: *getmaxcolor: word;*

2.38. SETCOLOR

Este procedimento permite que seja estabelecida uma nova cor para uma paleta. Estas cores podem variar de 0 até 15 de acordo com o driver e modo gráfico correntes. Sintaxe: *setcolor(<cor>:word);*

2.39. GETBKCOLOR

Esta função retorna o índice utilizado pela cor de fundo utilizado pela paleta corrente.
Sintaxe: getbkcolor:word;

2.40. SETBKCOLOR

Este procedimento permite alterar a cor de fundo para uma paleta. Estas cores podem variar de 0 a 15 de acordo com o modo e o driver gráfico atuais. Sintaxe: *setbkcolor(<cor>:word);*

2.41. GETPALETTE

Este procedimento retorna a paleta corente e o seu tamanho. É passado como parâmetro uma variável do tipo palletetype. Sintaxe: *getpalettetype(var <palette>:palettetype);*

2.42. SETPALETTE

Este procedimento troca em uma das paletas a sua cor corrente.
Sintaxe: *setpalette (<numcor>:word;<cor>:shortint);*

2.43. GETDEFAULTPALETTE

Este procedimento permite que se retorne a definição do registro de paleta tal qual fora inicializado com o procedimento INIGRAPH. Sintaxe: *getdefaultpalette(var <palette>:palettetype);*

2.44. SETALLPALETTE

Este procedimento permite que sejam trocadas todas as cores das paletas de uma única vez.
Sintaxe: *setallpalette(var <palette>: palettetype);*

2.45. OUTTEXT

Este procedimento envia um determinado texto para o dispositivo de vídeo e na posição corrente do ponteiro de vídeo. Este texto será truncado caso ultrapasse a borda e sempre será impresso no tipo de texto corrente. Sintaxe: *outtext(<texto>:string);*

2.46. OUTTEXTXY

Este procedimento é bastante similar ao procedimento OUTTEXT, com a diferença de que, neste caso, estabelecendo as coordenadas x e y, nas quais deverá ser iniciada a escrita do texto.
Sintaxe: *outtextxy(<x>,<y>:integer,<texto>:string);*

2.47. GETPALETTESIZE

Esta função retorna o valor do tamanho da paleta corrente de acordo com o número de cores desta paleta. Sintaxe: *getpalettesize: integer;*

2.48. SETRGBPALETTE

Este procedimento permite que se altere as entradas das paletas para drivers de videos IBM8514 e VGA. É evidente que para que este procedimento seja executado corretamente, deve-se ter presente, além dos drivers, o hardware. A sigla RGB indica um padrão de cores formado por três cores básicas (red, green and blue). Sintaxe: *setrgbpalette(<cor>,<valorvermelho>,<valorverde>,<valorazul>: integer);*

2.49. CLEARDEVICE

Este procedimento permite eliminar o conteúdo do dispositivo de saída de vídeo, colocando o ponteiro corrente nas coordenadas 0,0. O vídeo será limpo, usando-se a cor de fundo definida no procedimento SETBKCOLOR. Sintaxe: *cleardevice;*

2.50. SETTEXTJUSTIFY

Os procedimentos OUTTEXT e OUTTEXTXY escrevem texto passado como parâmetro de acordo com a justificação corrente. A justificação implica em como o texto será escrito, ou seja, na horizontal à esquerda do ponteiro de vídeo corrente, no centro ou à direita, e verticalmente, abaixo do ponteiro de vídeo corrente, ao centro ou acima do mesmo. Este procedimento permite alterar tais justificações através de dois valores, um para o parâmetro horizontal e outro para o vertical. Sintaxe: *settextjustify(<horizontal>,<vertical>:word);*

2.51. SETTEXTSTYLE

Assim como é possível estabelecer a posição do texto em relação ao ponteiro de vídeo, pode-se também determinar o estilo do texto a ser escrito no vídeo. Esta é a função deste procedimento que permite estabelecer a fonte de caracteres (de 0 a 4), a direção em que o texto deve ser escrito (entre 0 e 1) e o tamanho do texto a ser impresso (entre 0 e 4).
Sintaxe: *settextstyle(<fonte>,<direcao>:word,<tamanho>:word);*

2.52. GETTEXTSETTINGS

Este procedimentoretorna o estilo de texto corrente, ou seja, fonte, direção, tamanho, justificação horizontal e vertical. Sintaxe: *gettextsettings(var <info>:textsettingtype);*

2.53. TEXTHEIGHT

Esta função retorna a altura de uma string em pixels, podendo ser utilizada para calcular o ajustamento do espaço que a string irá ocupar. Esta função leva em conta a fonte e o fator de multiplicação utilizado em SETSTEXTSTYLE. Sintaxe: *textheight(<texto>:string): word;*

2.54. TEXTWIDTH

Esta fiunçào é bastante semelhante à anterior. A diferença é que nesta o valor retornado é o da largura do texto em número de pixels. Esta função também leva em conta o tamanho da fonte de caracteres e o fator de multiplicação. Sintaxe: *textwidth(<texto>:string):word;*

2.55. GETPIXEL

Esta função retorna a cor de um ponto nas coordenadas x e y. Sintaxe: *getpixel(<x>,<y>:integer):word;*

2.56. PUTPIXEL

Este procedimento permite a colocação de um ponto nas coordenadas x e y e na cor previamente definida. Sintaxe: *putpixel(<x>,<y>:integer;<cor>:word);*

2.57. GETLINESETTINGS

Este procedimento retorna o estilo de linha corrente que pode ser de 0 a 4, sendo 0 sólido, 1 em pontos, 2 centralizado, 3 tracejado e 4 definido pelo usuário. O procedimento também retorna o modelo de linha utilizado e ainda a espessura da linha. A variável passada como parâmetro é do tipo LINESETTINGSTYPE. Sintaxe*: getlinesettings(var <linha>: linesettingstype);*

2.58. SETLINESTYLE

Este procedimento permite alterar o estilo de linha, seu molde e sua espessura. O molde será ignorado quando o estilo de linha for diferente de 4 (userbitln) e a espessura poderá apenas assumir valores 1 (NORMWIDTH) ou 3 (THICKWIDTH). Quando for utilizado o estilo de linha 4, deve-se estabelecer um valor de 16 bits para o parâmetro do modelo PATTERN.
Sintaxe: *setlinestyle(<estilo>:word;<modelo>:word;<espessura>:word);*

2.59. PIESLICE

Este procedimento desenha e preenche uma “torta” ou um pedaço de uma “torta”, com ponto central nas coordenadas x e y, ângulo inicial, ângulo final e raio.
Sintaxe: *pieslice(<x>,<y>:integer;<angini>,<angfim>,<raio>:word);*

2.60. SETFILLPATTERN

Este procedimento permite que se defina um novo modelo para preeenchimento de áreas usadas nops procedimentos: FILLPOLY, FLOODFILL, BAR, BAR3D E PIESLICE, onde são definidos os moldes e cores para preenchimento. Sintaxe: *setfillpattern(<molde>:fillpatterntype;<cor>:word);*

2.61. GETFILLPATTERN

Este procedimento retorna os parâmetros estabelecidos no procedimento SETFILLPATTERN em uma variável do tipo FILLPATTERNTYPE. Sintaxe: *getfillpattern(<molde>);*

2.62. SETFILLSTYLE

Este procedimento permite alterar o estilo e a cor para preenchimento nos procedimentos FILLPOLY, BAR, BAR3D e PIESLICE. Os possíveis estilos variam de 0 até 12.
Sintaxe: *setfillstyle(<molde>:word,<cor>:word);*

2.63. GETFILLSETTINGS

Este procedimento retorna os valores na última chamada de SETFILLSTYLE. Para tanto deve ser passado como parâmetro uma variável do tipo FILLSETTINGSTYPE.
Sintaxe: *getfillsettings()var <estilo>:fillsettingstype;*

2.64. REGISTERBGIDRIVER

Este procedimento retorna um valor menor que zero em caso de erro, ou um número do arquivo de driver passando isto para uma alocação de memória. Sintaxe: *registerbgidriver(<drv>:pointer):integer;*

2.65. REGISTERBGIFONT

Esta função retorna um valor negative, caso o arquivo de fonts não tenha sido carregado previamente na memória, ou umnúmero interno da fonte e o registro passado para a memória dinâmica. Desta forma a fonte é carregada do disco para o heap. Sintaxe: *registerbgifont(<fonte>:pointer):integer;*

2.66. INSTALLUSERDRIVER

Esta função permite que seja instalado um driver gráfico gerado pelo usuário.
Sintaxe: *installuserdriver(<nome>:string;<autodet>:pointer):integer;*

2.67. INSTALLUSERFONT

Esta função permite instalar uma nova fonte de caracteres e retorna o valor da nova fonte instalada. Sintaxe: *installuserfont(<nomearq>:string):integer;*

2.68. SETUSERCHARSIZE

Este procedimento permite estabelecer um novo fator para a fonte de caracteres ativa, isto é, feito pela seguinte relação: X e fator X, Y e fator Y. Sintaxe: *setusercharsize(<multx>,<divx>,<multy>,<divy>:word);*

2.69. SETWRITEMODE

Este procedimento estabelece como sera o modo desenho de linhas, ou desenhos que se utilizam linhas. Permite dois modos de operação, passando-se o valor COPYPUT equivalente ao valor 0, que coloca em situação de se desenhar uma linha, copiando-se o estilo corrente para o dispositivo de saída. A segunda forma é passando-se a constante predefinida XORPUT equivalente ao valor 1, que executa uma operação XOR quando desenhada uma linha, ou seja, se for encontrado um ponto onde deveria ser desenhado um novo ponto, este será apagado e vice-versa. Caso uma linha seja desenhada duas vezes e o procedimento SETWRITEMODE estiver setado em XORPUT, uma vez esta será desenhada e outra apagada.
Sintaxe: *setwritemode(<modo>:integer);*

2.70. SETVIEWPORT

Este procedimento permite determiner uma janela dentro do vídeo corrente cujas coordenadas forem passadas como parâmetro, outro parâmetro é em relação à “clipagem”, para tal, é passado como parâmetro uma variável bboleana. A partir do momento em que for executada esta rotina, os comandos gráficos serão relativos às novas coordenadas se “clip” estiver em true. Caso contrário, as coordenadas serão absolutas. Após a execução deste comando, o ponto corrente ficará nas coordenadas 0,0.
Sintaxe: *setviewport(<x1>,<y1>,<x2>,<y2>:integer;<moldura>:boolean);*

2.71. CLEARVIEWPORT

Este procedimento permite limpar apenas o “viewport” corrente, definido pela rotina SETVIEWPORT. Sintaxe: *clearviewport;*

2.72. GETVIEWSETTINGS

Este procedimento nos retorna os valores do “viewport” ativo, suas coordenadas de canto superior esquerdo e canto inferior direito e, se a “clipagem” está ligada ou não. Para executar este procedimento é necessário passar como parâmetro uma variável do tipo VIEWPORTTYPE.
Sintaxe: *getviewsettings(var <coord>:viewporttype);*

2.73. GRAPHDEFAULTS

Este procedimento permite que sejam retornados aos valores defaults e as coordenadas correntes na posição 0,0. Sintaxe: *graphdefaults*;

2.74. RESTORECRTMODE

Este procedimento permite que se volte para o modo de vídeo anterior ao procedimentoo INITGRAPH. Pode ser utilizado em conjunto com o procedimento SETGRAPHMODE, assim alterando-se modo gráfico e modo texto. Sintaxe: *restorecrtmode;*

2.75. IMAGESIZE

Esta função retorna o número de bytes necessários para armazenar uma área retangular da tela. Sintaxe: *imagesize(<x1>,<y1>,<x2>,<y2>:integer):word;*

2.76. GETIMAGE

Este procedimento permite que seja salvo uma determinada área retangular do vídeo. Para tanto devem ser passadas como parâmetros, as coordenadas do canto superior esquerdo e do inferior direito além de uma variável sem tipo, que deverá ser pelo menos 4 vezes maior que a área definida como região. Os 2 primeiros bytes desta variável armazenarão a largura e a altura da região preestabelecida. Normalmente usa-se um ponteiro para esta variável sem tipo.
Sintaxe: getimage*(<x1>,<y1>,<x2>,<y2>:integer;var <mapa>);*

2.77. PUTIMAGE

Este procedimento permite retornar uma imagem na tela. São passados como parâmetros para esta rotina, as coordenadas do canto superior esquerdo, o mapa da imagem e a forma como esta será colocada na tela. Esta forma pode ter valores de 0 até 4. Sintaxe: *putimage(<x>,<y>:integer;<mapa>;<forma>:word);*

2.78. SETACTIVEPAGE

Este procedimento permite estabelecer uma nova página de vídeo. A partir de então todos os procedimentos gráficos serão executados na nova página selecionada, porém só podem ser utilizados com suporte gráfico EGA (256K), VGA e HERCULES. Este procedimento é usado especialmente em animações de imagens. Sintaxe: *setactivepage(<pag>:word);*

2.79. SETVISUALPAGE

Este procedimento permite a visualização de uma nova página de vídeo, e assim como no procedimento anterior, só poderá ser utilizado com suporte gráfico adequado. Sintaxe: *setvisualpage(<pag>:word);*

Observação: Para maiores informações sobre como manipular a parte reáfica do Turbo Pascal, ver Apêndice E.

CAPÍTULO XI – A UNIT OVERLAY

Este capítulo tem por objetivo orientar o leitor na utilização da unit OVERLAY do Turbo Pascal.

---

1. Unit OVERLAY

Por meio desta unit é possível gerenciar as atividades de overlay de um programa. O processo chamado de overlay nada mais é do que se aproveitar uma mesma área de memória para várias rotinas diferentes. Com este procedimento pode-se diminuir consideravelmente a memória requerida para execução de um programa. As partes geradas pelo Turbo Pascal como overlay têm extensão .OVR e para que esta unidade possa ser executada com correção, deve-se utilizar as diretivas de compilação $O+ e $F+. A primeira permite geração de código de overlay e a segunda permite as chamadas distantes (FAR CALLS). A posição do programa em que deverá ser colocada a rotina que utiliza overlay é marcada pela diretiva de compilação {$O nomeunit}. Para uma rotina poder ser carregada na memória em overlay, esta deverá ser previ amente compilada como uma unit.

2. Procedimentos Diversos

2.1. OVRINIT

Este procedimento permite a inicialização do gerenciador de overlays e também abrir um arquivo .OVR. Este arquivo será procurado no diretório corrente, e caso esteja rodando em uma versão de sistema operacional, também será pesquisado no PATH especificado no ambiente DOS. Em caso de erro, este será armazenado na variável OVRRESULT. Sintaxe: *OVRINIT (<nomearq>: STRING);*

2.2. OVRINITEMS

Este procedimento é bastante similar ao procedimento OVRINIT, exceto que neste caso, a rotina será levada à memória expandida. Este procedimento só tem suporte em padrão EMSLIMEXTENDEDMEMORYSPECIFICATION, LOTUS/INTEL/MICROSOFT. Sintaxe: *OVRINITEMS*;

2.3. OVRGETBUF

Esta função retorna o valor em bytes da memória ocupada pela rotina de overlay. Este valor é ajustado automaticamente quando o programa é executado. Sintaxe: *OVRGETBUF: LONINT;*

2.4. OVRSETBUF

Este procedimento aloca uma área de memória maior ou igual à memória alocada pelo programa. Em caso de erro de execução, o programa terá sua continuidade normal, porém o tamanho do buffer de memória não será trocado. Sintaxe: *OVRSETBUF(<tamanho>:LONGINT);*

2.5. OVRCLEARBUF

Este procedimento libera a memória alocada pelo overlay e não deve nuinca ser chamado de dentro de uma rotina em overlay. Sintaxe: *OVRCLEARBUF;*

2.6. OVRGETRETRY

Esta função retorna o valor em bytes livres da área alocada pelo overlay.
Sintaxe: *OVRGETRETRY: LONGINT:*

2.7. OVRSETRETRY

Este procedimento permite determinar p quanto de memória é possível deixar livre dentro da área alocada pelo overlay. Sintaxe: *OVRSETRETRY(<tamanho>: LONGINT);*

CAPÍTULO XII – TIPOS DEFINIDOS PELO USUÁRIO

Este capítulo tem por objetivo demonstrar ao leitor a criação de tipos de dados especiais, definidos pelo usuário.

---

1. Tipos de Dados

Os tipos byte, integer, real, boolean e char são pré-definidos em Turbo Pascal e constituem conjunto de valores inteiros, reais, lógicos, caracter. O Turbo Pascal permite a criação de novos tipos além daqueles citados anteriormente. Um novo tipo é criado através de uma definição que determina um conjunto de valores associados a um identificador. Uma vez definido, o tipo passa a ser referenciado pelo seu identificador.

O novo tipo criado pelo usuário também é denominado tipo enumerado. Os identificadores que criamos para um tipo enumerado não representam mais nenhum valor, exceto seus próprios. Os tipos enumerados constituem ferramentas extremamente úteis na criação de programas limpos e auto-documentados, podendo ser referenciados em qualquer parte do programa.

2. Definição

Um tipo enumerado é uma seqüência ordenada de identificadores definidos pelo usuário, que forma um tipo ordinal. A palavra reservada para criação de tipos enumerados é type.

Type semana = (segunda, terça, quarta, quinta, sexta, sábado, domingo);

Uma das mais importantes características de um tipo enumerado é a ordem na qual os valores são ordenados. Além de estabelecer os próprios identificadores, a declaração do mesmo define a ordem dos identificadores no tipo.

3. Operações com Tipos Enumerados

Diversas operações são válidas usando os tipos enumerados. Veja o programa exemplo:

Exemplo:

```
uses crt;
type cores = (preto, azul, vermelho, amarelo, verde, rosa, branco, roxo, lilas);
var
  uma_cor: cores;
begin
   clrscr;
   uma_cor := roxo;                  {atribuir um identificador a variável de tipo}
   uma_cor := succ(uma_cor);         {recebeu lilás, o sucessor}
   uma_cor := pred(uma_cor);         {recebeu roxo novamente, o antecessor}
   uma_cor := succ(azul);            {recebeu vermelho, sucessor de azul}
   if succ(uma_cor)>=lilás then
        uma_cor := preto;
   writeln(ord(uma_cor));            {escreveu 2, a ordem do vermelho}
   uma_cor := preto;
   writeln(ord(uma_cor));            {escreveu 0, a ordem do preto}
   for uma_cor := preto to rosa do
        writeln(ord(uma_cor));       {escreveu as posições, do preto ao rosa}
end.
```

As variáveis de tipos enumerados não podem ser lidas ou escritas.

Não é válida nenhuma operação aritmética com os elementos de um tipo enumerado, como somar 1, diminuir 2 ou somar duas constantes ou mais constantes do tipo.

Podemos usar as constantes dos tipos enumerados para servir de índice de vetores / matrizes e também no loop for.

A ordem do primeiro elemento do tipo tem valor 0.

4. Tipo Derivado Faixa

Um tipo pode ser definido como uma parte de um dado tipo escalar. Esta parte ou faixa define o domínio de um tipo escalar que está associado a um novo tipo, que é chamado de tipo escalar.

A definição de uma faixa (subrange) indica quais os valores mínimo e máximo que podem ser assumidos. A descrição de uma faixa se faz com a indicação dos seus limites separados por dois pontos (..), onde o limite inferior deve ser menor que o limite superior.

Exemplo:

```
uses crt;
type semana = (seg, ter, qua, qui, sex, sab, dom):      {tipo enumerado}
      coluna = 1..80;                                   {tipo escalar associado: byte}
      maius = 'A'..'Z';                                 {tipo escalar associado: char}
      d_util = seg..sex;                                {tipo escalar associado: semana}
      num = -128..130;                                  {tipo escalar associado: integer}
var
  letra: maius;
  dia: d_util;
begin
   clrscr;
   for letra := 'J' to 'P' do
        write(letra:2);
   for dia := seg to sex do
        case dia of
              seg: writeln('Segunda');
              ter: writeln('Terça');
              qua: writeln('Quarta');
              qui: writeln('Quinta');
        other
              writeln('Sexta');
        end;
    readkey;
end.
```

Exemplo:

```
uses crt;                {observe o resultado visual desse programa}
var
  linha: 1..25;
  coluna: 1..80;
  cor: 0..15;
  caracter: char;
begin
  clrscr;
  repeat
        linha := random(24)+1;
        coluna := random(80)+1;
        cor := random(16);
```

```
        caracter := chr(random(256));
        gotoxy(coluna,linha);
        TextColor(cor);
        write(caracter);
  until keypressed;
end.
```

CAPÍTULO XIII – PROCEDURES

Este capítulo tem por objetivo preparar o leitor para a criação de procedures nos seus programas.

---

1. Procedimentos

O conjunto de sentenças básicas da linguagem visto nos capítulos anteriores é adequado para escrever pequenos programas. Porém, algo mais é necessário se desejarmos efetivamente escrever e testar programas importantes. Isto cria a necessidade de algum método de organizar o nosso trabalho de programação, de maneira que:

- Diferentes partes do programa possam ser independentes, de forma que possam ser escritas e testadas separadamente;

- Trechos de programas possam ser escritos de forma a serem reusados em diferentes partes do programa;

- Permita que programas complexos possam ser montados a partir de unidades menores já prontas e testadas.

Esse conjunto de necessidades é satisfeito pelas PROCEDURES (procedimentos) em Pascal. Uma procedure é definida em uma parte do programa, podendo ser chamada ou invocada em outras posições do mesmo. A execução da procedure se comporta como se suas instruções fossem copiadas para o ponto de onde foi chamada. Uma procedure também é chamada de procedimento, subprograma, subrotina ou rotina.

2. Definição

A forma feral para definição de uma procedure é:

```
PROCEDURE nome_da_proc(lista de parâmetros e tipos);
const   {constantes locais}
type    {tipos locais}
var     {variáveis locais}
begin
        {comandos}
end;
```

Nesta representação da sintaxe, “nome_da_proc” é o identificador que criamos como nome da procedure e toda chamada à mesma será feita por esse identificador. As declarações const, type e var, dentro do bloco da procedure, são opcionais e diferem as constantes, tipos e variáveis locais para a rotina.

A lista de parâmetros e tipos, delimitada pelos parenteses no cabeçalho da procedure, também é opcional. Se presente, a lista determina as variáveis que receberão os valores de argumentos enviados à rotina.

O termo parâmetro refere-se às variáveis listadas nos cabeçalhos das procedures e o termo argumento refere-se aos valores que realmente são passados para a rotina por uma chamada. As variáveis listadas nos cabeçalhos das procedures também são chamadas de parâmetros formais, e os valores listados numa chamada à rotina, de parâmetros reais.

Uma lista de parâmetros especifica o número e o tipo dos valores que devem ser passados à procedure. Cada nome de variável, na lista, é seguido por um identificador de tipo. Os parâmetros, na lista, são separados por ponto e vírgula (;). Essas variáveis estão definidas para uso exclusivo dentro da rotina, não estando disponíveis em mais nenhuma parte do programa.

Exemplo:

```
uses crt;

Procedure centra_titulo (tit:string;linha:byte);
var
   tam,posição: byte;
begin
   tam := length(tit);
   posicao := 40 – round(tam/2);
   gotoxy(posicao,linha);
   write(tit);
end;

var
   tit: string;
   linha: byte;
begin
   clrscr;
   write('Entre com a frase: ');          readln(tit);
   write('Diga o número da linha: ');     readln(linha);
   centra_titulo(tit,linha);              {chamada à procedure}
   readkey;
end.
```

3. Chamadas a Procedures

Um comando de chamada direciona o Turbo Pascal para executar a procedure e, opcionalmente, enviar valores de argumentos a ela (veja o exemplo acima). A forma geral de chamada é: *Nome_da_procedure(lista de argumentos);*

A definição da rotina (os seus comandos) deve estar localizada acima da chamada real à rotina.

A lista de argumentos, se existir, é uma lista dos valores que serão enviados para a rotina. Os argumentos aparecem entre parênteses e separados por vírgulas. A lista de argumentos deve conter o mesmo número e tipo de valores, e na mesma ordem que os especificados no cabeçalho da rotina a ser chamada.

No programa anterior, se fizessemos a chamada: *centra_titulo('Pascal',13);*
A string 'Pascal' é passada para o parâmetro tit;
O valor byte 13 é passado para o parâmetro linha.

No momento da chamada da procedure, o programa é interrompido naquele ponto e a procedure é executada. Após o término de sua execução, o programa continua no comando seguinte ao chamador da procedure.

> A área de memória usada na execução de um programa Pascal varia dinamicamente durante sua execução. Quando o programa principal é iniciado, é reservado na pilha um espaço para as variáveis globais. A medida que as procedures são iniciadas, novos espaços na pilha são alocados para armazenar as variáveis e os parâmetros das subrotinas. Ao terminar uma procedure, seus espaços alocados são automaticamente retirados da pilha.

4. Parâmetros das Procedures

Existem dois tipos de parâmetros em Turbo Pascal:

- O tipo valor (value).
- O tipo referência ou variável (var).

Com parâmetros do tipo valor qualquer modificação sofrida pela variável dentro da procedure não afetará os valores das variáveis dentro do programa principl, ou seja, apenas serão passados os valores para os parâmetros.

Um parâmetro do tipo var é um ponteiro para o endereço a variável para qual o parâmetro referencia, ou seja, tanto a variável quanto o parâmetro apontam para o mesmo endereço de memória. Isto significa que ao modificar o conteúdo do parâmetro na procedure, o conteúdo da variável no programa principal também se modificará. Veja o exemplo na página seguinte:

Exemplo:

```
uses crt;

Procedure msg(var tit1:string;tit2:string);
begin
   tit1 := tit1 + 'Adicional';
   tit2 := tit2 + 'Adicional';
   writeln('Na procedure (tit1) como parâmetro de referência: ',tit1);
   writeln('Na procedure (tit2) como parâmetro de referência: ',tit2);
end;      {fim da procedure}

var
   tit1, tit2: string;
begin
   clrscr;
   write('Digite uma mensagem (tit1): ');     readln(tit1);
   write('Digite outra mensagem (tit2): ');   readln(tit2);
   writeln;
   msg(tit1,tit2);
   writeln('Após a procedure (tit1): ',tit1);
   writeln('Após a procedure (tit2): ',tit2);
   readkey;
end.
```

Ao executar o programa listado logo acima, você perceberá a diferença entre um parâmetro de referência e um parâmetro de valor. Duas mensagens são digitadas e, na procedure, as duas ganham uma string 'Adicional'. Após a execução da procedure, as duas mensagens são listadas na tela. Você perceberá que, a primeira mensagem ficou com o 'Adicional' e a segunda mensagem não. Isso se deve ao fato de que a primeira mensagem foi passada como parâmetro de referência e, por isso, seu valor também foi alterado. Ao contrário da segunda mensagem, que foi passada como valor e, por isso, seu valor continua intacto.

5. Localidade

Como já vimos, todos os tipos de declarações usadas no programa principal, type, const, var podem também ser usadas dentro da área da procedure. Essas declarações definem objetos que são locais a esta procedure. Estes estão indefinidos fora de sua área e, portanto, não podem ser usados fora da procedure.

A região de validade de um identificador é chamada de escopo. Em Pascal, escopo de um objeto é a procedure onde ele foi definido ou declarado.

Se acontecer o caso de dois objetos (um local e outro global) com o mesmo nome, então a regra é que o objeto local terá precedência sobre o global. Mesmo com os nomes iguais, os endereços das variáveis locais e globais são diferentes.

Exemplo:

```
uses crt;
var
   c,d: byte;                    {estas variáveis são globais a todo programa}

Procedure proc_a;
begin
   writeln('Procedure A');       {aqui, se fosse necessário, poderia ser chamado 'proc_b"}
end;                             {o 'proc_c' não pode ser chamado aqui}

Procedure proc_b;
   Procedure proc_a;
   Begin
      Writeln('Procedure A');
   End;
begin
   writeln ('Procedure B');
   Proc_c;                       {apenas aqui é possível chamar 'proc_c', pois o}
end;                             {mesmo é local a 'proc_b'}

var a,b: byte;    {estas variáveis são locais apenas ao programa principal}
begin
   proc_a;
   proc_b;        {'proc_b' chamará 'proc_c', pois aqui não é posível chamar proc_c}
end.
```

Se as procedures estiverem paralelas entre si, uma pode chamar a outra ('proc_a' e 'proc_b');
Se as procedures estiverem aninhadas entre si (uma interna a outra), apenas a externa imediata pode chamar a interna, nenhuma outra ('proc_b' e 'proc_c').

CAPÍTULO XIV – FUNCTIONS

Este capítulo tem por objetivo preparar o leitor para a criação de funções nos seus programas.

---

1. Funções

A estrutura de uma função (function) é muito similar à de uma procedure. A diferença principal é que a function é explicitamente definida para retornar um valor de um tipo especificado. As diferenças básicas entre uma function e uma procedure:

- A function sempre retorna um valor de um tipo especificado;
- Na definição da function, seu tipo deve ser especificado;
- O nome da function representa o valor que a mesma retorna;
- Dentro do corpo da function deverá existir um comando de atribuição que identifique o valor exato que a função irá retornar.

2. Definição

A forma geral para definição de uma function é:

```
FUNCTION nome_da_func(lista de parâmetros e tipos):tipo_retorno;
const   {constantes locais}
type    {tipos locais}
var     {variáveis locais}
begin
   {comandos da function}
   nome_da_func := valor_retorno;
end;
```

As regras para a lista de parâmetros são as mesmas das procedures. Observe que o cabeçalho da function define o tipo da mesma, ou seja, o tipo do valor que a mesma irá retornar. Veja logo abaixo um programa de exemplo de function:

Exemplo:

```
uses crt;

Function cubo (num: real): real;
var
  result: real;
begin
  result := num * num * num;
  cubo := result;          {comando de retorno da function}
end;

var
  num: real;
begin
  clrscr;
  write('Entre com um número: ');      readln(num);
  write('Seu cubo é: ', cubo(num));
  readkey;
end.
```

Observe no exemplo que a função cubo foi colocada em um comando de saída. Isto prova que a função retorna um valor, ou seja, no local onde está escrita aparecerá o seu valor de retorno. Para cada função, o seu valor de retorno será o resultado de todo o processamento interno da função.

Todas as regras de chamada, parâmetros e localidade de functions seguem as mesmas das procedures. A chamada da function não é independente como um comando, mas ao invés disso, aparece como parte de um comando maior (cálculo ou operação de saída). Geralmente, uma function está relacionada a uma seqüência de ações que conduzem a um único valor calculado.

É obrigatório atribuir a função a uma variável, comando de saída ou expressão, nos seus programas.

Exemplo:

```
uses crt;

Function minusculo (msg: string): string;
var
   i: byte;
   temp: string;
begin
   temp:='';
   for I:=1 to length(msg) do
        if msg[i] in [#65..#90] then
                temp := temp + chr(ord(msg[i]+32))      {se maiúsculo, transforma}
        else
                temp := temp + msg[i];                  {caso contrário, nenhuma  alteração}
   minusculo := temp;                                   {comando de retorno}
end;

var
  frase: string;
begin
  clrscr;
  write('Entre com uma frase: ');
  readln(frase);
  writeln('A frase em minúsculo: ',minúsculo(frase));
  readkey;
end.
```

Exemplo:

```
uses crt;

Function tan(ângulo: real): real;
begin
  tan := sin(angulo*pi/180)/cos(angulo*pi/180);         {comando de retorno}
end;

var
  ang: real;
begin
   clrscr;
   write('Entre com o valor de um ângulo: ');
   readln(ang);
   writeln('Sua tangente é: ',tan(ang));
   readkey;
end.
```

Exemplo:

```
uses crt;

Function fatorial(num: integer): real;
var
   temp: real;
   i: integer;
begin
   temp. := 1;
   if num>1 then
          for i:=2 to num do
                 temp := temp *i;
   fatorial := temp;                    {comando de retorno}
end;

var
   n: real;
begin
    clrscr;
    write('Entre com um número qualquer: ');    readln(n);
    writeln('Seu fatorial é: ',fatorial(n));
    readkey;
end.
```

CAPÍTULO XV – ARRAYS UNIDIMENSIONAIS

Este capítulo tem por objetivo instruir o leitor na utilização de vetores de uma única dimensão nos programas.

---

1. Introdução

O array é a forma mais familiar de um tipo estruturado. Um array é simplesmente um agregado de componentes do mesmo tipo. Suponha que se deseja ler um conjunto de 150 notas de alunos de uma disciplina e depois coloca-las em ordem e imprimi-las. Se tivéssemos que declarar 150 variáveis, cada uma com um nome diferente, teríamos um problema com o controle do uso das mesmas, sem contar com o trabalho de escrever 150 nomes em diversas áreas do programa.

A solução para um caso como este, está em usar um único nome para todas as 150 notas, como se fosse uma única variável. Na realidade, podemos chamar o array de variável indexada, pois cada elemento será identificado por um seletor ou índice.

Notas

1	5.0
2	9.3
3	2.1
...	...
150	2.3

Nomes

1	João
2	Lúcia
3	Áurea
4	Lara
5	Ana

Idade

1	21
2	35
3	18

Cada índice representa um endereço na memória. A forma usada para indicar um elemento de um array é o nome da variável junto com o índice entre colchetes.

Exemplos:

```
nota[10] := nota[5] + nota[100];
nomes[4] := 'Lara';
read(idade[i]);
for i:=1 to 150 do
    readln(notas[i]);          {este commando irá ler todas as notas !}
```

Os arrays se dividem em unidimensionais e multidimensionais. O array unidimensional é chamado de vetor e o multidimensional de matriz.

2. Definição e Declaração de um Array Unidimensional

A estrutura do tipo vetor é a mais conhecida dos programadores por ser a estrutura mais simples de ser implementada. Suas características básicas são:

É uma estrutura homogênea, isto é, formada de elementos de mesmo tipo, chamado tipo base.
Todos os elementos da estrutura são igualmente acessíveis, isto é, quer dizer que o tempo e o tipo de procedimento para acessar qualquer um dos elementos do vetor são iguais.

Cada elemento componente da estrutura tem um nome próprio, que é o nome do vetor seguido do índice.

A declaração de um vetor pode utilizar um tipo definido pelo usuário ou simplesmente a especificação de array para uma variável (veja s seguir os três exemplos de declaração):

Exemplo:
```
type notas = array[1..150] of real;
var
   nota: notas;
```

Exemplo:
```
nota: array[1..150] of real;
```

Exemplo:
```
const max = 150;
type notas = array[1..max] of real;
var
  nota: notas;
```

Exemplo:
```
uses crt;
const max = 50;
var
  nomes: array[1..maximo] of string[20];
  notas: array[1..maximo] of real;
  soma, media: real;
  i: 1..máximo;
begin
  clrscr;
  soma := 0;
    for i:=1 to Maximo do
    begin
          write('Entre com o nome do aluno: ',i,'de ',maximo,': ');   readln(nomes[i]);
          write('Entre com sua nota: ');     readln(notas[i]);
          soma := soma + notas[i];          {acumulador de notas}
    end;
  media := soma/Maximo;         {media geral da turma}
  write('Média geral da turma: ',media:5:2);
  readkey;
end.
```

Observe que para acessar os elementos de um vetor, necessitamos de uma estrutura de repetição, a exemplo for. O índice do vetor é a variável de controle da estrutura (o contador).

Exemplo:
```
uses crt;
var
  prod: array[1..10] of string[30];          {observe neste programa que a falta da}
  preco: array[1..10] os real;               {constante prejudica a alteração da}
  quant: array[1..10] of byte;               {quantidade de produtos a serem lidos. }
  soma, total, media, s_p: real;             {O trabalho é muito maior.            }
  i: 1..10;
begin
  clrscr;
  soma:=0;
  s_p:=0;
  for i:=1 to 10 do
  begin
          write('Nome do produto: ');              readln(prod[i]);
          write('Seu preço unitário (R$): ');      readln(preco[i]);
          write('Quantidade (unid): ');            readln(quant[i]);
          total := preco[i]*quant[i];              {total de cada produto}
          writeln('Total em R$: ',total:8:2);
          soma := soma + total;                    {acumulador de total}
          s_p := s_p + preço[i];                   {acumulador de preço unitário}
   end;
   media := s_p/10;                                {média geral de preços}
   clrscr;
```

```
   writeln('O somatório geral em R$: ',soma:14:2);
   writeln('Média geral de preços em R$: ',media:8:2);
   writeln('Lista de Produtos com preço acima da média');
   for i:= 1 to 10 do
         if preco[i]> media then
                  writeln('Nome do produto: ',prod[i],' = R$ ', preço[i]:8:2);
   readkey;
end.
```

- Cuidado com o trabalho com vetores. Caso o seu limite seja ultrapassado, o programa pode travar e junto com ele o sistema.
- O Pascal só acusará erro se você tentar atribuir a um vetor um dado qualquer com índice fora da faixa escrita no programa. Mas isso não acontecerá se a origem do índice for alguma expressão.
- Os tipos array em Pascal são estáticos, ou seja, o comprimento e as dimensões alocadas são fixas em tempo de compilação e não podem ser redefinidas durante a execução do programa.

3. Constante Array Unidimensional

No Turbo Pascal podemos definir uma constante array unidimensional da seguinte forma:

```
type apart = array[1..10] of string[3];
const numeros: apart = ('101','102','103','104','105','106','107','108','109','110');
…
write(numeros[3]);        {escreverá 103}
```

Exemplo:

```
uses crt;
const meses: array[1..13] of string[3]={'Jan','Fev','Mar','Abr','Mai','Jun','Jul','Ago','Set','Out','Nov','Dez','Inv'};
var
   i:1..13;
begin
   clrscr;
   for i:=1 to 13 do
         writeln('Mês ',i,': ', meses[i]);
   readkey;
end.
```

Exemplo:

```
uses crt;
var
   i,j: byte;
   num_apt,num_pes: array[1..20] of array[1..3] of string[3];        {vetor de um vetor}
   soma: integer;
begin
   clrscr;
   soma := 0;
   for i:= 1 to 20 do        {um vetor de um vetor = uma matriz}
   begin
         write('Número de apartamento do ',i,'andar e bloco ', j,': ');        readln(num_apt[i,j]);
         soma := soma + num_pes[i,j];
   end;
   clrscr;
   writeln('Número total de pessoas do edifício: ',soma);
   writeln('Média de pessoas por apartamento: ',soma/60:3:0);
   readkey;
end.                        {no próximo capítulo será estudada a estrutura matriz}
```

CAPÍTULO XVI – ARRAYS MULTISIMENSIONAIS

Este capítulo tem por objetivo instruir o leitor na utilização de matrizes de duas ou mais dimensões nos programas.

---

1. Matrizes

Os tipos dos componentes de um array não precisam necessariamente ser inteiros, reais ou outro tipo de dados simples. Na verdade, um array pode conter dados de qualquer tipo, até mesmo outros arrays. Se um vetor contém outro vetor como tipo, ele é denominado array multidimensional ou simplesmente matriz.

```
type apart = array[1..10] of array['A'..'C'] of string[40];
var
  propriet: apart;
```

Cada elemento na vertical de 'propriet' é um novo array. O mesmo foi definido com 10 linhas (1..10) e 3 colunas ('A'..'C').

	A	B	C
1			
2		Luciana Gomes	
3			
4			
5	Janaína Pereira		
6			
7			
8			
9			
10			Fátima de Souza

Para acessar ou gravar dados em qualquer um dos 30 apartamentos do edifício acima, basta utilizar os dois índices, linha e coluna:

```
propriet[2,'B'] := 'Luciana Gomes';
propriet[10,'C'] := 'Fátima de Souza';
propriet[5,'A'] := 'Janaína Pereira';
```

O uso de matrizes é tão comum que o Pascal permite uma forma curta para a definição.

```
type apart = array[1..10,'A'..'C'] of string[40];
var
  propriet: apart;
```

1.1. Acessando Elementos da Matriz

Um elemento pode ser selecionado pela expressão geral matriz[i,j], onde matriz é uma variável do tipo matriz, i é o índice das linhas e o j o índice das colunas.

Considere a matriz a seguir e veja se possíveis operações que podem ser feitas para acessar seus elementos:

	1	2
1		10
2		
3	15	
4		0

```
Mat[1,2]:=10;        Mat[3,1]:=15;        Mat[4,2]:=0;     {atribuição}
Readln(mat[3,2]);    readln(mat[4,1]);    {leitura individual}

For i:= 1 to 4 do                  {usando loops for aninhados é possível}
   For j:=1 to 2                   {fazer a leitura de todos os elementos}
        Readln(mat[I,j]);          {da matriz}
```

Exemplo:
```
uses crt;
var
  i,j: byte;
  mat:array[1..4,1..2] of integer;
begin
  clrscr;
  for i:= 1 to 4 do                {linhas}
    for j:=1 to 2 do               {colunas}
        readln(mat[I,j]);
end.
```

2. Matriz Constante

Uma matriz constante deve ser definida da seguinte forma:

```
type apart = array[1..5,1..2] of string[3];
const num: apart = (('101','102'), ('103','104'), ('105','106'), ('107','108'), ('109','110'));
...                    {   1 linha      2 linha      3 linha      4 linha      5 linha   }
write(num[3,2]);          {106}
write(num[1,1]);          {101}
```

Exemplo:
```
uses crt;
const alunos: array[1..3,101..103] of string[10] =
      (('Cenira','Joana','Marcia'),                {primeira linha}
      ('Lourdes','Nágyla','Luciana'),              {segunda linha}
      ('Patrícia','Marcos','Angélica'));           {terceira linha}
var
  i,j: byte;
begin
  clrscr;
  for i:=1 to 3 do
    for j:=101 to 103 do
        writeln('Aluno(a)',I,' da turma ',j,': ',alunos[i,,j]);          {listará por linha}
  writeln;
  for j:=101 to 103 do
    for i:=1 to 3 do
        writeln('Aluno(a)',I,' da turma ',j,': ',alunos[i,,j]);          {listará por coluna}
  readkey;
end.
```

3. Aplicações com Matrizes

A seguir serão descritos programas para desenvolver as seguintes operações com matrizes na matemática:

a) construção de matrizes;
b) soma de matrizes;
c) matrizes transpostas.

3.1. Construção de Matrizes

Construir a matriz B_{2x4} tal que b_{ij} = i2-j, se i ≠ j ou 5+2j, se i=j.

Exemplo:
```
Program construir_matrizes;
uses crt;
const linhas = 2;
```

```
        colunas = 4;
type matriz = array[1..linhas,1..colunas] of integer;
var
  b: matriz;
  i:1..linhas;
  j: 1..colunas;
begin
  clrscr;
  for i:=1 to linhas do
    for j:=1 to colunas do
    begin
        gotoxy(j*5,i*2);
        if (i=j) then
                B[i,j]:=5+2*j
        else
                B[i,j] := I*I-j;
    end;
  readkey;
end.
```

O programa ao lado apenas gera a matriz, não listando seu conteúdo na tela. Para listar, basta usar loops for aninhados.

3.2. Somando Duas Matrizes

Só podemos somar matrizes de mesmo tipo, ou seja, o número de linhas e colunas é o mesmo nas duas.

Exemplo:

```
Program somar_matrizes;
uses crt;
const linhas = 4;          {constants globais}
      colunas = 4;
type matriz = array[1..linhas,1..colunas] of integer;        {tipo global}


Procedure leia_matriz(var m1:matriz; msg:string; x,y:byte);
var
  i,j: 1..linhas;
begin
  gotoxy(x,y-1);
  write(msg);              {na linha de cima, escreva a mensagem}
  for i:=1 to linhas do
  begin
    for j:=1 to colunas do
    begin
       gotoxy(x,y);
       read(m1[i,j]);
       inc(x,4);
    end;
   dec(x,4*colunas);
   inc(y);
  end;
end.


Procedure soma(m1,m2: matriz; var soma: matriz);
var
  i,j: 1..linhas;
begin
  for i:=1 to linhas do
      for j:=1 to colunas do
          soma[i,j]:=m1[i,j]+m2[i,j];
end.
```

```
Procedure escreva_matriz(msg:string; m1:matriz; x,y:byte);
var
  i,j: 1..linhas;
begin
  gotoxy(x,y-1);
  write(msg);              {na linha de cima, escreva mensagem}
  for i:=1 to linhas do
  begin
      for j:=1 to colunas do
      begin
          gotoxy(x,y);
          write(m1[i,j]);
          inc(x,4);
      end;
      dec(x,4*colunas);
      inc(y);
  end;
end.

var
 mat1,mat2,soma: matriz;       {variáveis locais ao programa principal}
begin
  clrscr;
  leia_matriz(mat1,'Matriz Numero 1',1,2);          {começando na coluna 1, linha 2}
  leia_matriz(mat2,'Matriz Numero 2',40,2);         {começando na coluna 40, linha 2}
  soma(mat1,mat2,soma);
  escreva_matriz('Matriz soma',soma,1,12);
  readkey;
end.
```

3.3. Matrizes Transpostas

Matriz transposta é toda a matriz onde são trocadas as linhas pelas colunas, ou vice-versa.

Exemplo:

```
Program matriz_transposta;
uses crt;
const linhas=3;
        colunas=2;
type matriz = array[1..linhas,1..colunas] of integer;
      transp = array[1..colunas,1..linhas] of integer;

Procedure leia_matriz(var m1: matriz);
var
  i,j:byte;
begin
  for i:= 1 to linhas do
    for j:=1 to colunas do
    begin
         gotoxy(j*5,2+i*2);
          read(m1[i,j]);
    end;
end;

Procedure transposta(m1: matriz;var t: transp);
var
  i,j:byte;
begin
  for i:= 1 to linhas do
    for j:=1 to colunas do
         t[i,j] := m1[i,j];
end;
```

```
Procedure escreva_matriz(m1: transp);
var
   i,j:byte;
begin
   for i:= 1 to linhas do
      for j:=1 to colunas do
      begin
            gotoxy(j*5,12+i*2);
             read(m1[i,j]);
      end;
end;


var
   mat: matriz;
   t: transp;
vegin
   clrscr;
   writeln('Matriz Transposta');
   leia_matriz(mat);
   transposta(mat,t);
   escreva_matriz(t);
   readkey;
end.
```

CAPÍTULO XVII – TIPOS ESTRUTURADOS - REGISTRO

Este capítulo tem por objetivo orientar o leitor na utilização da estrutura heterogênea Record (registro).

---

1. Introdução

Um vetor de strings pode ser usado para guardar os nomes dos empregados de uma firma. Um vetor de reais pode ser usado para guardar seus respectivos salários. Entretanto, uma matriz bidimensional não pode ser usada para guardar os nomes e os salários dos empregados, porque todos os elementos de uma matriz devem ser do mesmo tipo.

No entanto, estruturas que usam elementos de tipos diferentes, mas que estão logicamente relacionados entre si, são muito comuns e necessárias em várias áreas. Observe a ficha abaixo:

INSC	NOME	SALÁRIO	OPTANTE
121	Lúcia Nunes	650,00	Sim

Integer	string	real	char

Uma estrutura desse tipo é definida em Turbo Pascal como um Record. A cada elemento dá-se o nome de campo ou componente. Esse conjunto de informações do empregado deverá ser referenciado por um identificador, como por exemplo, ficha.

Records correspondem a conjuntos de posições de memória conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições.

```
write(ficha.insc);      {irá imprimir o campo insc do registro ficha}
ficha.salario := 650;   {irá armazenar no campo salário do registro ficha, o valor 650}
```

2. Declaração

Há duas maneiras de se declarar uma variável Record.

Exemplo:
```
type ficha = record
        insc: integer;
        nome: string;
        salario: real;
        optante: char;
end;
var func: ficha;
```

Exemplo:
```
var func: record
        insc: integer;
        nome: string;
        salario: real;
        optante: char;
end;
```

- Os campos podem pertencer a qualquer tipo padrão ou definido pelo usuário, e não existe nenhum limite para o número de campos da estrutura Record.
- O fim da definição está marcado com a palavra reservada End.
- Após a definição de tipo, podemos declarar uma ou mais variáveis que pertençam ao tipo Record.

Exemplo:

```
type apt = record
        num: byte;
        propriet: string[30];
end;
var
  moradores,sindicos: apt;
```

3. Operações com tipo Record

Veja o programa abaixo que ilustra as principais operações com informações do tipo record.

Exemplo:

```
Program uso_record;
uses crt;
type ficha = record;
        codigo: integer;
        nome: string[45];
        tel: string[10];
end;
var
   aluno,func:ficha;
begin
   clrscr;
   write('Código do aluno: ');      readln(aluno.codigo);
   write('Nome do aluno: ');        readln(aluno.nome);
   write('Telefone do aluno: ');    readln(aluno.tel);
   with func do
      begin
         write('Código do funcionário:');          readln(codigo);
         write('Nome do funcionário:');            readln(nome);
         write('Telefone do funcionário:');        readln(tel);
      end;
   if aluno.nome = func.nome then
         aluno := func     {atribuição de toda estrutura numa operação simples}
   else
         writeln('Aluno e Funcíonários são pessoas diferentes !);
   readkey;
end.
```

1. É possível transferir toda a estrutura numa única operação de atribuição como feito no programa acima.
2. O Turbo Pascal não permite comparação direta de variáveis Record, apenas de seus campos.
3. Para comparar dois records, é preciso comparar individualmente todos os campos dos records.
4. Não é possível ler diretamente variáveis Record, é preciso ler campo a campo.
5. O comando WITH pode ser usado para simplificar o trabalho com dados Record.

4. O comando WITH

O comando WITH permite que os campos de um Record sejam referenciados sem listar toda vez o nome do identificador de variável Record (e o ponto), como aplicado no programa anterior.

Exemplo:

```
with aluno, diretor do
begin
   write('Nome do aluno: ');        readln(nome);          {aluno.nome}
   write('Idade: ');                readln(idade);         {aluno.idade}
   write('CPF do diretor: ');       readln(cpf);           {diretor.cpf}
   write('Telefone: ');             readln(tel);           {diretor.tel}
end;
```

Caso dois records possuam campos em comum, devemos separar cada record com seu with, ou então especificar o identificador Record para que não haja confusão nos dados:

Exemplo:

```
with aluno, diretor do
begin
   write('Nome do aluno: ');   readln(aluno.nome);      {aluno.nome}
   write('Idade: ');           readln(aluno.idade);     {aluno.idade}
   write('CPF do diretor: ');  readln(diretor.cpf);     {diretor.cpf}
   write('Telefone: ');        readln(diretor.tel);     {diretor.tel}
end;
```

Caso não seja colocado o identificador Record, será assumido apenas a última variável nome do comando with junto com o último identificador declarado (diretor), no caso ao lado, diretor.nome

5. Record dentro de Record

Observe o trecho de programa a seguir:

```
type ficha = record
        codigo: integer;
        nome: string[45];
        ender: Record
                rua: string[30];
                num: Word;
                bairro: string[20];
        end;
        tel: string[10];
end;
var
  cliente: ficha;
begin
   clrscr;
   with cliente, cliente.ender do
   begin
        write('Código do cliente: ');     readln(codigo);
        write('Nome: ');                  readln(nome);
        write('Endereço - Rua: ');        readln(rua);
        write('Número: ');                readln(num);
        write('Bairro: ');                readln(bairro);
        write('Telefone: ');              readln(tel);
   end;
   if cliente.ender.rua = 'Av. Alberto Torres' then      {referência de um Record interno a outro}
        writeln('Mora no centro');
   else
        if cliente.nome = 'Joao Gomes' then              {record comum}
                writeln('Gente boa');
   readkey;
end.
```

4. Observe que utilizamos um record dentro de outro record. Isto é cabível na programação Pascal, bastando apenas que façamos a referência à variável Record mais externa, depois à mais interna e finalmente o identificador de campo.

5. Caso tenhamos dois identificadores Record que possuam campos internos iguais, é aconselhável diferencia-los para que não haja confusão nas operações (rua1, rua2, ender1, ender2, etc).

6. Constante Record

O Turbo Pascal permite que se defina uma constante tipo Record.

Exemplo:
```
type ficha = record
        codigo: integer;
        nome: string[45];
end;
const fornecedor: ficha = (codigo:4; nome:'Lucia Santos');
```

Dentro do parentêses, a ordem dos campos deve ser a mesma que aparece na definição do Record e os valores dos campos devem ser do mesmo tipo daqueles definidos na declaração do record.

Exemplo:
```
type ficha = record
        nome: string[45];
        ender: record
                rua: string[30];
                num: Word;
        end;
        tel: string[10];
end;
const aluno: fich = (nome:'Maria'; ender:(rua:'Av. Bambina';num:10); tel:'7222222');
```

Observe no trecho acima como fazer quando o Record possui outro Record inteiro.

7. Array de Records

Já aprendemos que o tipo de um array pode ser um tipo qualquer, portanto, um Record pode ser um tipo de um array. Veja o trecho de programa abaixo:

Exemplo:
```
type ficha = record
        codigo: integer;
        nome: string[45];
        ender: record
                rua: string[30];
                num: Word;
                bairro: string[20];
        end;
        tel: string[10];
end;
var
  clientes: array[1..100] of ficha;
  i: 1..100;
begin
  clrscr;
  for i:=1 to 100 do
```

```
  with cliente[i], cliente[i].ender do
  begin
        write('Código do cliente: ');      readln(codigo);
        write('Nome: ');                   readln(nome);
        write('Endereço - Rua: ');         readln(rua);
        write('Número: ');                 readln(num);
        write('Bairro: ');                 readln(bairro);
        write('Telefone: ');               readln(tel);
  end;
  readkey;
end.
```

8. Record com Variante

Em muitas aplicações do cotidiano de um programador, são necessárias estruturas flexíveis de dados. Uma lista fixa de campos de uma variável Record, como usamos até agora, pode não satisfazer as necessidades de um determinado conjunto de dados. Felizmente, o Turbo Pascal permite criar uma estrutura que contenha alguns campos que variam de acordo com a situação. Tal estrutura é conhecida como Record com variante.

Exemplo:

```
Program Record_com_variante;
uses crt;
var
  aluno: record
        codigo: word;
        nome: string[40];
        case tipo: byte of
              0: (pai,mae:string[40]);
              1: (cpf,ident:string[12]);
        end;
begin
  clrscr;
  with aluno do
  begin
        write('Código cadastrado: ');          readln(codigo);
        write('Nome: ');                       readln(nome);
        write('Endereço 1 - Rua: ');           readln(rua);
        write('[0] Aluno  [1] Funcionário: '); readln(tipo);
        if tipo = 0 then
        begin
              write('Nome do pai: ');          readln(pai);
              write('Nome da mãe: ');          readln(mae);
        end;
        else
        begin
              write('Identidade: ');           readln(ident);
              write('Num.cpf: ');              readln(cpf);
        end;
   end;
   readkey;
end.
```

- O identificador de campo que vem após a palavra CASE é denominado por muitos de “campo de etiqueta”.
- Os campos variantes correspondentes a um particular valor do campo etiqueta devem ser colocados entre parênteses.
- Um campo variante vazio é denotado por ().
- O campo etiqueta pode ser qualquer identificador de tipo escalar.
- Não há um END especial para a parte variante porque esta é finalizada pelo END que finaliza o Record inteiro. Portanto, a parte variante de um Record sempre vem no fim do mesmo.
- Cada identificador de campo de um Record, tenha ou não uma parte variante, deve ser diferente de qualquer tipo de campo naquele Record. Portanto, não é possível tem no mesmo Record dois identificadores de campo iguais.

CAPÍTULO XVIII – TURBO DEBBUGER

Este capítulo tem por objetivo demonstrar ao leitor aomo depurar um programa por meio do Debbuger do Turbo Pascal..

---

1. Definição

Um debugger ou um depurador é um programa que permite a execução gradativa e controlada de um programa, mediante comandos do programador pelo teclado, sendo a sua ferramenta mais valiosa, depois do cérebro. O Turbo Debugger acompanha o Turbo Pascal, e permite que um programa seja executado uma linha de cada vez.

2. Execução Linha-a-linha Usando o Debugger

a) Para começar, execute manualmente o programa abaixo e faça um esquema da sua tela de saída.

```
PROGRAM QuadradoComX;
{Mostra na tela um quadrado feito com letras X.
Feito por Fulano de Tal, em 11/11/99}

uses CRT;

begin
  clrscr;
  writeln( 'XXXXX' );
  writeln( 'X   X' );
  writeln( 'X   X' );
  writeln( 'X   X' );
  writeln( 'XXXXX' );
end.
```

b) Agora digite, compile e execute o programa, comparando a tela obtida com a prevista.

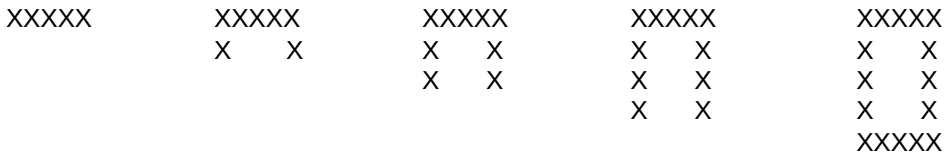
c) Agora vamos executar o programa usando o debugger. Com a janela do programa-fonte ativa, tecle F7. Isto inicia o debugger e a execução linha-a-linha. Uma barra horizontal surge na tela, posicionada sobre o **begin**. Essa barra indica qual será a próxima linha de instruções que será executada. Tecle F7 novamente. A barra estará sobre o *clrscr*. A próxima operação a ser efetuada é limpar a tela. Qual tela? Bem, a tela do seu programa não está aparecendo; você está no editor. Para ver a tela de saída do seu programa, tecle Alt-F5. Tudo que o seu programa fizer vai aparecer aí. O que você vê agora, antes da execução da primeira instrução, é a tela DOS, contendo o que aconteceu antes de você entrar no Turbo Pascal ou o resultado do último programa executado. Tecle qualquer coisa para voltar ao programa-fonte.

d) Tecle de novo F7, para executar o primeiro *writeln*, e novamente Alt-F5. Veja o que houve na tela DOS e tecle algo para retornar.

e) Novamente F7, para executar o próximo *writeln*. Veja o que apareceu na tela.

f) Continue teclando F7, até chegar ao **end**.

Observe que a figura foi *construída* na tela uma linha de cada vez na seguinte seqüência:



> *Usar o debugger não é útil somente para encontrar erros, mas também para você ver as instruções funcionando e assim aprender mais rapidamente.*

CAPÍTULO XIX – I/O CHECKING

Este capítulo tem por objetivo demonstrar ao leitor o controle dos erros de entrada e saída nos programas.

---

1. Run-time Error

A expressão run-time errors indica erros em tempo de execução, sabemos que nenhum programa está livre de receber dados indesejáveis dos usuários como, por exemplo, receber um nome de funcionário em um local onde era para ser digitado o seu salário.

Da mesma forma, se tentarmos abrir um arquivo no disco, se o mesmo não existir, será gerado um erro em tempo de execução, o programa será abortado e sem dúvida alguma, o usuário não irá entender nada do que aconteceu.

Exemplo:

```
Program teste;
uses crt;
var
  num: integer;
begin
  clrscr;
  {$I-}
  readln(num);
  {$I+}
  writeln(ioresult);
  readkey;
end.
```

Se você digitar um número inteiro qualquer, o programa irá funcionar normalmente e será impresso na tela 0 (zero), o resultado da operação de I/O (entrada/saída – input / output). Mas se, por exemplo, você digitar seu nome, o programa também irá funcionar normalmente, só que o resultado da operação será 106, que indica Formato Numérico Inválido (Invalid Numeric Format), basta olhar na tabela de erros em tempo de execução (ver Apêndice A).

As conclusões:

EXPRESSÃO/COMANDO	AÇÃO
{$I-} (diretiva de checagem de I/O)	Desliga a checagem de proced. de entrada e saída de dados
{$I+} (diretiva de checagem de I/O)	Liga a checagem de I/O de dados (valor padrão)
Ioresult	Retorna o código do erro, caso haja, ou retorna 0 (zero), caso a operação seja bem sucedida. (ver tabela pág. 4-5)

Exemplo:

```
uses dos,crt;
var
  f: file of byte;
begin
  {Com "ParamStr(1)" obtém-se o nome do arquivo direto na linha de comando}
  Assign(F,ParamStr(1));
  {$I-}
  Reset(F);
  {$I+}          {tenta abrir o arquivo para leitura}
  if IOResult = 0 then          {se não houve erros}
        writeln('Tamanho de arquivo em bytes: ',FileSize(F))
  else
        writeln('Arquivo não encontrado');
end.
```

CAPÍTULO XX – O TIPO TEXT - ARQUIVOS

Este capítulo tem por objetivo orientar o leitor na manipulação de arquivos-texto nos programas.

---

1. Introdução

Todas as operações de entrada e saída são executadas através de arquivos. Para simplificar essas operações o Turbo Pascal abre automaticamente dois arquivos-texto denominamos input e output, no início de cada execução do programa.

Graças a essas variáveis de arquivos predefinidas, a procedure READ aceita, por default, os dados pelo teclado e a procedure WRITE envia os dados para a tela.

Já vimos em capítulos anteriores que a sintaxe da procedure READ é Read(disp,var), onde disp é um dispositivo qualquer de I/º Como por exemplo, o dispositivo LST é um arquivo-texto direcionando a saída para a impressora.

2. Arquivos de Dados

Um arquivo de dados tem a propriedade de ser independente dos programas. É separado de qualquer programa e pode ser acessado e usado por muitos programas diferentes. Na maioria dos casos, usamos um único programa para introduzir as informações no arquivo de dados e gravá-lo em disco. A partir deste momento, podemos ler as informações daquele arquivo de dados usando muitos programas diferentes, cada um executando uma atividade diferente com o arquivo.

Suponha que você tenha necessidade de criar um arquivo de dados, para guardar os nomes, endereços e telefones de seus amigos. Inicialmente, você precisará de um programa para introduzir as informações no arquivo e adicionar novos dados ao mesmo. Um outro programa poderá ser criado para listar as informações do arquivo. Outro permitirá você selecionar um número de telefone do arquivo usando o nome do amigo como critério de seleção. Você pode criar outro programa para mudar os endereços e os telefones. Outro para imprimir etiquetas contendo as informações do arquivo. As possibilidades continuam...

O arquivo de dados é gravado no disco, separadamente dos programas, num lugar específico, e pode ser acessado por muitos programas diferentes. Cada programa descrito acima, copia as informações gravadas no disco para memória do computador e esses dados são manuseados em memória de acordo com as necessidades de cada programa. Alternativamente, o programa poderá transferir as informações da memória do computador para serem gravadas no disco.

3. Tipos de Arquivos

O Turbo Pascal admite três tipos de arquivos:

- Arquivo Typed (arquivo tipado): pode conter praticamente todos os tipos de dados, menos o tipo FILE. Isto quer dizer que podemos criar um arquivo para guardar reais, arrays, records, mas não podemos criar um arquivo de arquivo. Cada elemento pertencente ao arquivo tipado pode ser acessado diretamente pelo número, isto é, um registro pode ser acessado imediatamente sem que tenhamos que acessar todos os registros anteriores. Esse tipo de acesso é denominado randômico ou direto. Um arquivo tipado é um arquivo randômico.

- Arquivo Text: pertence a um tipo predefinido do Pascal denominado Text. Os arquivos Text são armazenados no disco como linhas de caracteres ASCII e só podem ser acessados de forma seqüencial, isto é, só podem ser lidos ou escritos do início para o fim do arquivo. Um arquivo seqüencial em disco está projetado para ser lido do início ao fim, toda vez que for aberto. Em outras palavras, não existe uma forma de pular diretamente de um determinado registro para outro num ponto qualquer do

arquivo. Por essa razão, os arquivos seqüenciais são melhor usados em aplicativos que executam tarefas sobre todo um conjunto de dados ao invés de executá-las sobre um determinado registro.

- Arquivo Untyped (sem tipo): pode ser associado a qualquer arquivo de disco, sendo que o tamanho do registro seja de 128 bytes. As operações de entrada e saída usando um arquivo sem tipo transferem dados diretamente de um arquivo em disco para uma variável de memória e vice-versa, sem passar pela memória temporária (buffer), o que, além de economizar espaço na memória, também torna a operação ligeiramente mais rápida. Seu uso é muito limitado. Quando um arquivo é aberto apenas para uma operação de remoção ou troca de nome, é aconselhável usar um arquivo sem tipo.

Os procedimentos para manipulação de qualquer arquivo são:
- Definir uma variável de arquivo e associar a variável ao nome do arquivo no disco.
- Abrir o arquivo, para leitura ou para escrita ou para ambos.
- Ler os dados do arquivo ou gravar dados no mesmo.
- Fechar o arquivo quando o processo terminar.

4. Declaração e Assinalamento de Arquivos-Texto

O Pascal usa a palavra reservada TEXT como identificador padrão de arquivos-texto. A sintaxe para definir variáveis desse tipo:

```
var
  arq_alunos: text;
  arq_func: text[512];
```

Os arquivos-texto são definidos inicialmente com memória temporária de 128 bytes. Isto quer dizer que o Pascal trabalha com 128 bytes de informações de cada vez. Esse tamanho de memória temporária é adequado para a maioria das aplicações. Entretanto, no caso de programas que utilizam repetidamente informações guardadas em disquetes, como é o caso de programas de banco de dados, pode ser conveniente aumentar a capacidade de memória temporária. Para estabelecer o tamanho da memória temporária, basta colocar o número de bytes desejado entre colchetes depois da palavra TEXT, na declaração da variável de arquivo (como no exemplo acima).

Normalmente, declaramos uma variável de memória, do mesmo tipo de dados que vão ser gravados ou lidos no arquivo. É essa variável que guarda em memória os conteúdos dos registros que vão ser lidos do arquivo ou que vão ser gravados no arquivo. Todo o manuseio com os dados é feito em memória. O arquivo de dados é utilizado apenas para gravar ou ler informações.

Exemplo:
```
type dados = record
        nome: string[30];
        tel: string[10];
end;
var
  arq_alunos: text;
  registro: dados;
```

Para utilizarmos uma variável de arquivo para acessar um arquivo-texto, temos que associar a variável a um determinado nome de arquivo em disco. Denominamos esse processo de assinalamento. A procedure predefinida que executa esta tarefa é ASSIGN. Assign associa o nome da variável de arquivo a um nome de arquivo em disco. Nunca deve ser usada em arquivos que já estejam em uso.

Exemplo:
```
assign (arq_alunos,'ALUNOS.DAT');
```

A partir desse comando, toda referência a “arq_alunos” no programa será dirigida ao arquivo no disco chamado “alunos.dat”.

Caso um programa tenha mais de uma variável de arquivo, teremos que assinalar individualmente cada variável, isto é, teremos tantos assign quanto forem as variáveis de arquivo.

Assinalar uma variável arquivo é criar um buffer específico para aquela variável. Um buffer é uma parte da memória do computador, com determinada capacidade, que é criada para atuar como intermediário entre o programa e o arquivo de dados no disco. A informação que se encontra no disco, primeiro é copiada para buffer, uma determinada quantidade de bytes de cada vez, estando, portanto, disponível ao programa para manipulação. Do mesmo modo, as informações, para serem gravadas em disco, são primeiramente acumuladas no buffer. Quando o buffer está cheio, as informações são copiadas para o disco.

ALUNOS.DAT ←→ BUFFER ←→ MEMÓRIA

Graças aos buffers, os dados são movimentados corretamente, para os arquivos especificados.

O default de arquivos que podem ser abertos simultaneamente no Pascal é 16. Se você deseja aumentar o número de arquivos que podem permanecer abertos basta usar a diretiva F do compilador. {$F20}.

Você também pode alterar o número de arquivos abertos ao dar "boot" no sistema, uando a declaração FILES do arquivo CONFIG.SYS, além também de poder estabelecer o número de buffers utilizáveis (BUFFERS).

Exemplo:

```
{$F20}
uses crt;
type dados = record
          nome: string[30];
          tel: string[10];
end;
var
  arq_alunos: text;
  registro: dados;
begin
   clrscr;
   assign(arq_alunos,'ALUNSO.DAT');
```

5. Abrindo um Arquivo-Texto

Um determinado arquivo-texto pode ser aberto para leitura ou gravação, mas não para ambas operações ao mesmo tempo.

O Turbo Pascal fornece três procedures diferentes para abertura de um arquivo-texto:

a) Rewrite (var de arquivo): cria um novo arquivo e o abre para operações de escrita. Após a abertura, podemos usar procedures WRITE e WRITELN para escrever os dados e linhas de texto no arquivo. Caso o nome do arquivo já exista no disco, este procedimento destrói o arquivo antigo e cria outro vazio e com o mesmo nome, isto significa que todos os dados do arquivo existente serão perdidos.

b) APPEND (var de arquivo): abre um arquivo-texto já existente, de modo que podemos adicionar novos registros no fim do arquivo. Ocorre um erro de I/O se o arquivo não existir no disco. Após a abertura, podemos usar as procedures WRITE e WRITELN para escrever dados no arquivo.

c) RESET(var de arquivo): abre um arquivo já existente para leitura. Após a abertura do arquivo, podemos usar as procedures READ e READLN para lermos dados e linhas de texto do arquivo.

Em contraste com a procedure REWRITE que cria um arquivo novo destruindo o antigo, caso exista, as procedures APPEND e RESET assumem a existência de um arquivo nomeado no disco. Se o arquivo não puder ser localizado, em ambos os casos, resultarão em erros de I/O em tempo de execução.

Exemplo:

```
uses crt;
type dados = record
        nome: string;
        tel: string;
end;
var
  registro: dados;
  arq_reg: text;
begin
  clrscr;
  assign(arq_reg,'ALUNOS.DAT');
  {$I-}
  reset(arq_reg);
  {$I+}
  if IOResult = 0 then
        write('Arquivo "ALUNOS.DAT" aberto para leitura');
  else
  begin
        rewrite(arq_reg);
        write('Arquivo ALUNOS.DAT criado');
  end;
  readkey;
  close(arq_reg);
end.
```

6. Escrevendo Dados num Arquivo-Texto

Veja o programa-exemplo abaixo:

```
uses crt;
type dados = record
        nome: string;
        tel: string;
end;
var
  registro: dados;
  arq_reg: text;
  conf: char;
begin
  clrscr;
  assign(arq_reg,'ALUNOS.DAT');
  {$I-}
  append(arq_reg);
  {$I+}
  if IOResult = 0 then
        writeln('Arquivo "ALUNOS.DAT" aberto para leitura');
  else
  begin
        rewrite(arq_reg);          write('Arquivo ALUNOS.DAT criado');
  end;
  readkey;
  repeat
    write('Entre com um nome: ');              readln(registro.nome);
    write('Seu telefone: ');                   readln(registro.tel);
    write('Confirma gravação ? <S/N>: ');      conf := upcase(readkey);
    if conf = 'S' then
        writeln(arq_reg,registro.nome,#32,registro.tel);
  until conf='N';
  close(arq_reg);
end.
```

Observe que o comando WRITELN automaticamente insere ao final da linha um caracter de retorno de carro (#13) e um de alimentação de linha (#10).

Podem ser usados como caracteres delimitadores de campos: espaço (#32) e tabulação (#9).

7. Fechando um Arquivo-Texto

Para fechar qualquer arquivo aberto, o Pascal dispõe da procedure CLOSE, como foi visto nos exemplos anteriores de programas.

Notas sobre o comando close:

- Se vários arquivos foram abertos simultaneamente, teremos que fechá-los individualmente, usando um comando close para cada um.

- O comando CLOSE fecha o arquivo externo em disco, mas não termina com o assinalamento feito pelo comando ASSIGN. Isto significa que o programa pode usar um comando REWRITE, RESET ou APPEND após o comando CLOSE se necessário.

- CLOSE é um comando muito importante, pois é usado para manter a integridade e exatidão dos arquivos de dados. Relembre que o buffer atua como um intermediário entre o programa e o arquivo em disco. Quando se executa uma operação de gravação, os dados são enviados primeiramente para o buffer. Quando o buffer está cheio, os dados são gravados em disco. Freqüentemente essa operação é chamada de atualização de arquivos em disco.

O que acontece quando o buffer só está parcialmente cheio e não existem mais dados para preenchê-lo. Se você está esperando que o buffer semi-cheio simplesmente transfira seu conteúdo para o disco quando o programa termina a sua execução, você está enganado. Os dados de um buffer semi-cheio não são necessariamente gravados no arquivo.

O comando CLOSE força o buffer a transferir o seu conteúdo para o arquivo de disco, mesmo que o buffer não esteja cheio.

8. Lendo Dados de um Arquivo-Texto

Veja o programa-exemplo abaixo:

```
uses crt;
type dados = record
        nome: string;
        tel: string;
end;
var
  registro: dados;
  arq_reg: text;
  conf: char;
begin
  clrscr;
  assign(arq_reg,'ALUNOS.DAT');
  {$I-}
  reset(arq_reg);
  {$I+}                         {tentando abrir para leitura}
  if IOResult <> 0 then
        exit;                   {algum problema, fim}
  with registro do
  begin
    repeat
```

```
      readln(arq_reg,nome,tel);  {lendo do arquivo}
      write(nome,' ',tel);       {listando na tela}
    until eof(arq_reg);          {repita até que o final do arquivo seja atingido}
  end;
  close(arq_reg);
  readkey;
end.
```

- Quando o programa está lendo uma seqüência de valores de um arquivo texto, a procedure READ reconhece os caracteres delimitadores usados durante o processo de gravação.

- O arquivo-texto foi projetado para conter só caracteres ASCII. Mesmo assim, as procedures READ e READLN podem ler apropriadamente valores de dados numéricos ou strings. É evidente que durante um processo de leitura, não podemos colocar valor de uma variável string numa variável numérica.

- Já foi dito que um arquivo-texto é um arquivo seqüencial, por isso é necessário sabermos quando o fim de arquivo foi atingido. A quantidade de linhas gravadas pode ser controlada pelo programador, através de técnicas, porém, o modo mais prático para se descobrir o fim do arquivo é usado a função booleana EOF, como foi aplicado no exemplo acima.

A função EOF retorna true se o ponteiro estiver no final do arquivo. Caso contrário, a função retornará false, indicando que o fim do arquivo não foi atingido.

APÊNDICE A – ERROS DE COMPILAÇÃO

Este apêndice apresenta uma relação dos 170 possíveis erros de compilação do Turbo Pascal.

ERRO	DESCRIÇÃO	TRADUÇÃO
1	Out of memory	Não existe espaço em memória
2	Identifier expected	Identificador esperado
3	Unknown identifier	Identificador desconhecido
4	Duplicate identifier	Identificador já existente
5	Syntax error	Erro de sintaxe
6	Error in real Constant	Erro na constante real
7	Error in integer Constant	Erro na constante inteira
8	String Constant exceeds line	Constante string excede a linha
9	Too many nested files	Muitos arquivos aninhados
10	Unexpected end of file	Fim de arquivo não esperado
11	Line too long	Linha muito longa
12	Type identifier expected	Espera a identificação do tipo
13	Too many open files	Muitos arquivos abertos simultaneamente
14	Invalid file name	Nome de arquivo inválido
15	File not found	Arquivo não encontrado
16	Disk full	Disco cheio
17	Invalid file name	Diretiva de compilação inválida
18	Too many files	Excesso de arquivos
19	Undefined type in pointer def	Ponteiro nunca antes declarado
20	Variable identifier expected	Identificador de variável esperado
21	Error in type	Erro de tipo
22	Structure too large	Estrutura muito larga
23	Set base type out of range	Faixa de valores inválida para a faixa
24	File components may not be files or objects	Componentes do arquivo não devem ser arquivos ou objetos
25	Invalid string length	Tamanho de string inválido
26	Type mismatch	Tipos misturados / incompatível
27	Invalid subrange base type	Faixa de valores inválida
28	Lower bound > than upper bound	Limite inferior > limite superior
29	Ordinal type expected	Tipo escalar esperado
30	Integer Constant expected	Constante inteira esperada
31	Constant expected	Constante esperada
32	Integer or real Constant expected	Constante inteira ou real esperada
33	Pointer type identifier expected	Identificador de tipo ponteiro esperado
34	Invalid function result type	Tipo de resultado da função inválido
35	Label identifier expected	Identificador Label esperado
36	Begin expected	Begin esperado
37	End expected	End esperado
38	Integer expression expected	Expressão inteira esperada
39	Ordinal expression expected	Expressão ordinal esperada
40	Boolean expression expected	Expressão booleana esperada
41	Operand types do not match	Operando incompatível com o operador
42	Error in expression	Erro na expressão
43	Illegal assignment	Associação ilegal
44	Field identifier expected	Identificador de campo esperado
45	Object file too large	Arquivo objeto muito grande
46	Undefined extern	Extern indefinido
47	Invalid object file Record	Objeto de registro de arquivo inválido
48	Code segment too large	Segmento de código muito grande
49	Data segment too large	Segmento de dados muito grande
50	Do expected	Do esperado
51	Invalid public definition	Definição public inválida
52	Invalid extern definition	Definição extern inválida

53	Too many extern definitions	Excesso de definições extern
54	Of expected	Of esperado
55	Interface expected	Interface esperada
56	Invalid relocatable reference	Referência relocável inválida
57	Then expected	Then esperado
58	To or Downto expected	To ou Downto esperado
59	Undefined forward	Forward indefinido
60	Too many procedures	Excesso de procedures
61	Invalid typecast	Tipo de resultado inválido
62	Division by zero	Divisão por zero
63	Invalid file type	Tipo de arquivo inválido
64	Cannot read or write vars of this type	Variáveis desse tipo não podem ser lidas / escritas
65	Pointer variable expected	Variável do tipo ponteiro esperada
66	String variable expected	Variável do tipo String esperada
67	String expression expected	Expressão string esperada
68	Circular unit reference	Referência circular da unit
69	Unit name mismatch	Nome da unit incompatível
70	Unit version mismatch	Versão da unit incompatível
71	Duplicate unit name	Nome da unit duplicada
72	Unit file format error	Erro no formato do arquivo unit
73	Implementation expected	Implementation esperado
74	Constant and case types don't match	Constante e expressão do case incompatíveis
75	Record variable expected	Variável do tipo Record esperada
76	Constant out of range	Constante fora de faixa
77	File variable expected	Variável de arquivo esperada
78	Pointer expression expected	Expressão tipo ponteiro esperada
79	Integer or real expression expected	Expressão inteira ou real esperada
80	Label not within current block	Label for a do bloco atual
81	Label already defined	Label já foi definido
82	Undefined label in stmt part	Label não declarado
83	Invalid @@ argument	Argumento @@ inválido
84	Unit expected	Unit esperado
85	";" expected	";" esperado
86	":" expected	":" esperado
87	"," expected	"," esperado
88	"(" expected	"(" esperado
89	")" expected	")" esperado
90	"=" expected	"=" esperado
91	":=" expected	":=" esperado
92	"[" or "(" expected	"[" or "(" esperado
93	"]" or ")" expected	"]" or ")" esperado
94	"." Expected	"." Esperado
95	".." expected	".." esperado
96	Too many variables	Variáveis demais
97	Invalid for control variable	Variável de controle loop for inválida
98	Integer variable expected	Variável do tipo integer esperada
99	Files and procedure types are not allowed here	Arquivos e procedimentos não permitidos aqui
100	String length mismatch	Comprimento da string incompatível
101	Tamanho da String incompatível	
102	Erro de ordenação	
103	Constante String esperada	
104	Variável Integer ou real esperada	
105	Variável ordinal esperada	
106	Erro em INLINE	
107	Expressão character esperada	
108	Excesso de itens relocados	
109	Estouro em operação aritmética	
110	Não pode ser utilizado em FOR, WHILE ou REPEAT	
112	Estouro da tabela de informação de Debug	
113	Constante CASE fora da faixa	

114	Erro na sentença
115	Procedimento de interrupção não pode ser chamado
116	A diretiva de compilação 8087 deve estar ativada
117	Endereço de destino não encontrado
118	Inclusão de arquivos não permitida neste ponto
119	Método de hierarquia não pode ser utilizado aqui
121	Qualificação inválda
122	Referência de variável inválida
123	Excesso de símbolos
124	Parte de comando muito grande
126	Arquivos devem ter parâmetro var
127	Excesso de símbolos condicionais
128	Diretiva condicional em local não apropriado
129	Falta diretiva endif
130	Erro na definição condicional
131	Cabeçalho incompatível com definição anterior
132	Erro crítico em disco
133	Não é possível avaliar esta expressão
134	Expressão terminada incorretamente
135	Formato especificado inválido
136	Referência indireta inválida
137	Não são permitidas variáveis estruturadas neste local
138	Não pode avaliar falta da Unit System
139	Não pode acessar esta símbolo
140	Operação com ponto flutuante inválida
141	Não pode compilar overlays em memória
142	Espera variável procedure ou function
143	Referência a procedure ou function inválida
144	Não pode tornar esta unit em overlay
145	Excesso de aninhamento
146	Erro em acesso a arquivo
147	Tipo object esperado
148	Objeto local não é permitido
149	Virtual esperado
150	Identificador method esperado
151	Virtual constructors não são permitidos
152	Identificador constructor esperado
153	Identificador destructor esperado
154	Um único Fail dentro de constructors
155	Operação inválida entre operando e operadores
156	Referência de memória esperada
157	Não pode somar ou subtrair em símbolos relocáveis
158	Combinação inválida no registro
159	Instruções do 286/287 não habilitadas
160	Referência a símbolo inválida
161	Erro na geração do código
162	ASM esperado
163	Método dinâmico com índice duplicado
164	Identificador de resourse duplicado
165	Índice export duplicado ou inválido
166	Procedimento ou função esperado
167	Símbolo não exportável
168	Nome de export inválido
169	Cabeçalho de arquivo executável muito grande
170	Excesso de segmentos

APÊNDICE B – ERROS DE EXECUÇÃO

Este apêndice apresenta uma relação dos principais erros em tempo de execução (run-time errors)do Turbo Pascal.

ERRO	DESCRIÇÃO	TRADUÇÃO
1	Invalid function number	Função numérica inválida
2	File not found	Arquivo não encontrado
3	Path not found	Caminho não encontrado
4	Too many open files	Muitos arquivos abertos
5	File Access denied	Acesso ao arquivo negado
6	Invalid file handle	Arquivo handle inválido
12	Invalid file Access code	Código de acesso de arquivo inválido
15	Invalid drive number	Número do drive inválido
16	Cannot remove current directory	Impossível remover diretório atual
17	Cannot rename across drives	Impossível renomear diretório
18	Não há mais arquivos	
100	Disk read error	Erro de leitura do disco
101	Disk write error	Erro de gravação do disco
102	File not assigned	Arquivo não assinalado
103	File not open	Arquivo não aberto
104	File not open for input	Arquivo não está aberto para entrada
105	File not open for output	Arquivo não está aberto para saída
106	Invalid numeric format	Formato numérico inválido
150	Disk is write-protect	Disco protegido
151	Erro interno do dispositivo DOS	
152	Drive not ready	Drive não pronto para leitura
154	CRC error in data	Erro de CRC nos dados
156	Disk seek error	Erro de pesquisa no disco
157	Unknown media type	Tipo de mídia desconhecida
158	Sector not found	Setor não encontrado
159	Printer out of paper	Impressora sem papel
160	Device write fault	Falha no dispositivo de escrita
161	Device read fault	Falha no dispositivo de leitura
162	Hardware failure	Falha no hardware
200	Division by zero	Divisão por zero
201	Range check error	Erro de faixa
202	Stack overflow error	Erro de estouro de pilha
203	Heap overflow error	Erro de estouro de heap
204	Invalid pointer operation	Operação de ponteiro inválida
205	Floating point overflow	Estouro de ponto flutuante
207	Invalid floating point operation	Operação de ponto flutuante inválida
208	Overlay manager not installed	Gerenciador de overlay não instalado
209	Overlay file read error	Erro na leitura de arquivo overlay
210	Object not initialized	Objeto não inicializado
211	Call to abstract method	Chamada a um método abstrato
213	Collection index out of range	Índice de coleção fora da faixa
214	Collection overflow error	Erro de estouro da coleção
215	Arithmetic overflow error	Erro de estouro aritmético
216	General protection fault	Falha de proteção geral

APÊNDICE C – PALAVRAS RESERVADAS

Este apêndice apresenta uma relação das palavras reservadas do Turbo Pascal.

Absolute: declara uma variável em um endereço absoluto de memória.

And: permite a operação matemática "AND" entre bit's, ou lógica entre operandos, sendo verdadeira somente quando ambos forem verdadeiros.

Asm: permite acessar a linguagem assembler dentro do programa.

Array: utilizado para definir uma tabela.

Begin: marca o início de um bloco.

Case: é uma expressão que define um seletor de opções.

Const: define a área onde são definidas as constantes do programa.

Constructor: é uma forma especial de inicializar uma estrutura objeto pelo método virtual.

Destructor: é uma forma de liberar memória utilizada por uma estrutura objeto.

Div: expressão aritmética que executa uma divisão inteira.

Do: é uma palavra reservada usada em três estruturas: WHILE, FOR e WITH.

Downto: é utilizado em um laço FOR para decrementar o passo.

Else: utilizado nos comandos IF e CASE.

End: utilizado para terminar um bloco, ou o comando CASE, ou ainda um registro.

Export: cria uma lista de procedimento e funções a serem exportados para uma biblioteca DLL.

External: indica se um procedimento ou função será compilado em separado.

File: identifica uma variável do tipo arquivo, pode ser tipada ou não.

For: laço de repetição controlado por variáveis.

Forward: permite informar ao compilador que uma determinada rotina será declarada após a sua chamada.

Function: é uma rotina do programa que retorna um determinado valor de acordo com o tipo.

Goto: desvio incondicional do programa para um determinado parágrafo.

If: estrutura de condição.

Implementation: define como públicas algumas rotinas de uma unidade (unit).

In: operador lógico de conjunto.

Inherited: permite que se use a notação de seu antecessor para um tipo objeto.

Inline: permite que se escreva diretamente em código de máquina.

Interface: declaração de que aquela área é pública.

Interrupt: declara uma rotina como sendo de interrupção para o compilador.

Label: declaração de parágrafos (rótulos) utilizados em um programa.

Library: inicializa o cabeçalho de uma biblioteca DLL.

Mod: operador aritmético que retorna o módulo de uma divisão, o resto.

Nil: constante nula do tipo ponteiro "pointer", compatível com qualquer tipo ponteiro.

Not: permite a operação matemática "NOT" entre bit's, invertendo-os, ou lógica, invertendo o resultado. Verdadeiro quando operando falso, e falso quando operando verdadeiro.

Object: é uma estrutura de dados que pode ter, além dos campos normais de um registro, também "procedures" e "functions".

Of: é utilizada em conjunto com as declarações: ARRAY, SET, FILE e o comando CASE.

Or: permite a operação matemática "OR" entre bit's, ou lógica entre operandos, sendo verdadeira quando pelo menos um dos operandos for verdadeiro.

Packet: prefixo para tabelas, não tem função no Turbo.

Procedure: é a declaração de uma sub-rotina, permitindo que sejam passados parâmetros e tem a mesma estrutura de um programa.

Program: identifica o cabeçalho de um programa.

Record: define um tipo especial que pode conter diversos campos com tipos diferentes.

Repeat: comando que controla um laço de comandos, que será executado até que a expressão seja verdadeira.

Set: permite a construção de conjuntos tipados e com valores ordinais, podendo ter de 0 até 255 valores, de qualquer tipo.

Shl: operador aritmético que permite a rotação n vezes à esquerda dos bit's de uma variável numérica x.

Shr: operador aritmético que permite a rotação n vezes à direita dos bit's de uma variável numérica x.

String: é a declaração de uma seqüência de caracteres que pode ter uma quantidade, variando entre 1 e 255.

Then: utilizado em conjunto com o comando IF.

To: utilizado em conjunto com o comando FOR.

Type: declaração dos tipos definidos pelo usuário, podendo ser mencionado no programa.

Unit: declaração de unidade que permite a criação de um módulo do programa, para facilitar a criação de programas muito grandes, ou ainda, na geração de bibliotecas próprias.

Until: utilizado em conjunto com o comando REPEAT.

Uses: declaração das unidades, UNIT, utilizadas em um programa ou unidade.

Var: área de declaração de variáveis de um programa, unidade ou rotinas.

Virtual: repete um determinado bloco enquanto a expressão for verdadeira.

With: relaciona as variáveis do bloco com as variáveis do tipo registro.

Xor: permite a operação matemática "XOR" entre bit's, ou lógica entre operandos, sendo verdadeira quando somente um dos operandos for verdadeiro.

APÊNDICE D – TECLAS DE FUNÇÃO

Este apêndice apresenta uma descrição de todas as teclas de função do Turbo Pascal.

TECLA	FUNÇÃO
F1	Help sensitive.
F2	Salva o arquivo corrente no editor.
F3	Carrega um arquivo do disco.
F4	Executa o programa até a posição do cursor.
F5	Zoom, aumenta e diminui a janela ativa.
F6	Troca a janela ativa.
F7	Executa um programa passo a passo.
F8	Executa um programa rotina a rotina.
F9	Executa um “make” no programa.
F10	Vai para o Menu.
ALT-0	Chama a lista de arquivos carregados “pick”.
ALT-F1	Chama o último help.
ALT-F3	Fecha a janela ativa.
ALT-F5	Exibe a tela de usuário.
ALT-F9	Compila o seu programa.
ALT-C	Ativa o Menu Compile.
ALT-D	Ativa o Menu Debug.
ALT-E	Ativa o editor.
ALT-F	Ativa o Menu File.
ALT-H	Ativa o Menu Help.
ALT-O	Ativa o Menu Options.
ALT-R	Ativa o Menu Run.
ALT-S	Ativa o Menu Search.
ALT-W	Ativa o Menu Window.
ALT-X	Sair do Turbo para o DOS.
CTRL-F1	Ativa o help, usando como chave a palavra que estiver sobre o cursor.
CTRL-F2	Termina uma sesão de debug.
CTRL-F3	Mostra a lista de “stack” que estiver sendo depurada.
CTRL-F4	Exibe o valor de uma variável e permite que a mesma tenha seu valor alterado.
CTRL-F7	Adiciona uma expressão na “watch window”.
CTRL-F8	Troca o “breakpoint”.
CTRL-F9	Executa o programa que estiver na tela de edição.
CTRL-DEL	Remove o texto seleccionado.
CTRL-L	Repete a última troca ou busca de texto.
CTRL-INS	Copia o texto seleccionado para o “clipboard”.
SHIFT-F1	Mostra o índice do help.
SHIFT-F5	Configurável pelo usuário para “tool”.
SHIFT-F6	Troca a janela ativa.
SHIFT-F10	Configurável pelo usuário para “tool”.
SHIFT-DEL	Remove o texto selecionado para o “clipboard”.
SHIFT-INS	Coloca no texto o conteúdo do “clipboard”.

APÊNDICE E – GRÁFICOS NOS PASCAL

Este apêndice apresenta uma descrição de como se trabalhar com gráficos no Turbo Pascal.

---

1. Introdução

No início de todos os programas gráficos existirão certos passos que devem ser seguidos. Por exemplo, temos que ajustar o hardware de vídeo para o modo gráfico. Além disso, temos de restaurar a tela do computador para o modo normal de exibição no final de todos os programas gráficos. Na seção abaixo, veremos os componentes básicos necessários em um programa gráfico típico desenvolvido em Pascal.

2. Inicialização da BGI

O primeiro passo que todos os programas gráficos devem dar é inicializar o hardware gráfico. Por hardware entendemos a placa do adaptador de vídeo que deve estar presente no PC. Existem diversas placas gráficas diferentes disponíveis para o PC, cada qual com uma variedade de modos gráficos. A BGI (Borland Graphic Interface) oferece suporte a muitos destes "modos". Utilizaremos o modo default da rotina de inicialização da BGI, o que por sua vez nos permitirá ajustar o adaptador gráfico instalado no seu modo de maior resolução.

A procedure da BGI usada para configurar o seu computador no modo gráfico chama-se **InitGraph** e possui a seguinte declaração: ***procedure*** *InitGraph(var GraphDriver, GraphMode: integer; DriverPath: string);*

A procedure **InitGraph** está definida na unidade **Graph** do Pascal que deve ser incluída na seção **uses** de todos os programas gráficos. Este arquivo contém as estruturas de dados, constantes, procedures e funções da BGI.

Os dois primeiros parâmetros especificados em initgraph identificam a placa de vídeo instalada e o modo gráfico a ser utilizado. O terceiro parâmetro especifica o path onde se encontram os drivers para gráfico do Turbo Pascal. (Existe pelo menos um driver gráfico para cada adaptador gráfico). Os arquivos dos drivers são os que terminam todos com a extensão **.BGI.**

No próximo exemplo, **InitGraph** pesquisa o diretório atual para encontrar os arquivos de drivers:
InitGraph*(Gdriver, Gmode, ' ');*

1.1. Escrita de um Programa Básico Usando a BGI

O programa a seguir é provavelmente o menor programa gráfico que podemos escrever em Turbo Pascal.

Este programa é apenas para enfatizar a estrutura básica de um programa gráfico típico em Pascal.

Podemos usá-lo como esqueleto para nossos próprios programas gráficos.

```
Program SmallestGraphicsProgram;
Uses Graph;    { unidade gráfica do Turbo Pascal }
Const
  Gdriver: integer = Detect;
Var
  Gmode: integer;
Begin
  InitGraph(Gdriver, Gmode, 'c:\bp\bgi');   { Inicializa o modo Gráfico }
        { Coloque os comandos para desenho nesta área !! }
  CloseGraph;   { Finaliza o Modo Gráfico }
End.
```

A tabela abaixo demonstra os drivers e suas respectivas resoluções

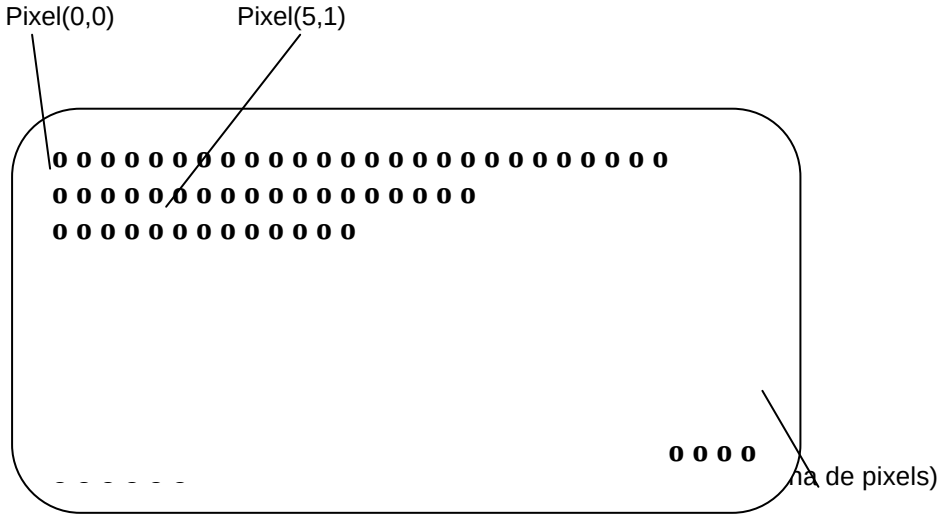
Valor	Resolução	Valor	Resolução
CGAC0	320x200	EGALo	640x200
CGAC	320x200	EGAHi	640x350
CGAC2	320x200	EGA64Lo	640x200
CGAC3	320x200	EGA64Hi	640x350
CGAHi	640x200	ATT400C0	320x200
MCGAC0	320x200	ATT400C1	320x200
MCGAC1	320x200	ATT400C2	320x200
MCGAC2	320x200	ATT400C3	320x200
MCGAC3	320x200	ATT400Med	640x200
MCGAMed	640x200	ATT400Hi	640x400
MCGAHi	640x480	IBM8514Lo	640x480
EGAMonoHi	640x350	IBM8514Hi	1024x768
HercMonoHi	720x348	PC3270Hi	720x350
VGALo	640x200	VGAHi	640x480
VGAMed	640x350		

1.2. Trabalhando com Coordenadas

Nos gráficos, acompanhamos as nossas posições usando *coordenadas de pixels*. Os pixels são pontos muitos pequenos que se vê na tela e que podem ser endereçados usando um sistema de coordenadas familiar à forma por que acessamos as linhas e colunas de uma estrutura de dados bidimensional.

O Turbo Pascal apresenta duas funções que obtém o número máximo de pixels tanto no eixo x (**GetMaxX)** quanto no eixo y (**GetMaxY**).

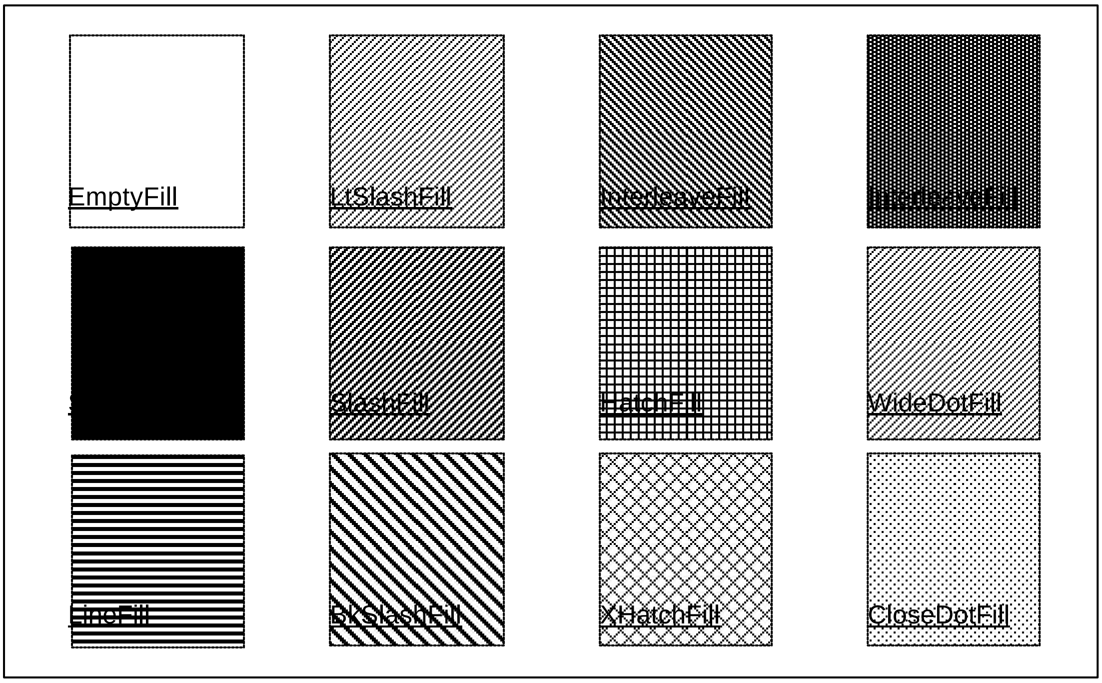
1.3. Endereçamento por linha e coluna da tela



3. Padrões de Preenchimento Pré-Definidos

Constante	Valor	Descrição
EmptyFill	0	Preenche com a cor de fundo
SolidFill	1	Preenche completamente com a cor de preenchimento
LineFill	2	Preenche usando linhas horizontais
LtSlashFill	3	Preenche usando linhas finas inclinadas
SlashFill	4	Preenche usando linhas espessas inclinadas
BkSlashFill	5	Preenche usando linhas espessas inclinadas da direita p/ a esquerda
LtBkSlashFill	6	Preenche usando linhas finas inclinadas da direita p/ a esquerda
HatchFill	7	Preenche usando um padrão hachurado leve
XhatchFill	8	Preenche usando um padrão xadrez espesso
InterleaveFill	9	Preenche usando um padrão formado por linhas intercaladas
WideDotFill	10	Preenche usando pontos largamente espaçados
CloseDotFill	11	Preenche usando pontos bastante próximos

4. Padrões de preenchimento pré-definidos



5. Estilos de Linhas

Constante	Valor
SolidLn	0
DottedLn	1
CenterLn	2
DashedLn	3
UserBitLn	4

	NormalWidth	ThickWidth
Solidln		
DottedLn		
CenterLn		
DashedLn		