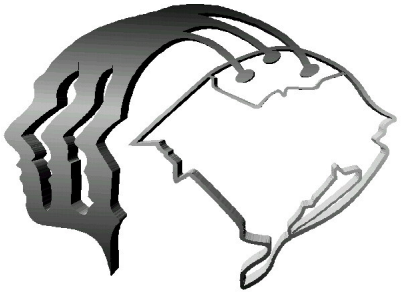



UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES

CAMPUS ERECHIM

DEPARTAMENTO DE ENGENHARIAS E CIÊNCIA DA COMPUTAÇÃO

CURSO DE CIÊNCIA DA COMPUTAÇÃO


# Linguagem de Programação II

## C, Pascal e Delphi


**Prof. Evandro Preuss (autor)**
preuss@fw.uri.br
http://www.uri.br/~preuss

**Prof. Neilor Tonin**
nat@uri.com.br
http://www.inf.uri.com.br/neilor



**1º Semestre/2003**

**SUMÁRIO**

# 1. DADOS

## 1.1 Elementos da Linguagem

Normalmente uma linguagem de programação possui dois tipos de elementos: os elementos definidos pela linguagem e os elementos definidos pelo próprio usuário:

### 1.1.1 Elementos definidos pela linguagem:

#### 1.1.1.1 Letras (alfanuméricas) - PASCAL e C:

A até Z
a até z

#### 1.1.1.2 Dígitos (numéricos) - PASCAL e C:

0 até 9

#### 1.1.1.3 Símbolos Especiais

Todas as linguagens possuem símbolos especiais que são diferentes em cada linguagem, mas que tem a mesma finalidade:

| Pascal | C | Significado |
|---|---|---|
| + | + | adição |
| - | - | subtração |
| * | * | multiplicação |
| / | / | divisão |
| = | = = | comp. igualdade |
| > | > | maior que |
| < | < | menor que |
| >= | >= | maior ou igual |
| <= | <= | menor ou igual |
| <> | != | diferente |
| := | = | atribuição |
| ( ) | ( ) | parênteses |
| { ou (* | /* | início de comentário |
| } ou *) | */ | final de comentário |
| ; | ; | separador |
| ' | " ou ' | demarca strings ou caracteres |

#### 1.1.1.4 Palavras Reservadas ou Palavras Chave

Palavras Reservadas são símbolos que possuem significado definido na linguagem, não podendo ser redefinidos ou usado como nome de identificador.

| PASCAL | | | C | | |
|---|---|---|---|---|---|
| and | array | begin | auto | break | case |
| case | const | div | char | const | continue |
| do | downto | else | default | do | double |
| end | file | for | else | enum | extern |
| function | goto | if | float | for | goto |
| in | label | mod | if | int | long |
| nil | not | of | register | return | short |
| or | packed | procedure | signed | sizeof | static |
| program | record | repeat | struct | switch | typedef |
| set | then | to | union | unsigned | void |
| type | until | var | volatile | while | |
| while | with | | | | |

Na linguagem **C**, o restante dos comandos são todos funções (da biblioteca padrão ou não). Todas as palavras reservadas devem ser escritas em minúsculo.

A linguagem Pascal tem ainda alguns identificadores predefinidos pela linguagem conhecidos como **Identificadores Padrão**. Podem ser constantes, tipos, variáveis ou subprogramas (procedimentos ou funções) e podem ser escritos tanto em minúsculo como em maiúsculo:

| | | | | | |
|---|---|---|---|---|---|
| abs | arqtan | boolean | char | chr | cos |
| eof | eoln | exp | false | input | integer |
| ln | maxint | odd | ord | output | pred |
| read | readln | real | reset | rewrite | run |
| sin | sqr | sqrt | str | succ | text |
| true | trunc | write | writeln | | |

#### 1.1.1.5 Delimitadores

Os elementos da linguagem (identificadores, números e símbolos especiais) devem ser separados por pelo menos um dos seguintes delimitadores: branco, final de linha ou comentário.

### 1.1.2 Elementos definidos pelo Usuário

#### 1.1.2.1 Identificadores

Um identificador é um símbolo definido pelo usuário que pode ser um rótulo (label), uma constante, um tipo, uma variável, um nome de programa ou subprograma (procedimento ou função).

Os identificadores normalmente devem começar com um caractere alfabético e não pode conter espaços em branco.

O número máximo de caracteres que podem format o identificador varia de compilador para compilador. No PASCAL padrão somente os 8 primeiros caracteres são válidos; no TURBO PASCAL pode-se usar identificadores de até 127 caracteres sendo todos significativos e não há distinção entre maiúsculas e minúsculas.

No **C** somente os 32 primeiros caracteres são significativos e há diferença entre maiúsculas e minúsculas. Em **C** Cont é diferente de cont que é diferente de CONT.

#### 1.1.2.2 Comentários

Os comentários não tem função nenhuma para o compilador e serve apenas para aumentar a legibilidade e clareza do programa.

#### 1.1.2.3 Endentação

A endentação também não tem nenhuma função para o compilador e serve para tornar a listagem do programa mais clara dando hierarquia e estrutura ao programa.

### 1.2 Tipos de Dados

Um Tipo de Dado define o conjunto de valores que uma variável pode assumir e as operações que podem ser feitas sobre ela.Toda variável em um programa deve ser associada a um e somente um tipo de dado. Esta associação é feita quando a variável é declarada na parte de declaração de variáveis do programa.

#### 1.2.1 Tipos predefinidos pela linguagem

| PASCAL | | |
|---|---|---|
| **tipo** | **intervalo de representação** | **tamanho** |
| Shortint | -128 a 127 | 1 byte |
| Byte | 0 a 255 | 1 byte |
| Integer | -32.768 a 32.767 | 2 bytes |
| Word | 0 a 65.535 | 2 bytes |
| Longint | -2.147.483.648 a 2.147.483.647 | 4 bytes |
| Real | $2.9 \times 10^{-39}$ a $1.7 \times 10^{38}$ | 6 bytes |
| single * | $1.5 \times 10^{-45}$ a $3.4 \times 10^{38}$ | 4 bytes |
| double * | $5.0 \times 10^{-324}$ a $1.7 \times 10^{308}$ | 8 bytes |
| Extended * | $1.9 \times 10^{-4951}$ a $1.1 \times 10^{4932}$ | 10 bytes |
| comp * | $-9.2 \times 10^{18}$ a $9.2 \times 10^{18}$ | 8 bytes |
| Char | os caracteres da tabela ASCII | 1 byte |
| Boolean | TRUE ou FALSE | 1 byte |
| String | tipo estruturado composto por um conjunto de elementos tipo char | quantidade de caracteres x 1 byte |

* os tipos assinalados somente podem ser utilizados em máquinas com co-processador matemático (8087, 80287, 80387, 80487) ou com chip processador 80486 DX ou superior.

| DELPHI | | |
|---|---|---|
| **tipo** | **intervalo de representação** | **tamanho** |
| Shortint | -128 a 127 | 1 byte |
| Byte | 0 a 255 | 1 byte |
| Smallint | -32.768 a 32.767 | 2 bytes |
| Word | 0 a 65535 | 2 bytes |
| integer ou longint | -2.147.483.648 a 2.147.483.647 | 4 bytes |
| cardinal ou longword | 0 a 4294967295 | 4 bytes |
| int64 | $-2^{63}$ a $2^{63}$ | 8 bytes |
| real * | $2.9 \times 10^{-39}$ a $1.7 \times 10^{38}$ | 6 bytes |
| Single | $1.5 \times 10^{-45}$ a $3.4 \times 10^{38}$ | 4 bytes |
| double | $5.0 \times 10^{-324}$ a $1.7 \times 10^{308}$ | 8 bytes |
| Extended | $1.9 \times 10^{-4932}$ a $1.1 \times 10^{4932}$ | 10 bytes |
| Comp ** | $-2^{63}$ a $2^{63}$ | 8 bytes |
| Char | os caracteres da tabela ASCII | 1 byte |
| Boolean | TRUE ou FALSE | 1 byte |
| String | tipo estruturado composto por um conjunto de elementos tipo char | quantidade de caracteres x 1 byte |

* Apenas para manter compatibilidade com Pascal. Este tipo não é nativo para processadores Intel e as operações com este tipo são mais lentas que os demais.

** O mesmo que um inteiro de 64 bits (int64)

| C | | |
|---|---|---|
| **tipo** | **intervalo de representação** | **tamanho** |
| Char | -128 a 127 | 1 byte |
| Int | -32.768 a -32767 | 2 bytes |
| float | 3.4 E-38 a 3.4 E38 | 4 bytes |
| double | 1.7 E-308 a 1.7 E308 | 8 bytes |
| void | – | |
| **Modificadores de Tipo** | | |
| ***Modificador Long*** | | |
| **tipo** | **intervalo de representação** | **tamanho** |
| long int | -2.147.483.647 a 2.147.483.647 | 4 bytes |
| long double | 1.2 E-4932 a 1.2 E4932 | 10 bytes |
| ***Modificador Unsigned*** | | |
| **tipo** | **intervalo de representação** | **tamanho** |
| unsigned char | 0 a 255 | 1 byte |
| unsigned int | 0 a 65.535 | 2 bytes |
| unsigned long int | 0 a 4.294.967.295 | 4 bytes |

### 1.2.2 Tipos definidos pelo usuário

Os tipos definidos pelo usuário são aqueles que usam um grupo de tipos predefinidos ou um subgrupo de algum tipo. Este tipo é chamado de tipo enumerado de dados e representa uma escolha dentre um pequeno número de alternativas.

#### 1.2.2.1 Tipo enumerado discreto

Em Pascal temos o comando **TYPE** para definir o tipo de dados que queremos:

```
TYPE tpdias = (segunda, terça, quarta, quinta, sexta,
     sábado, domingo);

VAR  diasem: tpdias;
     diasem:= segunda;  if (diasem = terça) then ...
```

Em C e C++ temos o comando **enum**

```
enum tpdias { segunda, terça, quarta, quinta, sexta,
     sábado, domingo } diasem
```

#### 1.2.2.2 Tipo enumerado contínuo

O tipo enumerado contínuo pode ser definido como um intervalo de um tipo enumerado discreto já definido ou de um tipo padrão.

```
TYPE tpdias = (segunda, terça, quarta, quinta, sexta, sábado, domingo);
     TYPE tpfimsem = sábado..domingo;
TYPE tpdiautil = segunda..sexta;
VAR  fimsem: tpfimsem;
     fimsem:= sábado;  if (fimsem = domingo) then ...
```

### 1.3 Conversões de tipos de dados

As operações que usam comandos que envolvem variáveis de diferentes tipos são chamadas de operações de modo misto.

Ao contrário das outras linguagens, C e C++ executam conversões automáticas de dados para um tipo maior ou trunca o valor para um tipo menor.

Devemos ter cuidado quando usamos operações de modo misto em C ou C++, pois os valores podem ser truncados, enquanto que no Pascal devemos prever exatamente o tipo da variável que necessitamos, por exemplo:

| **PASCAL:** | **C:** | **C:** | **C:** |
|---|---|---|---|
| `a, b:integer;` | `int a, b;` | `float a, b;` | `int a;` |
| `c: real;` | `float c;` | `int c;` | `float b,c;` |
| `a:=5;` | | `a=5;` | `a=5;  a=5;` |
| `b:=3;` | | `b=3;` | `b=3;  b=3;` |
| `c:=a/b;` | `c=a/b;` | `c=a/b;` | `c=a/b;` |
| `c = 1,66666667` | `c=1,000000` | `c=1` | `c=1,666667` |

A linguagem C e C++ permitem a conversão temporária dos tipos de variáveis através do operador de conversão.

Sempre que você necessitar a mudar o formato de uma variável temporariamente, simplesmente preceda o identificador da variável com o tipo entre parênteses para aquele que quiser converter. Se utilizarmos o primeiro exemplo de C acima podemos obter o resultado esperado usando um operador de conversão ou ***cast***:

```
int a, b;
float c;
a=5;
b=3;
c=(float)a/b;

c=1,666667
```

### 1.4 Constantes e Variáveis

#### 1.4.1 Constantes

Constantes são valores declarados no início do programa e que não se alteram na execução do programa. Podem ser expessas em qualquer base, desde que seguidas algumas regras simples:

1. Constantes em octal na linguagem C devem sempre iniciar com um 0, como em **mem = 01777**

2. Constantes em hexa na linguagem C devem sempre iniciar com 0x ou 0X, como em **mem = 0x1FA**

3. Constantes em hexa na linguagem Pascal devem sempre iniciar com $, como em **mem = $1FA**

4. Constantes em decimal são escritas de modo convencional sem se inicarem com 0

#### 1.4.2 Variáveis

Uma declaração de variável consiste do nome do tipo seguido do nome da variável (em C e C++) ou do nome da variável seguido do nome do tipo (em Pascal).

Todas as variáveis devem ser declaradas antes de serem usadas. As variáveis devem ser declaradas no início de cada função, procedimento ou início do programa. Não podem ocorrer declarações de variáveis após a primeira sentença executável de uma rotina.

As variáveis podem ser globais ou locais.

Variáveis globais são declaradas fora de qualquer função, valem em qualquer ponto do código, são inicializadas com zero automaticamente e uma única vez, são armazenadas na memória.

Variáveis locais são declaradas dentro das funções, existem apenas enquanto a função na qual foi declarada está ativa. Dentro de funções variáveis locais com mesmo nome de variáveis globais tem preferência, não são inicializadas automaticamente, ou seja, deve-se informar um valor inicial, são alocadas na pilha ("stack").

Os parâmetros das funções são tratados como se fossem variáveis locais, são inicializados com o valor passado na chamada, são declarados na lista de parâmetros, são passados por valor, somente tipos escalares de dados podem ser parâmetros.

### 1.4.3 Classes de armazenamento

Em C e C++ as variáveis tem diferente classes de armazenamento:

**auto:** variável automática. É criada quando uma rotina é chamada e destruída quando a rotina termina. É a classe default para variáveis locais a funções.

**register:** pede ao compilador para colocar uma variável em registrador. O compilador pode, ou não, seguir a "dica" dada pelo programador. É muito comum seu uso em variáveis de controle de laços "for". O operador endereço (&) não pode ser aplicada a uma variável de classe register.

**static:** é o contrário da variável "auto". Uma variável "static" é alocada na área de dados e sempre existe, mantendo seu conteúdo entre as chamadas de uma rotina.

```
ex:  void teste()
     {
     static int x=0;
     x++;
     }
```

Neste caso a inicialização somente será feita na primeira evocação da rotina. O valor de "x" sobreviverá de uma chamada para outra da rotina (cada vez que for chamada, "x" aumentará de valor).

Se uma variável global possui o atributo "static" significa que ela somente poderá ser usada no arquivo onde está declarada, sendo invisível a outros arquivos que componham o sistema.

**extern:** significa que a variável está declarada em outro arquivo, onde sua área é alocada. É utilizada para variáveis globais a diferentes arquivos componentes de um mesmo projeto (programa).

**volatile:** informa ao compilador para não otimizar o uso de uma variável colocando-a em registrador. É utilizado quando uma variável pode ser atualizada concorrentemente por mais de um processo.

## 1.5 Operadores

### 1.5.1 Operadores aritméticos

| PASCAL e DELPHI | | C | |
|---|---|---|---|
| + | Soma | + | soma |
| – | Subtração | – | subtração |
| * | Multiplicação | * | multiplicação |
| / | divisão | / | divisão |
| MOD | resto da divisão inteira | % | resto da divisão inteira |
| DIV | inteiro da divisão | | |

### 1.5.2 Operadores de atribuição

Em Pascal e Delphi temos o operador de atribuição:

**:=**

Em C e C++ temos os seguintes operadores de atribuição:

**=  +=  –=  *=  /=  %=**

```
Ex:  i=2;           -> atribui o número 2 à variável i
     i+=4;               -> i=i+4;
     x *= y+1;      -> x=x*(y+1);
     p %= 5;        -> p = p % 5;
```

### 1.5.3 Operadores relacionais e lógicos

Os operadores relacionais são operadores binários que devolvem os valores lógicos verdadeiro e falso.

| PASCAL e DELPHI | | C | |
|---|---|---|---|
| > | maior que | > | maior que |
| < | menor que | < | menor que |
| >= | maior ou igual | >= | maior ou igual |
| <= | menor ou igual | <= | menor ou igual |
| = | igual | = = | igual |
| <> | diferente | != | diferente |

Os operadores lógicos são usados para combinar expressões relacionais. Também devolvem como resultado valores lógicos verdadeiro ou falso.

| PASCAL e DELPHI | | C | |
|---|---|---|---|
| and | e | && | e |
| or | ou | \|\| | ou |
| not | não | ! | não |
| xor | ou exclusivo | ^ | ou exclusivo |

Uma expressão relacional ou lógica em C ou C++, retornará zero para o valor lógico **falso** e um para o valor lógico **verdade**. No entanto, qualquer valor diferente de zero será considerado um valor **verdade** quando inserido em uma expressão lógica.

### 1.5.4 Operadores bit a bit

Em C e C++ temos os operadores bit a bit. São operadores capazes de alterar os valores dos bits de uma variável. Funcionam apenas com os tipos **char** e **int**.

| PASCAL e DELPHI | C | OPERAÇÃO |
|---|---|---|
| and | & | e |
| or | \| | ou |
| not | ~ | não |
| xor | ^ | ou exclusivo |
| SHR | >> | shift para direita (divisão por 2) |
| SHL | << | shift para esquerda (multiplicação por 2) |

Exemplos:

***C*** **Pascal**

char a, b, c; *a, b, c: byte*

```
a=1; b=3;          a:=1; b:=3
```

| | |
|---|---|
| (C) c = a & b  00000001<br>& 00000011<br>00000001<br>*(Pascal) c:= a and b* | (C) c = a \| b  00000001<br>\| 00000011<br>00000011<br>*(Pascal) c:= a or b* |
| (C) c = ~a  ~ 00000001<br>11111110<br>*(Pascal) c:= not a* | (C) c = a ^ b  00000001<br>^ 00000011<br>00000010<br>*(Pascal) c:= a xor b* |
| (C) c = b >> 1  00000011<br>00000001<br>*(Pascal) c:= b shr 1* | (C) c = b << 1  00000011<br>00000110<br>*(Pascal) c:= b shl 1* |

### 1.5.5 Operadores de incremento e decremento

Em C e C++ também temos os operadores de incremento e decremento:

**++** incremento de um

**--** decremento de um

Escrever "m++" ou "++m" quando estes se encontram isolados em uma linha não faz diferença. Quando estiverem sendo usados em conjunto com uma atribuição, entretanto:

Ex:

```
int m, n;
m = 5; n = 4;

m = n++;                m = ++n;

Res:  m = 4             m = 5
      n = 5             n = 5
```

Obs.: A declaração: **`printf`**`("%d %d %d",n,n++,n+1);`

está correta, porém o resultado pode variar de acordo com o compilador dependendo da ordem em que os parâmetros são retirados da pilha (stack).

### 1.5.6 Operador Condicional

O operador condicional ternário pode ser utilizado em C e C++ quando o valor de uma atribuição depende de uma condição. O operador ternário é sibolizado pelo operador:

**?**

Exemplo:

```
if (x>3) k = k + 1;
else k = s - 5;
```

pode ser substituído por:

```
k=(x>3)? k+1: s-5;
```

### 1.5.7 Operador Vírgula

Em C e C++ o operador vírgula avalia duas expressões onde a sintaxe permite somente uma. O valor do operador vírgula é o valor da expressão à direita. É comumente utilizado no laço **for**, onde mais de uma variável é utilizada. Por exemplo:

```
for(min=0, max=compr-1; min < max; min++, max--)
     {
     ...
     }
```

## 2. ESTRUTURA DO PROGRAMA

### 2.1 Estrutura de um Programa em Pascal

Normalmente um programa Pascal possui três partes distintas: Identificação do programa, Bloco de declarações e Bloco de comandos.

#### 2.1.1 Identificação do programa

A identificação ou cabeçalho do programa em Pascal dá um nome ao programa e lista seus parâmetros. É a primeira linha do programa.

PROGRAM <identificação> ( <lista de parâmetros> );

#### 2.1.2 Bloco de Declarações

Todo identificador definido pelo usuário em um programa deve ser declarado antes de referenciado (usado), caso contrário o compilador acusará um erro na fase de compilação por desconhecer o identificador.

O Bloco de Declarações define todos os identificadores utilizados pelo Bloco de Comandos, sendo todos opcionais. Quando o Bloco de Delarações existir sempre estará antes do Bloco de Comandos.

Em Pascal o Bloco de Declarações é formado por cinco partes:

##### 2.1.2.1 Parte de Declarações de Rótulos:

Rótulos (labels) existem para possibilitar o uso do comando de desvio incondicional GOTO. Este comando comando gera a desestruturação do programa e não é aconselhado seu uso.

LABEL <rotulo1>, ... , <rotulo*n*>;

##### 2.1.2.2 Parte de Declarações de Constantes:

Define os identificadores que terão valores constantes dureante toda a esxecução do programa, podendo ser números, seqüências de caracteres (strings) ou mesmo outras constantes.

A declaração de constantes inicia pela palavra reservada CONST, seguida po uma seqüência de: identificador, um sinal de igual, o valor da constante e um ponto e vírgula:

CONST <identificador> = <valor> ;

##### 2.1.2.3 Parte de Declarações de Tipos:

Serve para o programadore criar seus próprios tipos de dados.

A declaração dos tipos é iniciadas pela palavra reservada TYPE, seguida de um ou mais identificadores separados por vírgula, um sinal de igual, um tipo e um ponto e vírgula:

TYPE <tipoident1> , ... , <tipoident*n*> = <tipo> ;

##### 2.1.2.4 Parte de Declarações de Variáveis:

Quando declaramos uma variãvel temos que definir, além do nome, seu tipo. O tipo especifica os valores que poderão ser atribuídos a esta variável. O tipo pode ser algum dos tipos predefinidos pela linguagem ou um tipo definido pelo usuário.

A declaração das variáveis é iniciada pela palavra reservada VAR, seguida de um ou mais identificadores, separados por vírgula, um tipo um ponto e vírgula:

VAR < ident1> , ... , <ident*n*> : <tipo> ;

#### 2.1.2.5 Parte de Declarações de Subprogramas:

Nesta parte são declarados e implementados os subprogramas (funções e procedimentos) e devem estar declarados e implementados antes da sua chamada em qualquer parte do programa principal ou de subprogramas.

### 2.1.3 Bloco de Comandos

O Bloco de Comandos é a última parte que compõe um programa em Pascal e especifica as ações a serem executadas sobre os objetos definidos no Bloco de Declarações.

O Bloco de Comandos é também conhecido como programa principal e é iniciado pela palavra reservada BEGIN seguida de por uma seqüência de comandos e finalizada pela palavra reservada END seguida um ponto.

## 2.2 Estrutura de um Programa em C

Normalmente um programa em C possui três partes distintas: Bloco de Diretivas de Compilação, Bloco de declarações e Bloco de Implementação

### 2.2.1 Bloco de Diretivas de Compilação

Como a linguagem C não possui nenhum comando de entrada e saída incorporado à linguagem, todas essas operações são realizadas através de funções que encontram-se nas bibliotecas da linguagem.

Para utilizar essas funções dentro do programa é necessário incluir o cabeçalho das funções no início do programa através da diretiva de compilação **#include**:

#include <stdio.h> /* inclui a biblioteca padrão de comandos de entrada e saída que se encontra no diretório padrão*/

#include “outros.h” /*inclui uma outra bliblioteca criada pelo usuário que
se encontra no diretório corrente */

Também nesse bloco podemos definir as macros para o nosso programa. Uma macro pode ser simplesmente a substituição de um texto como a implementação de uma pequena função, por exemplo:

```
#define MAXINT  32767

#define triplo(x) ((x)*3)

#define pqt Pressione Qualquer Tecla Para Continuar...
```

### 2.2.2 Bloco de Declarações:

No bloco das declarações são declaradas todas as variáveis globais, tipos definidos pelo usuário, estruturas, uniões e declaradas todas as funções (exceto a main) que estão implementadas no programa, através de um protótipo (cabeçalho) da mesma.

```
int soma(int x, int y);

int num, quant;

char nome[50];
```

### 2.2.3 Bloco de Implementação:

No bloco de implementações são implementadas todas as funções que compõem o programa.

Inicialmente se implementa a função principal, que é a primeira a ser executada e logo abaixo todas as demais funções.

```
void main()
        {
        printf("Olá mundo!");
        }
```

## 3. COMANDOS

### 3.1 Comandos Simples

Os comandos simples ou não estruturados, caracterizam-se por não possuírem outros comandos relacionados.

#### 3.1.1 Comandos de Entrada e Saída

##### 3.1.1.1 Comandos de E/S em Pascal

A linguagem Pascal tem definido os comandos de entrada e saída na própria linguagem.

###### 3.1.1.1.1 Read e Readln

Os comandos de Leitura (entrada) associam valores lidos do teclado ou de arquivos para as variáveis. São eles:

READ ( <var1>, ... , <var*n*> );

READLN ( <var1>, ... , <var*n*> );

###### 3.1.1.1.2 Write e Writeln

Os comandos de gravação (saída) transferem para os dispositivos de saída (disco, video ou impressora) os valores das variáveis.

Para mostrar valores no video temos os seguintes comandos:

WRITE ( <var1> , <´texto1´> , ... , < ´texto *n*´>, <var*n*>);

WRITELN ( <var1> , <´texto1´> , ... , < ´texto *n*´>, <var*n*>);

Para imprimir valores na impressora temos os seguintes comandos:

WRITE (LST, <var1> , <´texto1´> , ... , < ´texto *n*´>, <var*n*>);

WRITELN (LST, <var1> , <´texto1´> , ... , < ´texto *n*´>, <var*n*>);

Para gravar valores em arquivos temos os seguintes comandos:

WRITE (<arq>, <var1> , <´texto1´> , ... , < ´texto *n*´>, <var*n*>);

WRITELN (<arq>, <var1> , <´texto1´> , ... , < ´texto *n*´>, <var*n*>);

###### 3.1.1.1.3 Readkey

O comando READKEY lê um caracter do teclado e associa a uma variável do tipo char.

#### 3.1.1.2 Comandos de E/S em C

A linguagem C não possui nenhum comando de entrada e saída predefinido na linguagem. Todas as operações de E/S são realizadas por funções que encontram-se nas mais diversas bibliotecas. As principais funções são:

##### 3.1.1.2.1 A função "printf()"

A função "printf" é a função para saída formatada de dados e funciona da seguinte forma: o primeiro argumento é uma string entre aspas (chamada de string de controle) que pode conter tanto caracteres normais como códigos de formato que começam pelo caracter de porcentagem. Caracteres normais são apresentados na tela na ordem em que são encontrados. Um código de formato informa a função "printf" que um item não caracter deve ser mostrado. Os valores correspondentes encontram-se no segundo argumento (lista de argumentos).

SINTAXE: printf("<string de controle>",<lista de argumentos>);

Obs.: Além de códigos de formato e caracteres normais a string de controle pode conter ainda caracteres especiais iniciados pelo símbolo "\".

Exemplos:

```
printf("O preço é R$ %d,00",preco);
printf("São %d horas e %d minutos.", hora, minuto);
printf("O nome é %s.",nome);
printf("%d dividido por %d é igual a %f", n1, n2, (float)n1/n2);
printf("O código de %c é %d", letra, letra);
```

Códigos de formato:

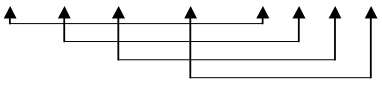
Normalmente os códigos de formato são utilizados na sua forma mais simples:

| | | | |
|---|---|---|---|
| %c | → caracter simples | %d | → decimal |
| %ld | → inteiro "longo" | %f | → ponto flutuante |
| %o | → octal | %s | → cadeia de caracteres |
| %x | → hexadecima | %lf | → double |

Obs.: Deve haver uma variável ou constante para cada código de formato! O tipo das variáveis ou constantes também deve coincidir com os códigos de formato.

```
int a;
float b;
char c;
double d;
```

```
printf("%d %f %c %lf",a,b,c,d);
```



Sintaxe completa de um código de formato:

%[flags][largura][.precisão][tamanho]<tipo>

<tipo> indica o tipo do valor a ser exibido. Listados acima.

[flags]:

"-" → indica alinhamento pela esquerda

"+" → força os números a começarem por "+" ou "-"

" " → os negativos ficam com o sinal de "-" e os positivos com um espaço em branco no lugar do sinal.

[largura]:

n → indica o número máximo de casas a ocupar.

0n → idêntico ao anterior, apenas com a diferença de que as casas não ocupadas serão preenchidas com zeros.

[precisão]:

n → indica o número de casas após a vírgula.

[tamanho]:

"l" → long

Caracteres especiais:

Alguns caracteres, como o de tabulação, não possuem representação gráfica na tela. Por razões de compatibilidade, a linguagem C fornece constantes iniciadas pelo caracter "\" para indicar esses caracteres.

\n → nova linha
\r → retorno do carro
\t → tabulação
\b → retrocesso (backspace)
\" → aspas
\\ → barra
\0 → nulo
\a → sinal sonoro
\xn → caracter de código **n** (em hexadecimal)

Exemplos:

```
printf("Olá!\n");
printf("Linha1\nLinha2\n");
printf("\tParágrafo\nHoje é %d/%d/%d\n,dia,mes,ano);
printf("Este é o \"\\\" backslach\n");
printf("\xB3\n);
printf("Atenção!!!\a\a\a Erro!!!\a\n");
```

#### 3.1.1.2.2 A função "scanf()"

É a função de entrada formatada de dados pelo teclado. Sua sintaxe é similar à da função "printf".

scanf("<expr. de controle>", <lista de argumentos>);

A expressão de controle pode conter tanto códigos de formatação precedidos pelo sinal "%", que indicam o tipo dos dados a serem lidos, como caracteres de espaçamento.

a) Códigos de formato:

%c → lê um caracter
%d → lê um inteiro decimal
%e → lê um número em notação científica
%f → lê um número de ponto flutuante
%s → lê uma string
%u → lê um decimal sem sinal
%l → lê um inteiro longo
%lf → lê um double

Sintaxe: [largura][código de formato]

b) Caracteres de Espaçamento:

São considerados caracteres de espaçamento o espaço em branco, o caracter "\t" e o "\n". Sempre que um destes surgir na string de controle de um comando "scanf" ele indicará que o mesmo deve ser considerado como separador dos valores a serem entrados Esses caracteres serão então lidos pelo "scanf", porém, não armazenados.

Normalmente quando o usuário está entrando com valores atendendo a um "scanf", quando o mesmo digita um destes caracteres de espaçamento, o mesmo é lido e não armazenado. A diferença é que se o caracter de espaçamento aparece na string de controle, então o scanf nõ é encerrado enquanto o mesmo não for digitado.

Toda entrada correspondente um comando "scanf" deve sempre ser finalizado por <ENTER>.

Ex.:

```
void main()
     {
     float anos, dias;
     printf("Digite sua idade em anos: ");
     scanf("%f",&anos);
     dias=anos*365;
     printf("Você já viveu %f dias\n",dias);
     }
```

Obs.: Por enquanto vamos assumir que todas as variáveis da lista de argumentos, com exceção das variáveis string, deverão ser antecedidas do operador "&". Mais adiante vamos entender a razão do operador "&" e quando o mesmo deve ser utilizado.

c) "Search set"

É possível ainda, no caso de entradas de **strings** determinar o conjunto de caracteres válidos (todos os caracteres que não aparecerem nesse conjunto serão considerados separadores).

Sintaxe: %[search set]

Exemplos:

%[A-Z] → todos os caracteres de A até Z

%[abc] → apenas os caracteres a, b ou c

%[^abc] → todos os caracteres menos a, b, c

%[A-Z0-9a-z] → maiúsculas + minúsculas + dígitos

#### 3.1.1.2.3 As funções “getche()” e “getch()”

São funções que lêem caracteres do teclado sem esperar a tecla <ENTER>. Ambas não recebem argumentos e devolvem o caracter lido para a função que os chamou. A diferença entre as duas reside no fato de que “getche” ecoa o caracter lido no vídeo.

Exemplo:

```
void main()
     {
     char ch;
     printf("Digite algum caracter: ");
     ch=getch();
     printf("\nA tecla digitada foi %c e seu valor na tabela ASCII é %d.",ch,ch);
     }
```

Obs.: Devido à maneira diferenciada como tratam o buffer de teclado, o uso das rotinas “getch” e “scanf” no mesmo programa pode trazer problemas. Para contorná-los, a função “fflush(stdin)” garante que o buffer do teclado (stdin - entrada padrão) esteja vazio.

### 3.1.2 Comandos de Desvio Incondicional

Comandos de desvio incondicional são comandos que alteram a seqüência normal de execução em um bloco de comandos, transferindo o processamento para um ponto no programa fonte marcado com o rótulo especificado no comando GOTO.

**Em Pascal:**

```
label <rótulo>;
begin
GOTO  <rótulo>;
. . .
<rótulo> :
<comandos>;
end.
```

**Em C:**

```
{
GOTO  <rótulo>;
. . .
<rótulo> :
<comandos>;
}
```

Devemos evitar sempre que possível o uso de comandos desse tipo.

Exemplo em Pascal:

```
uses crt;
label inicio, fim_ok, fim_qm, fim;
var x:char;
begin
clrscr;
inicio:
writeln('R-Repetir, F-finalizar por bem, Outra tecla- Finalizar de qualquer
maneira');
x:=upcase(readkey);
if (x='R') then goto inicio;
if (x='F') then goto fim_ok;
goto fim_qm;
fim_ok:
writeln('Finalizando por bem');
goto fim;
```

```
fim_qm:
writeln('Finalizando de qualquer maneira');
fim:
readkey;
end.
```

Exemplo em C:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

char x;

void main()
{
clrscr();
inicio:
printf("R-Repetir, F-finalizar por bem, Outra tecla- Finalizar de qualquer
maneira\n");
x=toupper(getch());
if (x=='R') goto inicio;
if (x=='F') goto fim_ok;
goto fim_qm;
fim_ok:
printf("Finalizando por bem");
goto fim;
fim_qm:
printf("Finalizando de qualquer maneira");
fim:
getch();
}
```

### 3.2 Estruturas de Controle

#### 3.2.1 Seqüência

Seqüência finita de instruções são agrupamentos de comandos, onde cada comando é executado um após o outro e sem desvios. Em Pascal a seqüência é delimitada pelas palavras reservadas BEGIN no início e END no final e seus comando são separados pelo delimitador ";" (ponto e vírgula). Em C a seqüência é delimitada pelos símbolos **{** no início e **}** no final e seus comando também são separados pelo delimitador ";" (ponto e vírgula).

#### 3.2.2 Comandos condicionais

##### 3.2.2.1 IF

O comando **if** é usado para executar um segmento de código condicionalmente. A forma mais simples do comando if é:

```
if (expressão)
        ação;
```

Neste caso a ação somente será executada se a expressão ou o conjunto de expressões lógicas for verdadeira.

No caso do comando **if-else** o programa pode ter duas ações distintas. Se a expressão for verdadeira, será executado o conjunto de ações do comando1. Se for falsa será executado o conjunto de ações do comando2.

Em Pascal a implementação do comando **if** é:

IF *condição* THEN
    <comando>;

IF *condição* THEN
    <comando1>
    ELSE
    <comando2>;

Exemplos em Pascal

```
uses crt;
var x: integer;
begin
x:=10;
if x>15 then
    begin
    writeln('X é maior que
15');
    end;
end.
```

```
uses crt;
var x, y: integer;
begin
x:=10;
if (x>15) and (y>15) then
    begin
    writeln('X e Y são maiores que 15');
    end
else
    begin
    writeln('X e Y não são maiores que
15');
    end;
end.
```

Em C a implementação do comando **if** é:

if (expressão)
    {
    <comando>;
    }

if (expressão)
    {
    <comando1>;
    }
else {
    <comando2>;
    }

Exemplos em C:

```
#include <stdio.h>
#include <conio.h>
int x;
void main()
{
x = 10;
if (x>15)
   {
   printf("X é maior que
15\n");
   }
}
```

```
#include <stdio.h>
#include <conio.h>
int x, y;
void main()
{
x = 10;
y = 20;
if (x>15 && y>15)
   {
   printf("X e Y são maiores que 15\n");
   }
else
   {
   printf("X e Y não são maiores que
15\n");
   }
}
```

Obs.: Deve-se tomar cuidado com os comando if-else aninhados. O **else** sempre está associado ao **if** mais próximo dentro do mesmo nível de comandos. Blocos mais internos não são considerados.

O comando **if** não necessita de uma expressão lógica no lugar do teste. Em C, qualquer expressão que resultar ZERO será considerada como FALSA e qualquer outro valor é considerado VERDADEIRO. Em Pascal somente é aceito os valores booleanos TRUE ou FALSE.

Em C também temos o comando **if – else if - else** que é freqüentemente utilizado para executar múltiplas comparações sucessiva. Sua forma geral é:

```
if (expressão1)
        ação1;
else if (expressão2)
        ação2;
else if (expressão3)
        ação3;
```

Logicamente, cada ação poderia ser um bloco composto exigindo seu próprio conjunto de chaves. Este tipo de controle de fluxo lógico avalia a expressão até que encontre uma que é VERDADEIRA. Quando isto ocorre, todos os testes condicionais restantes serão desviados. No exemplo anterior, nenhuma ação seria tomada se nenhuma das expressões fosse avaliada como VERDADEIRA.

Para executar uma ação padrão no caso de não satisfazer nenhuma das expressões declaradas pode-se colocar um **else** sem expressão de teste para realizar a ação pretendida, por exemplo:

```
if (expressão1)
        ação1;
else if (expressão2)
        ação2;
else if (expressão3)
        ação3;
else
        ação_padrão;
```

Exemplo:

```
#include <stdio.h>
#include <conio.h>

int x;

void main()
{
x = 16;
if (x == 5)
     {
     printf("X vale 5\n");
     }
else if (x == 10)
     {
     printf("X vale 10\n");
     }
else if (x == 15)
     {
     printf("X vale 15\n");
     }
else
     {
     printf("X não vale 5, 10 ou 15\n");
     }
}
```

#### 3.2.2.2 Comando de Seleção múltipla: SWITCH ou CASE

Quando se deseja testar uma variável ou uma expressão em relação a diversos valores usamos o comando de seleção múltipla.

O valor da expressão seletora é comparado com cada elemento da lista de valores. Se existir um valor igual será executada somente a seqüência relacionada ao valor. Caso contrário, ou nenhuma seqüência será executada, ou a seqüência relacionada á cláusula padrão será executada se ela existir.

Em Pascal o comando de seleção múltipla é o comando CASE:

CASE <seletor> OF

<valor1> : <comandos1>;

<valor2> : <comandos2>;

...

<valor*n*> : <comandos*n*>;

ELSE : <comandos_padrão>;

END;

Exemplo:

```
uses crt;
var x:integer;

begin
x := 15;
case x of
     5:   begin
          writeln('X vale 5');
               end;
     10:  begin
          writeln('X vale 10');
          end;
     15:  begin
          writeln('X vale 15');
          end;
     else begin
          writeln('X nao vale 5, 10 ou 15');
          end;
     end;
end.
```

Em C o comando de seleção múltipla é o comando **SWITCH**:

Em C devemos tomar um pouco de cuidado, pois o comando **switch** possui algumas peculiaridades. Sua sintaxe é:

switch (expressão) {

case <valor1>:

<comandos1>;

break;

case <valor2>:

<comandos2>;

break;

```
                    ...
                    case <valor n>:
                            <comandos n>;
                            break;
                    default:
                            <comandos_padrão>;
                    }
```

Exemplo:

```
#include <stdio.h>
#include <conio.h>

int x;

void main()
{
x = 15;
switch(x)
     {
     case 5:
          {
          printf("X vale 5\n");
          break;
          }
     case 10:
          {
          printf("X vale 10\n");
          break;
          }
     case 15:
          {
          printf("X vale 15\n");
          break;
          }
     default:
          {
          printf("X nao vale 5, 10 ou 15\n");
          }
     }
getch();
}
```

Devemos tomar bastante cuidado com o comando obrigatório **break**, que faz a porção restante do comando switch ser pulada. Caso ele seja removido do segmento de código, seriam executados todos os comandos abaixo dele.

### 3.2.3 Comandos de Repetição

Os comandos de repetição são caracterizados por permitir que uam seqüência de comandos seja executada um número repetido de vezes.

#### 3.2.3.1 For

O comando **for** é utilizado para executar uma seqüência de comandos repetidamente e com um número conhecido de vezes.

Em Pascal a sintaxe do comando é:

FOR <var> := <limite_inferior> TO <lim_superior> do

<comando>

ou

FOR <var> := <limite_superior> DOWNTO <limite_inferior> do

<comando>

Exemplos:

```
for i:= 1 to 10 do
     begin
     writeln(i, 'x', 7, '=', i*7);
     end;

for i:= 10 downto 1 do
     begin
     writeln(i, 'x', 7, '=', i*7);
     end;
```

Em C a sintaxe do comando é:

for (expr_inicialização; expr_teste; expr_incremento)

<comando>;

Exemplos:

```
for (i=1; i<=10; i++)
     {
     printf("%d x 7 = %d \n", i, i*7);
     }

for (i=10; i>=1; i--)
     {
     printf("%d x 7 = %d \n", i, i*7);
     }
```

Quando o comando de laço **for** é encontrado, a *expr_inicialização* é executada primeiro, no início do laço, e nunca mais será executada. Geralmente esse comando fornece a inicialização da variável de controle do laço. Após isso é testada a *expr_teste*, que é chamada de cndição de término do laço. Quando *expr_teste* é avaliada como VERDADEIRA, o comando ou comandos dentro do laço serão executados. Se o laço foi iniciado, *expr_incremento* é executada após todos os comandos dentro do laço serem executados. Contudo, se *exp_teste* é avaliada como FALSA, os comandos dentro do laço serão ignorados, junto com *expr_incremento*, e a execução continua no comando que segue o final do laço. O esquema de endentação para os laços **for** com diversos comandos a serem repetidos é assim:

```
for (expr_inicialização; expr_teste; expr_incremento)
     {
     comando_a;
     comando_b;
     comando_c;
     }
```

As variáveis de controle de um laço de **for** podem ter seu valor alterado em qualquer ponto do laço.

Qualqur uma das expressões de controle do laço de **for** pode ser omitida desde que sejam mantidos os “;”.

As variáveis utilizadas nas três expressões de controle não precisam ter relação entre si.

Ex.:

```
void main()
     {
     char c;
     int x,y;
     for(c=9;c>0;c--)
          {
          printf("%d",c);
          }
     for(c=9;c>0; )
          {
          printf("%d",c);
          c--;
          }
     for(x=0,y=0; x+y<100; x=x+4, y++)
          {
          printf("%d+%d=%d\n",x,y,x+y);
          }
     for(;;)
          {
          printf("não saio deste laço nunca!!!!\n");
          }
     }
```

#### 3.2.3.2 While

Assim como o laço *for*, **while** é um laço com teste no iníco. Isto significa que a expressão teste é avaliada antes dos comandos dentro do corpo do laço serem executados. Por causa disto, os laços com teste no início podem ser executados de zero a mais vezes.

Geralmente estas estruturas são usadas quando um número indefinido de repetições é esperado.

Em Pascal a sintaxe é:

WHILE <condição> do
<comandos>;

Exemplo:

```
uses crt;
var a,b,ano: integer;
begin
a:=1500;
b:=2000;
ano:=0;
while a < b do
     begin
     a := a * 1.05;
     b := b * 1.02;
     ano++;
     end;
writeln(ano, ' anos');
end.
```

Em C a sintaxe é:

```
while (expr_teste)
        <comandos>;
```

Exemplo.:

```
void main()
{
     int a, b,ano;
     a=1500;
     b=2000;
     ano=0;
     while(a<b)
          {
          a=a*1.05;
          b=b*1.02;
          ano++;
          }
     printf("%d anos",ano);
}
```

#### 3.2.3.3 Do While ou Repeat

Estes comandos, assim como o comando WHILE, são usados quando não é conhecido o número de vezes que uma seqüência de comandos deverá ser executada. Porém a seqüência será executada pelo menos uma vez.

Em Pascal a sintaxe do comando é:

```
REPEAT
        <comandos>;
UNTIL <cond_teste>;
```

Exemplo:

```
var a:char;
Begin
repeat
     clrscr;
     writeln('1 - Executar');
     writeln('2 - Sair');
     a := readkey;
     if a = ´1´ then executar;
until a =´2´;
end.
```

Em C a sintaxe do comando é:

```
do
        <comandos>
while(cond_teste);
```

Exemplo.:

```
Void main()
     {
          char a;
          do {
               clrscr();
               printf("1 - Executar\n");
               printf("2 - Sair\n");
               a = getche();
               if (a == ´1´) executar();
          } while (a != ´2´);
     }
```

Observe que em **Pascal** usamos o comando **repeat** e **until** fazendo com que o laço repita até que a condição seja satisfeita e em **C** usamos o comando **do** e **while** fazendo com que o laço repita enquanto a condição esteja sendo satisfeita.

#### 3.2.3.4 Comandos de desvio

Nas linguagem **C** e **Pascal** temos os comandos de desvio que interrompem a execução de um laço.

##### 3.2.3.4.1 Comando break

O comando **break** pode ser usado para sair de um laço **for**, **while** ou **do-while (repeat)** mais interno, desde que no mesmo subprograma, passando a seqüência da execução para a primeira linha após o laço.

Exemplo em Pascal:

```
uses crt;
var x: char;
i: integer;
begin
for i:= 1 to 100 do
      begin
      write('Digite um numero entre 0 e 9: ');
      x:=readkey;
      if (x < #48) or (x > #57) then break;
      writeln(x, ' foi digitado!');
      end;
end.
```

Exemplo em C:

```
void main()
{
char x, i;
for(x=1;x<=100;x++)
     {
     printf("Digite um número de 0 a 9:");
     x = getch();
     if (y < 48 || y > 57) break;
     printf("%d foi digitado \n",x);
     }
}
```

##### 3.2.3.4.2 Comando continue

O comando **continue** causa o fim de um dos laços de uma repetição e o retorno imediato ao teste.

Exemplo em Pascal:

```
uses crt;
var
i: integer;
begin
for i:= 1 to 10 do
      begin
      if (i mod 2 <> 0) then continue;
      writeln(i, ' e par!');
      end;
end.
```

Exemplo em C:

```
void main()
{
int i;
for(i=1;i<=10;i++)
     {
     if ( i % 2 != 0) continue;
     printf("O número %d é par!",i);
     }
}
```

#### 3.2.3.4.3 Função Exit

A função **exit** causa a imediata interrupção do programa e o retorno ao sistema operacional. Em C, o valor do parâmetro é retornado ao processo que o chamou que geralmente é o sistema operacional. O valor 0 ( `exit(0);`) geralmente indica que o processo terminou sem problemas.

Exemplo em Pascal:

```
uses crt;
begin
while true do
      begin
      write('Xx ');
      if keypressed then exit;
      end;
end.
```

Exemplo em C:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void main()
{
while(1)
     {
     printf("Xx ");
     if ( kbhit()) exit(0);
     }
}
```

#### 3.2.3.4.4 Comando return

O comando return causa uma interrupção no fluxo de comandos de uma função e o retorno a função chamadora. Pode ser usado com ou sem argumento dependendo do tipo de função em que é utilizado, porém não é possível em uma única função seu uso com e sem argumentos.

## 4. FUNÇÕES E PROCEDIMENTOS

As funções formam o alicerce da programação em C e C++. Conforme vamos aumentando a prática em programação, os programas começam a tomar uma aparência modular quando programamos com funções.

Podemos fazer toda a programação em C e C++ dentro de uma função. Isto porque todos os programas devem incluir **main**, que é uma função.

As funções são similares aos módulos de outras linguagens. Pascal utiliza procedimentos e funções. Fortran utiliza somente funções, e a linguagem Assembly utiliza somente procedimentos. O modo como as funções trabalham determina o alto grau de eficiência, legibilidade e portabilidade do código do programa em C.

### 4.1 Procedimentos

Um procedimento é um tipo de subprograma utilizado na linguagem Pascal, que se assemelha em muito com um programa em Pascal. Possui um cabeçalho de identificação com o nome do procedimento, uma lista opcional de parâmetros de comunicação, um bloco de declarações e um bloco de comandos.

#### 4.1.1 Passagem de parâmetros

Quando a variável que se quer trabalhar não é visível dentro do procedimento ela deve ser passada como parâmetro:

##### 4.1.1.1 Passagem de parâmetros por valor

Quando uma variável é passada por valor para um prodedimento, seu valor original não é alterado.

Declaração:

```
procedure <nome>(<variáveis>:<tipo>)
```

##### 4.1.1.2 Passagem de parâmetros por referência

Quando uma variável é passada por referência para um prodedimento, seu valor original é alterado na execução do procedimento.

Declaração:

```
procedure <nome>(var <variáveis>:<tipo>)
```

### 4.2 Funções

Em Pascal as funções podem ser vistas como um procedimento que retorna um único valor.

Ex.:

```
PROGRAM FUNCAO;
VAR pr, imp: REAL;

FUNCTION calc_imposto(preco: REAL):REAL;
     BEGIN
     calc_imposto = preco * 0.17;
     END;

BEGIN
READLN(pr);
```

```
imp = calc_imposto(pr);
WRITELN('Preço: ',pr,' Imposto: ',imp);
END.
```

Em C uma função **void** ou que retorna um valor nulo, pode ser comparada com um procedimento utilizado na linguagem Pascal, porém todas os subprogramas em C são funções e devem retornar um valor (ou retornar void, vazio).

### 4.2.1 Estilos e protótipos das funções

As declarações de função começam com o protótipo da função C e C++. O protótipo de função é simples e é incluído no início do código do programa para notificar o compilador do tipo e do número de argumentos que uma função utilizará.

Embora outras variações sejam legais, sempre que possível você deve utilizar a forma do protótipo de função que é uma réplica da linha de declaração da função. Por exemplo:

***tipo_de_retorno nome_da_função ( tipo(s)_argumento nome(s)_argumento);***

### 4.2.2 Argumentos das funções

Os argumentos, ou parâmetros, passados às funções são opcionais; algumas funções podem não receber argumentos enquanto outras podem receber diversos. Os argumentos podem ser misturados, sendo possível o uso de qualquer um dos tipos de dados escalares padronizados, podendo ser int, float, double, char e ponteiros.

Existem situações onde é necessário que as sub-rotinas retornem valores calculados internamente. A linguagem C apresenta dois mecanismos para tanto:

#### 4.2.2.1 Passagem de parâmetros por valor

As funções recebem parâmetros por valor quando na lista de argumentos existem valores ou variáveis com valores. Esses argumentos são criados na pilha no momento em que a função é chamada.

O comando **return** permite que uma função retorne um único valor. Esse valor é obtido na rotina chamadora na medida em que a chamada na função é feita através de uma atribuição (Ex. 2).

Ex. 1:

```
...
     int v;
     v=30;
     func1(v,25);
...

void func1(int a, int b)
     {
     int x;
     a = a + 10;
     b = b - 10;
     x = a + b;
     printf("%d\n",x);
     }
```

Ex. 2:

```
void main()
     {
     float pr, imp;
     scanf("%f",&pr);
     imp=calc_imposto(pr);
     printf("Preço: %f, Imposto: %f\n",pr,imp);
```

```
     }

float calc_imposto(float preco);
     {
     float imposto;
     imposto=preco * 0.17;
     return(imposto);
     }
```

No exemplo 1 acima, o valor da variável **v** e o valor **25** são passados respectivamente para os argumentos **a** e **b** da função **func1**. Apesar do valor de **a** e **b** ser alterado dentro da função, essa mudança não se refletirá no valor da variável **v** pois o que é passado para a função é apenas uma cópia do valor da variável.

#### 4.2.2.2 Passagem de parâmetros por referência

A passagem de parâmetros por referência faz com que os valores das variáveis passadas por referência sejam alterados dentro da função.

Em C, quando queremos que isto aconteça, ao invés de passarmos o valor como parâmetro de uma função, passamos o **endereço** dessa variável (que não deixa de ser um valor). O endereço, no caso, faz as vezes de "referência" para a variável e nos permite alterar o conteúdo da mesma dentro da sub-rotina.

Ex:

```
void main()
     {
     int a;
     func1(&a);
     printf("%d",a);
     }

void func1(int *p)
     {
     int x;
     scanf("%d",&x);
     *p = x * 2;
     }
```

Observe que o argumento da função func1 é um ponteiro para inteiro. O que se passa na chamada dessa função, não é o valor da variável **a**, mas sim seu endereço (até porque nesse momento **a** nem ao menos foi inicializado). Dentro da função **func1**, o valor digitado pelo usuário é multiplicado por 2 e é armazenado, não na variável **p** que contém o endereço de **a**, mas na própria variável **a**. Desta forma, quando a função acaba e o controle volta à rotina chamadora, o valor correto já está armazenado em **a**.

Note-se, então, que o uso de um ***** antes de uma variável ponteiro em uma expressão significa que não estamos nos referindo ao valor do ponteiro, mas sim ao valor para o qual aponta.

Resumindo, podemos dizer que:

**&a** → é o endereço de **a**

***p** → é o conteúdo da variável apontada por **p**

### 4.2.3 Tipos de funções

As funções podem ser:

Funções do tipo void: As funções são do tipo void quando indicam explicitamente a ausência de argumentos na função.

EX.:

```
#include <stdio.h>
#include <math.h>

void impressao(void);

main()
     {
     printf("Este programa extrai uma raiz quadrada\n");
     impressao();
     return(0);
     }

void impressao(void)
     {
     double z=5678.0;
     double x;
     x=sqrt(z);
     printf("A raiz quadrada de %lf é %lf\n",z,x);
     }
```

Funções do tipo char: As funções são do tipo char quando recebem um caracter como argumento.

Ex.:

```
#include <stdio.h>
#include <conio.h>

void impressao(char c);

void main()
     {
     char meucaracter;
     printf("Informe um caracter pelo teclado\n");
     meucaracter=getch();
     impressao(meucaracter);
     }

void impressao(char c)
     {
     int i;
     for(i=0;i<10;i++)
          printf("O caracter é: %c\n",c);
     }
```

Funções do tipo int: As funções são do tipo int quando aceitam e retornam um inteiro como argumentos.

Funções do tipo long: As funções são do tipo long quando aceitam e retornam long int como argumentos.

Funções do tipo float: As funções são do tipo float quando aceitam e retornam float como argumentos.

Ex.:

```
#include <stdio.h>
#include <math.h>

void hipotenusa(float x, float y);
```

```
main()
     {
     float cateto_y, cateto_x;
     printf("Altura do triângulo retângulo: ");
     scanf("%f",cateto_y);
     printf("Base do triângulo retângulo: ");
     scanf("%f",cateto_x);
     hipotenusa(cateto_y,cateto_x);
     return(0);
     }

void hipotenusa(float x, float y)
     {
     double minhahipo;
     minhahipo = hipot((double)x,(double)y);
     printf("A hipotenusa do triângulo é: %f\n",minhahipo);
     }
```

Funções do tipo double: As funções que aceitam e retornam um tipo double, ou seja um float com muita precisão são do tipo double. Todas as funções que estão definidas em ***math.h*** aceitam e retornam tipo double.

### 4.3 Argumentos passados a programas

Nomalmente as linguagens de programação permitem que o programa receba parâmetros passados na linha de comando no momento da sua ativação. Assim, quando usamos o comando **format a:** ou **scan a:** passamos o argumento **a:** para o programa.

#### 4.3.1 Argumentos passados a programas em PASCAL

Os argumentos passados aos programas em Pascal são verificados através de dois comandos (variáveis que estão armazenados os parâmentros):

PARAMCOUNT: armazena o número de argumentos passados na linha de comando. Paramcount igual a zero indica que não há argumentos na linha de comando.

PARAMSTR: armazena as strings de argumentos da linha de comando. ParamStr(1) é o primeiro argumento.

```
program parametros;
uses crt;
var I: integer;
begin
if ParamCount = 0 then
   begin
   Writeln('Este programa não tem argumentos!');
   exit;
   end;
for I := 1 to ParamCount do
   Writeln('O ',I,'º parâmetro é: ',ParamStr(I));
end.
```

#### 4.3.2 Argumentos passados a programas em C

C e C++ podem aceitar inúmeros argumentos da linha de comando. A função **main** recebe os parâmetros com a seguinte declaração:

```
main(int argc, char *argv[])
```

O primeiro argumento é um inteiro que fornece o número de termos da linha de comando. O nome do programa executável conta como sendo o primeiro, isto é, todo programa terá **argc** com valor 1 se não tiver nenhum parâmetro e valor *num_parâmetros + 1* quando tiver parâmetros.

O segundo argumento é um ponteiro para as strings **argv**. Todos os argumentos são strings de caracteres, de modo que são conhecidos no programa como:

```
argv[1]     primeiro argumento
argv[2]     segundo argumento
...
```

Ex.:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
     {
     int i;
     if (argc < 2)
          {
          printf("Este programa não tem argumentos!");
          exit(1);
          }
     for (i=1;i < argc; i++)
          printf("O %dº argumento é %s\n",i,argv[i]);
     }
```

Os argumentos são recebidos da linha de comando e impressos na tela na mesma ordem. Se números forem informados na linha de comando, eles serão interpretados como strings ASCII e impressos como caracteres. Se desejarmos fazer cálculos com esses números devemos convertê-los de string para número com as funções de conversão apropriadas (**atoi**, **atol**, **atof** que encontram-se definidas em stdlib.h)

## 5. MATRIZES

Matrizes são variáveis indexadas que contêm diversos itens de dados do mesmo tipo. Cada matriz possui um nome, e seus elementos são acessados com a associação de um índice ao nome da matriz.

Uma matriz possui quatro propriedades básicas:

a) Os itens de dados individuais na matriz são chamados de elementos.

b) Todos os elementos devem ser do mesmo tipo de dados.

c) Todos os elementos são armazenados contiguamente na memória do computador, e em linguagem C o índice do primento elemento sempre será zero.

d) O nome da matriz é um valor constante que representa o endereço do primeiro elemento na matriz.

Como todos os elementos são do mesmo tamanho, não podemos definir matrizes usando uma mistura de tipos de dados, pois a localização de um elemento é feita com base no endereço do primeiro elemento mais o deslocamento proporcional ao índice.

As matrizes podem ser **unidimensionais** quando o acesso a um de seus elementos é feito através de um único índice, também conhecida como *vetores* em Pascal, ou **multidimensionais** quando possuem mais que uma dimensão, sendo necessário mais de um índice para acessar um de seus elementos.

### 5.1 Matrizes em Pascal

Uma matriz unidimensional ou vetor em Pascal é definido com o uso da palavra reservada **array**, seguida de seus limites inferior e superior entre colchetes, da palavra reservada **of** e do tipo de componente do vetor.

<nomeVetor> : ARRAY [<lim_inf> .. <lim_sup>] of <TipoComponente>

Ex:

```
Alunos: Array[1..40] of integer;
```

Uma matriz multidimensional pode ser comparada a uma tabela de duas ou mais dimensões, sendo necessários tantos índices quantas forem as dimensões.

Uma matriz multidimensional em Pascal é definida com o uso da palavra reservada **array**, seguida de seus limites inferior e superior para cada dimensão, separada por vírgulas, entre colchetes, da palavra reservada **of** e do tipo de componente do vetor.

Ex:

```
Notas: Array[1..40,1..3] of integer;
```

Uma matriz pode ser declarada e inicializada declarando-se no bloco de constantes:

```
const
num1: array [1..5] of integer = (2,4,6,8,10);
num2: array [1..2,1..5] of integer = ((10,20,30,40,50),(15,25,35,45,55));
```

### 5.2 Matrizes em C

Uma matriz em C é definida escrevendo-se o tipo da matriz, seguida de um nome, e um par de colchetes contendo uma expressão constante que define o tamanho da matriz.

<tipo> <nomematriz> [<tamanho>]

Ex:

```
int alunos[40];
```

Uma matriz multidimensional é definida escrevendo-se o tipo da matriz, seguida de um nome, e tantos pares de colchetes quantas forem a dimensões, contendo uma expressão constante que define o tamanho da matriz em cada dimensão.

Ex:

```
int Notas[40][3];
```

Em C o primeiro elemento sempre será o elemento de índice 0.

Uma matriz pode ser inicializada explicitamente quando é declarada:

Ex.:

```
#include <conio.h>
#include <stdio.h>

int numeros[3] = {2,4,6};
char vogais[5] = {'a','b','c','d','e'};
int num[2][5]={{2,4,6,8,10},{1,3,5,7,9}};

void main()
{
int i,j;
for (i=0;i<2;i++)
     {
     printf("\n");
     for (j=0;j<5;j++)
          {
          printf("  %d",num[i][j]);
          }
     }
}
```

Observe qua não se pode usar uma variável na definição da matriz para dimensionar o seu tamanho, pois o compilador precisa alocar a memória necessária no momento da compilação.

### 5.3 Strings

Em Pascal temos o tipo de dado **string** predefinido na linguagem que nada mais é que um vetor de caracteres.

A linguagem C não possui este tipo de dado. Toda a vez que queremos uma string em C devemos declará-la como:

***char <nome> [<tamanho>];***

Quando precisamos de um vetor de string, declaramos como uma matriz de caracteres, por exemplo, se queremos uma lista de 250 nomes com 80 letras cada:

```
char nomes[250][80];
```

Toda a vez que precisamos fazer operações com strings em C, como comparação, cópia de valores, concatenação, devemos usar as funções predefinidas em **string.h**:

| Função | Sintaxe e Função |
|---|---|
| **strcat** | char *strcat(char *dest, const char *src);<br>Adiciona uma string a outra strcat adiciona uma códia da string de origem para o final da string de destino. |

| | |
|---|---|
| **Strchr** | char *strchr(char *s, int c);<br>Procura numa string a primeira ocorrência de um caracter. Retorna um ponteiro para a primeira ocorrência do caracter dentro da string. Se o caracter não for encontrado, strchr retorna NULL.. |
| **strcmp** | int strcmp(const char *s1, const char *s2);<br>Compara uma string com outra e retorna:<br>< 0 se s1 for menor que s2<br>= 0 se s1 for igual s2<br>> 0 se s1 for maior que s2 |
| **strcpy** | char *strcpy(char *dest, const char *src);<br>Copia uma string para outra. Copia string src para dest. Retorna um ponteiro para a string dest. |
| **strcspn** | size_t strcspn(const char *s1, const char *s2);<br>Procura em s1 até que qualquer um dos caracteres contidos em s2 é encontrado. O número de caracteres que são lidos em s1 é o valor retornado. |
| **stricmp** | int stricmp(const char *s1, const char *s2);<br>Compara uma string com outra sem distinção entre maiúscula e minúscula e retorna:<br>< 0 se s1 for menor que s2<br>= 0 se s1 for igual s2<br>> 0 se s1 for maior que s2 |
| **strlen** | size_t strlen(const char *s);<br>Calcula e retorna o tamanho de uma string. |
| **strlwr** | char *strlwr(char *s);<br>Converte letras maiúsculas em minúsculas numa string. |
| **strncat** | char *strncat(char *dest, const char *src, size_t n);<br>Adiciona uma porção de uma string a outra. strncat copia *n* caracteres de *src* para o final de *dest* e acrescenta um caractere NULL. |
| **strncmp** | int strncmp(const char *s1, const char *s2, size_t n);<br>Compara uma porção de uma string com uma porção de outra string.<br>strncmp faz a mesma comparação que strcmp, mas analisa no máximo os *n* primeiros caracteres e retorna:<br>< 0 se s1 for menor que s2<br>= 0 se s1 for igual s2<br>> 0 se s1 for maior que s2 |
| **strncpy** | char *strncpy(char *dest, const char *src, size_t n);<br>Copia um um determinado número de bytes (n) de uma string para outra.. |
| **strnicmp** | int strnicmp(const char *s1, const char *s2, size_t n);<br>Compara uma porção de uma string com uma porção de outra string sem distinção de maiúsculas com minúsculas. strncmp faz a mesma comparação que stricmp, mas analisa no máximo os *n* primeiros caracteres.<br>Retorna:<br>< 0 se s1 for menor que s2<br>= 0 se s1 for igual s2<br>> 0 se s1 for maior que s2 |
| **strnset** | char *strnset(char *s, int ch, size_t n);<br>Altera os *n* primeiros caracters de uma string para um caractere específico (*ch*). |
| **strrchr** | char *strrchr(const char *s, int c);<br>Procura numa string pela última ocorrência de um caracter (*c*). |
| **strrev** | char *strrev(char *s);<br>Reverte uma string. strrev muda todos os caracteres numa string para a ordem reversa, com exeção do caracter nulo de terminação. |
| **strset** | char *strset(char *s, int ch);<br>Altera todos os caracteres numa string para um determinado caracter (*ch*). |
| **strspn** | size_t strspn(const char *s1, const char *s2);<br>Procura numa string s1 pelo primeiro segmento que difere de s2, retornando a posição onde inicia a diferença. |
| **strstr** | char *strstr(const char *s1, const char *s2);<br>Procura numa string s1 pela ocorrência de uma dada substring s2. |
| **strupr** | char *strupr(char *s);<br>Converte letras minúsculas numa string para maiúsculas. |

### 5.4 Matrizes e vetores como parâmetros de funções

Como já se viu anteriormente, uma função em **C** só aceita parâmetros por valor. Não é possível, portanto, passar um vetor ou matriz (tipos de dados estruturados) como parâmetro. A solução utilizada quando da necessidade de se trabalhar com os dados contidos em um vetor não global dentro de uma função é passar um ponteiro para este vetor como parâmetro. Para trabalhar com os elementos do vetor dentro da função procede-se de forma normal já que sempre que estamos trabalhando com vetores estamos usando um ponteiro para a primeira posição da área de memória alocada para o mesmo.

Exemplo de passagem de parâmetro por valor de um vetor :

```
#include <stdio.h>
#include <conio.h>

void imprime(int vet2[10]);

void main()
{
     int a, vet[10];

     clrscr();
     for (a=0;a<10;a++)
          {
          scanf("%d",&vet[a]);
          }
     imprime(vet);
     getch();
}

void imprime(int vet2[10])
{
     int a;
     for (a=0;a<10;a++)
          printf("%d ",vet2[a]);
}
```

Exemplo de passagem de parâmetro por referência de um vetor :

```
#include <stdio.h>
#include <conio.h>

void imprime(int *vet2, int n);

void main()
{
     int a, vet[10];

     clrscr();
     for (a=0;a<10;a++)
          {
          scanf("%d",&vet[a]);
          }
     imprime(vet,10);
     getch();
}

void imprime(int *vet2, int n)
{
     int a;
     for (a=0;a<n;a++)
          printf("%d ",vet2[a]);
}
```

Exemplo de passagem de passagem de parâmetro por valor de uma matriz:

```
#include <stdio.h>
#include <conio.h>

void imprime(int mat2[3][3]);

void main()
{
     int i, j, mat[3][3];
     clrscr();
     for (i=0;i<3;i++)
          for (j=0;j<3;j++)
               scanf("%d",&mat[i][j]);

     imprime(mat);
     getch();
}

void imprime(int mat2[3][3])
{
     int a, b;
     for (a=0;a<3;a++)  {
          printf("\n");
          for (b=0;b<3;b++)
               {
               printf("%d ",mat2[a][b]);
               }
          }
}
```

Exemplo de passagem de passagem de parâmetro por referência de uma matriz. No caso de matrizes, pelo menos uma das dimensões tem que ser conhecida.

```
#include <stdio.h>
#include <conio.h>

void imprime(int mat2[][3], int l);

void main()
{
int i, j, mat[3][3];

clrscr();
for (i=0;i<3;i++)
     {
     for (j=0;j<3;j++)
          {
          scanf("%d",&mat[i][j]);
          }
     }
imprime(mat,3);
}

void imprime(int mat2[][3], int linha)
{
int a, b;
for (a=0;a<linha;a++)
     {
     printf("\n");
     for (b=0;b<3;b++)
          {
          printf("%d ",mat2[a][b]);
          }
     }
}
```

## 6. PONTEIROS

### 6.1 Definição de ponteiros

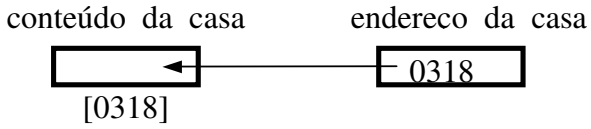
Ponteiro é uma variável que pode manter um endereço de uma variável.

Um modo conveniente e muito eficiente de acessar uma variável é fazer referência a ela através de uma segunda variável que contém o endereço da variável a ser acessada (ponteiro).

Por exemplo, suponha que você tenha uma variável **int** chamada *conteúdo_da_casa* e outra chamada *endereço_da_casa* que pode conter o endereço de uma variável do tipo **int**. Em C, preceder uma variável com o operador de endereços **&** retorna o endereço da variável em lugar do seu conteúdo. A sintaxe para atribuir o endereço da variável em lugar do seu conteúdo. A sintaxe para atribuir o endereço de uma variável a uma variável que contém o endereço é:

```
endereço_da_casa = &conteúdo_da_casa
```

Uma variável que pode manter um endereço, como *endereço_da_casa*, é chamada de *variável ponteiro* ou simplesmente um *ponteiro*.




Esta figura ilustra este relacionamento. A variável *conteúdo_da_casa* foi colocada na memória no endereço 0318. Após a execução do comando anterior, o endereço de *conteúdo_da_casa* foi atribuído para a variável ponteiro *endereço_da_casa*.

Pode-se expressar este relacionamento dizendo que *endereço_da_casa* aponta para *conteúdo_da_casa*.

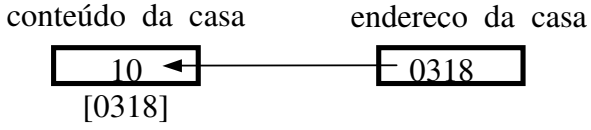
Para acessar o conteúdo da célula cujo endereço está armazenado em *endereço_da_casa*, somente preceda a variável ponteiro com um *, como em **endereço_da_casa*.

Por exemplo, se você executar os seguintes comandos:

```
endereço_da_casa = &conteúdo_da_casa;

*endereço_da_casa = 10;
```

O valor da célula (variável) chamada conteúdo da casa será 10. Podemos pensar no * como uma instrução para seguir a seta (figura) para encontrar a célula referenciada. Note que, se *endereço_da_casa* mantiver o endereço de *conteúdo_da_casa*, ambos os comandos que seguem terão o mesmo efeito, isto é, ambos armazenarão o valor 10 em *conteúdo_da_casa*:

```
conteúdo_da_casa = 10;

*endereço_da_casa = 10;
```



### 6.1.1 Declaração de variáveis tipo ponteiro

Como em outras linguagens, C exige uma definição para cada uma das variáveis. O seguinte comando define uma variável ponteiro *endereço_da_casa*, que pode manter o endereço de uma variável **int**:

```
int *endereço_da_casa;
```

Na verdade, há duas partes separadas para esta declaração. O tipo de dado de *endereço_da_casa* é

```
int *
```

e o identificador para a variável é:

```
endereço_da_casa
```

O asterisco após a **int** significa "aponta para", isto é, o tipo de dado **int *** é uma variável ponteiro que pode manter um endereço para um **int**.

Este é um conceito muito importante para memorizar. Em C, ao contrário de muitas outras linguagens, uma variável ponteiro guarda o endereço de um tipo de dado particular. Aqui está um exemplo:

```
char *endereço_para_char;
      int *endereço_para_int;
```

O tipo de dado *endereço_para_char* é diferente do tipo *endereço_para_int*. Os erros em tempo de execução e advertências durante a compilação podem ocorrer num programa que define um ponteiro para um tipo de dado e então o utiliza para apontar para algum outro tipo de dado. É também uma prática **inadequada** na programação definir um ponteiro de um modo e depois utilizá-lo de outro:

```
int *int_ptr;
float valor_real = 23.45;
      int_ptr = &valor_real;
```

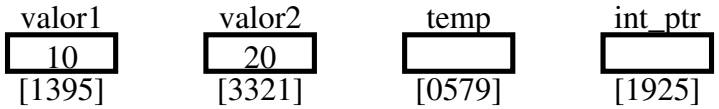
Aqui, a variável *int_ptr* foi definida ser do tipo **int ***, significando que pode conter o endereço de uma célula de memória do tipo **int**. O terceiro comando tenta atribuir a *int_ptr* o endereço *&valor_real* de uma variável **float** declarada.

### 6.1.2 Usando variáveis tipo ponteiro

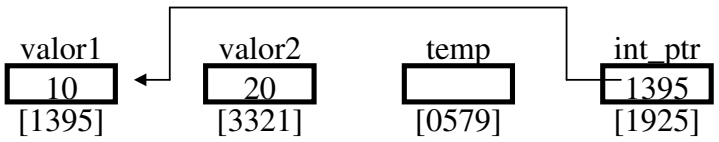
O seguinte exemplo de código trocará os conteúdos das variáveis *valor1* e *valor2* usando os operadores de ponteiros:

```
int valor1 = 10, valor2 = 20, temp;
int *int_ptr;

int_ptr = &valor1;
temp = *int_ptr;
*int_ptr = valor2;
      valor2 = temp;
```

A primeira linha do programa contém as definições e inicializações padrões. O comando aloca três células para manter um único **int**, dando a cada célula um nome, e inicializa duas delas. É assumido que a célula chamada *valor1* está localizada no endereço 1395, que a célula chamada *valor2* está localizada no endereço 3321 e que a célula chamada *temp* está localizada no endereço 0579.

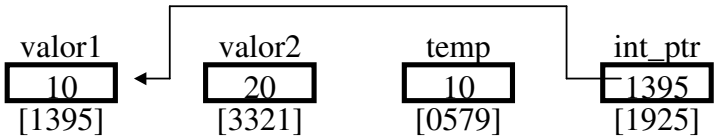


O segundo comando no programa define *int_ptr* como um ponteiro para um tipo de dado **int**. O comando aloca a célula e lhe dá um nome (colocado no endereço 1925). Lembre-se que, quando o asterisco é combinado com o tipo de dado (neste caso **int**), a variável contém o endereço de uma célula do mesmo tipo de dado. Como *int_ptr* não foi inicializada, ela não aponta para nenhuma variável **int** em particular. O terceiro comando atribui a *int_ptr* o endereço de *valor1*:




O próximo comando no programa:

```
temp = *int_ptr;
```

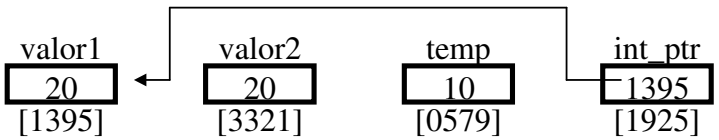



usa a expressão ***int_ptr*** para acessar o conteúdo da célula à qual *int_ptr* aponta: *valor1*. Portanto, o valor inteiro 10 é armazenado na variável *temp*. Se omitíssemos o asterisco na frente de *int_ptr*, o comando de atribuição ilegalmente armazenaria o conteúdo de *int_ptr* (o endereço 1395), na célula chamada *temp*, que somente pode armazenar um inteiro, e não um endereço.

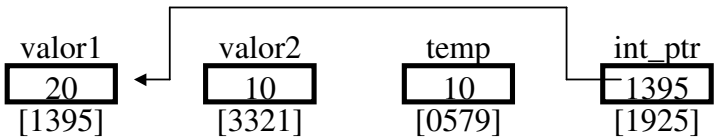
O quinto comando no programa:

```
*int_ptr = valor2;
```

copia o conteúdo da variável *valor2* na célula apontada pelo endereço armazenado em *int_ptr*.




O último comando do programa simplesmente copia o conteúdo da variável inteira *temp* em outra variável inteira *valor2*.




### 6.1.3 Inicializando variáveis do tipo ponteiro

As variáveis do tipo ponteiro, como muitas outras variáveis em C podem ser inicializadas em sua definição. Por exemplo, os seguintes dois comandos alocam espaço para duas células *valor1* e *int_ptr*:

```
int valor1;
int *int_ptr = &valor1;
```

Observe que não foi inicializado **int_ptr* (que poderia ser um inteiro qualquer), mas foi inicializado *int_ptr* (que precisa ser um endereço para um **int**). Esta inicialização na declaração deixa a interpretação do código bastante confusa, pois o exemplo acima poderia ser escrito com mais clareza, apesar de utilizar uma linha a mais assim:

```
int valor1;
int *int_ptr;
int_ptr = &valor1;
```

Já que as strings em C são vetores, podemos inicializar strings nos programas associando-as ao endereço do primeiro caracter, como no exemplo abaixo:

```
#include <stdio.h>
#include <conio.h>

void main()
     {
     char *palindroma = "Socorram o Marrocos";
     int indice;

     for(indice=0;indice < strlen(palindroma); indice++)
          printf("%c",palindroma[indice]);
     printf("\n");
     for(indice=strlen(palindroma)-1;indice>=0; indice--)
          printf("%c",palindroma[indice]);
     printf("\n");
     printf("%s",palindroma);
     printf("\n");
     printf(palindroma);
     }
```

### 6.1.4 Limitações no operador de endereços

O operador de endereços **&** não pode ser utilizado em expressões nos seguintes casos:

a) Não pode ser utilizado com constantes:

```
endereco_variavel = &23;
```

b) Não pode ser utilizado com expressões envolvendo operadores como + / * -:

```
endereco_variavel = &(valor1 + 10);
```

c) Não pode ser utilizado precedendo variáveis declaradas como **register**:

```
register int reg1;
        endereco_variavel = &reg1;
```

### 6.1.5 Ponteiros para matrizes

Ponteiros e matrizes estão intimamente relacionados. O nome de uma matriz é uma constante cujo valor representa o endereço do primeiro elemento da matriz.

Por esta razão, um comando de atribuição ou qualquer outro comando não pode modificar o valor do nome de uma matriz.Com as declarações:

```
float classe[10];
float *float_ptr;
```

o nome da matriz *classe* é uma constante cujo valor é o endereço do primeiro elemento da matriz de 10 **floats**. O comando a seguir atribui o endereço do primeiro elemento da matriz à variável ponteiro *float_ptr*:

```
float_ptr = classe
```

Um comando equivalente é:

```
float_ptr = &classe[0];
```

Entretanto, como *float_ptr* contém o endereço de um **float**, os seguintes comandos são **ilegais** pois tentam atribuir um valor para a constante *classe* ou seu equivalente *&classe[0]*:

```
classe = float_ptr;
&classe[0] = float_ptr;
```

### 6.1.6 Ponteiros para ponteiros

Em C podemos definir variáveis ponteiro que apontam para outra variável ponteiro, que, por sua vez, aponta para os dados.

Isto é necessário para programação em ambientes operacionais multitarefa, como Windows, Windows NT e OS/2 que foram projetados para maximizar o uso da memória. Para compactar o uso da memória, o sistema operacional precisa ser capaz de mover objetos na memória quando necessário.

Se seu programa aponta diretamente para a célula de memória física onde o objeto está armazenado, e o sistema operacional o mover, haverá desastre. Em vez disso, sua aplicação aponta para o endereço da célula de memória que não será modificada enquanto o programa estiver sendo executado (um endereço virtual), e a célula de memória *endereço_virtual* mantém o *endereço_físico_atual* do objeto de dados, atualizada automaticamente pelo sistema operacional.

Para definir um ponteiro para um ponteiro em C, simplesmente aumenta-se o número de asteriscos que precedem o identificador como por exemplo:

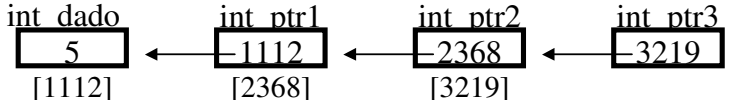
```
char **char_ptr;
```

Neste exemplo, a variável *char_ptr* é definida como um ponteiro para um ponteiro que aponta para um tipo de dados **char**.

Cada asterisco é lido como “ponteiro para”. O número de ponteiros que precisa ser seguido para acessar o item de dados, ou o número de asteriscos que deve ser colocado na variável para referenciar o valor ao qual ela aponta, é chamado de *nível de indireção* da variável ponteiro. O nível de indireção de um ponteiro determina quanta desreferenciação deve ser feita para acessar o tipo de dados na definição.

Observe o exemplo:

```
int int_dados = 5;
int *int_ptr1;
int **int_ptr2;
int ***int_ptr3;
int_ptr1 = &int_dados;
int_ptr2 = &int_ptr1;
int_ptr3 = &int_ptr2;
```



### 6.1.7 Aritmética com ponteiros

Em C também é possíver a realização de aritmética com ponteiros, fazendo-se adição e subtração com variáveis do tipo ponteiro.

Ex.:

```
int *ptr_int;
float *ptr_float;

int um_inteiro;
float um_float;

ptr_int = &um_inteiro;
ptr_float = &um_float;

ptr_int++;
ptr_float++;
```

Observe apenas que se o valor de ptr_int antes do incremento fosse 2000, após o incremento esse valor passou para 2002, pois um incremento de **int** corresponde a 2 bytes, da mesma forma que ptr_float passaria de 3000 para 3004.

## 6.2 Ponteiros para funções

Em **C** é possível que uma função seja ativada através de seu endereço de colocação em memória, ao invés de seu nome. Desta forma, pode-se declarar um ponteiro para uma função (um variável que armazena o endereço de uma função) e evocá-la através desse ponteiro.

Ex.:

```
int (*ptr_func)();
```

Neste caso, *ptr_func* é um ponteiro para uma função que retorna inteiro e não possui parâmetros. Se, por exemplo, o programa contiver uma função chamada *calcula*:

```
int calcula(void);
```

*ptr_func* irá armazenar o endereço de *calcula* se fizermos:

```
ptr_func = calcula;  /*observe que nao é calcula() */
```

A função é ativada através do ponteiro quando se escreve:

```
(*ptr_func)();
```

Outros exemplos:

```
void (*func)(int);
```

***func*** é um ponteiro para uma função **void** que possui um inteiro como parâmetro;

```
float (*ptf)(int,double);
```

***ptf*** é um ponteiro para uma função que retorna **float** e possui como parâmetros um **int** e um **double**.

Os ponteiros para funções têm importantes usos. Por exemplo, considere a função **qsort** (ordena um vetor), que tem como um de seus parâmetros um ponteiro para uma função. Nós não podemos passar uma função por valor, isto é, passar o próprio código, mas podemos passar um ponteiro para o código, ou seja, um ponteiro para a função.

Exemplos:

```
#include <stdio.h>
int soma(int a, int b);
int sob(int a, int b);

void main()
     {
     int (*pf)(int,int);  /* ponteiro para função /*
     int i, a, b, r;

     for(i=0;i<4;i++)
          {
          pf=(i%2==0)?soma:sub;
          printf("\nDigite um valor: ");
          scanf("%d",&a);
          printf("\nDigite outro valor: ");
          scanf("%d",&b);
          r=(*pf)(a,b);
          printf("%d\n", r);
          }
     }

int soma(int a, int b)
     {
     printf("A soma é: \n");
     return(a+b);
     }

int sub(int a, int b);
     {
     printf("A subtração é: \n");
     return(a-b);
     }
```

Outro caso típico de aplicação são os vetores de ponteiros para funções:

```
#include <stdio.h>

void zero();
void um();
void dois();

void main()
     {
     void (*vet[3])(); /* vetor de ponteiros para funções */
     int op;

     vet[0] = zero;
     vet[1] = um;
     vet[2] = dois;
     do
          {
          printf("Digite um número entre 0 e 2: ");
          scanf("%d",&op);
          (*vet[op])();
          }
     while (op<0 || op>2);
     }

void zero()
```

```
     {
     printf("Zero!\n");
     }

void um()
     {
     printf("Um!\n");
     }

void dois()
     {
     printf("Dois!\n");
     }
```

### 6.3 Ponteiros em Pascal

Em Pascal temos praticamente todas as funcionalidades dos ponteiros de C e C++, com pequenas diferenças na sintaxe. A declaração de um ponteiro em Pascal é:

***nome_do_ponteiro*** **: ^*tipo_de_dado*** **;**

Exemplo:

```
px : ^integer;           {ponteiro para um inteiro}
z  : ^real;        {ponteiro para um real}
mm : ^char;        {ponteiro para um char}
```

Para associar um endereço de uma variável a um ponteiro usa-se o operador **@**

```
px := @x;
```

Para mostrar o valor apontado pelo ponteiro usa-se o nome do ponteiro seguido de um **^**

```
writeln( px^ );
```

Exemplo:

```
uses crt;
var
x,y: integer;
px: ^integer;
begin
clrscr;
write('x: ');
readln(x);
write('y: ');
readln(y);
px := @x;
writeln('Valor apontado por px: ', px^);
px := @y;
writeln('Valor apontado por px: ', px^);
writeln('X: ',x);
writeln('Y: ',y);
readkey;
end.
```

Para usar um ponteiro para vetor ou para uma matriz é necessário primeiro criar um tipo para o qual o ponteiro possa apontar e declarar uma variável ponteiro que aponte para o tipo criado. Podemos observar que uma matriz nada mais é do que o endereço do primeiro elemento, portanto o tipo criado não precisa ter o tamanho da matriz a que se quer associar ao ponteiro, mas deve ter as mesmas **dimensões** da matriz:

```
type
y= array[1..1,1..1] of integer;
```

```
var
px : ^y;
```

O ponteiro pode então receber o endereço de uma matriz do mesmo tipo para o qual ele aponta

```
px := @x;
```

A partir dessa associação, pode-se acessar os elementos da matriz através do nome seguido dos índices ou do ponteiro seguido de ^ e dos índices

```
                  writeln( x[8,5] );
é o mesmo que     writeln( px^[8,5] );
```

Exemplo:

```
uses crt;

type
y= array[1..1,1..1] of integer;

var
x: Array[1..10,1..10] of integer;
px: ^y;
i, j: integer;

begin
clrscr;
for i:= 1 to 10 do
    begin
    for j:= 1 to 10 do
        begin
        x[i,j]:=i*10+j;
        end;
    end;
for i:= 1 to 10 do
    begin
    for j:= 1 to 10 do
        begin
        write(x[i,j] :5);
        end;
    writeln;
    end;
writeln('É o mesmo que: ');
px := @x;
for i:= 1 to 10 do
    begin
    for j:= 1 to 10 do
        begin
        write(px^[i,j] :5);
        end;
    writeln;
    end;
readkey;
end.
```

Em Pascal também pode-se utilizar aritmética com ponteiro, de modo que incrementando-os em uma unidade eles são incrementados em tantos bytes quanto o tipo para quem eles apontam. Outra forma de mostrarmos uma matriz usando artimética com ponteiros é:

```
uses crt;
```

```
var
x: Array[1..10,1..10] of integer;
px: ^integer;
i, j: integer;
begin
clrscr;
for i:= 1 to 10 do
    begin
    for j:= 1 to 10 do
        begin
        x[i,j]:=i*10+j;
        end;
    end;
px := @x[1,1];
for i:= 1 to 10 do
    begin
    for j:= 1 to 10 do
        begin
        write(px^ :5);
        px^:=px^+1;
        end;
    writeln;
    end;
readkey;
end.
```

## 7. ESTRUTURAS, UNIÕES E ITENS DIVERSOS

### 7.1 Estruturas

Nós podemos pensar em estruturas como sendo uma matriz ou um vetor de itens intimamente relacionados. Entretanto, ao contrário da matriz ou vetor, uma estrutura permite que se armazene diferentes tipos de dados. O conceito de estrutura de dados é comum no dia-a-dia. Um arquivo de fichas com informações sobre o cliente é uma estrutura de itens relacionados.

Nós podemos criar uma estrutura usando a palavra-chave ***struct*** e a seguinte sintaxe:

```
struct tipo
        {
        tipo var1;
        tipo var2;
        ...
        tipo varn;
        } var_tipo,...;
```

Por exemplo se queremos criar uma estrutura (tipo definido pelo usuário) ***cadastro*** e definirmos uma variável **cliente** que é do tipo (estrutura) **cadastro**:

```
#include <stdio.h>
#include <conio.h>

struct cadastro
     {
     char nome[30];
     char endereco[50];
     char cidade[20];
     } cliente;

void main()
     {
     printf("Nome: ");
     gets(cliente.nome);
     printf("Endereço: ");
     gets(cliente.endereco);
     printf("Cidade: ");
     gets(cliente.cidade);
     clrscr();
     printf("%s mora na rua %s, na cidade de %s", cliente.nome,
cliente.endereco,cliente.cidade);
     }
```

#### 7.1.1 Passando uma estrutura para uma função

Podem ocorrer situações que teremos que passar informações de estruturas para funções. Quando uma estrutura é passada para uma função, as informações da estrutura são passadas por valor, e assim a função não altera a estrutura original. Por exemplo se a impressão dos dados do exemplo anterior fosse passada para uma função:

```
#include <stdio.h>
#include <conio.h>

struct cadastro
     {
     char nome[30];
```

```
     char endereco[50];
     char cidade[20];
     };

void imprime(struct cadastro pessoas);


void main()
     {
     struct cadastro cliente

     printf("Nome: ");
     gets(cliente.nome);
     printf("Endereço: ");
     gets(cliente.endereco);
     printf("Cidade: ");
     gets(cliente.cidade);
     imprime(cliente);
     }

void imprime(struct cadastro pessoas)
     {
     clrscr();
     printf("%s mora na rua %s, na cidade de %s", pessoas.nome, pessoas.endereco,
pessoas.cidade);
     }
```

### 7.1.2 Matriz de Estruturas

Podemos criar uma matriz de estruturas, o que é semelhante a um fichário que contém diversas fichas de clientes. Por exemplo:

```
#include <stdio.h>
#include <conio.h>

struct cadastro
     {
     char nome[30];
     char endereco[50];
     char cidade[20];
     } cliente[5];

void main()
     {
     int i;

     for(i=0;i<5;i++)
          {
          printf("Nome: ");
          gets(cliente[i].nome);
          printf("Endereço: ");
          gets(cliente[i].endereco);
          printf("Cidade: ");
          gets(cliente[i].cidade);
          }
     clrscr();
     for(i=0;i<5;i++)
          {
          printf("%s mora na rua %s, na cidade de %s \n", cliente[i].nome,
cliente[i].endereco, cliente[i].cidade);
          }
     }
```

### 7.1.3 Estruturas dentro de estruturas

Podemos aninhar estruturas, isto é, tornar uma estrutura parte de uma segunda estrutura. Por exemplo:

```
#include <stdio.h>
#include <conio.h>

struct data
     {
     int dia;
     int mes;
     int ano;
     };

struct cadastro
     {
     char nome[30];
     char endereco[50];
     char cidade[20];
     struct data dnasc;
     };

void main()
     {
     int i;
     cadastro cliente[5];

     for(i=0;i<5;i++)
          {
          printf("Nome: ");
          gets(cliente[i].nome);
          printf("Endereço: ");
          gets(cliente[i].endereco);
          printf("Cidade: ");
          gets(cliente[i].cidade);
          fflush(stdin);
          printf("Data de Nascimento: ");
          scanf("%d%d%d", &cliente[i].dnasc.dia, &cliente[i].dnasc.mes,
&cliente[i].dnasc.ano);
          fflush(stdin);
          }
     clrscr();
     for(i=0;i<5;i++)
          {
          printf("%s mora na rua %s, na cidade de %s\n", cliente[i].nome,
cliente[i].endereco,cliente[i].cidade);
          printf("E nasceu no dia %d/%d/%d\n", cliente[i].dnasc.dia,
cliente[i].dnasc.mes, cliente[i].dnasc.ano);
          }
     }
```

### 7.1.4 Ponteiros para estruturas

A utilização de ponteiros para estruturas é uma técnica tão comum que um novo operador, conhecido cmo operador seta pelos programadores de C, foi desenvolvido, e tem este aspecto: **->** . Em outras palavras, se definirmos uma estrutura denominada meus_dados que seja semelhante a:

```
struct grande_estrutura_dados
{
  int chave;
  int grande_vetor[20];
} meus_dados;
```

Então poderemos declarar um ponteiro denominado ***meu_ponteiro*** para ***meus_dados*** como este:

```
struct grande_estrutura_dados *meu_ponteiro;
```

E atribuir um valor a ele desta forma:

```
meu_ponteiro = &meus_dados;
```

A partir de agora, podemos nos referir aos campos em ***meus_dados*** como ***meus_dados.chave***, como:

```
meu_ponteiro->chave = 5;
```

É o mesmo que o comando a seguir:

```
meus_dados.chave = 5;
```

Em outras palavras, ***meu_ponteiro*** é um ponteiro para uma estrutura; para alcançar um campo na estrutura para a qual ele aponta, utilize o operador seta, ->. Essa é a razão pela qual o operador -> foi desenvolvido: para permitir escolher campos isolados da estrutura que está sendo apontada. Foram incluídos os operadores * e & expressamente para aqueles casos em que apontamos para uma estrutura, porque & ou * sozinhos não podem acessar elementos. Ou seja, não existe algo como ***meu_ponteiro.chave*****;** temos que utilizar em seu lugar ***meu_ponteiro->chave***.

Uma estrutura somente pode ser passada para uma função por referência, através de um ponteiro para a estrutura. Por exemplo:

```
#include <stdio.h>
#include <conio.h>

struct produto
     {
     int codigo;
     char descr[80];
     float preco;
     }p1, p2;

void leprod(produto *ptr);
void imprimeprod(produto *ptr);

void main()
     {
     leprod(&p1);
     leprod(&p2);
     clrscr();
     imprimeprod(&p1);
     imprimeprod(&p2);
     getch();
     }

void leprod(produto *ptr)
     {
     int aux;
     float aux2;
     fflush(stdin);
     printf("Codigo: ");
     scanf("%d",&aux);
     ptr->codigo = aux;
     fflush(stdin);
     printf("Descricao: ");
     gets(ptr->descr);
     printf("Preco: ");
     scanf("%f",&aux2);
     ptr->preco = aux2;
     }
```

```
void imprimeprod(produto *ptr)
     {
     printf("%d\n",ptr->codigo);
     puts(ptr->descr);
     printf("%0.2f\n",ptr->preco);
     }
```

### 7.1.5 Estruturas em Pascal e Delphi

A declaração RECORD permite-nos definir um registro:

**<identificador> = RECORD**
**<campo1> : tipo;**
**<campo2> : tipo;**
**...**
**<campon> : tipo;**
**END;**

REGISTRO é um grupo de informações relativas a uma mesma entidade. Estas informações podem ter características diferentes, como em um R.G., onde temos diversos campos com nome, nome do pai, nome da mãe, número do registro, etc.

O registro tem uma particularidade em relação a uma matriz, enquanto em uma matriz podemos ter vários elementos de um único tipo, no registro podemos ter elementos com tipos diferentes. A definição de um registro no Turbo Pascal só pode ser feita na área de tipos (TYPE) e assim sendo, quando definimos um registro, na verdade, estamos criando um tipo novo. Para podermos utilizar este tipo, temos que declará-lo na área de variáveis (VAR), já a manipulação dos campos de um arquivo deve ser feita através da referência do nome do registro e o nome do campo unidos por um ponto (**.**).

```
Exemplos :
TYPE
registro = RECORD
     nome : STRING[30];
     ende : STRING[25];
     fone : STRING[8];
     idade : BYTE;
END;

tela     = RECORD
     atributo : BYTE;
     caracter : CHAR;
END;
```

Exemplo de referência de um campo do registro :

```
PROGRAM teste_reg;

TYPE

registro = RECORD
     nome     :STRING[30];
     ende     :STRING[25];
     fone     :STRING[8];
     idade    :BYTE;
END;

VAR
     reg  : registro;

BEGIN
reg.nome := 'Roberto';
```

```
reg.ende := 'Rua Anônima, 0';
reg.fone := '999-9999';
reg.idade:= 30;
writeln('Nome: ', reg.nome);
writeln('Endereço: ', reg.ende);
writeln('Fone: ', reg.fone);
writeln('Idade: ', reg.idade);
write('Nome: ');
readln(reg.nome);
write('Endereço: '
readln(reg.ende);
write('Fone: '
readln(reg.fone);
write('Idade: '
readln(reg.idade);
writeln('Nome: ', reg.nome);
writeln('Endereço: ', reg.ende);
writeln('Fone: ', reg.fone);
writeln('Idade: ', reg.idade);
END.
```

Pode-se também armazenar os registros em memória, através do uso de vetores ou matrizes de estrutras.

```
Exemplo:
Program ExemploRegistroSeletivo;
uses crt;
type

cadastro = record
         codigo: integer;
         nome: string[30];
         rg: string[15];
         end;

var
cad: Array [1..10] of cadastro;
i: integer;

begin
clrscr;
for i:= 1 to 10 do
    begin
    write('Codigo: ');
    readln(cad[i].codigo);
    write('Nome: ');
    readln(cad[i].nome);
    write('RG: ');
    readln(cad[i].rg);
    end;
clrscr;
for i:= 1 to 10 do
    begin
    writeln(cad[i].codigo, ' - ', cad[i].nome, ' - ', cad[i].rg);
    end;
readkey;
      end.
```

### 7.2 Uniões

Uma União é um tipo de dados muito similar à uma estrutura, com a diferença de que todos os membros de uma união compartilham a mesma área de armazenamento, enquanto cada membro de uma estrutura possui a sua própria área.

As uniões são úteis para economizar memória, em situações nas quais os campos de uma estrutura são mutuamente exclusivos.

Ex.:

```
union algum
     {
     int i;
     char c;
     } u;
```

Neste caso, a variável **"u"** somente poderá conter o valor do campo **"i"** ou do campo **"c"**, mas não de ambos. O compilador reservará área suficiente para o maior tipo contido na união.

Em Pascal e Delphi, podemos ter uma estrutura de registro seletivo :

```
Exemplo:
Program ExemploRegistroSeletivo;
uses crt;
type

  Classe  = (Num, Str);

  uniao = record
        Nome: string[30];
        case campo: Classe of
              Num: (N: real);
              Str: (S: string);
        end;

var
un: uniao;
op: char;

begin
clrscr;
write('Nome: ');
readln(un.nome);
writeln('O que voce quer armazenar: [N] Numero - [S] - String?');
op:=readkey;
if upcase(op)='N' then
   begin
   write('Digite um numero: ');
   readln(un.N);
   writeln(un.nome);
   writeln(un.N:0:6);
   end
else
   begin
   write('Digite uma string: ');
   readln(un.S);
   writeln(un.nome);
   writeln(un.S);
   end;
readkey;
end.
```

Neste tipo de registro, temos a possibilidade de variar a estrutura. dependendo da condição encontrada no decorrer do programa para um campo-sinal previamente declarado como tipo escalar, esta estrutura de seleção é chamada de união discriminada, cabendo desta forma ao programador manter válida.

### 7.3 Itens diversos

#### 7.3.1 typedef

Podemos associar novos tipos de dados com os tipos de dados existentes, usando **typedef**. Por exemplo, podemos definir um tipo de dado chamado ***real*** que é do tipo **float**:

```
#include <stdio.h>

typedef float real;

void main()
     {
     real a, divisao;
     int b;
     a = 7;
     b = 4;
     divisao = a/b;
     printf("%f",divisao);
     }
```

#### 7.3.2 enum

O tipo de dado **enum** permite que criemos um tipo de dado com uma escolha de itens. **enum** é útil quando a informação pode ser melhor representada por uma lista de valores inteiros,

## 8. ALOCAÇÃO DINÂMICA DE MEMÓRIA

Quando um programa em C ou Pascal é compilado, a memória do computador é dividida em quatro zonas que contêm o código do programa, todos os dados globais, a pilha, e o heap. O heap é a área de memória livre (algumas vezes chamada de *armazém livre*) que é manipulada com as funções de alocação dinâmica **malloc** e **free** (C) e **Getmem** e **Freemem** (Pascal).

Quando **malloc** (C) ou **Getmem** (Pascal) é chamada, ela aloca um bloco contíguo de armazenagem para o objeto especificado e então retorna um ponteiro para o início do bloco. A função **free** (C) ou **Freemem** (Pascal) retorna a memória alocada previamente para o heap, permitindo que a porção da memória seja realocada.

O argumento passado para **malloc** é um inteiro não-sinalizado que representa o número necessário de bytes de armazenagem. Se houver memória disponível, **malloc** retornará **void *** que pode ser convertido para o tipo desejado de ponteiro. Um ponteiro **void** significa um ponteiro de tipo desconhecido ou genérico. Um ponteiro **void** não pode ser usado para referenciar qualquer coisa (pois ele não aponta para qualquer tipo específico de dado), mas pode conter um ponteiro de qualquer outro tipo. Portanto podemos converter qualquer ponteiro num ponteiro **void** e reconverter sem qualquer perda de informação.

O seguinte exemplo aloca armazenagem para um número variável de valores **float**:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

float *fpt;
int i, qtd;

void main()
{
clrscr();
printf("Quantidade de valores: ");
scanf("%d",&qtd);
fpt = (float *) malloc( sizeof(float) * qtd);
for (i=0;i<qtd;i++)
     {
     printf("%d§ valor: ",i+1);
     scanf("%f",&fpt[i]);
     }
for (i=0;i<qtd;i++)
     {
     printf("%d§ valor: %6.2f\n",i+1,fpt[i]);
     }
getch();
free(fpt);
}
```

A função **malloc** obtém armazenagem suficiente para tantas vezes o tamanho de um **float** quanto for solicitado. Cada bloco de armazenagem requisitado é inteiramnte separado e distinto de todos os outros. Não podemos fazer suposições sobre onde os blocos serão alocados. Os blocos são tipicamente identificados com algum tipo de informação que permite que o sistema operacional gerencie sua localização e tamanho. Quando o bloco não é mais necessário, podemos retorná-lo para o sistema operacional por meio do seguinte comando:

**free( <** *nome_ponteiro* **> )**

Em Pascal as funções são semelhantes no modo de operar mas um pouco diferentes na sintaxe. O procedimento **Getmem** recebe dois parâmetros, sendo o primeiro o nome do ponteiro e o segundo a quantidade de bytes a serem alocados. Uma diferença a ser observada é que o procedimento de liberação de memória **Freemem** precisa também da informação da quantidade de bytes a serem liberados.

O exemplo abaixo em Pascal é semelhante ao anterior escrito em C:

```
uses crt;
type
  vetor = Array [1..1] of real;
var
fpt: ^vetor;
i, qtd: integer;

begin
clrscr;
write('Quantidade de valores: ');
readln (qtd);
GetMem(fpt, sizeof(real) * qtd);
for i:= 1 to qtd do
      begin
      write(i,'§ valor: ');
      readln(fpt^[i]);
      end;
for i:= 1 to qtd do
      begin
      writeln(i,'§ valor: ', fpt^[i]:6:2);
      end;
readkey;
FreeMem(fpt,sizeof(integer)*qtd);
end.
```

Quando precisamos de variáveis com um tamanho desconhecido na compilação, devemos usar alocação dinamica de memória, no heap. Devemos porém tomar o cuidado de liberar a memória quando não precisamos mais da variável, pois o compilador só devolve automaticamente a memória ocupada pelas variáveis locais. Se não liberarmos a memória corretamente, o programa pode vir a travar.

A grande vantagem da utilização de alocação dinâmica é que permite-se criar estruturas semelhantes a matrizes com dimensões desconhecidas no momento da compilação. Devido a uma limitação da linguagem Pascal, uma matriz ou vetor não pode ocupar mais do que 64 Kbytes de memória, portanto, quando necessitamos de uma matriz ou vetor com mais de 64 Kbytes devemos usar alocação dinâmica de memória.

Pode-se inclusive criar matrizes com número variado de colunas para cada linha, como por exemplo:

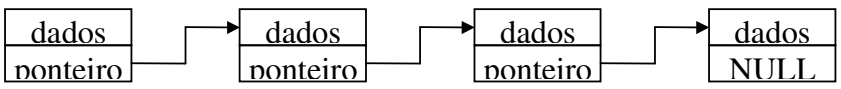
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>

int **matriz;
int *colunas;
unsigned long lin, col, i, j;

void main()
{
clrscr();
printf("Memoria livre: %lu bytes\n", (unsigned long) coreleft());
printf("Linhas: ");
scanf("%d",&lin);
matriz=(int **) malloc(sizeof(int *) * lin);
colunas=(int *) malloc(sizeof(int) * lin);
printf("Memoria livre: %lu bytes\n", (unsigned long) coreleft());
if (matriz==NULL)
     {
     printf("Nao foi possivel alocar memoria\n");
     exit(1);
     }
for(i=0;i<lin;i++)
     {
```

```
     printf("Colunas na linha %d: ",i);
     scanf("%d",&col);
     matriz[i]=(int *) malloc(sizeof(int)*col);
          if(matriz[i]==NULL)
               {
               printf("Nao foi possivel alocar memoria\n");
               exit(1);
               }
     colunas[i]=col;
     printf("Memoria livre: %lu bytes\n", (unsigned long) coreleft());
     }
printf("Memoria livre: %lu bytes\n", (unsigned long) coreleft());
for(i=0;i<lin;i++)
     {
     printf("Linha %d:\n",i);
     for(j=0;j<colunas[i];j++)
          {
          printf("%d§ Valor: ",j);
          scanf("%d",&matriz[i][j]);
          }
     }
for(i=0;i<lin;i++)
     {
     printf("\n");
     for(j=0;j<colunas[i];j++)
          {
          printf("%5d",matriz[i][j]);
          }
     }
for(i=0;i<lin;i++)
     {
     free(matriz[i]);
     }
free(matriz);
free(colunas);
printf("\nMemoria livre: %lu bytes\n", (unsigned long) coreleft());
getch();
}
```

### 8.1 Lista encadeada com alocação dinâmica de memória:

Uma lista encadeada é a forma ideal de armazenamento de dados quando não se sabe antecipadamente quantos itens de dados terão de ser armazenados. Ela funciona ssim: para cada item, existe também um ponteiro direcionado para o item de dados seguinte. A qualquer momento, pode-se acrescenta outro item de dados à lista, contanto que o que o último ponteiro seja atualizado a fim de apontar para o novo item de dados. Partindo-se do início da lista, é possível percorrê-la utilizando-se o ponterio de cada item, que aponta para o próximo item de dados. O útlimo ponteiro da lista é geralmetne um ponteiro NULL, portando, sabemos que estamos no final da lista quando encontramos um ponteiro NULL. É preciso também termos 3 ponteiros auxiliares, para a alocação dinâmica da memória: um ponteiro que aponta para o **primeiro** elemento, para saber onde inicia a lista, um ponteiro que aponta para o **último** elemento da lista, para colocarmos o valor do novo ponteiro quando é inserido um elemento sem ter que percorrer a lista inteira, e um ponteiro **auxiliar**, que aponta para a nova área de memória alocada.

Esquematicamente, uma lista encadeada é semelhante a:



Podemos observar no exemplo a seguir, uma lista encadeada com alocação dinâmica de memória.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

typedef struct produto
     {
     int codigo;
     char descr[40];
     float preco;
     struct produto *prox;
     }produto;

produto *prim, *ult, *aux;
int cod;

void ledados(produto *p, int c);
void imprime();
void liberamem();


void main()
     {
     prim=ult=NULL;
     while(1)
          {
          clrscr();
          printf("Codigo: ");
          scanf("%d",&cod);
          if (cod<0) break;
          aux=(produto *) malloc(sizeof(produto));
          if (aux==NULL)
               {
               printf("Sem memoria");
               exit(1);
               }
          ledados(aux,cod);
          if(prim==NULL)
               {
               prim=aux;
               ult=prim;
               }
          else
               {
               ult->prox=aux;
               ult=aux;
               }
          }
     imprime();
     getch();
     liberamem();
     }

void ledados(produto *p, int c)
     {
     float precoaux;
     clrscr();
     p->codigo=c;
     fflush(stdin);
     printf("Codigo: %d\n",p->codigo);
     printf("Descricao: ");
     gets(p->descr);
     fflush(stdin);
     printf("Preco: ");
     scanf("%f",&precoaux);
```

```
     p->preco=precoaux;
     p->prox=NULL;
     }

void imprime()
     {
     aux=prim;
     while(aux!=NULL)
          {
          printf("%d - %s - %f\n",aux->codigo, aux->descr, aux->preco);
          aux = aux->prox;
          }
     }

void liberamem()
     {
     aux=prim;
     while(aux!=NULL)
          {
          free(aux);
          aux = aux->prox;
          }
            }
```

O exemplo abaixo é a implementação de uma lista encadeada com alocação de memória simples, sem ordenação sendo toda nova informação incluída no final da lista:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <ctype.h>

void liberar();
void listar();
void insere(unsigned char infoaux);

struct lista
{
unsigned char info;
struct lista *elo;
};


struct lista *pri, *aux, *ult, *pen;
unsigned char infoaux;

void main()
{
clrscr();
pri=NULL;
ult=NULL;
do
     {
     gotoxy(1,1);
     printf("Digite uma Letra: ");
     infoaux=toupper(getch());
     if ((infoaux>=65)&&(infoaux<=90)) insere(infoaux);
     listar();
     }
while ((infoaux>=65)&&(infoaux<=90));
getch();
liberar();
}

void listar()
{
```

```
struct lista *lult;
lult=pri;
gotoxy(1,10);
while(lult!=NULL)
     {
     printf("%c ",lult->info);
     lult=lult->elo;
     }
}


void liberar()
{
struct lista *lult;
lult=pri;
gotoxy(1,10);
while(lult!=NULL)
     {
     aux=lult;
     lult=lult->elo;
     free(aux);
     }
}


void insere(unsigned char infoaux)
{
aux=(struct lista *) malloc(sizeof(struct lista));
if (aux==NULL)
     {
     printf("Nao foi possivel alocar memoria\n");
     exit(1);
     }
if (pri==NULL)
     {
     aux->info=infoaux;
     aux->elo=NULL;
     pri=aux;
     ult=pri;
     }
else
     {
     aux->info=infoaux;
     aux->elo=NULL;
     ult->elo=aux;
     ult=aux;
     }
}
```

O exemplo abaixo é a implementação de uma lista encadeada com alocação de memória com ordenação das informações, ou seja, as informações são inseridas na lista de forma que ela sempre está com os dados em ordem crescente:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <ctype.h>

void liberar();
void listar();
void insere(unsigned char infoaux);

struct lista
{
unsigned char info;
struct lista *elo;
```

```
};


struct lista *pri, *aux, *ult, *pen;
unsigned char infoaux;

void main()
{
clrscr();
pri=NULL;
do
     {
     gotoxy(1,1);
     printf("Digite uma Letra: ");
     infoaux=toupper(getch());
     if ((infoaux>=65)&&(infoaux<=90)) insere(infoaux);
     listar();
     }
while ((infoaux>=65)&&(infoaux<=90));
getch();
liberar();
}

void listar()
{
struct lista *lult;
lult=pri;
gotoxy(1,10);
while(lult!=NULL)
     {
     printf("%c ",lult->info);
     lult=lult->elo;
     }
}


void liberar()
{
struct lista *lult;
lult=pri;
gotoxy(1,10);
while(lult!=NULL)
     {
     aux=lult;
     lult=lult->elo;
     free(aux);
     }
}


void insere(unsigned char infoaux)
{
int inseri;

aux=(struct lista *) malloc(sizeof(struct lista));
if (aux==NULL)
     {
     printf("Nao foi possivel alocar memoria\n");
     exit(1);
     }
if (pri==NULL)
     {
     aux->info=infoaux;
     aux->elo=NULL;
     pri=aux;
     }
else
```

```
     {
     if(pri->info > infoaux)
          {
          aux->info=infoaux;
          aux->elo=pri;
          pri=aux;
          }
     else
          {
          if(pri->elo == NULL)
               {
               aux->info=infoaux;
               aux->elo=NULL;
               pri->elo=aux;
               }
          else
               {
               inseri=0;
               pen=pri;
               ult=pri;
               while(inseri == 0 && ult->elo != NULL)
                    {
                    if (ult->elo != NULL)
                         {
                         pen=ult;
                         ult=ult->elo;
                         }
                    if (ult->info > infoaux)
                         {
                         aux->info=infoaux;
                         aux->elo=ult;
                         pen->elo=aux;
                         inseri=1;
                         }
                    }
               if (inseri==0)
                    {
                    aux->info=infoaux;
                    aux->elo =NULL;
                    ult->elo =aux;
                    }
               }
          }
     }
}
```

## 9. ARQUIVOS

Para começarmos a manipular arquivos, necessitamos lembrar alguns conceitos básicos a respeito dos mesmos. Quando pensamos em arquivos, a primeira coisa que nos vem à memória é a idéia de um arquivo físico igual aos arquivos encontrados nos escritórios. De forma geral um arquivo serve para armazenar informações através de fichas. Em informática, estes conceitos são perfeitamente válidos, pois para o armazenamento de informações em um arquivo, temos que dar uma forma ao arquivo e dar forma às informações que estarão contidas nele.

### 9.1 Arquivo tipado em Delphi e Pascal

#### 9.1.1 Declaração de arquivos

A primeira preocupação é com relação ao conteúdo de um arquivo. Em um arquivo físico, este conteúdo é feito normalmente através de fichas que têm informações de uma mesma forma. Para nós, esta ficha é um registro e como já vimos anteriormente, o registro é uma estrutura que deve ser definido, na área de tipos, porém só a definição de um registro não implica na formação de um arquivo. Para formá-lo devemos ter um conjunto destas fichas, no nosso caso, devemos declarar uma variável do tipo arquivo e esta variável pode ser declarada de duas formas básicas :

**FILE**

Esta declaração define uma variável como sendo arquivo. Sua sintaxe :

**identificador : FILE OF tipo;**

ou

**identificador : FILE;**

Vejamos um exemplo de definição de um tipo arquivo:

```
TYPE
registro = RECORD
          nome : STRING[30];
          cep : LONGINT;
          rg : STRING[8];
          cic : STRING[11];
     END;

VAR
arquivo_pessoal : FILE OF registro;
arquivo_numerico : FILE OF BYTE;
arquivo_sem_tipo : FILE;
```

Porém, somente a declaração de uma variável do tipo arquivo não quer dizer que já podemos manipular o arquivo, assim como no arquivo físico, aquele de aço, aqui teremos tarefas semelhantes, como por exemplo abrir uma gaveta, retirar ou colocar uma ficha, fechar uma gaveta, identificar o arquivo através de uma etiqueta qualquer, organizar as informações, etc.

Veremos então os comandos para fazermos tais tarefas:

O primeiro comando que veremos é o que nos permite associar a variável do tipo arquivo ao nome externo deste arquivo, ou seja, o nome local que este deve estar.

**ASSIGN**

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

**ASSIGN(VAR <arq>, <nomearq> : STRING);**

arq, deve ser uma variável do tipo arquivo. por exemplo:

```
Exemplo:
PROGRAM teste_assign;
VAR
arq1, arq2 : FILE OF BYTE;

BEGIN
ASSIGN(arq2,'teste.dat');
ASSIGN(arq1,'');    {o nome externo do arquivo omitido}
END.                {determina que o dispositivo de saida }
                    {será o standart   }
```

### 9.1.2 Funções de abertura e fechamento de arquivos

Com a definição de uma variável do tipo arquivo e a sua associação a um nome de arquivo externo, já temos um bom caminho andado, porém ainda nos faltam alguns comandos básicos para, efetivamente, podermos manipular os arquivos. Voltando à analogia com um arquivo de aço, veremos que a primeira coisa que faremos para dar início a manipulação deste arquivo é abri-lo. Pois bem, no nosso arquivo, também deveremos ter esta atividade, ou seja, abrir o arquivo e para isto temos dois comandos básicos:

**RESET**

Este procedimento permite-nos abrir um arquivo já existente. No caso do uso deste, para a tentativa de abertura de um arquivo não existente, ocorrerá um erro de execução. Para que o procedimento tenha sucesso é necessário que antes de executá-lo, tenhamos utilizado o procedimento ASSIGN. Sua sintaxe :

**RESET(VAR <arquivo> [:FILE; <tamanho> : WORD]);**

```
Exemplo:
PROGRAM teste_reset;
Uses CRT;
VAR
arquivo : FILE OF BYTE;
nomearq : STRING[67];
BEGIN
WRITE('Entre com o nome do arquivo que quer abrir ') ;
READLN(nomearq);
ASSIGN(arquivo,nomearq);
RESET(arquivo);
END.
```

Este comando apenas permite-nos abrir um arquivo já existente. para que possamos abrir um arquivo novo, temos um outro comando:

**REWRITE**

Este comando permite criar e abrir um novo arquivo. Caso o arquivo já exista, terá seu conteúdo eliminado e será gerado um novo arquivo. Antes de executarmos este procedimento, devemos usar o ASSIGN, e caso a variável do tipo arquivo não seja tipada, o tamanho de cada registro será de 128 bytes, isto se não for especificado o tamanho do registro. Sua sintaxe:

**REWRITE(VAR <arquivo> [: FILE ; <tamanho> : WORD]);**

Vejamos um exemplo, usando o comando REWRITE :

```
PROGRAM teste_rewrite;
Uses Crt;
VAR
arquivo : FILE OF BYTE;
nomearq: STRING[67];

BEGIN
WRITE('Entre com o nome do arquivo que quer abrir ');
READLN(nomearq);
ASSIGN(arquivo,nomearq);
REWRITE(arquivo);
END.
```

Em ambos os exemplos anteriores, vemos uma deficiência. Enquanto um comando apenas nos permite a abertura de arquivos já existentes, o outro apenas abre arquivos novos ou destrói um possível conteúdo anterior.

Para resolver esse problema podemos usar as diretivas de compilação para checagem de erros de entrada/saída **{$I }**. Esta diretiva retorna um código de erro em uma função do Turbo Pascal chamada **IORESULT**. Para não abortar um programa quando ocorre um erro de I/O, usa-se a diretiva **{$I–}**. Para voltar ao padrão, usa-se a diretiva **{$I+}**.

Por exemplo:

```
PROGRAM teste_rewrite_ou_close;
Uses Crt;
VAR
arquivo : FILE OF BYTE;
nomearq: STRING[67];

BEGIN
WRITE('Entre com o nome do arquivo que quer abrir ');
READLN(nomearq);
ASSIGN(arquivo,nomearq);
{$I-}
RESET(arquivo);
{$I+}
if IORESULT <> 0 then REWRITE(arquivo);
END.
```

A segunda tarefa que devemos nos preocupar é o fato do arquivo permanecer aberto, ou abrirmos o mesmo arquivo mais de uma vez. Para termos um uso adequado de um arquivo, devemos abri-lo e depois de utilizado fechá-lo.

**RESET**

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

**CLOSE (VAR <arq>);**

Exemplo :

```
PROGRAM teste_close;

VAR
arquivo : FILE OF BYTE;
nomearq: STRING[67];

BEGIN
WRITE('Entre com o nome do arquivo que quer abrir');
READLN(nomearq);
ASSIGN(arquivo,nomearq);
REWRITE(arquivo);
CLOSE(arquivo);
END.
```

### 9.1.3 Funções de escrita e gravação

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

#### 9.1.3.1 WRITE

Este procedimento além de ser usado para exibir mensagens e variáveis no dispositivo padrão de saída (video), pode ser usado para gravar informações em outros dispositivos, como um arquivo. Sua sintaxe:

**WRITE(<arq>, <reg1>[,<reg2>,..., <regn>] ;**

Exemplo:

```
PROGRAM teste_write_arq;
CONST
nomearq = 'TESTE.DAT';
max     = 10;

TYPE
registro = RECORD
     nome : STRING[30];
     ender: STRING[25];
END;

VAR
arquivo : FILE OF registro;
reg    : registro;
ind    : BYTE;

BEGIN
ASSIGN(arquivo,nomearq);
REWRITE(arquivo);
FOR ind := 1 TO MAX DO
     BEGIN
     WRITE('Digite o ',ind,'o Nome ');
     READLN(reg.nome);
     WRITE('Digite o ',ind,'o Endereco ');
```

```
     READLN(reg.ender);
     WRITE(arquivo,reg);
     END;
CLOSE(arquivo);
END.
```

#### 9.1.3.2 READ

Como já vimos anteriormente, este procedimento permite que atribuamos a uma variável, um valor obtido por um dispositivo associado. Este dispositivo pode ser também um arquivo, e se no caso da leitura em arquivo, for detectado o fim do mesmo, será gerado um erro. Sua sintaxe:

**READ(<arq>, <reg>);**

Vejamos um exemplo simples de uma leitura em arquivos:

```
PROGRAM teste_read_arq;
Uses Crt;
CONST
nomearq = ´TESTE.DAT´;
max     = 10;

TYPE
registro = RECORD
     nome  : STRING[30];
     ender : STRING[25];
END;

VAR
arquivo: FILE OF registro;

reg: registro;
ind: BYTE;

BEGIN
ASSIGN(arquivo,nomearq);
RESET(arquivo);
FOR ind := 1 TO MAX DO
     BEGIN
     READ(arquivo,reg);
     WRITELN ( 'Este é o ', ind, '.º Nome ', reg.nome) ;
     WRITELN('Este é ' o ',ind,'.º Endereco ',reg.ender);
     END;
CLOSE(arquivo);
END.
```

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

#### 9.1.3.3 EOF

Esta função nos retorna o valor TRUE quando for encontrada a marca de fim de arquivo. Sua sintaxe :

**EOF(VAR <arq>) : BOOLEAN;**

Utilizando o exemplo anterior, com uma pequena variação :

```
PROGRAM teste_eof;
Uses crt;
CONST
     nomearq = 'TESTE.DAT';

TYPE
registro = RECORD
     nome : STRING[30];
     ender: STRING[25];
END;

VAR

arquivo: FILE OF registro;
reg: registro;
ind: BYTE;

BEGIN
ASSIGN(arquivo,nomearq);
RESET(arquivo);
ind := 1;
WHILE NOT EOF(arquivo) DO
     BEGIN
     READ(arquivo,reg);
     WRITELN ( 'Este é o ´,ind,'.º Nome ',reg.nome) ;
     WRITELN('Este é o ',ind,'.º Endereco ', reg.ender);
     ind := ind + 1;
     END;
CLOSE(arquivo);
END.
```

O laço é repetido até que seja encontrada a marca de fim de arquivo.

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

Quando estamos manipulando um arquivo, existe a todo momento um ponteiro que fica deslocando-se de acordo com as leituras e gravações, ou seja, quando abrimos um arquivo,este ponteiro indica a posição zero, ou que é o primeiro registro fisico do arquivo. A cada leitura ou gravação este ponteiro avança uma posição, mas existem algumas funções e procedimentos que permitem a manipulação deste ponteiro.

#### 9.1.3.4 SEEK

Este procedimento permite que movamos o ponteiro do arquivo para uma posição preestabelecida, podendo ser usado em arquivos de qualquer tipo exceto os de tipo TEXTO. Só pode ser usado em arquivos previamente abertos. Sua sintaxe :

**SEEK(VAR <arq>; <posicao> : LONGINT);**

Exemplo:

```
PROGRAM teste_seek;
CONST
max     = 10;

TYPE
registro = RECORD
     nome : STRING[30];
```

```
     ender: STRING[25];
END;

VAR
arquivo : FILE OF registro;
reg     : registro;
ind     : BYTE;

BEGIN
ASSIGN(arquivo,'TESTE.DAT');
REWRITE(arquivo);
FOR ind := 1 TO MAX DO
     BEGIN
     WRITE('Digite o ',ind,'o Nome ');
     READLN(reg.nome);
     WRITE('Digite o ',ind,'o Endereco ');
     READLN(reg.ender);
     WRITE(arquivo,reg);
     END;
SEEK(arquivo,4);
READ(arquivo,reg);
WRITELN ('Este é o 5º Nome ',reg.nome) ;
WRITELN ('Este é o 5º Endereco ', reg.ender);
CLOSE(arquivo);
END.
```

#### 9.1.3.5 FILEPOS

Esta função nos retoma a posição atual do ponteiro do arquivo. Assim que abrimos um arquivo, com RESET ou REWRITE, a posição do ponteiro é zero, ou seja, o primeiro registro, não pode ser usado em arquivos do tipo TEXTO. Sua sintaxe :

**FILEPOS(VAR <arq>) : LONGINT;**

#### 9.1.3.6 FILESIZE

Esta função retorna o tamanho de um arquivo em número de registros. Caso o arquivo esteja vazio, a função retornará 0. Não pode ser usada, em arquivos do tipo TEXTO. Sua sintaxe:

**FILESIZE(VAR <arq>) : LONGINT;**

Vejamos agora, um exemplo englobando estes últimos comandos:

```
PROGRAM  teste_ponteiro;
Uses Crt;
VAR
nomearq: STRING[67];
arquivo: FILE OF BYTE;
tamanho: LONGINT;

BEGIN
WRITE('Entre com o nome do arquivo ');
READLN(nomearq);
ASSIGN(arquivo,nomearq);
RESET(arquivo);
TAMANHO := FILESIZE(arquivo);
WRITELN('O tamanho do arquivo ',nomearq, ´ em bytes ',tamanho);
```

```
WRITELN(’Posicionando no meio do arquivo... ’);
SEEK(arquivo,tamanho DIV 2);
WRITELN ('A posicao atual é ', FILEPOS (arquivo) ) ;
CLOSE(arquivo);
READLN;
END.
```

#### 9.1.3.7 WITH

Este comando permite que a referência a uma variável do tipo registro seja simplificada. Podemos encadear até sete registros em um comando WITH. A referência aos campos dos registros pode ser efetuada com o uso deste comando sem a utilização do ponto, ligando o nome do registro ao nome do campo. Sua sintaxe:

**WITH <registro1>, [<registron>...] DO BLOCO;**

```
Exemplo
PROGRAM teste_with;
CONST
max     = 10;

TYPE
registro = RECORD
     nome : STRING[30];
     ender: STRING[25];
END;

VAR
arquivo : FILE OF registro;
reg    : registro;
ind    : BYTE;

BEGIN
ASSIGN(arquivo,’TESTE.DAT’);
REWRITE(arquivo);
WITH reg DO
     BEGIN
     FOR ind := 1 TO MAX DO
      BEGIN
      WRITE('Digite o ',ind,'° Nome ');
      READLN(nome);
      WRITE('Digite o ',ind,'° Endereco ');
      READLN(ender);
      WRITE(arquivo,reg);
      END;
     SEEK(arquivo,0);
     WHILE NOT EOF(arquivo) DO
           BEGIN
           READ(arquivo,reg);
           WRITELN ('Este é o ',filepos(arquivo),'° Nome ',nome) ;
           WRITELN ('Este é o ',filepos(arquivo),'° Endereco ', ender);
           END;
     END;
CLOSE(arquivo);
END.
```

### 9.2 Arquivos Texto em Pascal e Delphi

Uma outra forma de se trabalhar com um arquivo, é definindo-o como sendo do tipo Texto. Esta forma de definição permite-nos ter registros de tamanhos diferentes, e cada registro é identificado pela sequência CR LF, Carriage

Retum, e Line Feed, correspondentes aos caracteres ASCII $0D e $0A. O identificador TEXT define uma variável como sendo arquivo do tipo texto.

Exemplo:

```
PROGRAM teste_text;

USES CRT;

VAR
arq : TEXT;
ch : CHAR;

BEGIN
WRITELN('Escolha onde deseja que seja impressa a frase
WRITELN(' Impressora, Disco, ou Video, (I, D,V) ' ) ;
REPEAT
ch := UPCASE(READKEY);
UNTIL ch IN ['I','D','V'];
IF ch = 'I' THEN
     ASSIGN(arq,'PRN')
     ELSE
     IF ch = 'D' THEN
          ASSIGN(arq,'TESTE.DAT')
          ELSE
          ASSIGN(arq,'CON');
{$I-}
REWRITE(ARQ);
{$I+}
IF IORESULT <> O THEN
     WRITELN('Arquivo nao aberto ')
     ELSE
     BEGIN
     WRITELN(arq,'Teste de TEXT ');
     CLOSE(arq);
END;
WRITE('Tecle algo para continuar ');
READKEY;
END.
```

Este outro exemplo copia o arquivo "autoexec.bat" para um novo arquivo removendo todas as linhas marcadas com comentário (rem).

```
Program ExemploArquivoTexto;
Uses CRT, Strings;

var
arq1, arq2: text;
buffer: string;
aux: string[3];
i: integer;

begin
assign(arq1,'c:\autoexec.bat');
assign(arq2,'c:\autoexec.new');
{$I-}
reset(arq1);
{$I+}
if IORESULT <> 0 then
   begin
   writeln('Nao foi possivel abrir o arquivo C:\AUTOEXEC,BAT');
   end;
rewrite(arq2);
```

```
while not eof(arq1) do
      begin
      readln(arq1,buffer);
      aux:=copy(buffer,1,3);
      for i:= 1 to 3 do
          begin
          aux[i]:=upcase(aux[i]);
          end;
      if aux<>'REM' then
         begin
         writeln(arq2,buffer);
         end;
      end;
writeln(arq2,'rem - Alterado pela exemplo de Arquivos Texto usando Pascal!');
flush(arq2);
end.
```

### 9.2.1 Funções para manipulação de arquivos texto

#### 9.2.1.1 APPEND

Procedimento que permite a abertura de um arquivo do tipo TEXT, permitindo-se que se faça inclusão ao seu final. Se por acaso existir um caractere ^Z (ASCII 26) no arquivo, será assumido este ponto como fim do mesmo. O arquivo deverá ser previamente associado a um nome externo, e após o uso deste procedimento somente será permitida a inclusão de dados ao seu final. Sua sintaxe:

**APPEND(var <arq> : TEXT);**

Este procedimento não permite que se faça a abertura de um arquivo que ainda não existe. O seu uso correto se faz apenas quando o arquivo já existir. Se em seu disco de trabalho existe um arquivo chamado AUTOEXEC.BAT, faça o seguinte teste :

```
PROGRAM teste_append;
VAR
arq : TEXT; {define arq do tipo arquivo texto}
BEGIN
ASSIGN(arq,'\autoexec.bat'); {associa o nome externo a  arq}
APPEND(arq);
WRITELN(arq,'ECHO INCLUIDO PELO TURBO');
CLOSE(arq);
END.
```

#### 9.2.1.2 EOF(TEXT)

Esta função retorna verdadeiro se for encontrado o fim de um arquivo do tipo texto. A diferença entre EOF para Texto e EOF para arquivos tipados é que na primeira, a referência ao arquivo é opcional. Sua sintaxe:

**EOF[(VAR <arq> : TEXT)] : BOOLEAN;**

#### 9.2.1.3 SEEKEOF

Esta função retorna verdadeiro se encontrado status de final de arquivo. É bastante parecida com a função EOF, exceto que esta salta todos os brancos e tabs quando da leitura. Somente para arquivos do tipo texto. Sua sintaxe:

**SEEKEOF[(VAR <arq>:TEXT)] : BOOLEAN;**

#### 9.2.1.4 SEEKEOLN

Esta função retorna verdadeira se encontrada marca de fim de linha, salta todos os caracteres em branco e tabulações encontradas durante a leitura. Sua sintaxe:

**SEEKEOLN[(VAR <arq> : TEXT)] : BOOLEAN;**

#### 9.2.1.5 FLUSH

Este procedimento permite que seja descarregada a área de Buffer de um arquivo do tipo Texto. Em sua utilização, pode ser usada a função IORESULT, que retornará zero caso a operação for bem sucedida. Sua sintaxe :

**FLUSH(VAR <arq> : TEXT);**

```
Exemplo:
PROGRAM teste_flush;
var
arq : TEXT;

BEGIN
ASSIGN(arq,'\autoexec.bat');
APPEND(arq);
WRITELN(arq,'ECHO INCLUIDO PELO TURBO');
FLUSH(arq);
END.
```

No exemplo anterior, caso não tivéssemos nos utilizando da rotina FLUSH, a gravação não teria sido efetivada, pois a informação estaria apenas bufferizada.

#### 9.2.1.6 SETTEXTBUF

Este procedimento permite-nos associar um arquivo com tipo TEXTO a uma área de buffer na área de dados do programa. Utilize sempre que possivel múltiplos de 128 para o tamanho do Buffer. Normalmente, esta área já está disposta pelo sistema operacional e corresponde a 128 Bytes. Este comando deve ser usado antes que o arquivo seja aberto, podendo ser usado tanto para leitura como para gravação. Este procedimento acelera os procedimentos de leitura e de gravação. Sua sintaxe :

**SETTEXTBUF(VAR <arq>:TEXT; VAR <buffer> [;<tamanho> : WORD]);**

### 9.3 ARQUIVOS SEM TIPOS EM PASCAL E DELPHI

Às vezes, temos a necessidade de manipular arquivos sem saber qual seu tipo de conteúdo. Poderiamos defini-lo do tipo BYTE, ou ainda, do tipo CHAR, porém se o arquivo tiver um tamanho relativamente grande, sua manipulação será meio lenta e neste caso, temos algumas opções.

#### 9.3.1 Funções para manipulação de arquivos sem tipos

#### 9.3.1.1 BLOKREAD

Procedimento que permite a leitura de um ou mais registros (blocos de 128 bytes), num máximo de 65535 bytes (64 K) de um arquivo sem tipo em uma variável que pode ser de qualquer tipo. Sua sintaxe:

**BLOCKREAD(VAR <arq>: FILE; VAR <buffer>; <contador> : WORD; [VAR <resultado> : WORD];**

#### 9.3.1.2 BLOCKWRITE

Procedimento que permite a gravação de um ou mais registros (bloco de 128 bytes) de um arquivo sem tipo em uma variável que pode ser de qualquer tipo.

Sua sintaxe:

**BLOCKWRITE(VAR <arq>: FILE; VAR <buffer>; <contador> : WORD; [VAR <resultado> : WORD];**

Exemplo que utiliza estes comandos :

```
PROGRAM teste_biock;

VAR
arqent,arqsai    : FILE;
nomeent,nomesai  : STRING[79];
lidos, gravados  : WORD;
buf              : ARRAY[1..4096] OF CHAR;


BEGIN
WRITE('Entre com o nome do arquivo de Origem ');
READLN(nomeent);
ASSIGN(arqent,nomeent);
RESET(arqent,l);
WRITE ('Entre com o nome do arquivo de Destino ') ;
READLN(nomesai);
ASSIGN(arqsai,nomesai);
REWRITE(arqsai,l);
WRITELN('Copiando ',
FileSize(arqent),' bytes...');
REPEAT
     BLOCKREAD(arqent,buf,sizeof(buf),lidos);
     BLOCKWRITE(arqsai,buf,lidos,gravados);
UNTIL (lidos = 0) OR (gravados <> lidos);
Close(arqent);
Close(arqsai);
END.
```

É de fundamental importância, que ao abrir o arquivo coloque-se além da variável tipo File, também o tamanho 1.

#### 9.3.1.3 TRUNCATE

Este procedimento faz com que o arquivo seja truncado em seu tamanho, na posição corrente do ponteiro do arquivo. Sua sintaxe ;

**TRUNCATE(VAR <arq>);**

### 9.3.2 Arquivos com diversas estruturas em Pascal e Delphi

Um arquivo pode conter diversas estruturas. Para uma leitura correta desses arquivos é necessário conhecer o "layout" desse arquivo.

Por exemplo, um arquivo DBF, utilizado por dBase, Clipper, FoxPro e reconhecido por qualquer programa que importe base de dados, é dividido em 3 partes distintas: cabeçalho, atributos dos campos e dados.

O cabeçalho possui um registro com a seguinte definição:

```
str_dbf = record
        versao: byte;
        ano: byte;
        mes: byte;
        dia: byte;
        n_regs: longint;
        tam_cab: integer;
        tam_reg: integer;
        reservado: Array[1..20] of char;
        end;
```

A seguir para cada atributo (campo) há o seguinte registro:

```
descr_atr = record
          nome: array[1..11] of char;
          tipo: char;
          reservado1: array[1..4] of char;
          tamanho: byte;
          num_dec: byte;
          reservado2: array[1..14] of char;
          end;
```

A quantidade de atributos é dada pela fórmula:

*atributos := trunc((dbf.tam_cab-sizeof(str_dbf))/sizeof(descr_atr));*

Os dados são gravados a partir do 1º byte posterior ao tamano do cabeçalho (cab.tam_cab). No início de cada registro (conjunto de atributos) é gravado um byte para marcar se o registro está deletado (se o caracter for igual a 42 (*)) ou não (se o caracter for igual a 32 (espaço em branco)), e são todos gravados em modo texto, sendo que cada registro referente a cada atributo possui o tamanho definido na descrição do atributo (tamanho + num_dec).

```
Program LeDBF;
Uses CRT;

TYPE

str_dbf = record
        versao: byte;
        ano: byte;
        mes: byte;
        dia: byte;
        n_regs: longint;
        tam_cab: integer;
        tam_reg: integer;
        reservado: Array[1..20] of char;
        end;
```

```
descr_atr = record
          nome: array[1..11] of char;
          tipo: char;
          reservado1: array[1..4] of char;
          tamanho: byte;
          num_dec: byte;
          reservado2: array[1..14] of char;
          end;

VAR
desc: descr_atr;
dbf: str_dbf;
arq:FILE;
quant, i: integer;
aux: char;
nomeaux: Array[1..256] of string[12];
tipo: Array[1..256] of char;
tam, dec: Array[1..256] of byte;
buffer: char;
nome_arquivo: string;
tm, cont, k, qtd, l, tm1, tm2, op, tamanho, zz, byteslidos: integer;

BEGIN
clrscr;
write('Nome do Arquivo: ');
readln(nome_arquivo);
assign(arq,nome_arquivo);
{$I-}
reset(arq,1);
{$I+}
if IORESULT <> 0 then
   begin
   writeln('Nome de Arquivo Invalido!');
   readkey;
   exit;
   end;
BLOCKREAD(arq,dbf,32,byteslidos);
writeln('versao: ',dbf.versao);
writeln('ano:    ',dbf.ano);
writeln('mes:    ',dbf.mes);
writeln('dia:    ',dbf.dia);
writeln('n§ reg: ',dbf.n_regs);
writeln('t_cab:  ',dbf.tam_cab);
writeln('t_reg:  ',dbf.tam_reg);
writeln('reserv: ',dbf.reservado);
writeln('-------------------------------');
readkey;
quant := trunc((dbf.tam_cab-sizeof(str_dbf))/sizeof(descr_atr));
writeln('Blidos: ',byteslidos);
writeln('Quant: ',quant);
writeln('Tam cab: ',sizeof(str_dbf));
writeln('Tam campos: ',sizeof(descr_atr));
for i:=1 to quant do
    begin
    BLOCKREAD(arq,desc,sizeof(descr_atr),byteslidos);
    tipo[i]:=desc.tipo;
    tam[i]:=desc.tamanho;
    nomeaux[i]:=desc.nome;
    dec[i]:=desc.num_dec;
    writeln(desc.nome, ' - ',desc.tipo, ' - ',desc.tamanho, ' - ',desc.num_dec);
    end;

write('Ler quantos registros - (999 - todos)? ');
readln(qtd);
seek(arq,dbf.tam_cab);
if (qtd=999) then qtd:=dbf.n_regs;
cont:=1;
```

```
while ((cont<=qtd) and (not eof(arq))) do
      begin
      writeln('  Registro: ',cont,' -------------------------');
      BLOCKREAD(arq,aux,1,byteslidos);
      if (aux<>' ') then writeln('-> ',aux,' (deletado) ');
      for k:=1 to quant do
          begin
          write(nomeaux[k],': ');
          for l:= 1 to tam[k] do
              begin
              BLOCKREAD(arq,buffer,1,byteslidos);
              write(buffer);
              end;
          writeln;
          end;
      readkey;
      cont:=cont+1;
      end;
writeln('      ************* FIM ');

readkey;
clrscr;
end.
```

### 9.4 Arquivos em C

Um arquivo em "C" não possui a idéia de registro. Podemos gravar dados no formato de caracter (ASCII) ou em binário ("binary file"). A seqüência e interpretação dos dados é de inteira responsabilidade do programador do sistema. Um arquivo em "C" é uma seqüência contínua de butes gravados em memória secundária. Cada byte do arquivo possui um endereço único que é o deslocamento relativo ao início do arquivo.

Toda a manipulação de arquivos é realizada através de funções de biblioteca. Existem dois níveis de funções: as rotinas de "baixo nível" - mais eficientes, que utilizam diretamente os recursos do sitema operacional - e as de "alto nível" - mais fáceis de serem utilizadas - construídas sobre as funções de "baixo nível".

Para as rotinas de alto-nível, os arquivos são acessados através de ponteiros para estruturas do tipo "FILE", definidas em "stdio.h". Para as rotinas de baixo nível os arquivos são identificados por um número inteiro que é o código identificador do arquivo retornado pelo sistema operacional (file handle). Na estrutura FILE, o número identificador do arquivo é o campo "fd".

As rotinas de alto-nível possuem por sua vez dois conjuntos de rotinas: rotinas para manipulação de arquivos texto e rotinas para manipulação de arquivos binários. A diferença básica entre os dois conjuntos é que as rotinas de manipulação de arquivos binários armazenam uma cópia dos bytes da memória principal para a secundária enquanto que as de manipulação de arquivos texto armazenam a representação ASCII dos valores.

Principais rotinas de baixo nível (prototipadas em io.h)

| | |
|---|---|
| access | - Verifica se um arquivo existe e se pode ser lido/gravado. |
| chmod | - Modifica os atributos de um arquivo. |
| close | - Fecha um arquivo. |
| creat | - Cria um arquivo. Está em desuso |
| eof | - Verifica se a leitura chegou ao final de um arquivo. |
| filelenght | - Informa o tamanho de um arquivo. |
| getftime | - Informa a data de criação/acesso de um arquivo. |
| lock | - Coloca um arquivo em uso exclusivo. É utilizada para fazer controle de acesso concorrente em ambiente multi-usuário ou de rede. |
| lseek | - Descola o ponteiro de leitura para um byte específico no arquivo. |
| open | - Abre/cria um arquivo. |
| read | - Lê dados de um arquivo. |
| tell | - Informa em qual byte está o ponteiro do arquivo. |
| unlock | - Libera o arquivo do uso exclusivo. |
| write | - Escreve dados em um arquivo. |

Principais rotinas de alto nível:

| | |
|---|---|
| fclose | - Fecha um arquivo. |
| feof | - Informa se a leitura chegou ao fim de arquivo. |
| fflush | - Esvazia o buffer do arquivo. |
| fgetc | - Lê um caracter de um arquivo. |
| fgets | - Lê uma string de um arquivo. |
| fputc | - Escreve um caracter em um arquivo. |

fputs - Escreve uma string em um arquivo.

fread - Lê dados de um arquivo.

fscanf - Lê dados de um arquivo, como "scanf"

fseek - Desloca o ponteiro do arquivo para um determinado byte.

ftell - Indica em qual byte se localiza o ponteiro do arquivo.

fwrite - Escreve em um arquivo.

remove - Remove um arquivo.

rename - Muda o nome de um arquivo.

setvbuf - Modifica o tamanho interno do buffer do arquivo.

fflushall - Descarrega o buffer de todos os arquivos.

fprintf - Escreve em um arquivo, no mesmo formato que printf.

**Obs:** Existem cinco arquivos tipo "stream" pré-definidos que são automaticamente abertos quando um programa entra em execução:

stdin - Dispositivo padrão de entrada (geralmente teclado).

stdout - Dispositivo padrão de saída (geralmente vídeo).

stderr - Dispositivo padrão para saída de erros.

stdaux - Dispositivo auxiliar (porta serial, por exemplo).

stdprn - Impressora padrão do sistema.

Exemplo:

```
printf("%d",x);
```

é o mesmo que

```
fprintf(stdout, "%d", x);
```

### 9.4.1 Declaração de arquivos

Quando se usa o conjunto de rotinas de alto nível, utilizam-se ponteiros do tipo FILE para armazenar uma referência para um arquivo. FILE é na verdade uma estrutura que guarda uma série de informações sobre o arquivo. Uma estrutura desse tipo é alocada pela rotina "fopen" e uma referência para a mesma deve ser armazenada.

Exemplo:

```
FILE *arq, *arq1;
```

### 9.4.2 Funções de abertura e fechamento de arquivos

A função **fopen** é usada para a abertura de arquivos tanto no modo texto como no modo binário. Em seus parâmetros devemos indicar o nome com o qual o arquivo é (ou será) identificado no dispositivo de memória secundária e o modo que se deseja trabalhar com o mesmo (leitura, escrita, alteração, etc). A função **fopen** retorna um ponteiro para a estrutura do tipo FILE que descreve o arquivo.

**Função "fopen"**

Sintaxe:

*<fstream> = **fopen**(<nome_arquivo>,"<modo>");*

onde:

- fstream = ponteiro para o arquivo;
- nomearq = string que contém o nome do arquivo no disco;
- "modo" = string constante composta por dois caracteres que indica o modo de abertura do arquivo:

modos:

w – abre/cria para escrita. Se já existe, é destruído.

r – abre somente para leitura.

a – abre o arquivo para escrita a partir do fim do mesmo ou cria se o arquivo não existir.

r+ – abre um arquivo já existente tanto para leitura como para gravação. Se este arquivo já existir ele não será destruído.

w+ – abre um arquivo tanto para leitura como para gravação. Se este arquivo já existir ele será destruído.

a+ – abre um arquivo para leitura/gravação ao final do arquivo. Se o arquivo não existe ele é criado.

No MS-DOS ainda há como segundo caracter de modo as opções:

b – especifica que o arquivo é binário

t – especifica que o arquivo é texto

A função **fopen** retorna NULL se não puder abrir/criar o arquivo.

Exemplo:

```
...
char narq[20];

arq1 = fopen("texto.txt","wt");
if (arq1 == NULL) printf("Não consegui abrir o arquivo\n");

strcpy(narq,"dados");
arq2 = fopen(narq,"rb");
if (arq2 == NULL) printf("Não consegui abrir o arquivo %s",narq);
...
```

**Função "fclose"**

A função **fclose** deve ser usada toda a vez que não for mais necessário trabalhar com um arquivo. Libera os buffers alocados pelo sistema operacional para o gerenciamento do arquivo garantindo que todos os dados são efetivamente descarregados no dispositivo de memória secundária. Libera também a estrutura do tipo FILE.

Sintaxe:

***fclose*** *<fstream>*

onde:

- fstream = ponteiro para o arquivo

Exemplo:

```
fclose(arq1);
fclose(arq2);
```

### 9.4.3 Funções de escrita e gravação

#### 9.4.3.1 Função "fwrite"

A função **fwrite** copia uma porção da memória principal para um dispositivo de memória secundária. Recebe como parâmetros o endereço do início da área e a quantidade de bytes a ser copiada.

Sintaxe

***fwrite****(<ptr>, <tamanho>, <nro>, <fstream>);*

onde:

- ptr = ponteiro para o início dos dados a serem gravados
- tamanho = tamanho do registro lógico;
- nro = quantidade de registros lógicos (registro físico = nro x tamanho);
- fstream = ponteiro para o arquivo;

fwrite retorna o número de itens que conseguiu gravar no arquivo.

#### 9.4.3.2 Função "fread"

A função **fread** tem a função inversa da função **fwrite** carregando um conjunto de bytes do dispositivo de memória secundária para a memória principal. Possui os mesmos parâmetros que **fwrite** porém com o sentido inverso.

Sintaxe

***fread****(<ptr>, <tamanho>, <nro>, <fstream>);*

onde:

- ptr = ponteiro para o início da área onde devem ser armazenados os dados lidos;
- tamanho = tamanho do registro lógico;
- nro = quantidade de registros lógicos a serem lidos;
- fstream = ponteiro para o arquivo;

fread retorna o número de itens que conseguiu ler do arquivo.

#### 9.4.3.3 Função "feof" - Teste de Fim de Arquivo:

A função de teste de fim de arquivo **feof** retorna o valor lógico verdadeiro quando o programa tenta ler o arquivo após o último byte gravado.

Sintaxe:

*<x> = **feof** (<fstream>);*

onde:

fstream = ponteiro para o arquivo;

x = um inteiro que recebe o valor 1 se foi encontrado o fim de arquivo, senão recebe 0.

Ex.:

```
while(!feof(arq1))

        fread(&buf,sizeof(int),10,arq1);
```

Exemplo de um programa que grava e escreve na tela uma estrutura de nome e idade:

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <io.h>

typedef struct
     {
     char nome[30];
     int idade;
     }pessoa;

pessoa cliente;


FILE *arq1;
int i, tamanho;


void main()
     {
     clrscr();
     arq1 = fopen("clientes.dat","ab+");
     if (arq1==NULL) exit(1);
     for(i=0;i<4;i++);
          {
          printf("nome: ");
          fflush(stdin);
          gets(cliente.nome);
          fflush(stdin);
          printf("idade: ");
          scanf("%d",&cliente.idade);
          fwrite(&cliente,sizeof(cliente),1,arq1);
          }
     clrscr();
     rewind(arq1);
     while(!feof(arq1))
          {
          fread(&cliente,sizeof(pessoa),1,arq1);
          if(!feof(arq1))
               {
               printf("nome: %s - %d anos\n",cliente.nome, \
                    cliente.idade);
               }
          }
     getch();
     }
```

### 9.4.4 Funções de Escrita e Gravação em Arquivos Texto

#### 9.4.4.1 Função “fputc”

A funçao **fputc** permite que se armazene um caracter (um byte) em um arquivo texto.

Sintaxe:

***fputc***(*<char>,<fstream>);*

onde:

- char = caracter a ser gravado;
- fstream = ponteiro para o arquivo;

#### 9.4.4.2 Função “fgetc”

A funçao **fgetc** permite que se leia um caracter (um byte) de um arquivo texto.

Sintaxe:

*<c> = **fgetc**(<fstream>);*

onde:

- c = variável que recebe o caracter lido;
- fstream = ponteiro para o arquivo;

#### 9.4.4.3 Função “fputs”

A funçao **fputs** permite que se grave um string em um arquivo texto.

Sintaxe:

***fputs**(<str>,<fstream>);*

onde:

- str = ponteiro para o string a ser gravado (substitui o “\0” por um “\n” no momento da gravação);
- fstream = ponteiro para o arquivo;

#### 9.4.4.4 Função “fgets”

A funçao **fgets** permite que se leia um string inteiro de uma só vez de um arquivo texto. Lê todos os caracteres até encontrar um “\n” ou estourar o limite estabelecido nos parâmetros.

Sintaxe:

***fgets**(<str>,<tammax>,<fstream>);*

onde:

- str = ponteiro para o string a ser lido (carrega todos os caracteres até encontrar um “\n” e acrescenta um “\0” no fim do string).
- tammax = número máximo de caracteres que podemser lidos.
- fstream = ponteiro para o arquivo;

#### 9.4.4.5 Função “fprintf”

A funçao **fprintf** permite a gravação formatada de dados em um arquivo texto. Os dados são armazendados no arquivo da mesma maneira que seriam jogados no dispositivo de saída padrão. A sintaxe é igual a da função **printf** diferenciando-se apenas pela indicação do arquivo destino.

Sintaxe:

***fprintf*** *(<fstream>, “<string de controle>”, <lista de argumentos>);*

- funciona como o comando **printf** com a diferença que os caracteres são enviados para o arquivo especificado por fstream.

#### 9.4.4.6 Função “fscanf”

Possui a mesma filosofia de **fprintf** só que para a leitura de dados. Possui sintaxe idêntica a da função **scanf** diferenciando-se apenas pela indicação do arquivo fonte.

Sintaxe:

***fscanf*** *(<fstream>, “<string de controle>”, <lista de argumentos>);*

Funciona como o comando **scanf** com a diferença que os caracteres são lidos do arquivos específicado por fstream

O programa abaixo é um exemplo de programa que utiliza as funções ***fgets, fputs, fgetc, fputc, fscanf, fprintf***, para manipular um arquivo texto.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

FILE *arq1;
char i, caracter, meu_complemento[20], frase_do_arq[80];


void main()
{
arq1 = fopen("teste.txt","w+t");
if(arq1==NULL)
     {
     printf("Nao foi possivel abrir o arquivo!");
     exit(1);
     }
strcpy(meu_complemento,"Maravilhoso!");
fprintf(arq1,"Ola Mundo %s\n",meu_complemento);
fputs("Este e' o alfabeto: ",arq1);
for(i=65;i<=90;i++)
     {
     fputc(i,arq1);
     }
clrscr();
rewind(arq1);
fscanf(arq1,"%s",frase_do_arq);
printf("%s",frase_do_arq);
fgets(frase_do_arq,80,arq1);
printf("%s",frase_do_arq);
while(!feof(arq1))
     {
     caracter=fgetc(arq1);
     putchar(caracter);
     }
getch();
```

```
}
```

### 9.4.5 Funções "fseek" , "ftell" e "rewind"

As funções **fseek**, **ftell** e **rewind** são usadas para determinar ou modificar a localização do marcador de posição de arquivo.

A sintaxe é:

***fseek (****ponteiro_arquivo, bytes_de_deslocamento, de_onde****);***

A função **fseek** reinicializa o marcador de posição no arquivo apontado por por *ponteiro_arquivo* para o número de *bytes_de_deslocamento* a partir:

- do início do arquivo (*de_onde* = 0)
- da localização atual do marcador de posição no arquivo (*de_onde* = 1)
- do final do arquivo (*de_onde* = 2).

A função **fseek** retornará 0 se o reposicionamento for bem sucedido ou **EOF** se não.

Para determinar a quantidade de *bytes_de_deslocamento* pode-se usar a função **sizeof** que retorna o tamanho de uma variável ou de uma estrutura.

EX:

```
fseek(arq1,sizeof(float),0);
fseek(arq1,sizeof(clientes),1);
```

A função **ftell** retorna a posição atual do marcador de posição no arquivo apontado por *ponteiro_arquivo*. Esta posiçào é indicada por um deslocamento medido em bytes, a partir do início do arquivo.

A sintaxe é:

*variável_long = **ftell** (ponteiro_arquivo);*

A função **rewind** simplesmente reinicializa para o início do arquivo o marcador de posição no arquivo apontado por *ponteiro_arquivo*.

A sintaxe é:

***rewind****(ponteiro_arquivo);*

### 9.4.6 Arquivos com diversas estruturas

Um arquivo pode conter diversas estruturas. Para uma leitura correta desses arquivos é necessário conhecer o "layout" desse arquivo.

Por exemplo, um arquivo DBF, utilizado por dBase, Clipper, FoxPro e reconhecido por qualquer programa que importe base de dados, é dividido em 3 partes distintas: cabeçalho, atributos dos campos e dados.

O cabeçalho possui um registro com a seguinte definição:

```
typedef struct
{
```

```
char versao;
char      ano;
char mes;
char dia;
long int  n_regs;
int  tam_cab;
int  tam_reg;
char reservado[20];
} inf_DBF;
```

A seguir para cada atributo (campo) há o seguinte registro:

```
typedef struct
{
char nome[11];
char      tipo;
char      reservado[4];
char tamanho;
char num_dec;
char reservado2[14];
} descr_atrib;
```

A quantidade de atributos é dada pela fórmula:

atributos = (cab.tam_cab - sizeof(inf_DBF))/sizeof(descr_atrib);

Os dados são gravados a partir do 1º byte posterior ao tamano do cabeçalho (cab.tam_cab). No início de cada registro (conjunto de atributos) é gravado um byte para marcar se o registro está deletado (se o caracter for igual a 42 (*)) ou não (se o caracter for igual a 32 (espaço em branco)), e são todos gravados em modo texto, sendo que cada registro referente a cada atributo possui o tamanho definido na descrição do atributo (tamanho + num_dec).

O seguinte programa lê um arquivo DBF:

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>

typedef struct
{
char versao;
char ano;
char mes;
char dia;
long int n_regs;
int tam_cab;
int tam_reg;
char reservado[20];
} str_dbf;

typedef struct
{
char nome[11];
char tipo;
char reservado1[4];
char tamanho;
char num_dec;
char reservado2[14];
} descr_atr;

descr_atr desc;
str_dbf dbf;
FILE *arq;
int quant, i;
char aux;
```

```
char nomeaux[256][12];
char tipo[256];
char tam[256];
char dec[256];
int tm, j, k, qtd, l, tm1, tm2, op, tamanho, zz;


void main(int argc, char *argv[])
{
if (argc!=2)
     {
     printf("\nA sintaxe e' ledbf arquivo.dbf\n");
     exit(1);
     }
clrscr();
arq=fopen(argv[1],"rb");
if (arq==NULL)
     {
     printf("Arquivo Inexistente!");
     exit(1);
     }
fread(&dbf,sizeof(str_dbf),1,arq);
printf("versao: %d\n",dbf.versao);
printf("ano:    %d\n",dbf.ano);
printf("mes:    %d\n",dbf.mes);
printf("dia:    %d\n",dbf.dia);
printf("n° reg: %ld\n",dbf.n_regs);
printf("t_cab:  %i\n",dbf.tam_cab);
printf("t_reg:  %i\n",dbf.tam_reg);
printf("reserv: %s\n",dbf.reservado);
printf("-------------------------------\n\n");
tamanho=sizeof(str_dbf);
getch();
quant=(int)((dbf.tam_cab-sizeof(str_dbf))/sizeof(descr_atr));
for (i=0;i<quant;i++)
     {
     fread(&desc,sizeof(descr_atr),1,arq);
     tipo[i]=desc.tipo;
     tam[i]=desc.tamanho;
     strcpy(nomeaux[i],desc.nome);
     dec[i]=desc.num_dec;
     printf("\n%11s - %c - %2d - %2d",desc.nome, desc.tipo, desc.tamanho, desc.num_dec);
     tamanho += sizeof(descr_atr);
     }
printf("\n\nLer quantos registros - (999 - todos)? ");
scanf("%d",&qtd);
fseek(arq,dbf.tam_cab,0);
if (qtd==999) qtd=dbf.n_regs;
j=1;
while ((j<=qtd) &&(!(feof(arq))))
     {
     printf("\n\n  Registro: %d -------------------------",j);
     fread(&aux,1,1,arq);
     if (aux!=' ') printf("\n-> %c (deletado) ",aux);
     for (k=0;k<quant;k++)
          {
          printf("\n %12s: ",nomeaux[k]);
          for (l=1;l<=tam[k];l++)
               {
               fread(&aux,1,1,arq);
               printf("%c",aux);
               }
          }
     getch();
     j++;
     }
printf("\n     ************* FIM ");
getch();
clrscr();
}
```

**BIBLIOGRAFIA**


GUEZZI, Carlo & JAZAYERI, Mehdi. **Conceitos de Linguagem de Programação**. Rio de Janeiro, Ed. Campus. 1985.

KELLY-BOOTLE, Stan. **Dominando o Turbo C**. Rio de Janeiro, Ed. Ciência Moderna. 1989.

KERNIGHAM, Brian W. & RITCHIE, Dennis M. **C: A Linguagem de Programação**. Rio de Janeiro, Ed. Campus. 1986.

LONGO, Maurício B.; SMITH Jr., Ronaldo & POLISTCHUCK, Daniel. **Delphi 3 Total**. Rio de Janeiro, Brasport Livros e Multimídia Ltda. 1997.

MECLER, Ian & MAIA, Luiz Paulo. **Programação e Lógica com Turbo Pascal**. Rio de Janeiro, Ed. Campus. 1989.

PALMER, Scott D. **Guia do Programador - Turbo Pascal for Windows**. Rio de Janeiro, Editora Ciência Moderna. 1992.

PAPPAS, Chris H. & MURRAY, Wiliam H. **Borland C++**. São Paulo, Makron Books. 1995.

RINALDI, Roberto. **Turbo Pascal 7.0: Comandos e Funções**. São Paulo, Érica. 1993.

SCHILDT, Herbert. **Linguagem C: Guia Prático e Interativo**. São Paulo, McGraw-Hill. 1989.

WIENER, Richard S. **Turbo C Passo a Passo**. Rio de Janeiro, Ed. Campus. 1991