

Linguagem C

ABORDAGEM PRÁTICA - MÓDULO 1

Silvio Antonio Carro
Leandro Luiz de Almeida

1994/2018

ÍNDICE

PREFÁCIO.....	4
<i>Histórico</i>	<i>4</i>
ESTRUTURA BÁSICA DE UM PROGRAMA EM C.....	7
<i>Forma Geral de FUNÇÕES em C.....</i>	<i>7</i>
VARIÁVEIS	9
<i>Tipos de Variáveis.....</i>	<i>9</i>
<i>Declarando Variáveis</i>	<i>9</i>
<i>Inicializando Variáveis</i>	<i>10</i>
OPERADORES.....	11
<i>Aritméticos</i>	<i>11</i>
<i>Operadores Aritméticos de Atribuição</i>	<i>11</i>
<i>Relacionais.....</i>	<i>12</i>
<i>Lógicos.....</i>	<i>12</i>
<i>Bit a Bit</i>	<i>12</i>
FUNÇÕES E/S	13
<i>Funções de Saída:.....</i>	<i>13</i>
<i>Funções de Entrada</i>	<i>16</i>
COMANDOS DE ITERAÇÃO	18
<i>for(;;).....</i>	<i>18</i>
<i>while()</i>	<i>19</i>
<i>do ... while()</i>	<i>20</i>
<i>Comandos Auxiliares</i>	<i>20</i>
COMANDOS DE DECISÃO	22
<i>if.....</i>	<i>22</i>
<i>switch</i>	<i>26</i>
<i>Operador Condicional Ternário “ ? “</i>	<i>29</i>
FUNÇÕES	30
<i>Declaração das Funções.....</i>	<i>30</i>
<i>Tipos de Funções.....</i>	<i>31</i>
CLASSES DE ARMAZENAMENTO	32
<i>auto</i>	<i>32</i>
<i>extern.....</i>	<i>32</i>
<i>static</i>	<i>32</i>
<i>register.....</i>	<i>32</i>
DIRETIVAS	34
<i>#define</i>	<i>34</i>
<i>#undef</i>	<i>34</i>

<i>#include</i>	34
<i>#undef, #if, #ifdef, #ifndef, #else, #endif</i>	35
VETORES E MATRIZES	36
<i>Referência aos Elementos</i>	37
<i>Inicialização de vetores/matrizes</i>	37
<i>Verificando limites</i>	39
<i>Lendo strings</i>	41
<i>Imprimindo strings</i>	41
<i>Funções de manipulação de strings</i>	41
ESTRUTURAS	43
<i>Acessando a Estrutura</i>	43
<i>Matrizes de Estruturas</i>	43
<i>Inicializando uma Estrutura</i>	44
PONTEIROS	45
ARGUMENTO DE LINHA DE COMANDO	48
ALOCANDO MEMÓRIA	49
UNIÕES	50
OPERAÇÕES COM ARQUIVOS EM DISCO	51
<i>Abrir um Arquivo</i>	51
<i>Fechar um Arquivo - fclose()</i>	52
<i>Verificando final de arquivo - EOF</i>	53
<i>Funções para Ler/Gravar caracter a caracter</i>	53
<i>Funções para Ler/Gravar Strings</i>	54
<i>Funções para Ler/Gravar Strings Formatadas</i>	55
<i>Funções para Ler/Gravar Estruturas</i>	56
<i>Acesso Aleatório</i>	56
<i>Condições de Erro</i>	57
ANEXO 1 – FUNÇÕES GRÁFICAS	58
BIBLIOGRAFIA	62

PREFÁCIO

Esta apostila foi elaborada como guia de apresentação da linguagem C, mais precisamente o Turbo C 3.0 da Borland, por isso mostra-se de modo superficial e pouco detalhado.

Histórico






A linguagem C foi primeiramente criada por Dennis M. Ritchie e Ken Thompson no laboratório Bell em 1972, baseada na linguagem B de Thompson que era uma evolução da antiga linguagem BCPL.

A Linguagem C ficou contida nos laboratórios até o final da década de 70, momento que começou a popularização do Sistema Operacional UNIX e consequentemente o C (o UNIX é desenvolvido em C).

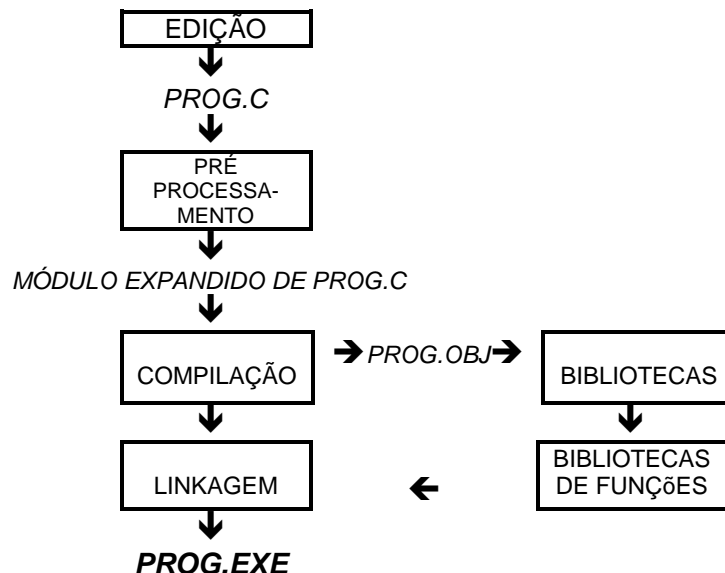
Os programas, então escritos em Assembler, como o dBase II e III, Wordstar, Lotus versão 1.0 dentre outros, tiveram seus fontes convertidos em C, originando como exemplo o dBase III Plus, Lotus versão 2.0, MS-Word e MS-Excel. É utilizado amplamente no desenvolvimento de sistemas operacionais (UNIX), SGBDs, aplicativos e utilitários.

Dentre algumas características do C, estão:

* C une conceitos de linguagem de montagem e programação de alto nível (o programador usufrui de recursos de hardware sem a necessidade de conhecer o assembly da máquina).

Alto Nível		ADA MODULA PASCAL COBOL FORTRAN BASIC
Médio Nível		C FORTH
Baixo Nível		MACROASSEMBLER ASSEMBLER

* Linguagem compilada. Os programas escritos em C geram, depois de compilados, programas-objetos pequenos e eficientes.

**O pré-processador**

O pré-processador atua apenas ao nível do código fonte, modificando-o. Trabalha apenas com texto. Algumas das suas funções são:

- remover os comentários de um programa;
- interpretar diretivas de compilação (#...);

O compilador

Alguns compiladores traduzem o código fonte (texto) recebido do pré-processador para linguagem assembly (também texto). No entanto são também comuns os compiladores capazes de gerar diretamente código objeto (instruções do processador já em código binário). Geram arquivos com extensão .OBJ

O linker

Se o programa referencia funções da biblioteca standard ou outras funções contidas em arquivos com código fonte diferentes do principal (que contém a função `main()`), o linker combina todos os objetos com o resultado compilado dessas funções num único arquivo com código executável. As referências a variáveis globais externas também são resolvidas pelo linker.

A utilização de bibliotecas

A linguagem C é muito compacta. Muitas das funções que fazem parte de outras linguagens não estão diretamente incluídas na linguagem C. Temos como exemplo as operações de entrada/saída, a manipulação de strings e certas operações matemáticas.

- * Utiliza conceitos de linguagem estruturada.

ESTRUTURADA	NÃO ESTRUTURADA
PASCAL	FORTRAN
ADA	BASIC
C	COBOL
MODULA-2	

* C está amplamente disponível. Há compiladores C disponíveis para a maioria dos computadores pessoais, microcomputadores e *mainframes* e *palm tops*.

* C independe da máquina. Programas escritos em C são facilmente transportados de um computador para outro (sistemas operacionais diferentes).

- * Possui poucos comandos, cerca de 32 na versão básica e 43 na versão para Turbo C.
- * Não possui críticas eficientes para erros de execução;

TURBO PASCAL x TURBO C

- * Termos básicos similares;
- * Algumas funções semelhantes;

- * C tem maior controle direto sobre o computador;
- * O C possui fraca verificação de erros.

IMPORTÂNCIA DA LINGUAGEM C

- * Linguagem mais utilizada atualmente nos EUA;
- * Utilizada no desenvolvimento de importantes softwares (Windows, Linux, Visual Basic, Clipper)
- * Ter subsídios (base) para o aprendizado da linguagem C++ e Java;
- * Linguagem padronizada (ANSI);
- * Entender a fundo a programação em ambientes Windows e Unix;
- * Programação de sistemas embutidos (hardware);
- * Linguagem preferida em concursos públicos.

"C é a linguagem de um programador"

Herbert Schildt

ESTRUTURA BÁSICA DE UM PROGRAMA EM C

Um programa C consiste em uma ou várias "funções", onde uma das quais precisa ser denominada *main*. O programa sempre começa executando a função *main*. Definições de funções adicionais podem preceder ou suceder a função *main*.

Regras Gerais:

- * Toda função em C (inclusive a *main*) deve ser iniciada por uma chave de abertura: "{", e encerrada por uma chave de fechamento: "}";
- * Toda função é precedida de parênteses "()";
- * Todo programa deverá conter a função *main* (é uma palavra reservada);
- * As instruções em C são sempre encerradas por um ponto-e-vírgula (;);
- * A formatação dos programas é absolutamente livre, mas temos por conveniência, manter a legibilidade;
- * Os comandos são executados na ordem em que foram escritos;
- * Os comentários devem ser delimitados por */** no início e **/* no final e podem ocupar desde algumas colunas entre as instruções a até várias linhas. Pode ser usando também os caracteres *//* para comentários de uma linha.

Forma Geral de FUNÇÕES em C

*/** Os comentários podem ser colocados em qualquer parte do programa **/*

declaração de variáveis globais

void main(void) ⇒ função principal (obrigatória)

```
{
    ⇒ inicia o corpo da função
    declarações de variáveis locais
    ----
    ---- comandos;
    ----
}
```

⇒ finaliza o corpo da função

tipo funcao(lista dos argumentos de entrada) ⇒ função

```
{
    declarações das variáveis locais a função
    ----
    ---- comandos;
    ----
}
```

EXEMPLO:

```
/* programa para calcular a area de um círculo */
#include <stdio.h>                                // inclusão do "header file" stdio.h

float processa(float r);

void main(void)
{
    float raio, area;                            /* declaração de variáveis */
    printf(" Raio = ? ");
    scanf("%f", &raio);
    area = processa(raio);
    printf(" Area = %f", area);
}

float processa (float r)                          /* definicao de funcao */
{
    float a;                                     /* declaracao de variavel local */
    a=3.1415*r*r;
    return(a);
}
```


VARIÁVEIS

“É um espaço de memória que pode conter, a cada tempo, valores diferentes”.

Regras Gerais:

- * Em C, todas as variáveis utilizadas no programa devem ser declaradas previamente;
- * A declaração indica, no mínimo, o nome e o tipo de cada uma delas;
- * Na ocorrência de mais de uma variável do mesmo tipo, podemos declará-las de uma única vez, separando seus nomes por vírgula;
- * As letras minúsculas são diferenciadas das respectivas maiúsculas;
- * O tamanho máximo significativo para uma variável é de 31 posições;
- * Todo nome de variável é iniciado por uma letra (a a z **ou** A a Z), ou o caracter *underscore* (`_`), o restante pode conter letras, o *underscore* ou números;
- * As palavras reservadas, descritas a seguir não podem ser usadas como nome de variável;

Palavras Reservadas:

<i>auto</i>	<i>break</i>	<i>case</i>	<i>char</i>	<i>const</i>	<i>continue</i>	<i>default</i>
<i>do</i>	<i>double</i>	<i>else</i>	<i>enum</i>	<i>extern</i>	<i>float</i>	<i>for</i>
<i>goto</i>	<i>if</i>	<i>int</i>	<i>long</i>	<i>main</i>	<i>register</i>	<i>return</i>
<i>short</i>	<i>signed</i>	<i>sizeof</i>	<i>static</i>	<i>struct</i>	<i>switch</i>	<i>typedef</i>
<i>union</i>	<i>unsigned</i>	<i>void</i>	<i>volatile</i>	<i>while</i>	<i>class</i>	<i>public</i>
<i>private</i>	<i>protected</i>	<i>virtual</i>				

Exemplo de nome de variáveis:

a, b, TOTAL_DE_DESPESA, VALOR1, valor_1, _nome, ano_1999, Horas, horas

Tipos de Variáveis

Tipo	Bits	Escala
void	0	<i>sem valor</i>
char	8	-128 a 127
int (*)	16	-32768 a 32767
float	32	3.4E-38 a 3.4E+38
double	64	1.7E-308 a 1.7E+308
long int	32	3.4E-38 a 3.4 E+38
unsigned char	8	0 a 255
unsigned int	16	0 a 65535

(*) depende da quantidade de bits do computador

Declarando Variáveis

A forma de declaração de variáveis é a seguinte:

tipo nome;

Exemplo:

```
int K;
double Valores;
float quadro;
char character;
```

Inicializando Variáveis

Podemos inicializar uma variável no momento de sua declaração, ou em qualquer momento do algoritmo. Exemplo:

```
int c=8; ⇒ global
void main(void)
{
    float tempo=27.25; ⇒ Local
    char tipo='C';      ⇒ Local
    int x;
    x=10;
    c=c+x;
} * Lembre-se que ao declarar variáveis em C: letras minúsculas são diferentes de letras maiúsculas e todas
    variáveis devem ser declaradas.
```

Casting...

No código abaixo, porquê o resultado de Result é zero?

```
int NDec=3;
int NPoints=1000;
float Result;
Result=(NDec/NPoints);
```

Toda vez que se mistura tipos numéricos (*float*, *int*, *long*, etc), devemos tomar cuidado com a forma com que o compilador entende a expressão matemática. No caso acima a operação foi realizada com dois inteiros NDec=3 e NPoints=1000, assim, o compilador assumiu que 3/1000 é zero. Depois, o compilador converte para *float* e armazena em Result, mas é tarde... Para obter o resultado esperado, que é 0.003, devemos instruir o compilador que desejamos um resultado tipo *float*. Existem vários métodos, o mais comum é o uso de um **cast**:

```
Result=(float)NDec/NPoints;
```

Outra alternativa é inserir um float na expressão:

```
Arq1=1.0*NDec/NPoints;
```

NOTE...

- * As variáveis deverão ser declaradas antes de qualquer instrução;
- * As variáveis do tipo caractere deverão receber valores entre aspas simples ('');
- * Em C, não existe o tipo *boolean* nem o tipo *string*;
- * O Valor zero (0) representa um valor falso em C, todos os demais valores (1,'a','z',-1,-0.7, por exemplo) são considerados verdadeiros;

OPERADORES

Aritméticos

Mais importantes:

	Significado	Exemplo
=	Atribuição	a=5; caracter='B';
+	Soma	a=a+1; total=salario+comissão
-	Subtração	total=subtotal-desconto
*	Multiplicação	nota=nota*2;
/	Divisão	comissão=lucro/4;
%	Módulo (retorna o resto da divisão)	resto=8%3;
++	Incrementa de 1 seu operando	i++;
--	Decrementa de 1 seu operando	j--;

Exemplo :

```
temp=(Fahr-32)*5/9;
i++;          /*equivale i=i+1*/;
```

Os operadores ++ e – podem ser utilizados antes ou após o operando, se este for utilizado antes do operando, este é atualizado antes da execução da linha de comando. Caso for utilizado após o operando, o mesmo é atualizado somente após a resolução da linha de comando .

Exemplo:

```
a = 10;
b = a++;
c = a;
d = ++a;
```

Valores das variáveis ao
término da execução

Variável	Valor
a	12
b	10
c	11
d	12

Operadores Aritméticos de Atribuição

Os operadores aritméticos de atribuição, além de realizar a operação, ele atribui o valor resultante a uma variável. O programa em C fica mais compacto e código de máquina mais eficiente.

	Significado
+=	Soma e atribui
- =	Subtrai e atribui
* =	Multiplica e atribui
/ =	Divide e atribui
%=	Atribui o resto da divisão

Exemplos:

```
teste = 10;
teste += 5;   equivale a  => teste = teste + 5;
teste *=2;    equivale a  => teste = teste * 2;
```

* A divisão e multiplicação têm precedência sobre a soma e a subtração.

Relacionais

	Significado
>	Maior
>=	Maior ou igual
<	Menor
<=	Menor ou igual
==	Igualdade
!=	Diferente

Lógicos

	Significado
&&	Lógico E
	Lógico OU
!	Lógico de negação

Bit a Bit

	Significado	Exemplo (*)
~	negação	n = 0110 1001 ⇨ 105 ~n = 1001 0110 ⇨ 150
&	and	a = 0110 1001 ⇨ 105 b = 0101 1100 ⇨ 92 a & b = 0100 1000 ⇨ 72
	or	a = 0110 1001 ⇨ 105 b = 0101 1100 ⇨ 92 a b = 0111 1101 ⇨ 125
^	or exclusivo	a = 0110 1001 ⇨ 105 b = 0101 1100 ⇨ 92 a ^ b = 0011 0101 ⇨ 53
<<	deslocamento para esquerda	a = 0101 1100 ⇨ 105 a << 3 = 1110 0000 ⇨ 224
>>	deslocamento para direita	a = 0101 1100 ⇨ 105 a >> 3 = 0000 1011 ⇨ 11

FUNÇÕES E/S

Funções de Saída:

printf()

Sintaxe:

```
printf("expr.controle", lista de argumentos);
```

Definição: Imprime uma expressão formatada na tela, permitindo definir a expressão e o formato em que as variáveis ou constantes serão apresentadas.

Onde: **expr.controle** é uma cadeia de caracteres, contendo códigos especiais, códigos para impressão formatada e caracteres que serão impressos na tela.

lista de argumentos são variáveis ou constantes que serão impressas de acordo com a expressão de controle definida.

Códigos Especiais:

Código	SIGNIFICADO
\n	Nova Linha
\a	Beep
\f	Nova tela ou nova página
\t	TAB
\b	Retrocesso
\0	Nulo
\"	Aspas
\\	Barra invertida

Códigos de Formatação:

Código	SIGNIFICADO
%c	Caractere simples
%d	Inteiros Decimais
%ld	Long
%e	Notação Científica
%f	Ponto Flutuante
%o	Octal
%s	Cadeia de Caracteres
%u	Decimal sem sinal
%x	Hexadecimal
%%	Símbolo de Porcentagem

Exemplo:

```
void main(void)
{
    char cidade_A='X';
    char cidade_B='Y';
    float distanc=25.5;
    printf("Cidade    = %c \n",cidade_A);
    printf("Distancia = %f Km\n",distanc);
    printf("A %s %c esta a %f Km \n da %s%c",
        "cidade",cidade_A,distanc,"cidade",cidade_B);
}
```

Resultado na Tela:

```
Cidade    = X
Distancia  = 25.5 Km
A cidade X esta a 25.5 KM
da cidade Y
```

Definindo o tamanho de campos na impressão

Em C é possível a definição de como um valor deve ser impresso, para isso deve-se colocar o tamanho do campo de impressão logo após o "%".

Exemplo:

```
void main(void)
{
    printf("Nota do 1o Bimestre - %3d\n", 10);
    printf("Nota do 2o Bimestre - %3d\n", 4);
}
```

Resultado na tela:

```
Nota do 1o. Bimestre - 10
Nota do 2o. Bimestre - 4
```

Também é possível definir a quantidade de casas decimais a ser apresentada. Caso a quantidade de casas decimais definidas for menor que a quantidade real, o valor é apresentado com arredondamento.

Exemplo:

```
void main(void)
{
    printf("Nota do 1o Bimestre - %4.2f\n", 10);
    printf("Nota do 2o Bimestre - %4.2f\n", 4.78);
    printf("Nota do 2o Bimestre - %4.1f\n", 4.78);
}
```

Resultado na Tela:

```
Nota do 1o. Bimestre - 10.00
Nota do 2o. Bimestre - 4.78
Nota do 2o. Bimestre - 4.8
```

Complementando com Zeros a Esquerda

Para fazer o complemento de zeros a esquerda, em valores inteiros, é necessário colocar após o "%" o valor 0 seguido da quantidade.

Exemplo:

```
void main(void)
{
    printf("\nCodigo do Produto - %04d\n", 13);
    printf("Codigo do Produto - %06d\n", 13);
    printf("Codigo do Produto - %6.4d\n", 13);
}
```

Resultado na Tela:

```
Codigo do Produto - 0013
Codigo do Produto - 000013
Codigo do Produto - 0013
```

Imprimindo Caracteres

Variáveis do tipo *char* podem ser apresentadas e representadas de várias formas. A forma de apresentação depende do código de formatação utilizado.

Exemplo:

```
printf("Impressão do A - %d %c %x %o \n", 'A' , 'A' , 'A' , 'A');
Resultado - Impressão do A - 65 A 41 101
printf("Varias letras A - %c %c %c %c \n", 'A', 65, 0x41,);
Resultado - Varias letras A - A A A A
```

putchar()

Sintaxe:

putchar(variável caracter); Imprime uma variável do tipo caracter no dispositivo de saída.

Exemplo :

```
void main(void)
{
    char letra='a';
    putchar(letra);    ⇒ imprime a letra a na tela
}
```

Funções de Entrada

scanf()

Sintaxe: `scanf("expr. de controle" , lista argumentos);`

Definição: Leitura de dados formatados a partir do teclado.

Onde: **expr. de controle** é uma expressão de controle contendo códigos de formatação, precedidos por um sinal %.

lista de argumentos, que consiste nos endereços das variáveis. C oferece um operador para tipos básicos chamado operador de endereço e referenciado pelo símbolo & que retorna o endereço do operando.

Códigos de Formatação:

CÓDIGOS	SIGNIFICADO
%c	Caractere simples
%d	Inteiro Decimal
%e	Notação Científica
%f	Ponto Flutuante
%o	Octal
%s	Cadeia de Caracteres
%u	Decimal sem sinal
%x	Hexadecimal
%l	Inteiro longo

Exemplos:

`scanf("%f" , &x);` ⇒ lê a variável x do tipo float

```
void main(void)
{
    float anos, dias;
    printf("Digite sua idade em anos: ");
    scanf("%f ", &anos);
    dias = anos*365;
    printf("Sua idade em dias e %.0f", dias);
}
```

Resultado na Tela:

Digite sua idade em anos: 4

Sua idade em dias e 1460.0

getche()

Sintaxe: `int getche(void);`

Definição: lê um caracter do teclado e o imprime na tela. Não requer o pressionamento de <ENTER> após o caracter digitado.

Exemplo:

```

letra = getche();

void main(void)
{
    char ch;
    printf("Digite algum character:");
    ch = getche();
    printf("\n A tecla que voce pressionou e %c.", ch);
}

```

Resultado na Tela:

Digite algum character: w
A tecla que voce pressionou e w.

getch()

Sintaxe: `int getch(void);`

Definição: Lê um character do teclado, mas não imprime o character lido na tela.

Exemplo:

```

letra = getch();

void main(void)
{
    char ch;
    printf("Digite algum character:");
    ch = getch();
    printf("\n A tecla que voce pressionou e %c,", ch);
    printf("\n e a sua sucessora em ASCII e %c.", ch+1);
}

```

Resultado na Tela:

Digite algum character:
A tecla que voce pressionou e G, e a sua sucessora em ASCII e H.

COMANDOS DE ITERAÇÃO

for(;;)

Sintaxe:

for (*inicialização*; *teste*; *incremento*) *comando*;

Definição: Permite que um conjunto de operações seja executado até que ocorra uma determinada condição. Engloba 3 expressões em uma única, e é útil principalmente quando queremos repetir algo um número fixo de vezes.

Onde: **inicialização** é geralmente um comando de atribuição com a função de colocar um valor na variável de controle do laço. Essa instrução é executada apenas uma vez antes do laço ser executado.

teste é uma expressão relacional que a instrução testa no princípio de cada uma das iterações. Pode-se utilizar mais de uma expressão, separando-as por “,” (vírgula).

incremento é uma expressão que define a maneira como a variável de controle da iteração será alterada após cada uma das alterações. É sempre executada após o término do laço.

Exemplo:

```
void main(void)
{
    for (a=1; a<=3; a++)
        printf("Número %d \n", a);
}
```

Resultado na Tela:

```
Numero 1
Numero 2
Numero 3
```

Laços for sem algum dos parâmetros

Os laços for não necessitam de todos os parâmetros, os parâmetros podem ser omitidos, ou seja, pode haver laços com 1 ou 2 dos parâmetros.

Exemplos:

```
for (x = 0 ; x !=123 ; )
    scanf ("%d", &x);
for ( ; x < 10; )
    printf ("%d\n", x
```

Laços Infinitos

Com a utilização do for é possível se fazer loops infinitos, para isso é necessário somente não colocar nenhum dos parâmetros.

Exemplo:

```
for(;;);
```

Laços sem corpo

Esse tipo de laço é utilizado quando se quer criar espaços de tempo, para isso é necessário deixar o bloco a ser repetido sem comando algum.

```
for(i = 1; i < 10000; i++);
```

Laços com múltiplas expressões em cada um dos parâmetros

Utilizado quando há a necessidade de várias variáveis no loop, ou quando há mais de uma condição de saída do Loop.

Exemplo:

```
for(i=0, j=9; i<=9, j>=0 ;i++, j--)  
    printf ("\n %d %d",i,j);
```

while()

Sintaxe:

```
while ( condição ) comando;
```

Definição: Permite que um conjunto de operações seja executado enquanto uma condição for satisfeita. Trabalha como um comando **for** com apenas a expressão do meio, o teste.

Onde: **condição** é geralmente uma expressão relacional que a instrução testa no princípio de cada uma das iterações. Portanto *comando* pode até mesmo não ser executado.

comando pode ser um simples comando ou um bloco de comandos iniciado por { e terminado por }.

Exemplo:

```

while (conta<10)
{
    total=total+conta;
    printf("conta=%d, total=%d", conta, total);
    conta++;
}

```

do ... while()*Sintaxe:*

do comando **while** (condição);

Definição: Permite que um conjunto de operações seja executado enquanto uma condição for satisfeita. A principal diferença entre o **do while** e **while** é que o **while** testa a condição antes de executar os comandos e o **do while**, os comandos são executados e após a execução é que a condição será verificada. Portanto os comandos que estão dentro do **do... while** são executadas ao menos uma vez.

Onde: **condição** é geralmente uma expressão relacional que a instrução testa no final de cada uma das iterações.

comando pode ser um simples comando ou um bloco de comandos iniciado por { e terminado por }.

Exemplo:

```

do {
    Y>>1;
    X<<1;
} while (Y);

```

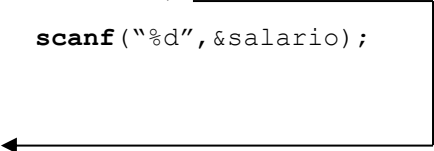
Comandos Auxiliares**break;***Sintaxe:*

break;

Definição: O comando **break** causa a saída imediata de uma estrutura de repetição. Esse comando pode ser utilizado dentro de qualquer comando de iteração, como também no **switch**.

Exemplo:

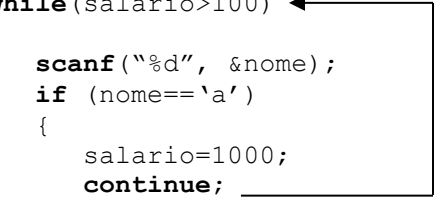
```
while(salario>100)
{
    scanf("%d", &nome);
    if (nome=='a')
        break;
    scanf("%d",&salario);
}
```


continue;*Sintaxe:***continue;**

Definição: O comando **continue** causa um desvio imediato do laço, passando para a próxima iteração, sem executar o código que estiver abaixo.

Exemplo:

```
while(salario>100)
{
    scanf("%d", &nome);
    if (nome=='a')
    {
        salario=1000;
        continue;
    }
    scanf("%d",&salario);
}
```



COMANDOS DE DECISÃO

if

Sintaxe:

```
if ( condição ) comando;
```

Definição: Permite que um comando ou conjunto de comandos seja executado quando uma determinada condição for satisfeita. Se o valor da condição entre parênteses for verdadeiro ele executa as instruções, caso seja falso, as instruções serão ignoradas.

Onde: **condição** é uma expressão lógica ou relacional, que deve ser satisfeita para que a(s) instrução(ões) contida(s) em *comando* seja(m) executada(s). É um teste, uma decisão.

comando é um comando ou conjunto de comandos que serão executados quando a condição for satisfeita. Quando for um conjunto de comandos deve-se colocá-los entre chaves "{}".

Observações:

- Lembrar que em C, não há um tipo booleano;
- Em C, o valor 0 (zero) é falso e qualquer outro valor é verdadeiro, portanto uma condição pode conter apenas um valor inteiro;
- Para verificar se um valor é igual ao outro não é utilizado o sinal de igualdade "=" e sim dois sinais "==";
- Para verificar se um determinado valor é diferente do outro não se utiliza "menor e maior" como em Pascal, utiliza-se a exclamação juntamente com o sinal de igualdade "!="
- Os operadores lógicos são && representando "e", || representando "ou" e para a negação de uma condição !;

Exemplo:

```
void main(void)
{
    char a, b;
    printf("Digite duas letras:");
    a = getche(); b = getche();
    if (a==b)
        printf("\n As duas letras são iguais");
    printf("\n Fim da Execução");
}
```

Resultado na Tela:

Digite duas letras:**ab**

Fim da Execução

Digite duas letras:**aa**

As duas letras são iguais

Fim da Execução

if else

Sintaxe:

```
if ( condição )
    comandoTrue;
else
    comandoFalse;
```

Definição: Este outro formato do comando if permite que um comando ou conjunto de comandos seja executado quando uma determinada condição for verdadeira e um outro conjunto de comandos seja executado caso essa condição seja falsa.

Onde: **condição** é uma expressão lógica ou relacional.

comandoTrue é um comando ou conjunto de comandos que serão executados quando a condição for verdadeira.

comandoFalse é um comando ou conjunto de comandos que serão executados quando a condição for falsa.

Exemplo:

```
void main(void)
{
    char a, b;
    printf("Digite duas letras:");
    a = getche(); b = getche();
    if (a==b)
        printf("\n As letras são iguais");
    else
        printf("\n As letras são diferentes");
    printf("\n As duas letras são iguais");
}
```

Resultado na Tela:

Digite duas letras:**ab**
 As letras são diferentes
 Fim da Execução

Digite duas letras:**aa**
 As letras são iguais
 Fim da Execução

Comandos ifs Aninhados

O padrão ANSI C define que até 15 ifs aninhados devem ser suportados

Exemplo:

```
void main(void)
{
    int x, y, z;
    scanf("%d", &x);
    scanf("%d", &y);
    scanf("%d", &z);
    if (X<Y)
        if (X<Z)
            printf("X é o menor !");
        else
            printf("Z é o menor !");
    else
        if (Y<Z)
            printf("Y é o menor !");
        else
            printf("Z é o menor !");
}
```

Resultado na Tela:

```
1
2
3
X é o menor

2
1
3
y é o menor
```


Utilização de else if

Quando se tem uma grande quantidade de if-else aninhados para evitar muito recuo na margem esquerda como acontece no caso abaixo:

```

if (nota >= 90)
    printf("conceito A");
else
    if (nota >= 70)
        printf("conceito B");
    else
        if (nota >= 50)
            printf("conceito C");
        else
            if (nota >= 30)
                printf("conceito D");
            else
                if (nota > 0)
                    printf("conceito E");
                else
                    printf("totalmente sem conceito");

```

Pode-se utilizar o formato apresentado abaixo.

```

if (nota >= 90)
    printf("conceito A");
else if (nota >= 70)
    printf("conceito B");
else if (nota >= 50)
    printf("conceito C");
else if (nota >= 30)
    printf("conceito D");
else if (nota > 0)
    printf("conceito E");
else
    printf("totalmente sem conceito");

```

switch

Sintaxe:

```
switch ( expressão ) {
    case constante1:
        comandos1;
        break;
    case constante2:
        comandos2;
        break;
    default:
        comandosdefault;
}
```

Definição: Permite que um determinado grupo de instruções seja escolhido entre diversos grupos disponíveis. A seleção é baseada no valor corrente de uma expressão incluída na instrução.

Onde: **expressão** é uma expressão inteira ou caracter, geralmente uma variável de um desses tipos.

constanteN são os valores constantes aos quais a expressão será comparada, só há teste de igualdade. São os possíveis valores que a expressão pode assumir.

comandosN são comandos ou conjuntos de comandos que serão executados de acordo com o valor da expressão, após os comandos a serem executadas em uma determinada opção deve-se colocar um **break**, para evitar que os comandos abaixo do mesmo seja executado;

comandosdefault são os comandos executados quando a expressão procurada não for encontrada entre as constantes definidas, ou seja, caso nenhuma das constantes for igual ao procurado;

Observações:

- A instrução **default** é opcional, se não colocada e nenhuma das opções forem iguais à procurada pela *expressão* não será executado código algum;
- O **switch** difere do **if**, pois ele só pode testar igualdade;
- Duas constantes **case** não podem ter valores iguais;

Exemplo:

```

void main(void)
{
    char opcao;
    printf("Digite V-Vermelho, B-Branco, A-Amarelo")
    opcao = getch();
    switch (opcao)
    {
        case 'V':
            printf("\nVERMELHO");
            break;
        case 'B':
            printf("\nBRANCO");
            break;
        case 'A':
            printf("\nAZUL");
            break;
        default :
            printf("\nCOR DESCONHECIDA !");
    }
}

```

Resultado na Tela:

Digite V-Vermelho, B-Branco, A-Amarelo **V**
VERMELHO

Digite V-Vermelho, B-Branco, A-Amarelo **B**
BRANCO

Se os **breaks** não tivessem sido colocados o resultado seria:

Digite V-Vermelho, B-Branco, A-Amarelo **B**
BRANCO
AZUL
COR DESCONHECIDA

Várias constantes para um mesmo conjunto de comandos

Quando há um conjunto de comandos que podem ser acessado através de várias constantes, pode se utilizar duas formas, que são exemplificadas no programa abaixo.

Exemplo:

```

void main(void)
{
    char op;
    int v1, v2, res = 0;
    printf("\nDigite o primeiro valor..:"); scanf("%d", &v1);
    printf("\nDigite o segundo valor..:"); scanf("%d", &v2);
    printf("\nOperação..:"); op = getche();
    switch (op)
    {
        case '/':, ':':
            res = v1 / v2;
            break;
        case '*':
        case 'X':
        case 'x':
            res = v1 * v2;
            break;
        case '+':
            res = v1 + v2;
            break;
        case '-':
            res = v1 - v2;
            break;
        default :
            printf("\nOPÇÃO INVÁLIDA !");
    }
    printf("\nResultado de %d  %c %d e %d",v1, op, v2, res);
}

```

Operador Condicional Ternário “ ? ”

Sintaxe:

condição ? expressãoT : expressãoF

Definição: Uma forma simples de se expressar a instrução **if-else**. Ele é chamado ternário pois trabalha com três operadores. É utilizado como uma forma condicional de atribuição.

Onde: **condição** é uma expressão relacional, que será avaliada pelo operador.

expressãoT é a expressão a ser avaliada quando a condição for verdadeira.

expressãoF é a expressão a ser avaliada quando a condição for falsa.

Exemplos:

```
void main(void)
{
    int a, b, maior;
    scanf("%d", &a);
    scanf("%d", &b);
    maior = (a > b) ? a : b;
    printf("Maior = %d", maior);
}
```

Não é restrito a atribuições:

```
void main(void)
{
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    (a > b) ? printf("maior e %d",a) : printf("maior e %d",b);
}
```

FUNÇÕES

Modularização:

A idéia principal do conceito de modularização é dividir o programa em sub-módulos, o que torna o trabalho de desenvolvimento e manutenção menos desgastante. Em C o conceito de modularização é implementado por meio de funções. As funções podem ter variáveis próprias ou utilizar as variáveis declaradas como globais. É importante lembrar que variáveis locais com o mesmo identificador (nome) de variáveis globais ocultam o acesso às variáveis globais.

Porque modularizar?

- Para permitir o reaproveitamento de código já construído (próprio ou de outros programadores);
- Para evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa;
- Para permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas dentro da função que se deseja;
- Para que os blocos do programa não fiquem grandes demais e, por consequência, mais difíceis de entender (garantir legibilidade);
- Para facilitar a leitura e depuração do programa fonte de uma forma mais fácil;
- Para separar o programa em partes (blocos) que possam ser logicamente compreendidos de forma isolada.

Funções em C

Uma função é uma unidade de código de programa autônoma desenhada para cumprir uma tarefa particular. A função em C, recebe um certo número de parâmetros e retorna apenas um valor. Da mesma forma que declaramos variáveis, devemos declarar a função. A declaração de uma função é chamada de **protótipo** e é uma instrução geralmente colocada no início do programa, que estabelece o tipo da função e os argumentos que ela recebe.

Técnicamente, quando o tipo_da_funcao é void esta função pode ser chamada de um PROCEDIMENTO (embora, a maior parte da literatura não use esta nomenclatura para a linguagem C).

Declaração das Funções

```

tipo nome_função (declaracao dos argumentos)    ⇒ declaração do
                                                    protótipo da função
                                                    deve preceder sua
                                                    definição e chamada.

void main(void)
{
    a=nome_função(lista dos argumentos a serem enviados); ⇒ chama
                                                    a função f1, a qual retorna um valor em a
}

tipo nome_função(lista dos parâmetros recebidos)
{
    declaração das variáveis locais
    comandos;
    return(valor);    ⇒ retorno de um valor
}

```

* Expressões em negrito correspondem a comandos opcionais.

Tipos de Funções

O tipo de uma função é determinado pelo tipo de valor que ela retorna pelo comando **return** e não pelo tipo de seus argumentos.

Se uma função for do tipo não inteira ela deve ser declarada. O valor default é int caso não for declarada.

Alguns tipos:

float - retorna um valor numérico real

int - retorna valor inteiro (não é necessária a sua declaração)

void - funções que não retornam valores.

Exemplo :

```
/* calcula a área de uma esfera */

#define PI 3.14159;
float area(int r); /* protótipo da função */

void main(void)
{
    int raio;
    float area_esfera;
    printf ("Digite o raio da esfera: ");
    scanf("%d", &raio);
    area_esfera=area(raio); /* chamada à função */
    printf("A área da esfera é %f", area_esfera);
}

float area(int r) /* definicao da funcao */
{
    return(4*PI*r*r); /* retorna float */
}
```

CLASSES DE ARMAZENAMENTO

Todas variáveis e funções em C têm dois atributos: um tipo e uma classe de armazenamento. Os tipos nós já conhecemos. As classes de armazenamento são 4:

*	<code>auto</code>	(automáticas)
*	<code>extern</code>	(externas)
*	<code>static</code>	(estáticas)
*	<code>register</code>	(em registradores)

auto

As variáveis declaradas dentro de uma função são automáticas por default. Variáveis automáticas são as mais comuns dentre as quatro classes, elas são criadas quando a função é chamada e destruídas quando a função termina a sua execução.

```
void main(void)
{
    auto int i;
    :
}
```

extern

Todas variáveis declaradas fora de qualquer função têm a classe de armazenamento `extern`. Variáveis declaradas com esse atributo serão conhecidas por todas as funções declaradas depois delas.

```
void main(void)
{
    extern int x;
    ...
}
```

A palavra **extern** não é usada para criar variáveis da classe **extern** e sim para informar ao compilador que uma variável em questão foi criada em outro programa compilado separadamente e será linkeditado junto a este para formar o programa final.

static

Variáveis *static* de um lado se assemelham às automáticas, pois são conhecidas somente as funções que as declaram e de outro lado se assemelham às externas pois mantêm seus valores mesmo quando a função termina.

Declarações *static* permitem a variáveis locais reterem seus valores mesmo após o término da execução do bloco onde são declaradas.

register

A classe de armazenamento *register* indica que a variável associada deve ser guardada fisicamente numa memória de acesso muito mais rápido chamada registrador. Um registrador da máquina é um espaço de 16 bits onde podemos armazenar um **int** ou um **char**.

Basicamente variáveis **register** são usadas para aumentar a velocidade de processamento.

Exemplo :

```
#include <time.h>

void main(void)
{
    int i,j;
    register int m,n;
    long t;

    t=time(0);
    for(j=0;j<5000;j++)
        for (i=0;i<5000;i++);

    printf("\n Tempo dos laços não register: %ld",time(0)-t);

    t=time(0);
    for(m=0;m<5000;m++)
        for (n=0;n<5000;n++);

    printf("\n Tempo dos laços register: %ld",time(0)-t);
}
```

DIRETIVAS

#define

A diretiva **#define** pode ser usada para definir constantes simbólicas com nomes apropriados.

Sintaxe : #define <identificador> <texto>

Exemplo :

```
#define PI 3.14159
#define begin }
#define end {
#define PRN(n) printf("%.2f\n",n)

void main(void)
begin
    float num;
    num+=PI;
    PRN(num) ;
end
```

Quando o compilador encontra a diretiva **#define**, procura em cada linha abaixo dela a ocorrência do identificador e a substitui pelo texto

#undef

A diretiva **#undef** remove a mais recente definição criada com **#define**.

Exemplo :

```
#undef PI /* cancela a definição de PI */
```

#include

A diretiva **#include**, como o nome sugere, inclui um programa-fonte ou um “header file” no fonte corrente, durante a compilação, ou seja, na verdade, o compilador substitui a diretiva **#include** do programa pelo conteúdo de arquivo indicado.

Exemplo :

```
#include <stdio.h>           ❶
#include "funcoes.c"         ❷
#include "c:\prog\mouse.c"   ❸
void main(void)
{
    int a,b;
}
```

Quando o arquivo a ser incluído estiver delimitado por “< >”, o caminho de procura será o diretório C:\TC\INCLUDE ou o diretório setado para conter os “header files” (1) . Quando delimitado por ASPAS, o caminho de procura será o diretório corrente (2) ou o especificado na declaração da diretiva #include (3).

Geralmente os arquivos de inclusão ou “header files” tem o nome com a extensão “.h” e estão gravados no diretório INCLUDE.

#undef, #if, #ifdef, #ifndef, #else, #endif

Estas diretivas são geralmente usadas em grandes programas. Elas permitem suspender definições anteriores e produzir arquivos que podem ser compilado de mais de um modo.

VETORES E MATRIZES

VETOR/MATRIZ

Um vetor/matriz é um tipo de dado usado para representar uma certa quantidade de variáveis de valores homogêneos. A linguagem C reserva o espaço de memória referente a *tam* vezes o **tipo** da variável, e esse espaço pode ser referenciado por um índice.

Sintaxe: **tipo** *nomevar* [*tam*] ;

Onde:

tipo é o tipo do vetor a ser construído, pode ser qualquer um dos tipos básico da linguagem C como float, int e char, ou até mesmo um tipo definido pelo usuário.

nomevar é o nome da variável que será utilizada como um vetor, ou seja, é o nome do vetor construído.

tam é o tamanho do vetor a ser criado, ou seja, é a quantidade de variáveis do tipo **tipo** que serão criadas, cada uma delas referenciadas por um índice, começando-se pelo 0.

NOTE...

- A linguagem C não valida limites em matrizes, cabe ao programador verificar o correto dimensionamento das mesmas;
- O primeiro índice de uma matriz é o índice 0;
- Vetores em C possui de uma ou mais dimensões, como convenção chamamos vetores bidimensionais de matrizes;
- O nome de uma matriz desacompanhado de colchetes representa o endereço de memória onde a matriz foi armazenada.

Exemplos:

```
int meses[12];
```

Resultado na Memória:

0	1	2	3	4	5	6	7	8	9	10	11
lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo	lixo

```
float tabela [2][2];;
```

Resultado na Memória:

	0	1
0	lixo	lixo
1	lixo	lixo

Referência aos Elementos

A linguagem C para referenciar uma determinada posição no vetor unidimensional utiliza-se o seguinte formato:

nomevar [*pos*]

Em um vetor bidimensional, e de maiores dimensões utiliza-se o seguinte padrão:

nomevar [*pos1*][*pos2*][*pos3*]..*[posn]*

Exemplo:

```
meses[0] = 8;
```

```
meses[5] = 6;
```

Resultado na Memória:

0	1	2	3	4	5	6	7	8	9	10	11
8	lixo	lixo	lixo	lixo	6	lixo	lixo	lixo	lixo	lixo	lixo

Exemplo:

```
tabela[1][0] = 3.14;
```

```
tabela[0][1] = 6.2;
```

Resultado na Memória:

	0	1
0	lixo	6.2
1	3.14	lixo

Inicialização de vetores/matrizes

Considerando uma variável inteira *numero*, pode-se inicializar a variável *numero* das seguintes formas:

```
int numero = 0;
numero = 0;
scanf ("%d", &numero);
```

A matriz como é uma variável, pode-se utilizar também esses formatos. Para fornecer o conteúdo de cada um dos elementos da matriz na sua declaração, utiliza-se o seguinte formato:

tipo nomevar [*tam*] = { *valor1*, *valor2*, *valor3*, ... *valorTam* };

Exemplos:

```
int tab[5] = {10, 20, 30, 40, 1};
```

Resultado na Memória:

0	1	2	3	4
10	20	30	40	1

```
int mat[2][2]={10,10},{20,20}};
```

Resultado na Memória:

	0	1
0	10	10
1	20	20

Pode-se fazer também atribuições das seguintes formas:

```
tab[0] = 0;
tab[1] = 0;
...
tab[4] = 0;
for ( i = 0 ; i < 5; i++)
    scanf ("%d", &notas[ i ] );
```

Exemplos:

Contar o número de vezes que um dado número aparece no vetor

```
#define TAM 20
void main()
{
    int i, numeros[TAM];
    int procurado, qte;
    for( i = 0; i < TAM; i++)
        scanf("%d", &numeros[i]);
    scanf("%d", &procurado);
    qte = 0;
    for( i = 0; i < TAM; i++)
        if(procurado == numeros[i])
            qte++;
    printf("o número procurado apareceu %d vezes", qte);
}
```

Programa para imprimir a média da classe e as notas dos alunos

```
#define N_ALUNOS 40

void main()
{
    int i;
    float notas[N_ALUNOS], media = 0;

    for( i = 0; i < N_ALUNOS; i++ )
    {
        printf ("entre com a nota %d", i+1);
        scanf ("%f", &notas[ i ]);
        media += notas [ i ];
    }
    printf (" Média = %f \n", media / N_ALUNOS);
    for ( i = 0; i < N_ALUNOS; i++ )
        printf ("\n Nota do aluno %d = %f \n", i+1, notas[ i ]);
}
```

Em C, é possível inicializar um arranjo sem que se defina a sua dimensão, indicando-se os valores iniciais, o compilador fixará a dimensão do arranjo

Exemplo:

```
int mat[2][2]={10,10},{20,20};
```

Resultado na Memória:

	0	1
0	10	10
1	20	20

Não se pode inicializar o i-ésimo elemento sem inicializar todos os anteriores, mas pode-se inicializar apenas os primeiros elementos de um vetor. Como mostra o exemplo abaixo.

```
int notas [ 5 ] = { , , 0, , } // ERRO
int notas [    ] = { , , 0, , } // ERRO

int notas [ 5 ] = {1, 2 }
```

A declaração acima tem o mesmo efeito desta próxima.

```
int notas [ 5 ] = {1, 2, 0, 0, 0 }
```

Verificando limites

A linguagem C não realiza verificação de limites em arranjos (overflow), nada impede o acesso além do fim do arranjo, esse acesso pode causar resultados imprevisíveis. Devido a esse problema sempre faça a verificação dos limites dos vetores.

Exemplo:

```
#define TAM 100
int notas [ TAM ], quantidade;
do {
    printf ( "quantas notas devo ler ?");
    scanf("%d", &quantidade);
} while ( quantidade > TAM );
```

Exemplo com enum:

```
#include <stdio.h>
enum mes { JAN, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT, NOV, DEZ };
void main()
{
    enum mes index;
    char *meses[12] = { "Janeiro", "Fevereiro", "Marco", "Abril", "Maio",
        "Junho", "Julho", "Agosto", "Setembro", "Outubro", "Novembro", "Dezembro" };
    for (index = JAN; index <= DEZ; index++)
        printf("\n%s", meses[index]);
}
```

STRINGS

String é uma útil e importante representação de dados em C. É usada para armazenar e manipular textos como palavras, nomes e sentenças.

String é uma *array* do tipo **char** terminada pelo caracter NULL ('\0'), ou seja, **string** é uma série de caracteres, onde cada caracter ocupa um byte de memória, são armazenados em sequência e terminados por um byte de valor zero ('\0'). Cada caracter é um elemento independente e pode ser acessado através de um índice.

Exemplos:

```
char nome[5]; ou
char nome[5]="Joao"; ou
char nome[5]="{'J','o','a','o'}";
```

0	1	2	3	4
'j'	'o'	'a'	'o'	'\0'

```
char vetor[]="abc"; sendo que:
```

```
vetor[0]='a'
vetor[1]='b'
vetor[2]='c'
vetor[3]='\0'           ⇒ indica final do vetor
```

```
char matriz[10][20] ⇒ duas dimensões (matriz)
```

```
char nome[3][6] =
    {'V','i','l','m','a'},
    {'D','i','r','c','e'},
    {'G','i','l','d','o'}}; ou
```

```
char nome[3][6]={"Vilma","Dirce","Gildo"};
```

	0	1	2	3	4	5
0	'V'	'i'	'l'	'm'	'a'	'\0'
1	'D'	'i'	'r'	'c'	'e'	'\0'
2	'G'	'i'	'l'	'd'	'o'	'\0'

Lendo strings

scanf()

A função `scanf` é bastante limitada para a leitura de strings. A função considera o caracter [espaço] como final de string, por exemplo a leitura do nome **Antônio Coimbra de Jesus** pelo `scanf()` (`scanf("%s",nome);`) resultará em apenas "Antônio".

gets()

Função própria para leitura de string, a leitura é delimitada pela tecla <ENTER>. *Exemplo :*

Sintaxe : `gets (string);`

Exemplo :

`gets (nome) ;`

Imprimindo strings

puts()

`puts(nome);` ⇒ Sérgio Ferreira
`puts(&nome[4]);` ⇒ gio Ferreira

printf()

`printf("%s",nome);` ⇒ Sérgio Ferreira

Funções de manipulação de strings

strlen()

Retorna o tamanho da string, a partir de um endereço da string até o caracter anterior a '\0'.

Sintaxe : `int strlen(string);`

Exemplo:

```
char nome[ ]="José Carlos";
strlen(nome);           ⇒ 11
strlen(&nome[2]);       ⇒ 09
```

strcat()

Concatena duas strings.

Sintaxe: `strcat(string1,string2);`

Exemplo:

```
char nome[]="Patricia";
char prof[]="estuda";
strcat(nome,prof);      ⇒ nome="Patricia estuda"
                        ⇒ prof continua sendo "estuda"
```

strcmp()

Compara duas strings.

Sintaxe :

```
int strcmp(string1,string2);
```

valor de retorno:

<0	⇒ string1<string2
0	⇒ string1=string2
>0	⇒ string1>string2

Exemplo :

```
printf("%d",strcmp("Ana","Alice"));
```

⇒ retornará a diferença em ASCII entre os primeiros dois caracteres diferentes, no caso um valor negativo.

strcpy()

Copia strings

Sintaxe : void strcpy(string_destino,string_origem);

Exemplo :

```
strcpy(nome,"ANA");  
strcpy(nome1,nome2);
```

ESTRUTURAS

Agrupamento de um conjunto de dados não similares sob um único nome, ou seja, estruturas são tipos de variáveis que agrupam dados geralmente desiguais. Os itens de dados de uma estrutura são chamados de membros.

```

    nome do tipo recém criado
      ↑
struct nome
{
    tipo variavel;
    ...
} membros da estrutura
};
struct nome nomex; → declaração da variável nomex como do tipo "nome"

```

Por meio da palavra-chave **struct** definimos um novo tipo de dado. Definir um tipo de dado significa informar ao compilador o seu nome, tamanho em bytes e o formato em que ele deve ser armazenado e recuperado na memória.

Após ter sido definido, o novo tipo existe e pode ser utilizado para criar variáveis de modo similar a qualquer tipo simples.

Definir uma estrutura não cria nenhuma variável, somente informa ao compilador as características de um novo tipo de dado. Não há nenhuma reserva de memória.

Exemplo :

```

struct livro
{
    int reg;
    char titulo[30];
    char autor[30];
};
struct livro livrox;

```

Acessando a Estrutura

```

livrox.reg=10;
gets(livrox.titulo);
strcpy(livrox.autor,"Alvares de Azevedo");

```

Matrizes de Estruturas

O processo de declaração de uma matriz de estruturas é perfeitamente análogo à declaração de qualquer outro tipo de matriz :

```

struct livro livrox[50];   ⇒ declara a estrutura livrox como sendo uma matriz de
                           50 elementos.

```

Para acessar a Estrutura :

```

livrox[2].reg=1;   ⇒ refere-se ao membro reg da 3ª estrutura da matriz de estruturas livrox.

```

Inicializando uma Estrutura

```
struct data
{
    char nome[80];
    int mes;
    int dia;
    int ano;
};

struct data aniversario[] = {"Ana",12,30,73,
                             "Carlos",5,13,66,
                             "Mara",11,29,70};
```

PONTEIROS

É uma das mais poderosas estruturas oferecidas pela linguagem C. O ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente. O mecanismo usado para isto é o endereço da variável, sendo o ponteiro a representação simbólica de um endereço.

Utilização dos ponteiros :

- * Manipular elementos de matrizes;
- * Receber argumentos em funções que necessitem modificar o argumento original;
- * Passar strings de uma função para outra; usá-los no lugar de matrizes;
- * Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde um item deve conter referências a outro;
- * Alocar e desalocar memória do sistema.

A memória do computador é dividida em bytes, e estes bytes são numerados de 0 até o limite da memória da máquina. Estes números são chamados **endereços** de bytes. Um endereço é a referência que o computador usa para localizar variáveis. Toda variável ocupa uma certa localização na memória, e seu endereço é o do primeiro byte ocupado por ela.

O C oferece dois operadores unários para trabalharem com ponteiros. Um é o operador de endereço (&) que retorna o endereço de memória da variável. O segundo é o operador indireto (*) que é o complemento de (&) e retorna o conteúdo da variável localizada no endereço (ponteiro) operando, isto é, devolve o conteúdo da variável apontada pelo operando.

Declaração: **tipo** *variável;

Exemplos de declaração ::

```
char *nome, *sexo;
int *idade;
float *medida;
```

⇒ declara que as variáveis nome, sexo, idade e medida são do tipo char, int e float respectivamente e são ponteiros.

```
char x='Z', y='K';
char *px, *py;
px = &x;
py=&y;

printf("%u",&px);
    ⇒ end. de px = 100
printf("%u",px);
    ⇒ cont. de px= 1000
printf("%d",*px);
    ⇒ cont. onde px aponta='Z'
printf("%d",*(px++));
    ⇒ cont. onde px+1 aponta='K'
```

endereço	variável	conteúdo
100	px	1000
101	py	1001
102		
M	M	M
1000	x	'Z'
1001	y	'K'
1002		
1003		

Programa exemplo :

```
void reajusta20(float *p, float *r);

void main(void)
{
    float preco, val_reaj;
    do
    {
        printf("Digite o preco atual");
        scanf ("%f",&preco);
        reajusta20(&preco, &val_reaj); /* envia o endereço dos
                                         parâmetros preco e val_reaj,
                                         que terão seu conteúdo
                                         atualizados pelos ponteiros */

        printf("Preco novo : = %f",preco);
        printf("\nAumento : = %f",val_reaj);
    }
    while (preco != 0.0);
}

void reajusta20(float *p, float *r)
{
    *r=*p*0.2;
    *p=*p*1.2;
}
```

Matrizes de Ponteiros

O uso mais comum de matrizes de ponteiros é a construção de matrizes de ponteiros para strings. Uma matriz de ponteiros é descrita a seguir :

```
char *semana[7]={ "Domingo", "Segunda", "Terça", "Quarta", "Quinta",
                  "Sexta", "Sábado"};
```

	0	1	2	3	4	5	6	7
0	'D'	'o'	'm'	'i'	'n'	'g'	'o'	'\0'
1	'S'	'e'	'g'	'u'	'n'	'd'	'a'	'\0'
2	'T'	'e'	'r'	'ç'	'a'	'\0'		
3	'Q'	'u'	'a'	'r'	't'	'a'	'\0'	
4	'Q'	'u'	'i'	'n'	't'	'a'	'\0'	
5	'S'	'e'	'x'	't'	'a'	'\0'		
6	'S'	'á'	'b'	'a'	'd'	'o'	'\0'	

Ponteiros para Estruturas

`struct livro *biblio;` ⇒ declara o ponteiro **biblio** como do tipo **struct livro**.

`biblio=&livro[0];` ⇒ aponta o ponteiro **biblio** para a primeira posição da estrutura **livro**.

Acessando os membros através do ponteiro

`livro->preço` ou `(*livro).preço`

Ponteiros x Vetores...

Vetor e ponteiro são quase a mesma coisa, inclusive podem ser tratados igualmente. Para entender melhor o que acontece em funções que recebem vetores como argumento é melhor aprender como funciona os ponteiros.

Quando se passa uma variável simples como argumento para uma função, a função faz uma cópia daquela variável e trabalha com ela. Se o valor dessa variável for alterado, ela não estará mudada quando sair da função.

Ao passar um vetor como argumento, não será passado todo o conteúdo do vetor para a função, na verdade será passado apenas o endereço de onde o vetor está alocado na memória (diferentemente de passar uma variável como argumento), e se dentro da função for alterado o conteúdo do vetor, quando sair da função o vetor estará alterado.

Declarando o seguinte vetor: `char nome[10];` a variável "nome" estará apontando para o início desses 10 bytes alocados na memória. Se for passado esse vetor como argumento, na verdade será enviado somente o endereço de onde começam esses 10 bytes alocados.

Para retornar vetores, o processo é semelhante. Não se retorna todo o conteúdo de um vetor, e sim apenas o endereço de memória onde o vetor foi alocado, então se deve retornar o endereço do vetor:

```
char *retVetor(void)
{
    char vet[30]; //ou char *vet;
    ...
    return(vet);
}
```

Mas lembre-se, somente um ponteiro poderá receber o retorno desta função. Não é possível alterar o endereço de um vetor depois de declarado. Esta é a principal diferença funcional entre vetores e ponteiros.

```
char *pvet;
pvet=retVetor(); // funciona!

e não:
char vetor[30];
vetor=retVetor(); // não funciona!
```

ARGUMENTO DE LINHA DE COMANDO

Alguns programas, ao ser ativado pelo sistema operacional, requer não apenas o nome digitado, mas também outros itens, como exemplo :

```
C:\> FORMAT A:
C:\> EDIT TEXTO.TXT
```

No exemplo, A e TEXTO.TXT é um argumento de linha de comando. Os itens digitados na linha de comando do DOS são chamados **argumentos de linha de comando**.

Os argumentos digitados na linha de comando são enviados pelo DOS como argumentos da função **main()**. Para que esta função possa reconhecê-los, é necessário declarar os argumentos **argc** e **argv**.

argc argumento inteiro que corresponde ao número de argumentos da linha de comando.

argv representa uma matriz de ponteiros para "string", onde cada "string" representa um argumento da linha de comandos.

Exemplo :

```
/* Programa teste */
void main(int argc, char *argv[])
{
    int i;
    printf("Nº de argumentos é %d", argc);
    for(i=0; i<argc; i++)
        printf("Argumento nº %d é %s", i, argv[i]);
}
```

Para Executar :

```
C:\> teste imprimindo argumentos
  ↓      ↓      ↓      ↓
prompt programa primeiro segundo
do DOS  gerado  argumento argumento
```

Resultado :

```
Nº de argumentos é 3
Argumento nº 0 é c:\> teste.exe
Argumento nº 1 é imprimindo
Argumento nº 2 é argumentos
```


ALOCANDO MEMÓRIA

A Função malloc()

Usado em ponteiros, a função malloc reserva espaço de memória livre. Toma um inteiro sem sinal como argumento. Este número representa a quantidade em bytes de memória requerida. A função retorna um ponteiro para o primeiro byte do novo bloco de memória que foi alocado. Caso não haja memória suficiente para satisfazer a exigência **malloc()** devolverá um ponteiro NULL. Não se esqueça de incluir o "header file" **<alloc.h>**

Exemplo :

```
pont=malloc(sizeof(struct livro)) ⇒ reserva uma área correspondente ao tamanho
                                   da estrutura livro
pont=malloc(500) ⇒ reserva 500 bytes de memória para o ponteiro pont
```

A Função calloc()

Há uma grande semelhança entre **malloc()** e **calloc()** que também retorna um ponteiro para **void** apontando para o primeiro byte do bloco solicitado. A função aceita dois argumentos do tipo **unsigned int**.

Exemplo :

```
long *mem;
mem = (long *) calloc(100, sizeof(long));
```

↓
↓
↓

①
②
③

- 1-"cast" que transforma p/ tipo long
- 2-número de células de memória desejada
- 3-tamanho de cada célula em bytes

A Função free()

É o complemento de **malloc()** e **calloc()**. Aceita como argumento um ponteiro para uma área de memória previamente alocada por **malloc()** ou **calloc()** e então libera esta área para uma possível utilização futura.

Exemplo :

```
free(mem); ⇒ Libera o espaço outrora reservado para o ponteiro "mem"
```

UNIÕES

Union é semelhante à estruturas, mas enquanto os membros das estruturas são armazenados em espaços diferentes da memória, membros da união compartilham da mesma localização da memória. Assim, as "union" são utilizadas para economizar memória. Mas, cuidado!, somente um membro da "union" pode conter valores no mesmo instante.

```

"etiqueta" da union
    ↑
union nome
{
    tipo variavel;
    ...
} membros da estrutura
|
union nome nomex; ⇒ declaração da variável nomex como do tipo "struct nome"
  
```

Exemplo :

```

union numero
{
    double dnum;
    float fnum;
    int inum;
    char cnum
};
  
```

Utilização :

Em situações que necessite um mesmo dado seja tratado como tipos diferente e quando à necessidade de economia de memória.

OPERAÇÕES COM ARQUIVOS EM DISCO

A linguagem C divide as categorias de acesso a disco em dois grandes grupos chamados de alto nível, ou leitura e gravação *bufferizada*, e de baixo-nível, ou leitura e gravação não *bufferizada*.

ALTO NÍVEL

Caractere: getc() ou fgetc(), putc() ou fputc()

String: fgets(), fputs()

Formatada: fscanf(), fprintf()

Registro: fread(), fwrite()

BAIXO NÍVEL

read(), write()

Arquivos de Texto

Um arquivo aberto em modo texto é uma seqüência de caracteres agrupadas em linhas. As linhas são separadas pelo caractere de final de linha lógico ou LF, de código ASCII 10 decimal.

No DOS um arquivo aberto em modo texto são seqüências de caracteres agrupadas em linhas e essas são separadas por dois caracteres, o caractere 13 decimal, ou CR, e o caractere 10 decimal ou LF.

O compilador C converte o par CR/LF em um único caractere de final de linha quando um arquivo aberto em modo texto é lido, e converte o caractere de final de linha no par CR/LF quando o arquivo é gravado em disco.

Se um arquivo é aberto em modo texto, ele reconhece a indicação de fim de arquivo enviada pelo DOS.

Abrir um Arquivo

Para trabalhar com um arquivo de dados, o primeiro passo é estabelecer um *buffer*, onde as informações serão armazenadas temporariamente entre a memória do computador e o arquivo de dados. Esse buffer permite que informações sejam lidas de/ou escritas em arquivo de dados mais rapidamente.

```
FILE *fptr
```

FILE (maiúsculo) é uma estrutura pré definida FILE (no *header file* stdio.h) que estabelece o buffer e *fptr* é uma variável ponteiro que indica o início de buffer.

Para gerar um código de programa que abre um arquivo, o compilador precisa conhecer 3 aspectos:

- 1º O nome do arquivo que será usado
- 2º O tipo de abertura
- 3º Onde guardar informações sobre o arquivo.

```
fptr=fopen("arquivo.txt","w");
```

③ ① ②

Abre um arquivo chamado "arquivo.txt" no disco corrente para escrita "w", a qual será armazenada na variável **fptr** declarada como ponteiro para o tipo FILE.

Quando é necessário colocar o caminho completo do arquivo, e a conseqüente presença de barras invertidas " \ " como em "c:\aluno\teste.txt", ao invés de colocar uma barra, deve-se colocar "\\", portanto ficando assim "c:\\aluno\\teste.txt";

A função *fopen* retorna o valor NULL caso o arquivo não possa ser aberto.

Exemplos:

```
FILE *cliente; //Definição do buffer cliente

FILE *fornecedor; //Definição do buffer fornecedor

cliente = fopen("clientes.txt", "w");

fornecedor = fopen("c:\\aluno\\fornec.txt", "r");
```

Tipos de abertura de arquivo:

CÓD.	DESCRIÇÃO
"r"	Abrir um arquivo texto para leitura. O arquivo deve estar presente no disco
"w"	Abrir um arquivo texto para gravação. Se o arquivo estiver presente ele será destruído e reinicializado. Se não existir, ele será criado.
"a"	Abrir um arquivo texto para inclusão (acréscimo de informações). Os dados serão adicionados ao fim do arquivo existente, ou um novo arquivo será criado.
"r+"	Abrir um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser atualizado.
"w+"	Abrir um arquivo texto para leitura e gravação. Se o arquivo existir ele será destruído e reinicializado. Se não existir, será criado.
"a+"	Abrir um arquivo texto para leitura e para adicionar dados ao fim do arquivo existente. Caso não exista, um novo arquivo será criado.
"rb"	Abrir um arquivo binário para leitura. O arquivo deve estar presente no disco.
"wb"	Abrir um arquivo binário para gravação. Se o arquivo estiver presente ele será destruído e reinicializado. Se não existir, ele será criado.
"ab"	Abrir um arquivo binário para inclusão. Os dados serão adicionados ao fim do arquivo existente, ou um novo arquivo será criado. Não permite leitura dos dados.
"rb+"	Abrir um arquivo binário para leitura e gravação. O arquivo deve existir e pode ser atualizado.
"wb+"	Abrir um arquivo binário para leitura e gravação. Se o arquivo existir ele será destruído e reinicializado. Se não existir, será criado.
"ab+"	Abrir um arquivo binário para atualizações e para adicionar dados ao fim do arquivo existente ou um novo arquivo será criado. Possibilita a leitura dos dados existentes.

Fechar um Arquivo - fclose()

Sintaxe: **fclose(nomebuf);**

A função *fclose* é utilizada para fechar um determinado arquivo após sua utilização, é recomendado deixar o arquivo aberto somente quando for utilizá-lo. Quando a função *fclose* é utilizada qualquer caracter que estiver no buffer é gravado em disco evitando perda de dados.

Onde: **nomebuf** é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo a ser fechado.

Verificando final de arquivo - EOF

EOF

EOF é a marca de fim de arquivo. É um constante inteira pré-definida no *header file* `stdio.h`, nos microcomputadores IBM-PC essa constante tem o valor -1. É utilizada quando a leitura é feita caractere a caractere, onde após cada uma das letras lidas pode-se verificar se já está no fim do arquivo, evitando erros no programa.

`feof()`

Sintaxe: `int feof (nomebuf)`

A função **`feof`** é utilizada para a verificação do final do arquivo. Essa função retorna um valor inteiro caso o ponteiro chegou no final do arquivo, caso contrário o retorno é zero. Essa função pode ser utilizada quando a leitura é feita caractere a caractere ou de qualquer outra forma

Onde: **`nomebuf`** é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo onde será feita a verificação de final de arquivo.

Funções para Ler/Gravar caractere a caractere

Leitura Caractere a Caractere - `getc()` ou `fgetc()`

Sintaxe:

`int fgetc(nomebuf);`

`int getc(nomebuf)`

As funções `fgetc` e `getc` são utilizadas para a leitura de caracteres de um arquivo texto, onde os caracteres são lidos um a um. As duas funções trabalham da mesma forma, é necessária a existência das duas para a compatibilidade entre sistemas. A função `getc` retorna o caractere lido do arquivo referenciado por **`nomebuf`** esse caractere pode ser armazenado em uma variável inteira ou caractere. É recomendado a utilização do inteiro para a comparação com a constante inteira **EOF**.

Onde:

`nomebuf` é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo a ser lido.

`varint` é a variável inteira ou caractere que irá armazenar o dado lido.

Gravação Caractere a Caractere - `putc()` ou `fputc()`

Sintaxe:

`fputc(caracter, nomebuf);`

`putc(caracter, nomebuf)`

As funções `fputc` e `putc` são utilizadas para a gravação de caracteres de um arquivo texto,

onde os caracteres são gravados um a um.

Onde:

nomebuf é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo a ser gravado, lembrar que este arquivo deve ser aberto para gravação com w ou a.

caracter é a variável caracter que será armazenada no arquivos especificado por nomebuf.

Funções para Ler/Gravar Strings

Ler strings - fgets()

Sintaxe: `char *fgets(stringlida, tamanho, nomebuf);`

A função *fgets* lê uma string do arquivo especificado por *nomebuf*. Essa função lê os caracteres de um arquivos até que um final de linha seja encontrado, ou até que a string de retorno **stringlida** contenha (**tamanho**-1) caracteres. O retorno da função pode ser comparado com NULL para verificar se já é final de arquivo ou se ocorreu um erro durante a leitura.

Onde:

nomebuf é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo a ser lido.

stringlida é a variável do tipo cadeia de caracteres ou um ponteiro para caracteres, onde serão armazenados os dados lidos do arquivo.

tamanho é a quantidade máxima de caracteres que serão lidos de uma vez.

Exemplo:

```
FILE *fornecedor;

char lido[80];

fornecedor = fopen("c:\\aluno\\fornec.txt", "r");

if (fgets(lido, 80, fornecedor) == NULL)

    printf("Ocorreu um erro na leitura do arquivo ou EOF");

else

    printf("%s", lido);
```

Gravar strings - fputs()

Sintaxe: `fputs(stringgrava, nomebuf);`

A função *fputs* grava a string contida em **stringgrava** do arquivo especificado por **nomebuf**. O final de linha é colocado, ou não dependendo do ambiente utilizado.

Onde:

nomebuf é o nome da variável que está sendo utilizada como ponteiro para o buffer do

arquivo a ser gravado, não esquecendo que este arquivo deve ser aberto para a gravação.

stringgrava é a variável do tipo cadeia de caracteres ou um ponteiro para caracteres, onde será armazenada no arquivo.

Exemplo:

```
FILE *fornecedor;

char gravar[80];

fornecedor = fopen("c:\\aluno\\fornec.txt", "w");

gets(gravar);

fputs(gravar, fornecedor);
```

Funções para Ler/Gravar Strings Formatadas

Ler Strings Formatadas- fscanf()

Sintaxe: **fscanf(nomebuf, formatação, variáveis);**

A função **fscanf** lê uma string formatada do arquivo especificado por **nomebuf**. Essa função lê os caracteres de um arquivo de acordo com a formatação definida por **formatação** e os coloca nas variáveis definidas por **variáveis**.

Onde:

nomebuf é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo a ser lido.

formatação é uma string de formatação, definindo como os dados serão lidos do arquivo. A formatação é definida na mesma forma do scanf.

Gravar Formatadas- Gravar - fprintf()

Sintaxe: **char *fscanf(nomebuf, formatação, variáveis);**

A função **fscanf** lê uma string formatada do arquivo especificado por **nomebuf**. Essa função lê os caracteres de um arquivo de acordo com a formatação definida por **formatação** e os coloca nas variáveis definidas por **variáveis**.

Onde:

nomebuf é o nome da variável que está sendo utilizada como ponteiro para o buffer do arquivo a ser lido.

formatação é uma string de formatação, definindo como os dados serão lidos do arquivo. A formatação é definida na mesma forma do scanf.

Funções para Ler/Gravar Estruturas

Estruturas- Ler - fread()

Sintaxe: fread(&estrutura, sizeof(estrutura), num_estr, nomebuf);

A função **fread** lê num_estr elementos gravados em disco, muito utilizado na leitura de estruturas gravadas em disco. A leitura dos dados depende do tipo de dados que está gravado, caso for um inteiro, ele irá ler o dado em dois bytes. Portanto um arquivo gravado com estruturas, inteiros, e qualquer outro dado que não seja caracter, será ilegível pelo DOS.

Onde:

estrutura é o nome da estrutura que será colocado o dado lido do disco. Antes do nome da estrutura deve-se colocar um & comercial, pois o fread necessita do endereço dessa estrutura.

sizeof(estrutura) é o tamanho em bytes que a estrutura a ser lida possui. Caso for um inteiro essa função retorna o valor 2.

num_estr é a quantidade de estruturas serão lidas de uma vez.

nomebuf é o nome da variável relacionada ao arquivo que será lido.

Gravar Estruturas- fwrite()

Sintaxe: fwrite(&estrutura, sizeof(estrutura), num_estr, nomebuf);

A função **fwrite** grava n elementos em disco, muito utilizado na gravação de estruturas em disco. A gravação dos dados depende do tipo de dados que está gravado, caso for um inteiro, ele irá gravar o dado em dois bytes do arquivo. Portanto um arquivo gravado com estruturas, inteiros, e qualquer outro dado que não seja caracter, será ilegível pelo DOS.

Onde:

estrutura é o nome da estrutura que contém os dados a serem gravados no disco. Antes do nome da estrutura deve-se colocar um & comercial, pois o fread necessita do endereço dessa estrutura.

sizeof(estrutura) é o tamanho em bytes que a estrutura a ser armazenada possui. Caso for um inteiro essa função retorna o valor 2.

num_estr é a quantidade de estruturas serão gravados de uma vez.

nomebuf é o nome da variável relacionada ao arquivo que será utilizado para a gravação das informações.

Acesso Aleatório

fseek()

Posiciona o ponteiro do arquivo na posição desejada.

Sintaxe: **fseek**(pont tipo FILE ,desl,modo)



1- ponteiro do tipo file da estrutura

2- faixa de deslocamento do arquivo, que consiste no número do registro a ser apontado vezes o tamanho da estrutura, a partir do 3º argumento.

3- determina a partir de onde o deslocamento começará a ser acessado

0 - SEEK_SET - começo do arquivo

1 - SEEK_CUR - posição corrente do arquivo

2 - SEEK_END - fim do arquivo

ftell()

Sintaxe: long **ftell**(pont. tipo FILE);

Retorna a posição do ponteiro de um arquivo binário em relação ao seu começo. Esta função aceita um único argumento, que é o ponteiro para a estrutura FILE do seu arquivo e retorna um valor do tipo long, que representa o número de bytes do começo do arquivo até a posição atual.

rewind(file)

Retorna o ponteiro do arquivo para o início do arquivo

Condições de Erro

ferror()

Determina se ocorreu algum erro na leitura/gravação, retorna não zero se ocorreu algum erro e 0 se nenhum erro ocorreu.

Sintaxe: int **ferror**(ponteiro do tipo FILE)

perror()

Imprime a mensagem do erro fornecida pelo sistema.

Sintaxe: **perror**(String);

Exemplo :

```

/* Tentativa de abrir para escrita um arquivo protegido */
FILE *fp;
fp=fopen("io.sys","r");
if (ferror(fp))
{
    printf("Arquivo não pode ser aberto");
    perror("Causa do Erro : ");
}
  
```

Resulta em :

```

Arquivo nao pode ser aberto
Causa do Erro : Permission Denied
  
```

ANEXO 1 – FUNÇÕES GRÁFICAS

• FUNÇÕES GRÁFICAS - TURBO C

CONTROLE DA PLACA GRÁFICA

initgraph: inicializa o modo gráfico

```
void far initgraph(int *graphdriver, int *graphmode, char *pathtodriver)
```

Graphics drivers:

CGA	MCGA	EGA
EGA64	EGAMONO	IBM8514
HERCMONO	ATT400	VGA
PC3270	DETECT (Auto detecção)	

Graphics modes:

Modo	Resolução
CGACO	320x200 palette 0
CGAC1	320x200 palette 1
CGAC2	320x200 palette 2
CGAC3	320x200 palette 3
CGAHI	640x200
MCGACO	320x200 palette 0
MCGACL	320x200 palette 1
MCGAC2	320x200 palette 2
MCGAC3	320x200 palette 3
MCGAMED	640x200
MCGAHI	640x400

Modo	Resolução
EGALO	640x200 -16 cores
EGAHI	640x350 -16cores
EGA64LO	640x200 -16 cores
EGA64HI	640x350 - 4cores
EGAMONOH	640x350
HERCMONOH	720x348
VGALO	640x200
VGAMED	640x350
VGAHI	640x480 -16 cores

DETECT: Detecta o tipo de placa gráfica do Hardware

Ex-.

```
int a=DETECT
int b;
initgraph(&a, &b);
```

detectgraph: determina a placa gráfica (graphic driver) e o modo de uso (graphic mode), checando o hardware.

```
void far detectgraph(int far *graphddver, int far *graphmode);
```

cleardevice: limpa a tela gráfica

```
void far cleardevice(void);
```

closegraph: desativa o modo gráfico

```
void far closegraph(void);
```

setviewport: determina um viewport para as saídas gráficas

```
void far setviewport(int esq, int acima,
    int direita, int abaixo, int dip);
    0 - não recorta
    1 - recorta
```

clearviewport: apaga o viewport corrente

```
void far clearviewport(void);
```

getmaxx: retorna a máxima coordenada no eixo x

getmaxy: retorna a máxima coordenada no eixo y

```
int far getmaxx(void);
```

```
int far getmaxy(void);
```

getx: retorna a coordenada (x) corrente

gety: retorna a coordenada (y) corrente

```
int far getx(void);
```

```
int far gety(void);
```

setcolor : seta o cor de desenho

```
void far setcolor (int cor);
```

setbkcolor : seta a cor de fundo

```
void far setbkcolor(int cor);
```

setfillstyle : seta cor e textura para preenchimento de figuras

```
void far setfillstyle(int padrão, int cor);
```

Padrões :

EMPTY_FILL

SOLID_FILL

LINE_FILL

LTSLASH_FILL

SLASH_FILL

BKSLASH_FILL

LTBKSLASH_FILL

HATCH_FILL

XHATCH_FILL

INTERLEAVE_FILL

WIDE_DOT_FILL

CLOSE_DOT_FILL

Cores :

Constante	Id	Cor
BLACK	0	Preto
BLUE	1	Azul
GREEN	2	Verde
CYAN	3	Roxo
RED	4	Vermelho
MAGENTA	5	Magenta
BROWN	6	Marrom
LIGHTGRAY	7	Cinza Claro
DARKGRAY	8	Cinza Escuro

Constante	Id	Cor
LIGHTBLUE	9	Azul Claro
LIGHTGREEN	10	Verde Claro
LIGHTCYAN	11	Roxo Claro
LIGHTRED	12	Vermelho Claro
LIGHTMAGENTA	13	Magenta Claro
YELLOW	14	Amarelo
WHITE	15	Branco
BLINK	128	

FIGURAS GEOMÉTRICAS

putpixel : desenha um ponto na tela

```
void far putpixel (int x1, int y1, int cor);
```

moveto : move o cursor gráfico

```
void far moveto(int x, int y);
```

line: desenha uma linha entre dois pontos especificados

```
void far line(int x1 , int y1 , int x2, int y2);
```

lineto: desenha uma linha da posição corrente do *cursor* até a posição (x,y)

```
void far lineto(int x, int y);
```

arc: desenha um arco

```
void far arc(int x, int y, int inic_angulo, int fim_angulo, int raio);
```

circle: desenha um círculo

```
void far circle(int x, int y, int raio);
```

ellipse: desenha uma elipse

```
void far ellipse(int x, int y, int inic_angulo, int fim_angulo,
                int xraio, int yraio);
```

fillellipse: desenha e preenche um arco

```
void far fillellipse(int x, int y, int xradius, int yradius);
```

pieslice: desenha e preenche um “pedaço de pizza”

```
void far pieslice(int x, int y, int inic_angulo, int fim_angulo, int raio);
```

bar: desenha uma barra : “retângulo preenchido”

```
void far bar(int esq, int topo, int direita, int base);
```

bar3d: desenha uma barra em 3-D

```
void far bar3d(int esq, int topo, int direita,
               int base int profundidade, int topflag);
```

rectangle: desenha um retângulo

```
void far rectangle(int esq, int topo, int direita, int base);
```

drawpoly : desenha um polígono a partir de um vetor de pontos. *Num_pontos* deve representar a quantidade de pontos do polígono.

```
void far drawpoly(int num_pontos, int *pontos);
```

fillpoly : desenha e preenche um polígono a partir de um vetor de pontos. Preenche o polígono usando a cor e textura corrente.

```
void far fillpoly(int numpoints, int far *polypoints);
```

floodfill : expande um ponto até encontrar uma cor de borda especificada. Usa as cores de preenchimento corrente. Retorna o valor -7 caso não possa ser ativado.

```
void far floodfill(int x, int y, int border);
```

TEXTOS GRÁFICOS

settextstyle: seta as características do texto

```
void far settextstyle(int fonte, int direcao, int tamanho);
```

fontes:

DEFAULT_FONT	TRIPLEX_FONT	SMALL_FONT
SANS_SERIF_FONT	GOTHIC_FONT	SCRIPT_FONT
SIMPLEX_FONT	TRIPLEX_SCR_FONT	COMPLEX_FONT
EUROPEAN_FONT		
BOLD_FONT		

direção

HORIZ_DIR	esquerda p/ direita	VERT_DIR	base para o topo
-----------	---------------------	----------	------------------

tamanho:

1 - 8x8 2 - 16x16 ... 10-80x80

outtext: imprime uma string no viewport / tela gráfica

```
void far outtext(char *textstring);
```

outtextxy: imprime uma string a partir de coordenada especificada

```
void far outtextxy(int x, int y, char *textstring);
```

textheight: retorna a altura da string em pixels

```
int far textheight(char *textstring);
```

textwidth: retorna o tamanho de uma string, em pixels

```
int far textwidth(char far *textstring);
```

settextjustify : alinha um texto gráfico a partir de um ponto configurado por esta função. O default é um ponto superior esquerdo

TEXTO

```
void far settextjustify(int horiz,  int vert);
                        LEFT_TEXT   BOTTOM_TEXT
                        CENTER_TEXT CENTER_TEXT
                        RIGHT_TEXT  TOP_TEXT
```

OUTROS

getimage : Salva uma região da tela em um vetor de caracteres

```
void far getimage(int esq, int topo, int direita, int base, char *imagem);
```

putimage : Restaura uma tela capturada pela função getimage

```
void far putimage(int esq, int topo, char far *imagem, int op);
                                     ↓
                                COPY_PUT    0      Normal
                                XOR_PUT     1      Or exclusivo
                                OR_PUT      2      Or
                                AND_PUT     3      And
                                NOT_PUT     4      Inverso
```

imagesize : retorna o tamanho em bytes necessário para armazenar uma região da tela especificada. O tamanho da região deve ser menor que 64K, senão o retorno será -1.

```
unsigned far imagesize(int esq, int topo, int direita, int base);
```

Exemplo:

```
#include <graphics.h>
void main(void)
{
    int a,b,i;
    char *palavra="Teste";
    a=DETECT,
    initgraph(&a,&b,"");
    setcolor(14);
    setbkcolor(2)
    rectangle(0,0,639,479);
    setcolor(4);
    for (i=50;i<300;i=i+50)
    {
        bar3d(i+90,i,i+60,200,30,5);
    }
    bar3d(50,50,50,400,1,1 00);
    getch();
    setcolor(BLUE);
    settextstyle(GOTHIC_FONT,HORIZ_DIR,5);
    outtextxy(300,320,palavra);
    pieslice(100,350,1,100,45);
    line(10,10,400,400);
    getch();
    closegraph();
}
```

BIBLIOGRAFIA

- Gottfried, Byron Stuart; Programando em C, Trad. Ana Beatriz Correa da Costa Parra, São Paulo, Makron Books, 1993.
- Mayer, R.C.; Linguagem C ANSI, Guia do Usuário, São Paulo, McGraw-Hill, 1989.
- Mizrahi, Victorine Viviane; Treinamento em Linguagem C++, vol. I e II, São Paulo, McGraw-Hill, 1994.
- Mizrahi, Victorine Viviane; Treinamento em Linguagem C, vol. I e II, São Paulo, McGraw-Hill, 1990.
- Schildt, Herbert, Guia do Usuário, Trad. Maria Cláudia de Oliveira Santos, São Paulo, McGraw-Hill, 1988.
- Schildt, Herbert, Guia Prático e Interativo, trad. Lars Gustav Erik Unonius, São Paulo, McGraw-Hill, 1989.
- Swan, Tom; Aprendendo C++, Trad. Carlos Montez e Helcio Tonnera Jr., Rio de Janeiro, Campus, 1993.
- Turbo C 2.0 - User Guide.
- Turbo C 2.0 - User Reference.
- Weiskamp, Keith; Programação Orientada para objeto com Turbo C++, São Paulo, Makron Books, 1993.
- Wiener, Richard; Turbo C, Passo a Passo, Rio de Janeiro, Editora Campus, 1991.