

---

# **LINGUAGEM C**

**POR**

**Prof. Edson J. R. Justino**

2007

---

## ÍNDICE

1. COMENTÁRIOS INICIAIS	01
2. CARACTERÍSTICAS	01
3. COMPILAÇÃO E GERAÇÃO DE CÓDIGO EXECUTÁVEL	01
4. ESTRUTURA DE PROGRAMA EM C	03
4.1. DIRETIVAS DE PRÉ-PROCESSAMENTO	03
4.2. DECLARAÇÃO	03
4.3. DEFINIÇÃO	03
4.4. EXPRESSÃO	04
4.5. COMANDO	04
4.6. FUNÇÃO	04
5. ENTRADA E SAÍDA	05
6. VARIÁVEIS E CONSTANTES	06
6.1. NOMES DE VARIÁVEIS	06
6.2. TIPOS DE DADOS	07
6.2.1. OS TIPOS DE VARIÁVEIS	07
6.2.2. OS TIPOS DE CONSTANTES	07
7. DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS	09
8. OPERADORES	10
8.2. OPERADORES ARITMÉTICOS	10
8.3. OPERADORES RELACIONAIS	11

---

8.4. OPERADORES LÓGICOS	12
8.5. OPERADORES LÓGICOS EM BINÁRIO	14
8.6. OPERADORES DE DESLOCAMENTO DE BITS (SHIFT)	15
8.7. OPERADOR CONDICIONAL	16
8.8. OPERADOR TAMANHO DE VARIÁVEL	
17	
8.9. OPERADOR DE CONVENÇÃO AUTOMÁTICA	18
 9.CONTROLE DE FLUXO	18
9.1. COMANDO IF	18
9.2. COMANDO SWITCH	19
9.3. COMANDO WHILE	20
9.4. COMANDO FOR	21
9.5. COMANDO DO-WHILE	22
9.6. COMANDOS BREAK E CONTINUE	23
 10.VETORES E APONTADORES	24
10.1. DECLARAÇÃO DE VETORES	24
10.2. VETORES MULTIDIMENSIONAIS	26
10.3. APONTADORES	27
 11. FUNÇÕES	28
 12. ESTRUTURAS	32
12.1. SINTAXE	32
12.2. ESTRUTURA COMPOSTA	35

---

12.3.PONTEIROS, FUNÇÕES E ESTRUTURAS	36
13. ARQUIVOS EM C	37
13.1. INTRODUÇÃO	
37	
13.2. ENTRADA E SAÍDA EM ALTO NÍVEL	38
13.3. ABRINDO UM ARQUIVO	39
13.4. LENDO UM CARACTERE DO ARQUIVO	40
13.5. FECHANDO ARQUIVOS	41
13.6. ENTRADA E SAÍDA DE DADOS EM LINHA (REGISTRO) EM ARQUIVOS	43
13.7. REDIRECIONAMENTO	45
13.8. ARQUIVOS FORMATADOS	45
13.9 ARQUIVOS FORMATADOS	46
13.10 ETRADA E SAÍDA DE DADOS EM BLOCOS OU REGISTROS	48
13.11 ARRAY DINÂMICO	51
ANEXOS	
A. BIBLIOTECA PADRÃO	58
BIBLIOGRAFIA	60

---

## 1- COMENTÁRIOS INICIAIS

A linguagem C foi desenvolvida a partir da necessidade de se escrever programas que utilizem os recursos de máquina de uma forma menos penosa e mais portátil que o assembler.

A popularidade da linguagem C deve-se exatamente à elegância em conciliar seu poder de programação em baixo nível com um alto grau de portabilidade, que torna os programas escritos em C compatíveis, independentemente da máquina utilizada.

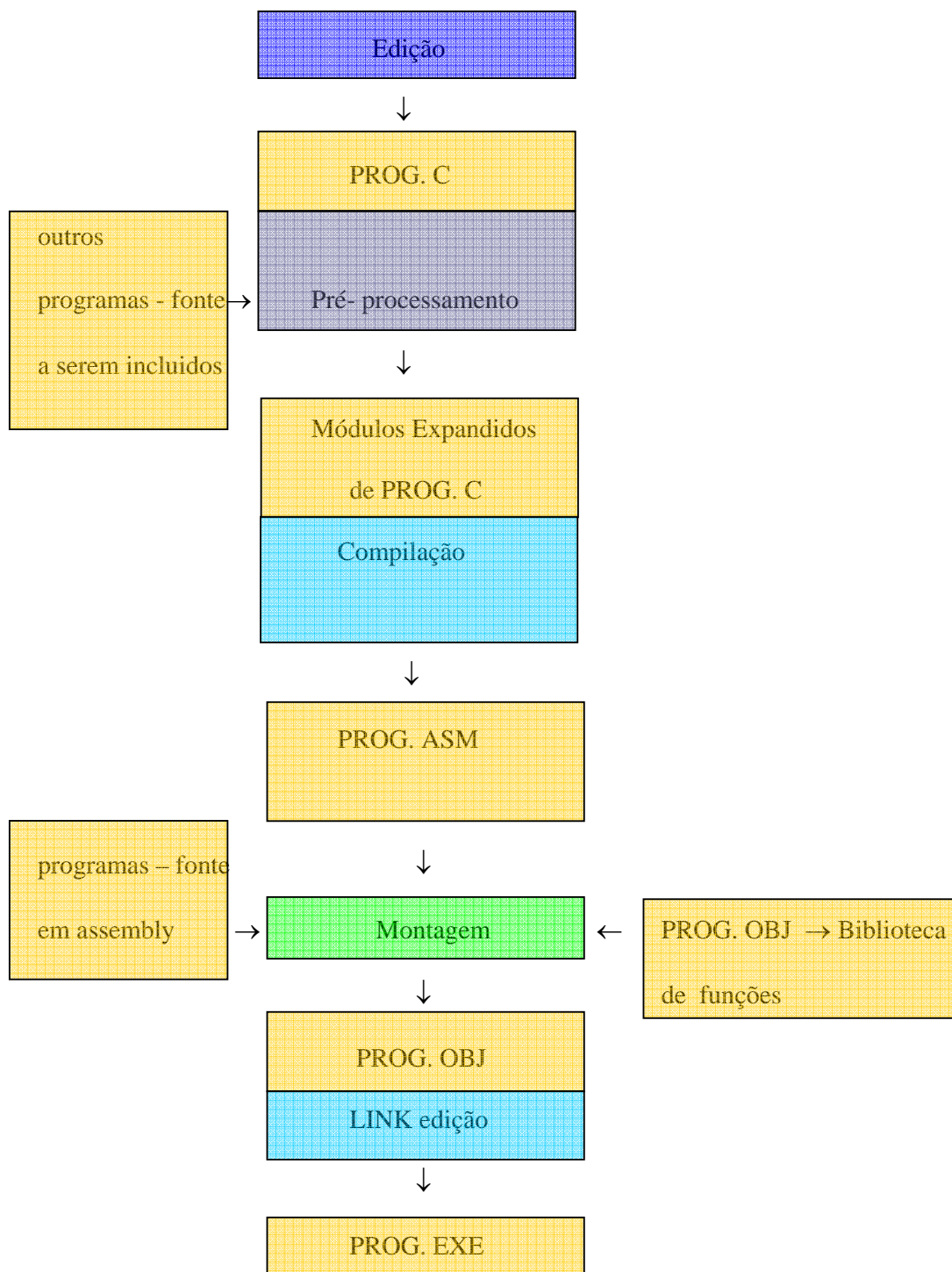
A linguagem C derivou-se do ALGOL 68, e foi projetada em 1972 nos laboratórios da BELL por Brian W. Kernigham e Dennis M. Ritchie para o sistema operacional UNIX.

## 2- CARACTERÍSTICAS

- Alto grau de portabilidade;
- É de uso geral, sendo eficiente tanto para programação de utilitários como para sistemas operacionais;
- Gera código executável compacto;
- É uma linguagem estruturada e modular.

## 3- COMPILAÇÃO E GERAÇÃO DE CÓDIGO EXECUTÁVEL

A linguagem C não dispõe de críticas demasiadas para os possíveis erros de execução, tais como estouro de dimensionamento de um vetor, divisão por zero e atribuição de valores com tipos diferentes. Além disso, o surgimento destes erros na maioria das vezes não interrompe o processamento, que continua normalmente.



**fig. 1.** Diagrama Compilação/ Montagem

---

## 4- ESTRUTURA UM DE PROGRAMA EM C

Todo programa em C contém diretivas de pré-processamento, definições, expressões e funções..

### 4.1- DIRETIVAS DE PRÉ - PROCESSAMENTO

Uma diretiva de pré-processamento é um comando dado ao C (é automaticamente executado na primeira etapa de compilação), para adicionar ou modificar o programa fonte. As diretivas mais comuns são *#define* , que substitui um texto por um identificador específico, e a diretiva *#include*, que inclui arquivos-fonte em C externos, no programa.

### 4.2- DECLARAÇÃO

A declaração estabelece os nomes e atributos de variáveis, funções e tipos usados nos programas. Variáveis Globais são declarações fora da função principal (*main* ). Variáveis locais são declaradas dentro da função avaliada.

### 4.3- DEFINIÇÃO

A definição estabelece o conteúdo de variáveis ou funções.

---

#### 4.4 - EXPRESSÃO

Uma expressão é a combinação de operadores e operandos gerando um valor simples.

#### 4.5- COMANDO

Comandos de Controle que controlam o fluxo de execução do programa.

#### 4.6- FUNÇÃO

Uma função é uma coleção de declarações, definição e comandos para execução de uma tarefa específica.

OBS: 1) A sintaxe da linguagem C diferencia letras maiúsculas de minúsculas. Assim, os termos “*main*” e “*MAIN*” são considerados diferentes. Uma prática padronizada é o uso de letras minúsculas para a formação de nomes das variáveis, funções e comandos, utilizando-se letras maiúsculas somente para a formação dos nomes de constantes simbólicas.

2) Comentário em C :

/ \* O comentário deve estar incluso entre os asteriscos e as barras \* /

3) Comentários me C++:

// Após as barras, todo o restante da linha será usado como espaço de comentário

Ambos podem ser usados em programas em C



## EXEMPLO 001

```

/*****
Exemplo Geral de um Programa em C
*****/

```

```

#include < stdio.h >          /* diretivas de pre-processamento */
                             /* include standart C header file */
#define PI 3.14              /* definição de constante simbólica */
float area;                  /* declaração de variável global */
long square (long r);        /* função prototipo */

```

```

void main ( ) {
    long radius_squared;      /* início da função principal e programa */
    long radius = 230;        /* declaração de variável local */
                              /* declaração e inicialização de variável */

    /* Corpo do programa */
    radius_squared = square(radius); /* passa o valor para a função */
    area = PI * radius_squared;    /* comando de atribuição */
    printf ( "area: %6.2f\n",area); /* imprime o resultado */
} /* Fim do corpo do Programa */

```

```

long square(long r) {
    long r_squared;           /* Rotulo da função */
                              /* variável local somente da função */
    /* Corpo da Função */
    r_squared = r * r;
    return(r_squared);        /* valor retornado */
} /* Fim da Função */

```

## 5. ENTRADA E SAÍDA

Uma das idéias mais interessantes da linguagem C é a inexistência de comandos específicos para entrada e saída de dados, sendo este processamento feito por funções definidas e armazenadas em uma biblioteca chamada padrão. Esta filosofia difere das demais linguagens que possuem comandos específicos para isso, como: BASIC, COBOL, PASCAL, etc.

Uma das vantagens da entrada e saída por função é que o usuário pode desenvolver sua própria biblioteca de entrada e saída, o que dará maior flexibilidade à programação.

---

## 6. VARIÁVEIS E CONSTANTES

### 6.1. NOMES DE VARIÁVEIS

Todo nome de variável deve iniciar com um caracter alfanumérico ou sublinha (\_).

Ex.: nome

\_nome

Pode possuir qualquer tamanho mas só será reconhecido os oitos (8) primeiros.

Ex.: resultado\_geral -> resultad

resultado\_parcial -> resultad

OBS: Alguns compiladores reconhecem apenas 6 e outras até 32 (Turbo C)

### 6.2. TIPOS DE DADOS

#### 6.2.1. OS TIPOS DE VARIÁVEIS EM C SÃO:

- 1) *char* - Variável que contém um caractere ASCII e que ocupam somente um byte (-128 a 127);
- 2) *int* - Valores numéricos inteiros que podem ser positivos ou negativos. Armazenado em quatro bytes nos compiladores atuais, representam valores entre -2.147.483.647 a 2.147.483.648;
- 3) *short* - Valores inteiros, “curtos”. Normalmente têm o mesmo tamanho dos inteiros, porém existem implementação onde são armazenados em um número menor de bytes para representar valores menores;
- 4) *long* - Inteiro “longo” eram utilizados quando era necessário representar valores inteiros muito grandes, que não podiam ser armazenados em variáveis do tipo *int*(*represetados antigamente por 2 bytes*). São armazenados normalmente em quatro bytes e representam valores entre -2.147.483.647 a 2.147.483.648;

---

5) *float* - Números reais em ponto flutuante. Assim como o tipo *long*, são armazenados em 4 bytes.

6) *double* - Valores numéricos reais como o *float*, porém armazenados em oito (8) bytes e permitem maior precisão na representação dos dados.

OBS: Os valores inteiros e caracteres, podem assumir apenas valores positivos, utilizando-se o adjetivo *unsigned*.

Ex: *unsigned char* -> 1 byte 0 a 255

*unsigned int* -> 4 bytes 0 a 4.294.967.294

*unsigned long* -> 4 bytes 0 a 4.294.967.294

## 6.2.2 OS TIPOS DE CONSTANTES EM C SÃO:

a) Caracteres - Constantes caracteres ficam entre aspas simples (') e são representadas pelo valor ASCII correspondente.

Ex: `'a'`, `'A'`, `'*'`, `'2'`, `'|'`

Outros tipos de caracteres:

`'\0'` - nulo- caracter indicador de byte zero (todos os oito (8) bits zerados). É utilizado para indicar o fim de uma cadeia de caracteres;

`'\n'` - newline- passa para a próxima linha do vídeo ou da impressora;

`'\t'` - tabulação- pula para a próxima coluna de tabulação no vídeo ou na impressora;

`'\\'` - barra invertida;

`'\''` - plic;

`'\xnn'` - representação de um byte em base hexadecimal. Neste caso, o mesmo caracter “escape”, será `'\x1b'`.

b) Cadeia de caracteres:

Ex: “Cadeia”

“linguagem c”

“pula 1 linha\npula 2 linhas\n\npulou”

#### c) Inteiro

Ex: -32.768 e 32767

#### d) Hexadecimal

Ex: 0 x 41, 0xffff, 0xa0, 0xC1

#### e) Ponto Flutuante

Ex: 3.14 ; 2.71 ; -5423.7265 ; 1. ; 3.286E2

#### EXEMPLO 002

\*\*\*\*\*

#### Exemplo de Representação de Cadeia de Caracteres

\*\*\*\*\*/

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

```
void main ( ) {
```

```
    printf("cadeia\n");
```

```
/* pula 1 linha (\n) */
```

```
    printf("linguagem C");
```

```
/* não pula linha */
```

```
    printf("Pula 1 linha\nPula 2 linhas\n\nPulo!!\n");
```

```
    printf("\x01\n");
```

```
/* representação hexadecimal */
```

```
}
```

## 7. DECLARAÇÃO E INICIALIZAÇÃO DE VARIÁVEIS

Para declarar uma variável basta seu tipo e seu nome na forma:

```
<tipo> <var1, var2, var3, ...>;
```

**EXEMPLO 003**

```

/*****
    Exemplo de Representação Numérica
*****/
int numero, valor;
char resp;

void main( ) {
    /* declaração */
    int aplic;
    char carac;
    float rel;

    /* inicialização */
    aplic = 2;
    carac = `A`;
    valor = 3;
    resp = `B`;
    rel = 333.22;

    /* impressão dos valores */
    printf ("aplic = %d\n",aplic);
    printf ("carac = %c\n", carac);
    printf ("valor = %d\n resp = %c\n rel = %3.2f\n",valor,resp,rel);
}

```

Inicializando as variáveis durante a declaração das mesmas:

**EXEMPLO 004**

```

/*****
    Inicializando Variáveis
*****/
#include <stdio.h>
int aplica = 10000;
char carac_atual = `$`;

void main ( ) {
    printf("Aplicação = %d\n\n", aplica);
    printf("Caracter = %c\n",Carac_atual);
}

```

**8. OPERADORES****8.1. OPERADORES DE ATRIBUIÇÃO SIMPLES**

a = 2 ( linguagem BASIC e FORTRAN)

---

$a := 2$  ( linguagem PASCAL)

Em C, o sinal é tratado como um operador, e não como um comando. Sendo um operador, o “=” termina por gerar um resultado, que é o próprio valor atribuído.

Ex. :  $a = 2;$  equivale a  $a \leftarrow 2$

$b = a = 2;$  equivale a  $a \leftarrow 2$  e  $b \leftarrow a$

## 8.2. OPERADORES ARITMÉTICOS

Soma (+)

Subtração (-)

Multiplicação (\*)

Divisão (/)

Resto da divisão inteira (%)

Ex:  $7 \% 2 \rightarrow 1$

$100 \% 8 \rightarrow 4$

Quando descrevemos um programa, é comum o uso da forma

$i = i + 1$                        $i = i - 1$

Em C podemos fazer esta operação utilizando somente um operador.

$i++$  ou  $++i$                        $i--$  ou  $--i$

Temos quatro formas de utilizar o incremento e o decremento:

$i++$  - pós-incremento; incrementa a variável de uma unidade e retorna como resultado o valor anterior ao incremento;

$++i$  - pré-incremento; incrementa a variável e retorna o resultado incrementado;

$i--$  - pós-decremento; decrementa a variável de uma unidade e retorna o valor anterior ao decremento;

$--i$  - pré-decremento ; decrementa e retorna o valor decrementado.

**EXEMPLO 005**

/\*\*\*\*\*

Exemplo de Pós-incremento e pré-incremento

\*\*\*\*\*/

#include &lt;stdio.h&gt;

int a, b;

void main( ) {

/\* Exemplo de Pós-incremento \*/

b = 10;

a = b++;

printf("a = %d\n b = %d\n\n", a, b);

/\* Exemplo de Pré-incremento \*/

b = 10;

a = ++b;

printf("a = %d\n b = %d\n\n", a, b);

}

**8.3. OPERADORES RELACIONAIS**

Quando queremos estabelecer uma comparação entre dois valores para tomar uma decisão, utilizamos as operações relacionais.

&gt; maior

&lt; menor

&gt;= maior ou igual

&lt;= menor ou igual

== igual

!= diferente (não igual)

OBS: Quando elaboramos programa, devemos tomar um cuidado especial para não confundirmos as formas:

a == b (Comparação)

a = b (Atribuição)

**EXEMPLO 006**

```
/******  
Exemplo de Operadores Relacionais  
******/  
#include <stdio.h>  
int a,  
    b  
void main ( ) {  
    b = 10;  
    a = 2;  
    if(a == b)  
        puts("iguais");  
    else  
        puts("diferentes");  
}
```

**8.4. OPERADORES LÓGICOS**

Em C temos três operadores lógicos, que utilizam normalmente operações relacionais como operadores.

&& (E)

|| (OU)

! (NÃO)

**EXEMPLO 007**

```
/******  
Exemplo de Operadores Relacionais  
******/  
#include <stdio.h>  
int a, b;  
void main ( ) {  
    a = 2;  
    b = 3;  
    if ( (a == 2) && (b == 3) )  
        printf("a = 2 e b = 3\n");  
    if ( (a == 2) || (b == 3) )  
        printf("a = 2 ou b = 3 ou ambos\n");  
    if (a != 3)  
        puts("a e diferente de 3\n");  
}
```



## 8.5. OPERADORES LÓGICOS EM BINÁRIO

Seus operadores não são tratados como verdadeiro e falso, mas sim como bytes onde as operações são feitas em nível de seus bits.

a) & (E)

a = 0x23F    00000010 00111111

b = 0xF3AC   11110011 10101100

c = a & b    00000010 00101100   0x022C

b) | (OU)

c = a | b    11110011 10111111   0xF3BF

c) ~ ( Complemento)

O operador unário ~ completa o seu operando, ou seja, transforma todos os bits 0 em bits 1 e vice-versa.

É interessante notar que podemos tornar um número negativo, fazendo o complemento de dois.

$$a = -b \Rightarrow a = \sim b + 1$$

↑

complemento a 1

```

EXEMPLO 008
/*****
    Exemplo de Operadores Binários
*****/
#include <stdio.h>
int a = 0x23f, b = 0xf3ac, c, d, f;
void main() {

    c = a & b;
    d = a ! b;
    f = ~a;
    printf("c = %x\nd = %x\nf = %x\n",c , d, f);
}

```

## 8.6. OPERADORES DE DESLOCAMENTO DE BITS (SHIFT)

Desloca os bits para a esquerda ou para a direita

>> ( direita )

<< (esquerda)

```
Ex.: a = 0xF66F    11110110    01101111

a << 2            11011001    10111100    0xD9BC

a >> 3            00011011    00110111    0x1B37
```

O deslocamento permite a multiplicação e a divisão do número por potência de 2.

$a = a * 16$        $a = a << 4$     ( $16 = 2^{**4}$ )

### EXEMPLO 009

\*\*\*\*\*

#### Exemplo de Deslocamento de Bits

\*\*\*\*\*/

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

```
int a = 0x00ff;
```

```
void main( ) {
```

```
    a <<= 2; /* ou a = a << 2 */
```

```
    printf("A = %x\n",a);
```

```
    a = a >> 2;
```

```
    printf("A = %x\n",a);
```

```
}
```

## 8.7. OPERADOR CONDICIONAL "? " "

```
a = 2;
b = 3;
z = (a > b) ? a : b;
```

```
ou
a = 2;
b = 3;
if ( a > b )
    z = a;
else
    z = b;
```

**EXEMPLO 010**

```

/*****

```

**Exemplo Operador Condicional**

```

*****/

```

```

#include <stdio.h>

```

```

#define ABS (a > 0) ? a : (-a)

```

```

int a;

```

```

void main( )

```

```

{

```

```

    a = -4;

```

```

    printf("a = %d\n",a);

```

```

    a = ABS;

```

```

    printf( "a = %d\n",a);

```

```

}

```

**8.8. OPERADOR TAMANHO DE VARIÁVEL**

```

sizeof( )

```

Retorna o número de bytes ocupados pela variável.

**EXEMPLO 011**

```

/*****

```

**Exemplo de Operador de Tamanho de Variável**

```

*****/

```

```

#include <stdio.h>

```

```

void main( ) {

```

```

    char          a;

```

```

    int           b;

```

```

    short         c;

```

```

    long int      d;

```

```

    unsined int   e;

```

```

    unsigned char h;

```

```

    float         f;

```

```

    double        g;

```

```

    puts("Número de bytes ocupados pelas variáveis");

```

```

    puts("===== ");

```

```

    puts ("tipo          - No. de bytes\n");

```

```

    printf("char          - %d\n",sizeof(a));

```

```

    printf("int           - %d\n",sizeof(b));

```

```

    printf("short         - %d\n",sizeof(c));

```

```

    printf("long int      - %d\n",sizeof(d));

```

```

    printf("unsined int   - %d\n",sizeof(e));

```

```

    printf("unsigned char - %d\n",sizeof(h));

```

```

    printf("float         - %d\n",sizeof(f));

```

```

    printf("double        - %d\n",sizeof(g));

```

```

}

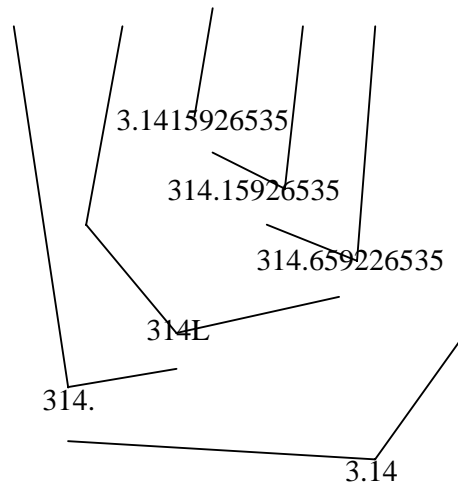
```

## 8.9. OPERADOR DE CONVERSÃO AUTOMÁTICA

Em C, é possível converter tipos de dados utilizando-se o operador (*tipo*), que transfere o valor no tipo especificado.

```
float valor = 3.1415926535;
```

```
valor = (float) ((long) (valor * 100. + 0.5)) / 100.;
```



## 9.0. CONTROLE DE FLUXO

### 9.1. COMANDO IF

Em C, a avaliação de expressão é *semiboleana*, quer dizer, uma expressão será “verdadeira” se, quando avaliada, produza um resultado diferente de zero, e “falso” se seu resultado for igual a zero.

Sintaxe

```
if (expressão)  ou  if (expressão)
    comando
                else
                    comando
```

**EXEMPLO 012**

```
/******  
Exemplo de estrutura IF  
*****/  
#include <stdio.h>  
int a, b, z;  
  
void main( ) {  
  
    b = 2;  
    a = 10;  
    if (a > 0)  
        printf ("valor positivo\n");  
    if (a < b){  
        z = a;  
        printf("A e menor que B\n Z = %d", z);  
    }  
    else {  
        z = b;  
        printf("B e menor que A\n Z = %d", z);  
    }  
}
```

**9.2. COMANDO SWITCH**

A linguagem C provê o programador de um comando que possibilita a tomada de múltiplas decisões baseadas em uma expressão.

Sintaxe

```
switch (expressão)  
{  
  
    case constante : comandos  
    case constante : comandos  
    .  
    case constante : comandos  
    default : comandos  
}
```

**EXEMPLO 013**

```

/*****
    Exemplo de Comando Switch
*****/
#include <stdio.h>
    char    ch;
    char    resp1[] = "opção escolhida foi a 1",
    resp2[] = "opção escolhida foi a 2",
    resp3[] = "opção escolhida foi a 3",
    resp4[] = "opção escolhida foi a default";
void main( ) {

    puts("Entre com um número <1,2 ou 3>:");
    ch = getchar ( );
    switch(ch) {
        case `1`: printf("%s\n", resp1);
                    break;
        case `2`: printf("%s\n", resp2);
                    break;
        case `3`: printf("%s\n", resp3);
                    break;
        default : printf("%s\n", resp4);
    }
}

```

**9.3. COMANDO WHILE**

O comando *while* é utilizado para a execução de um bloco de comandos enquanto uma condição lógica for “verdadeira”.

Sintaxe

While (expressão)

comando

**EXEMPLO 014**

```
/******  
Exemplo de Comando While  
*****/  
#include <stdio.h>  
  
int a = 10,  
    b = 0;  
void main( ) {  
  
    while ((a != 0) || (b != 10)){  
        printf("A = %d B = %d\n",a, b.);  
        a --;  
        b ++;  
    }  
}
```

**9.4. COMANDO FOR**

O *for* existe em uma forma bem parecida com a forma convencional, embora tenha uma filosofia elegante que lhe permite construções bem mais versáteis.

Sintaxe:

```
for (expressão1 ; expressão2 ; expressão3 )  
    comando
```

onde

expressão1 - é zero ou mais expressões separadas por vírgulas que serão executadas antes da interação do bloco de comandos associado.

expressão2 - é geralmente um teste lógico, no caso de ser avaliado “falso”, indica o fim da interação.

expressão3 - é zero ou mais expressões separadas por vírgulas que serão executadas no momento do término de uma interação.

**EXEMPLO 015**

```
/*  
*****  
Exemplo de Comando For  
*****  
*/  
#include <stdio.h>  
  
int i, val[10];  
  
void main( ) {  
  
    for (i = 1; i <= 10; i++){  
        val[i] = i;  
        printf("valor de VAL[ %d ] = %d\n",i,val[i]);  
    }  
}
```

**9.5. COMANDO DO-WHILE**

O do-while, mais conhecido em outras linguagens por uma variante com o nome de repeat-until, é outra forma estruturada para controle de fluxo.

Sintaxe:

```
do {  
  
    Comando  
  
} while (condição);
```

A execução ocorre do seguinte modo:

- 1) O bloco de comando é executado;
- 2) A expressão é avaliada;
- 3) Caso o valor seja “verdadeiro”, o fluxo de execução volta para o item 1;
- 4) Caso o valor seja “falso”, o “loop” terminará, e o processamento segue a partir do primeiro comando após o do-while.



**EXEMPLO 016**

```
/******  
Exemplo de Comando Do-While  
*****?  
#include <stdio.h>  
    int i;  
void main( ) {  
  
    i = 1;  
    do{  
        printf("i = %d\n",i);  
        i++;  
    }while(i < 10);  
}
```

**9.6. COMANDOS BREAK E CONTINUE**

Algumas vezes torna-se necessário o controle de execução de uma iteração independentemente do teste lógico efetuado.

O *break*, quando utilizado dentro do bloco de comandos associados aos comandos de “loop”, com o *for*, faz com que o “loop” seja imediatamente interrompido, transferindo o fluxo de execução para o próximo comando fora do “loop”.

O *continue*, desvia o fluxo de execução para a próxima iteração dos comandos.

**EXEMPLO 017**

```
/******  
Exemplo de Comando Continue  
*****/  
#include <stdio.h>  
    int i;  
void main( ) {  
  
    i = 1;  
    do{  
        i++;  
        if ((i % 2) == 0 )  
            continue;  
        printf("i = %d\n",i);  
    }while(i < 10);  
}
```

---

## 10. VETORES E APONTADORES

### 10.1. DECLARAÇÃO DE VETORES

Em C, os vetores são declarados da seguinte forma:

tipo nome [tamanho];

send

tipo - pode ser qualquer tipo de dado, simples ou composto.

Ex.: *int, char, etc;*

nome - é o nome que irá representar o conjunto de variáveis;

tamanho - é o número de elementos do vetor.

Ex.:

char mensagem[10];

int n\_ocorre[20];

int n\_tes[2] = {1,2};

neste último caso teremos os seguintes elementos:

n\_tes[0] = 1

n\_tes[1] = 2

char men[ ] = "abc";

neste caso teremos os seguintes valores:

men[0] = 'a'

men[1] = 'b'

men[2] = 'c'

men[3] = '\0'

char men[5] = "abc ";

neste caso teremos os seguintes valores:

men[0] = 'a'

men[1] = 'b'

```
men[2] = `c`
```

```
men[3] = `0`
```

```
men[4] = `0`
```

```
int num[ ] = {1,2,3,4};
```

neste caso teremos os seguintes elementos:

```
num[0] = 1
```

```
num[1] = 2
```

```
num[2] = 3
```

```
num[3] = 4
```

```
int num[4] = {1,2};
```

```
num[0] = 1
```

```
num[1] = 2
```

```
num[2] = 0
```

```
num[3] = 0
```

#### EXEMPLO 018

```
/******
```

```
Exemplo de Vetor
```

```
*****/
```

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

```
int i, quad[10];
```

```
void main( ) {
```

```
    for (i = 0; i < 10; i++)
```

```
        quad[i] = i * i;
```

```
    for (i = 0; i < 10; i++)
```

```
        printf("quadrado de %d = %d\n", i, quad[i]);
```

```
}
```

## 10.2.VETORES MULTIDIMENSIONAIS

Em C, podemos simular vetores com qualquer número de dimensões, sem nos preocuparmos com a forma como isto é feito.

Ex:                `char nomes[2][10] = { "joão vai", "maria não" };  
                    int matriz[2][3] = { {1,2,3},{4,5,6} };`

### EXEMPLO 019

```
/*  
*****  
Exemplo de Vetor Multidimensional  
*****  
*/  
#include <stdio.h>  
int i;  
char nome[5][10] =  
    {  
        "Carmen",  
        "Ricardo",  
        "Alexandre",  
        "Ana Paula",  
        "Eduardo"  
    };  
void main( ) {  
  
    for (i = 0; i < 5; i++)  
        printf("nome : %s\n", nome[i]);  
}
```

## 10.3. APONTADORES

Em C, podemos armazenar em uma variável o endereço de uma outra variável. Para isso, devemos declarar esta variável como um apontador.

Ex.: `char *pc;`

Na maneira acima, declaramos que *pc* é um apontador para uma variável do tipo *char*.

Em C, existe uma maneira para referenciar diretamente os endereços das variáveis, através do operador `&`.

Ex.:

```
void main( ) {
    int x, *px;
    .
    .
    px = &x;
}
```

Após a atribuição, o conteúdo do apontador *px* será o endereço da variável *x*.

```
EXEMPLO 020
/*****
Exemplo do Uso de ponteiros
*****/
#include <stdio.h>
void main( ) {

    int x = 10, y, *px;

    px = &x;
    y = *px + 1;
    printf("y = %d\n",y);
}
```

## 11. FUNÇÕES

Podemos resumir função, em um conceito mais moderno, como sendo um trecho independente de programa com atribuições definidas.

A função, uma vez desenvolvida e testada, passa a ser completamente transparente ao usuário, que só necessita saber sua utilidade, entrada e saída.

Sintaxe:

Qualquer função em C tem o seguinte formato:

```
tipo de dado de retorno nome (declaração dos parâmetros de passagem) {

    declaração de variáveis locais

    comandos

}
```

**EXEMPLO 021**

/\*\*\*\*\*

**Exemplo do Uso de Funções**

\*\*\*\*\*/

#include &lt;stdio.h&gt;

void pot(int x, int y);

void main( ){

int a,

b;

a = 10;

b = 3;

pot(a,b);

}

/\* Início da Função\*/

void pot(int x, int y){

int i,

potencia = 1;

for (i = 0; i &lt; y; i++)

potencia \*= x; /\* igual a potência = potência \* x \*/

printf("%d\n",potencia);

}

**EXEMPLO 022**

/\*\*\*\*\*

**Exemplo 2 do Uso de Funções**

\*\*\*\*\*/

#include &lt;stdio.h&gt;

void pont(long, long);

long a, b;

void main( ){

a = 10;

b = 3;

pot(a,b);

}

```

/* Início da Função */
void pot(long x, long y){
    int i,
    potencia = 1;
    for (i = 0; i < y; i++)
        potencia *= x; /* igual a potência = potência * x */
    printf("%d\n",potência);
}

```

Em C uma função pode retornar apenas um valor ou nenhum. Este retorno é feito pelo comando *return*.

#### EXEMPLO 023

```

/*****

```

##### Exemplo 3 do Uso de Funções

```

*****/

```

```

#include <stdio.h>

```

```

int a, b;

```

```

long pot(int, int), resposta;

```

```

void main( )

```

```

{

```

```

    a = 10, b = 3;

```

```

    resposta = pot(a, b);

```

```

    print("%d\n",resposta);

```

```

}

```

```

long pot (int x, int y)/* Início da Função */

```

```

{

```

```

    long i,

```

```

    potencia = 1;

```

```

    for (i = 0; i < y; i++)

```

```

        potencia *= x; /* igual a potência = potência * x */

```

```

    return(potencia);

```

```

}

```

Existe uma função particular chamada *main()*, que normalmente não recebe parâmetros de entrada, mas isso pode ocorrer. A característica principal, que a difere das demais funções, está no uso de dois parâmetro, *argv* e *argc*.

Quando você executa programas no ambiente do interpretador de comando do DOS (COMMAND.COM), você pode especificar argumentos a serem passados para o programa que será executado.

---

Ex. copy a:\meu.doc c:\

No exemplo, o programa copy recebe dois path como argumento, (a:\meu.doc) e (c:\). A função main do programa copy ,seria:

```
void main(argc,argv)

    int argc;

    char *argv[];
```

onde argc contém o número de parâmetros passados como argumento. O contador de argumento inclui também o nome do próprio programa. Já argv[] é um vetor de strings, que contém as strings passadas como parâmetros.

Ex:

```
copy a:\meu.doc c:\

argc = 3

argv[0] = "copy"

argv[1] = "a:\meu.doc"

argv[2] = "c:\"

argv[3] = NULL
```



**EXEMPLO 024**

```

/*****
Exemplo usando argv e argc
*****/
#include <stdio.h>

void main(int argc, char *argv[]) {

    int i;
    printf("argc = %d\n",argc);
    printf("\n");
    for (i= 0; i < argc ;++i){
        printf("argv[%d] -> \"%s\"\n",i, argv[i]);
    }
    printf("argv[%d] -> NULL\n",i);
    printf("\n");
}

```

**12. ESTRUTURAS****12.1. SINTAXE**

Normalmente a estrutura recebe um nome que passa a identificar um novo tipo de dado composto , correspondente ao que foi definido.

Ex:

```

struct funcionario {
    char   nome[30];
    char   endereco[40];
    char   conjuge[30];
    int    n_depen;
    char   cargo;
    float  salario;
    char   data_adm[10];
};

```

Após ter sido efetuada a declaração, um novo tipo passa a existir. Neste instante, basta declarar as variáveis para o tipo definido.

Ex:

```

struct funcionario    analista, programador ;

```

Este exemplo indica ao compilador que reserve espaço de memória para as variáveis criadas com o tipo *struct funcionario*.

Uma outra forma de declaração está em definir as variáveis diretamente na definição do tipo.

Ex:

```
struct funcionário {
    char   nome[30];
    char   endereco[40];
    char   conjugue[30];
    int    n_depen;
    char   cargo;
    float  salario;
    char   data_adm[10];
} analista, programador;
```

Esta prática, no entanto, não é recomendada, já que peca pelo aspecto documental.

Para nos referirmos ao campos das variáveis basta referenciarmos a variável e seu campo.

Ex:

```
analista.n_depen = 4;
programador.salario = 800.00;
```

#### EXEMPLO 025

\*\*\*\*\*

Exemplo usando estrutura

\*\*\*\*\*

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

```
struct reg {
    int    codigo;
    struct valor;
};
```

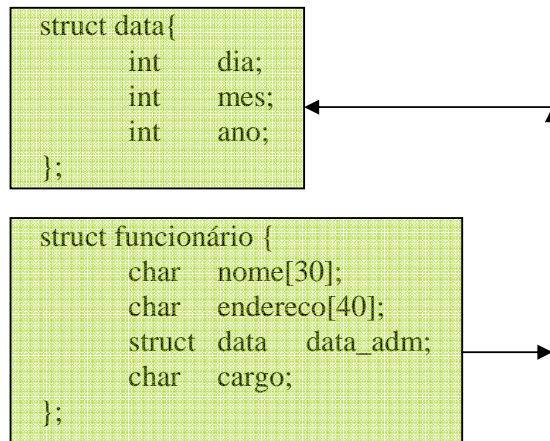
```
void main() {
    struct reg tabela[5];
    int i,c;
    float v;

    for(i=0;i<=4;i++){
        printf("codigo:");scanf("%d",&tabela[i].codigo);fflush(stdin);
        printf("valor:");scanf("%f",&tabela[i].tabela.valor);fflush(stdin);
    }
    for(i=0;i<=4;i++)
        printf("codigo: %d valor: %8.2f\n",tabela[i].codigo,tabela[i].valor);
}
```

## 12.2. ESTRUTURA COMPOSTA

Neste caso, teríamos uma estrutura dentro da outra, o que é uma prática muito comum no tratamento de dados em computação.

Ex:



Agora, para nos referenciarmos ao campo ano da estrutura data\_adm, utilizaríamos:

analista.data\_adm.ano

### EXEMPLO 026

\*\*\*\*\*

Exemplo usando estrutura composta

\*\*\*\*\*/

#include <stdio.h>

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

struct reg {int codigo; struct data data\_c; };

void main() {

struct reg tabela[5]; int i;

for(i=0;i<=4;i++) {

printf("codigo:"); scanf("%d",&tabela[i].codigo); fflush(stdin);

printf("dia:"); scanf("%d",&tabela[i].data\_c.dia); fflush(stdin);

printf("mes:"); scanf("%d",&tabela[i].data\_c.mes); fflush(stdin);

printf("ano:"); scanf("%d",&tabela[i].data\_c.ano); fflush(stdin);

}

for(i=0;i<=4;i++)

printf("codigo: %d data: %d/%d/%d\n",tabela[i].codigo,  
tabela[i].data\_c.dia,tabela[i].data\_c.mes, tabela[i].data\_c.ano);

}

## 12.3. PONTEIROS, FUNÇÕES E ESTRUTURAS

Como visto anteriormente, podemos utilizar ponteiros para variáveis dos tipos int, char, etc. Como uma estrutura é do tipo de dado como os outros, é possível a utilização de apontadores para ela.

Ex:

```
struct data *pt_data;
```

O ponteiro pt\_data, é capaz de apontar para o primeiro byte de uma variável do tipo data, permitindo, então, que seus campos sejam acessados isoladamente.

Ex: pt\_data->campo

### EXEMPLO 027

```
/******
```

Exemplo usando estrutura, ponteiro e função

```
*****/
```

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

```
struct movimento {
```

```
    long  numero;
```

```
    double valor;
```

```
    char  codigo;
```

```
};
```

```
void novo_mov(struct movimento *);
```

```
void main(){
```

```
    struct movimento reg_mov;
```

```
    novo_mov(&reg_mov);
```

```
    printf("Numero: %d\n", reg_mov.numero);
```

```
    printf("Valor: %f\n", reg_mov.valor);
```

```
    printf("Código: %c\n", reg_mov.codigo);
```

```
}
```

```
void novo_mov(struct movimento *pt_mov){
```

```
    pt_mov->numero = 9999;
```

```
    pt_mov->valor = 1000.00;
```

```
    pt_mov->codigo = 'c';
```

```
}
```

## 13. ARQUIVOS EM C

### 13.1. INTRODUÇÃO

A biblioteca C oferece três formas ou categorias para manipulação de arquivos, que são:

a) Alto Nível

---

São chamadas de *buffered stream* (rotinas de saída com área intermediária de memória) porque o buffer se interpõe entre o arquivo e o programa.

b) Médio Nível

São chamados de *no buffered*, porque o programa acessa o arquivo diretamente (através do sistema operacional SO). Não será abordada nesse estudo.

c) Baixo Nível

Usam níveis não portáteis do BIOS, não será abordada nesse estudo.

As rotinas de médio e alto nível possuem dois modos de manipulação de dados, o modo texto e o modo binário. O modo texto é usado como arquivo texto, ou arquivos que armazenam texto no formato ASCII (prog. fontes, etc). O modo binário, é usado em arquivos que armazenam dados na forma binária (imagens, sons, prog. executáveis, etc). No modo texto, o caractere Ctrl+z (um byte contendo o valor 0x1A) é a marca de final de arquivo. Nos arquivos binários Ctrl+z é uma parte **legal (dado)** do arquivo, o SO é quem guarda o tamanho do arquivo (no diretório).

## 13.2. ENTRADA E SAÍDA EM ALTO NÍVEL

Todas as funções de i/o em alto nível necessitam que se inicie o programa com *#include <stdio.h>*. Esse arquivo de header, contém as definições para *FILE*, o tipo que você usa para manipular arquivos.

Ex:

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

```
FILE *fp;
```

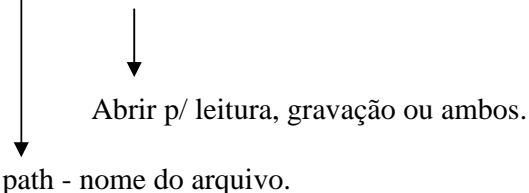
Sendo \*fp é um ponteiro para a estrutura do tipo FILE.

## 13.3. ABRINDO UM ARQUIVO

Antes de você acessar um arquivo para leitura , gravação ou ambos, você deve inicialmente abrir o arquivo. Para as rotinas de i/o com buffers, você deve usar um ponteiro para o arquivo e abrir o mesmo com a função *fopen( )*.

Ex:

```
fp = fopen(nomearq, operação);
```



path - nome do arquivo.

Abrir p/ leitura, gravação ou ambos.

Tabela de operação

modo	Descrição
r	Abre para leitura apenas. O arquivo deve existir.
w	Abre para gravação apenas. Cria o arquivo se não existir e apaga antigo se já existir e cria um novo.
a	Abre para inclusão. Grava novos dados no final do arquivo.
r+	Abre para leitura/gravação. O arquivo deve existir.
w+	Abre para leitura/gravação. Cria o arquivo se não existir e apaga se já existir criando um novo.
a+	Abre para leitura/gravação, Grava novos dados no final do arquivo. Cria um novo arquivo se não existir.

OBS: Cada arquivo aberto requer um ponteiro para si.

Ex:

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

```
.  
. FILE *fp_in, *fp_out;  
. .  
fp_in = fopen("teste.txt", "r");  
fp_out = fopen("teste2.txt", "w");
```

### 13.4. LENDO UM CARACTERE DO ARQUIVO

O `fgetc( )` é uma versão para arquivo da função `getchar( )`, ela retorna o próximo caractere lido do arquivo. Usando uma variável do tipo inteiro podemos identificar o final de arquivo facilmente.

Ex:

```
if((ch= fgetc(fp)) == EOF) {  
    /* Se for o final do arquivo */  
}
```

A chamada para `fopen( )` com o operador "rb" :

```
fp = fopen(argv[1], "rb")
```

é uma declaração específica do PC. O "rb" habilita a abertura de arquivo binários.

Tabela

modo	Descrição
t	text mode - converte carriage return/line feed em um line feed na leitura e o contrário na gravação. Ctrl+z é a marca de final de arquivo.
b	binary mode - Elimina as conversões. O sistema operacional guarda o tamanho do arquivo.

### 13.5. FECHANDO ARQUIVOS

O MS\_DOS suporta até 20 arquivos abertos simultaneamente. Você deve fechar cada arquivo aberto. Fechar um arquivo aberto para escrita significa atualizar a entrada do diretório para o arquivo e liberar o ponteiro de arquivo.

Ex:

```
if ( fclose(fp) == EOF) {

    /* Não habilitado p/ fechamento */

}
```

#### EXEMPLO 28

```

/*****
Abre um arquivo e busca por possíveis strings
*****/
#include <stdio.h> /* para FILE, BUFSIZ e EOF */
#include <ctype.h> /* para isprint() - verifica se caracter pode ser impresso */

void main (int argc, char *argv[]) {

    FILE *fp; char buf[BUFSIZ]; int ch, cont;
    if (argc != 2){
        fprintf(stderr,"use: file01 nomearq\n");
        exit(1);
    }
    if ((fp = fopen(argv[1],"rb")) == NULL){
        fprintf(stderr,"Não abriu %s\n",argv[1]);
        exit(1);
    }
    cont = 0;
    while ((ch = fgetc(fp)) != EOF){
        if (! isprint(ch) || cont >= (BUFSIZ - 1)){
            if (cont > 5){
                buf[cont] = 0;
                puts(buf);
            }
            cont = 0;
            continue;
        }
        buf[cont++] = ch;
    }
}

```



### 13.6. ACESSO RANDÔMICO EM ARQUIVOS

Aplicações mais sofisticadas, como busca em base de dados, vão necessitar de movimentações através do arquivo(uso do ponteiro de bytes do arquivo),partindo do princípio que esse arquivo seja uma sequência de bytes. A função `fseek( )` permite que você acesse vários elementos do arquivo, através da determinação da posição da próxima leitura ou gravação do arquivo.

Ex.:

```
fseek(fp, offset,origem);
```

↑ origem do deslocamento (veja OBS)

↑ deslocamento (diferença entre a posição atual e a nova posição)

↑ ponteiro para o arquivo

OBS: No arquivo de header *stdio.h*, existem as definições simbólicas para origem. Veja tabela abaixo:

tabela 1

0	SEEK_SET	início do arquivo
1	SEEK_CUR	posição corrente
2	SEEK_END	fim do arquivo

O deslocamento(offset), em bytes, diz à função `fseek( )` quanto deverá se deslocar no arquivo, deve ser um valor do tipo inteiro longo (*long*). A origem define a posição a partir da qual deverá haver o deslocamento(veja a tabela 1).

A função `fseek( )` retornará o valor longo ( -1L) caso não reposicione o ponteiro de bytes. Se a operação obtiver sucesso retornará a nova posição do ponteiro de bytes do arquivo, em relação ao início do arquivo( posição 0 = primeiro byte do arquivo) .

### 13.7. ENTRADA E SAÍDA DE DADOS EM LINHA(REGISTRO) EM ARQUIVOS

As funções `fgets( )`"file get string" e `fputs( )`"file put string", são similares a `gets( )` e `puts( )`. A função `fgets( )` lê linhas de texto do arquivo e `fputs( )` grava linhas de texto no arquivo.

Ex:

---

```

#include <stdio.h>
#define SIZE 512
.
.
FILE *fp_in, *fp_out;
char buf[SIZE];
.
.
/* abre fp_in para leitura e fp_out para gravação */
.
.
if ( fgets(buf , SIZE , fp_in) == NULL) {
    /* erro ou EOF */
}

/* a linha de texto está agora em buf */
.
.
if ( fputs( buf, fp_out) == EOF){          {
    /* erro na gravação */
}

```

onde

```
fgets(arg1, arg2, agr3);
```

arg1 - endereço do buffer de caracteres.

arg2 - número máximo de caracteres lidos para o buffer.

arg3 - ponteiro para o arquivo.

- fgets( ) lê o máximo de caracteres SIZE do buffer (buf) ou até o primeiro *newline* inclusive

e inclui um '\0'.

```
fputs( arg1, arg2 );
```

arg1 - endereço da string terminada com '\0'.

arg2 - ponteiro do arquivo de saída.

- fputs( ) grava uma string do buffer (buf) para fp\_out, juntamente com os *newline* da string.

**EXEMPLO 29**

```

/*****

```

Copia um arquivo, eliminando linhas em branco e principais espaços de linhas copiadas.

```

*****/

```

```

#include <stdio.h>    /* para FILE, BUFSIZ, NULL */

```

```

#include <ctype.h>    /* para isspace() */

```

```

void main(int argc, char *argv[]) {

```

```

    FILE *fp_in, *fp_out;

```

```

    char buf[BUFSIZ];

```

```

    char *cp;

```

```

    if (argc != 3){

```

```

        printf("use: ccopy arg_ent arg_sai\n");

```

```

        exit(1);

```

```

    {

```

```

        if ((fp_in = fopen(argv[1], "r")) == NULL){

```

```

            printf("Erro no arquivo %s de leitura\n", argv[1]);

```

```

            exit(1);

```

```

        }

```

```

        if ((fp_out = fopen(argv[2], "w")) == NULL){

```

```

            printf("Erro no arquivo %s de gravação\n", argv[2]);

```

```

            exit(1);

```

```

        }

```

```

        printf("Copiando e Comprimindo: %s -> %s ...", argv[1], argv[2]);

```

```

        while (fgets(buf, BUFSIZ, fp_in) != NULL){

```

```

            cp = buf;

```

```

            if (*cp == '\n') /*linha com brancos */

```

```

                continue;

```

```

            while(isspace(*cp)) /* função para identificar espaço em branco */

```

```

                ++cp;

```

```

            if (*cp == '\0') /*linha aberta */

```

```

                continue;

```

```

            if (fputs(cp, fp_out) == EOF){

```

```

                printf("\n Erro na gravação de %s. \n", argv[2]);

```

```

                exit(1);

```

```

            }

```

```

        }

```

```

        puts("Terminado\n");

```

```

    }

```

### 13.8. REDIRECIONAMENTO

Cada vez que um programa em C é executado, cria-se um ambiente no qual se tem acesso a três fluxos de dados:

- 1) Um de entrada (stdin), onde stdin significa entrada padrão de dados (normalmente o teclado);
- 2) Um de saída (stdout), onde stdout significa saída padrão de dados (normalmente o vídeo);
- 3) Outra de saída (stderr), onde stderr significa saída padrão de erros (normalmente o vídeo);

### 13.9. ARQUIVO FORMATADO

As funções de entrada e saída formatada para arquivos são similares as entradas e saídas padrão printf( ) e scanf( ) e chamam-se fprintf( ) e fscanf( ). São idênticas às primeiras, com uma exceção: Ambas necessitam de um ponteiro como primeiro argumento, indicando o endereço lógico do dispositivo( pode ser de entrada ou de saída).

Ex:

fprintf(fp\_out, controle, arg...);  
↓      ↗ mesmo da função printf( )  
Ponteiro para arquivo (endereço lógico do dispositivo)

fscanf(fp\_in, controle, endereço...);  
↓      ↗ mesmo da função scanf( )  
Ponteiro para arquivo (endereço lógico do dispositivo)

**EXEMPLO 30**

```

/*****
    Abre um arquivo e grava registros de dados
    com o nome e idade de alunos
*****/
#include <stdio.h>
#include <conio.h>

FILE *fp_out;
struct reg {
    char   nome[20];
    int    idade;
};
void main(){
    struct reg reg_pes;
    char op = 's';

    if ((fp_out = fopen("a:\\teste.dat","w")) == NULL){
        printf("\ns: Não abriu o arq.\n","a:\\teste.dat");
        exit(1);
    }
    while ((op != 'n') && (op != 'N')){
        clrscr();
        gotoxy(10,2);puts("Nome:");gets(reg_pes.nome);
        strncat(reg_pes.nome,"", (19 - strlen(reg_pes.nome)));
        gotoxy(10,4);puts("Idade:");scanf("%d",&reg_pes.idade);fflush(stdin);
        /* grava o registro no arq. no disco */
        fprintf(fp_out,"%s%d\n",reg_pes.nome,reg_pes.idade);
        printf("Continua(s/n): ",op = getch());fflush(stdin);
    }
    fclose(fp_out);
}

```

**EXEMPLO 31**

```

/*****
    Abre um arquivo para leitura de registros com
    dados contendo o nome e idade de alunos
*****/
#include <stdio.h>
#include <conio.h>

FILE *fp_out;
struct reg {
    char   nome[20];
    int    idade;
};
main(){
    struct reg reg_pes;
    char op = 's';
    char n; /* usado para eliminar o [cr/ln] (EOR) no final do registro */

    if ((fp_in = fopen("a:\\teste.dat","r")) == NULL){
        printf("\ns: Não abriu o arq.\n","a:\\teste.dat");
        exit(1);
    }
    while ((op != 'n') && (op != 'N')){
        clrscr();
        fgets(reg_pes.nome,20,fp_in);
        fscanf(fp_in,"%d", &reg_pes.idade);fflush(fp_in);
        fscanf(fp_in,"%c",n);fflush(fp_in); /* retira o EOR do registro lido */
        fprintf(stdout,"Nome: %s    Idade: %d",reg_pes.nome,reg_pes.idade);
        printf("\n Continua(s/n): "); op = getche();fflush(stdin);
    }
    fclose(fp_in);
}

```

**13.10. ENTRADA E SAÍDA DE DADOS EM BLOCOS OU REGISTRO**

Algumas vezes você necessitará gravar ou ler blocos de dados em byte, não importando se esses dados são linhas de textos ou uma estrutura de dados (registro). Para tanto duas funções estão disponíveis na biblioteca standard do "C". São elas, as funções `fread( )` e `fwrite( )`. Seu formato de uso está descrito abaixo:

Ex:

```
fread(buffer, size, count, fp_in);
```

↓      ↓      ↓      ↓      ponteiro para o arquivo  
          tamanho do bloco (em bytes)      número de blocos  
 ↓  
 endereço destino dos dados (origem fp\_in)

```
fwrite(buffer, size, count, fp_out);
```

↓      ↓      ↓      ↓      ponteiro para o arquivo  
          tamanho do bloco (em bytes)      número de blocos  
 ↓  
 endereço origem dos dados (destino fp\_out)

**EXEMPLO 32**

```
/******
```

Abre um arquivo e grava registros de dados  
com o nome e idade de alunos

```
*****/
```

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

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

```
FILE *fp_out;
```

```
struct reg {
    char nome[20];
    int idade;
};
```

```
main(){
```

```
    struct reg reg_pes;
```

```
    char op = 's';
```

```
    if ((fp_out = fopen("a:\\teste.dat","wb+")) == NULL){
```

```
        puts("\%s: Não abriu o arq.\n");
```

```
        exit(1);
```

```
    }
```

```
    while ((op != 'n') && (op != 'N')){
```

```
        clrscr();
```

```
        puts("Nome:"); gets(reg_pes.nome);
```

```
        puts("Idade:"); scanf("%d",&reg_pes.idade); fflush(stdin);
```

```
        /* grava o registro no arq. no disco */
```

```
        fwrite(&reg_pes,sizeof(reg_pes),1,fp_out);
```

```
        printf("Continua(s/n): ",op = getch()); fflush(stdin);
```

```
    }
```

```
    fclose(fp_out);
```

```
}
```

**EXEMPLO 33**

```

/*****
    Abre um arquivo para ler registros do arquivo
    com dados contendo o nome e idade de alunos
*****/
#include <stdio.h>
#include <conio.h>

FILE *fp_in;
struct reg {
    char   nome[20];
    int    idade;
};
main(){
    struct reg reg_pes;
    char op = 's';

    if ((fp_in = fopen("a:\\teste.dat","rb+")) == NULL){
        puts("\%s: Não abriu o arq.\n");
        exit(1);
    }
    while ((op != 'n') && (op != 'N')){
        clrscr();
        fread(&reg_pes,sizeof(reg_pes),1,fp_in);
        printf("Nome: %s    Idade: %d",reg_pes.nome,reg_pes.idade);
        printf("\n Continua(s/n): "); op = getche();fflush(stdin);
    }
    fclose(fp_in);
}

```

**EXERCÍCIO:**

1) Elaborar um programa em C para gerenciar uma agenda telefônica. O formato do registro de dados está descrito abaixo. Deverá ser possível incluir, consultar, alterar e excluir (lógica/física) dados da agenda. Deverá ser usado o método de PESQUISA SEQUENCIAL.



---

Arquivo de Dados

ST	Nome	Telefone
0	João da Silva	234-5670
0	Maria José	222-5678
1	Alberto Pereira	345-5543
0	João Fagundes	444-6778

Para a verificação da chave de pesquisa use a função em C, da biblioteca padrão

`strcmp( )` - compara duas cadeias de caracteres ( incluir *string.h* em seu programa)

### 13.11. ARRAY DINÂMICO

Usando as funções padrões da biblioteca do C, seu programa pode alocar memória, enquanto está sendo executado (Dinamicamente) e desta forma criar um *array dinâmico*.

As funções para a alocação dinâmica de memória, são as seguintes:

- `malloc( )` Alocação de memória;
- `calloc( )` Alocação de memória em blocos;
- `realloc( )` Realocação de blocos de memória;
- `free( )` Liberação de blocos de memória alocados.

Os tipos retornados por essas funções estão declarados no arquivo de header *malloc.h*

---

### 13.11.1. A função malloc( )

É a mais usada das funções. Ela recebe um argumento simples, que contém o número de bytes de memória a ser alocado, e retorna o endereço do primeiro byte dessa área. Se a função malloc( ) não puder alocar a área desejada, ela retorna um NULL.

Ex:

```
#include <stdio.h>
#include <malloc.h>
.
.
int *iptr;
unsigned int byte = 100;
.
.
if (( iptr = malloc(bytes)) == NULL)
    ↑ número de bytes desejados
    /* erro*/
```

### 13.11.2. A função free()

A função free( ), libera a memória alocada pela função malloc( ) ou calloc( ).

Ex:

```
free(iptr);
```

**EXEMPLO 34**

```

/*****
Exemplo do uso da função malloc(), para
alocação dinâmica de memória(array dinâmico)
*****/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>

int main(){

    char *str; /* ponteiro que receberá o endereço do bloco de memória alocado */

    /* Alocando memória para uma string */
    if((str = (char *) malloc (11)) == NULL) {
        puts("Não existe memória para o buffer");
        exit(1); /* termina a execução e retorna um código de erro
                  para o sistema operacional */
    }
    /* copia "Vamos ver!", para a área de memória alocada */
    strcpy(str,"Vamos ver!");
    /* imprime a cadeia de caracteres */
    printf("A string é %s\n",str);

    /* libera o bloco de memória */
    free(str);
    return(0);
}

```

A função `malloc()` retorna um ponteiro, que será ajustado através de um *type cast* (ex. `(char *)`), para o tipo de dado que você deseja alocar na memória. No exemplo, o retorno será um ponteiro para uma cadeia de caracteres.

### 13.11.3. A função `realloc()`

A função `realloc` copia pequenos ou grandes blocos de memória para uma outra área. Isto é, copia o conteúdo de uma área antiga de memória para uma nova, truncando se o novo tamanho for menor que o antigo e retornando o endereço da nova área.

Ex:

```

if (( iptr = realloc(iptr, byte*(cont +1))) == NULL) {
Ponteiro anterior ↑ |—————|
                                ↑ Tamanho em bytes
/* erro */
}

```

**EXEMPLO 35**

```

/*****

```

Exemplo do uso da função `realloc()`, para  
alocação dinâmica de memória(array dinâmico)

```

*****/

```

```

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>

```

```

int main(){

```

```

    char *str; /* ponteiro que receberá o endereço do bloco de memória alocado */

```

```

    /* Alocando memória para uma string */

```

```

    if((str = (char *) malloc (11)) == NULL) {
        puts("Não existe memória para o buffer");
        exit(1); /* termina a execução e retorna um código de erro para o
                  sistema operacional*/
    }

```

```

    /* copia "Vamos ver!", para a área de memória alocada */

```

```

    strcpy(str,"Vamos ver!");
    /* imprime a cadeia de caracteres e seu endereço*/
    printf("A string é %s\n e seu endereço é %p\n",str,str);
    str = (char *) realloc(str, 20);
    printf("A string é %s\n e seu endereço é %p\n",str,str);

```

```

    /* libera o bloco de memória */

```

```

    free(str);
    return(0);
}

```

**13.11.3. A função `calloc()`**

A função `calloc()`, permite a alocação de áreas de memória em blocos, numa quantidade definida previamente.

Ex:

```

if((iptr = calloc(ítems, sizeof(ítems)))== NULL) {
    ↑      ↑
    número de blocos    número de byte por bloco

    /* erro */

}

```

Cada bloco é inicializado com zero.

#### EXEMPLO 35

```

/*****
Exemplo do uso da função calloc( ), para
alocação dinâmica de memória(array dinâmico)
*****/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>

int main(){

    char *str; /* ponteiro que receberá o endereço do bloco de memória alocado */
    /* Alocando memória para uma string */
    if((str = (char *) calloc (11,sizeof(char))) == NULL) {
        puts("Não existe memória para o buffer");
        exit(1);
    }
    /* copia "Vamos ver!", para a área de memória alocada */
    strcpy(str,"Vamos ver!");
    /* imprime a cadeia de caracteres */
    printf("A string é %s\n",str);
    /* libera o bloco de memória */
    free(str);
    return(0);
}

```

No exemplo, alocamos onze bytes (caracteres).

## Exemplos usando estruturas.

## EXEMPLO 36

```

/*****
    Exemplo do uso da função malloc(), para
    alocação dinâmica de memória(array dinâmico) e
    estrutura
*****/

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>

    struct reg{
        char nome[40];
        char end[50];
        int idade;
        float sal;
    };

int main(){

    struct reg *str;

    /* Alocando memória para um registro */
    if (( str = (struct reg *) malloc(sizeof(struct reg))) == NULL){
        printf("Não existe memória para ser alocada\n");
        exit(1);
    }
    /* copia dados para o registro */
    strcpy (str->nome,"João da Silva");
    strcpy (str->end,"Rua França, 123");
    str->idade = 34;
    str->sal = (float) 2000.00;
    /* imprime o registro */
    printf ("Mome: %s\nEdereco: %s\nIdade: %d\nSalario: %5.2f",
        str->nome,str->end,str->idade, str->sal);

    /* libera a memória */
    free(str);

    return 0;
}

```

**EXEMPLO 37**

```

/*****

```

Exemplo do uso da função `calloc( )`, para alocação dinâmica de memória(array dinâmico) e estrutura

```

*****/

```

```

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>
struct reg
{
    char nome[40];
    char end[50];
    int idade;
    float sal;
};
int main() {
    struct reg *str,*str2;
    int reglen;
    /* Alocando memória para um registro */
    if ((str = (struct reg *) calloc(40,sizeof(struct reg))) == NULL){
        printf("Não existe memória para ser alocada\n");
        exit(1);
    }
    /* determina o número de bytes da estrutura */
    reglen = sizeof(struct reg);

    /* copia dados para o registro */
    strcpy (str->nome,"João da Silva");
    strcpy (str->end,"Rua França, 123");
    str->idade = 34;
    str->sal = (float) 2000.00;
    /* str2 recebe o endereço do próximo bloco */
    str2 = str + reglen;
    /* copia dados para o segundo registro */
    strcpy (str2->nome,"Maria José");
    strcpy (str2->end,"Rua Italia, 321");
    str2->idade = 64;
    str2->sal = (float) 3000.00;
    /* imprime o registro */
    printf ("Nome: %s\nEndereço: %s\nIdade: %d\nSalário: %5.2f\n",str->nome,
            str->end, str->idade, str->sal);
    printf ("Nome: %s\nEndereço: %s\nIdade: %d\nSalário: %5.2f\n",str2->nome,
            str2->end, str2->idade, str2->sal);
    /* libera a memória */
    free(str);
    return 0;
}

```

**EXERCÍCIOS:**

1) Dado o algoritmo abaixo para ordenação de dados, Método da Bolha. Elaborar um programa em C que, usando alocação dinâmica de memória, seja capaz de implementar as mesmas funções do algoritmo.

Na leitura, os elementos do vetor VET devem ser digitados em uma ordem qualquer. No resultado, VET deverá ser impresso ordenado(ordem crescente).

```

Procedimento Bolha;
Variáveis
    tipo v = vetor[1:100] de inteiros;
    v : VET;
    inteiro :AUX,      { auxiliar para troca de elementos}
                BOLHA, { indicador de mais alto elemento fora de ordem!}
                LSUP,  { indicador do tamanho do vetor a ser pesquisado,
                        sendo o vetor inicial igual a 100}
                J;      { indicador do elemento do vetor}

Início
    leia(VET);
    LSUP ← 20;
    Enquanto LSUP > 1 faça:
        BOLHA ← 0;
        Para J de 1 até LSUP - 1 faça:
            Se VET[J] > VET[J+1]
                então { troca elemento j com j+1}
                    AUX ← VET[J];
                    VET[J] ← VET[J+1];
                    VET[J+1] ← AUX;
                    BOLHA ← J;

            Fim-se;
        Fim-para;
        LSUP ← BOLHA; { aponta para última posição trocada}
    Fim-enquanto;
    Imprima(VET);
Fim.

```

2) Dado o vetor do exercício anterior, verificar se existe um elemento igual a K(chave) no vetor. Se existir, imprimir a posição onde foi encontrada a chave; se não, imprimir: “chave não encontrada”.

A chave K deve ser lida do teclado ( usar o algoritmo de pesquisa binária).



### Procedimento Pesquisa\_Binária;

## Variáveis

<u>inteiro</u> :	COMEÇO,	{ indicador do primeiro elemento da parte do vetor a considerar }
	FIM,	{ indicador do último elemento da parte do vetor a considerar }
	MEIO,	{ indicador do elemento do meio do vetor considerado }
	K;	{ elemento procurado }

```
tipo v = vetor[1:100] de inteiros;
```

v : VET;

## Início

Leia( K);

```
COMEÇO ← 1;
```

```
FIM ← 100;
```

Repita

```
MEIO ← (COMEÇO + FIM) div 2;
```

Se K < VET[MEIO]

então FIM  $\leftarrow$  MEIO - 1;

```
senão COMEÇO ← MEIO + 1;
```

Fim-se;

até VET[MEIO] = K ou COMEÇO > FIM;

$$\underline{\text{Se}} \text{ VET}[\text{MEIO}] \neq \text{K}$$

então imprima( “Não existe o elemento”);

senão imprima(“Está na posição:”, MEIO);

Fim-se;

Fim.

---

ANEXOS

## A . BIBLIOTECA PADRÃO

Nosso objetivo não é o de listarmos todas as funções disponíveis para a biblioteca padrão, nem tão pouco as das outras bibliotecas, já que para tanto existem disponíveis uma vasta literatura disponível e incluímos a elas os manuais dos fabricantes de compiladores. Vamos nos restringir a algumas funções mais importantes para o nosso estudo, o que dará a você o suporte suficiente para utilizar todas as demais.

## Entrada e Saída

Para acessarmos a biblioteca padrão devemos usar certas macros e estruturas que estão definidas no arquivo *stdio.h*, cujo conteúdo deve ser incluído no texto-fonte através da cláusula *#include* do pré-processamento, ou seja, incluir a seguinte linha no programa:

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

## Funções

*getchar()*

Função que retorna o próximo caractere disponível no fluxo de dados da entrada padrão.

```
c = getchar();
```

Caso encontre a indicação de fim de arquivo na entrada padrão, ela retorna o valor -1, que está definido no arquivo *stdin.h* como a constante simbólica EOF.

*putchar()*

Função que escreve o caractere *c* no fluxo da saída-padrão.

```
putchar(c);
```

## Exemplo 001

```

/*****
    uso das funções getchar e putchar
*****/
#include <stdio.h>

void main()
{
    int c;

    puts("Entre com um caracter e tecle entre: ");
    c = getchar();
    puts("O caracter digitado foi: ");
    putchar(c);
}

```

*gets()*

Função utilizada para ler uma cadeia de caracteres da entrada-padrão e que lê para a posição de memória apontada por *string* os próximos caracteres diferentes dos delimitadores \0, \n e EOF.

```
gets(string);
```

onde *char \*string;*

*puts()*

Função que imprime uma cadeia de caracteres na saída padrão.

```
puts(string);
```

onde *char \*string;*

OBS: *string* é um ponteiro para uma cadeia de caracteres que será transmitida para saída-padrão até que se encontre o caractere \0, terminador da cadeia.

**Exemplo 002**

```

/*****
    uso das funções gets e puts
*****/

#include <stdio.h>

void main()
{
    char nome[23];

    puts("Entre com um nome e tecle entre: ");
    gets(nome);
    puts("O nome e: ");
    puts(nome);
}

```

*printf()*

Função que fornece facilidades de formatação, conversão de valores, além de uma flexibilidade de impressão que abrange as outras duas funções anteriores.

Todavia, devido à maior complexidade, não se aconselha seu uso em aplicações onde suas facilidades não seja requeridas, pois implicaria em um código maior de programa.

caractere	Conversões
d	notação em base decimal
o	notação em base octal em sinal
x	notação em base hexadecimal
u	notação decimal sem sinal
c	caractere simples
s	cadeia de caracteres
e	float ou double em notação científica [-]Z.XXXE[+/-]WW
f	float ou double em notação decimal [-]ZZZ.XXX

OBS: se o caractere logo após o % não puder ser interpretado como sendo de conversão, ele será impresso normalmente como se fosse um caractere comum.

- . um sinal de menos (-)
  - ajusta o argumento na parte esquerda do campo disponível. (direita é o padrão)
- . um número
  - especifica o tamanho mínimo do campo a ser impresso.
- . um ponto (.)
  - separa o campo da próxima cadeia de dígitos.

---

```
printf(" dec= %d oct= %o hex= %x asc = %c cad= %s flo=
      %f",a,b,c,d,e,f);
```

#### Exemplo de cadeia

"jararaca deita e rola" (21 caracteres)

Controle	Saída
%10s	:jararaca deita e rola:
%- 10s	:jararaca deita e rola:
%25s	: jararaca deita e rola:
%-25s	:jararaca deita e rola :
%25.14s	: jararaca deita:
%-25.14s	:jararaca deita :
%.14s	:jararaca deita:

#### Exemplo geral

```
char string = "maracuja";
int x = 25;
```

Controle	Saída
("vamos ver")	:vamos ver:
("512")	:512:
("%c",65)	:A:
("%s",string)	:maracuja:
("4.2f",253.478)	:258.48:
("valor=%d",x)	:valor=25:
("%6d",532)	: 532:

*scanf()*

Função que permite a entrada de dados formata.

```
int n;
scanf("%d",&n);
```

Este trecho de código lê a entrada-padrão, atribuindo à variável *n* o valor lido e convertido para o tipo decimal.

OBS: *&n* - endereço de memória da variável *n*.

- Brancos ' ', tabulação '\t' e newline '\n' são ignorados.

- Caracteres comuns precisam combinar (coincidir) com os correspondentes na sequência de caracteres da entrada, caso contrário implicará em erro.

- Caracteres de conversão iniciam com %, seguidos da [largura máxima do campo] e do caractere de conversão que interpreta o fluxo de entrada com conforme a tabela:

Caractere	Conversões
d	inteiro decimal
x	inteiro octal
h	inteiro hexadecimal
c	inteiro curto (short int)
s	cadeia de caracteres
e	float ou double
f	float ou double

*fflush()*

Função que descarrega todos os *buffers* pendentes.

*fflush(stdin)*

onde

*stdin* é a entrada padrão (normalmente teclado)

#### Exemplo 003

```

/*****
    uso das funções printf(), scanf() e fflush()
*****/
#include <stdio.h>

void main()
{
    char string[25];
    int valor;
    float total;

    printf("Entre com um nome e tecle entre: ");
    scanf("%s",&string);fflush(stdin);
    printf("\nEntre com valor:");
    scanf("%d",&valor);fflush(stdin);
    printf("\nEntre com total:");
    scanf("%f",&total);fflush(stdin);
    printf("nome = %s  valor = %d  total = %10.2f",string,valor,total);
}

```

## BIBLIOGRAFIA

1. Microsoft QuickC Programming - The Waite Group - Mitchell Waite - Microsoft Press.
2. Linguagem C Programação e Aplicações - Antônio Roberto Ramos Nogueira - Livro Técnico e Científicos Editora S.A..
3. Algoritmos e Estruturas de Dados - Guimarães/Lages - Livros Técnicos e Científicos Editora S. A.
4. C Avançado Guia do Usuário - Herbert Schildt - McGraw-Hill