

# LINGUAGEM C: DESCOMPLICADA

Prof. André R. Backes

<b>1</b>	<b>Introdução</b>	<b>9</b>
1.1	A linguagem C . . . . .	9
1.1.1	Influência da linguagem C . . . . .	9
1.2	Utilizando o Code::Blocks para programar em C . . . . .	11
1.2.1	Criando um novo projeto no Code::Blocks . . . . .	11
1.2.2	Utilizando o debugger do Code::Blocks . . . . .	15
1.3	Esqueleto de um programa em linguagem C . . . . .	19
1.3.1	Indentação do código . . . . .	21
1.3.2	Comentários . . . . .	22
1.4	Bibliotecas e funções úteis da linguagem C . . . . .	23
1.4.1	O comando #include . . . . .	23
1.4.2	Funções de entrada e saída: stdio.h . . . . .	24
1.4.3	Funções de utilidade padrão: stdlib.h . . . . .	26
1.4.4	Funções matemáticas: math.h . . . . .	28
1.4.5	Testes de tipos de caracteres: ctype.h . . . . .	29
1.4.6	Operações em String: string.h . . . . .	29
1.4.7	Funções de data e hora: time.h . . . . .	30
<b>2</b>	<b>Manipulando dados, variáveis e expressões em C</b>	<b>32</b>
2.1	Variáveis . . . . .	32
2.1.1	Nomeando uma variável . . . . .	33
2.1.2	Definindo o tipo de uma variável . . . . .	35
2.2	Lendo e escrevendo dados . . . . .	39
2.2.1	Printf . . . . .	39

2.2.2	Putchar . . . . .	42
2.2.3	Scanf . . . . .	43
2.2.4	Getchar . . . . .	46
2.3	Escopo: tempo de vida da variável . . . . .	47
2.4	Constantes . . . . .	52
2.4.1	Comando #define . . . . .	53
2.4.2	Comando const . . . . .	53
2.4.3	seqüências de escape . . . . .	54
2.5	Operadores . . . . .	55
2.5.1	Operador de atribuição: “=” . . . . .	55
2.5.2	Operadores aritméticos . . . . .	58
2.5.3	Operadores relacionais . . . . .	60
2.5.4	Operadores lógicos . . . . .	62
2.5.5	Operadores bit-a-bit . . . . .	63
2.5.6	Operadores de atribuição simplificada . . . . .	66
2.5.7	Operadores de Pré e Pós-Incremento . . . . .	67
2.5.8	Modeladores de Tipos (casts) . . . . .	69
2.5.9	Operador vírgula “,” . . . . .	70
2.5.10	Precedência de operadores . . . . .	71
<b>3</b>	<b>Comandos de Controle Condicional</b>	<b>73</b>
3.1	Definindo uma condição . . . . .	73
3.2	Comando if . . . . .	75
3.2.1	Uso das chaves {} . . . . .	78
3.3	Comando else . . . . .	79
3.4	Aninhamento de if . . . . .	83

3.5	Operador ?	86
3.6	Comando switch	88
3.6.1	Uso do comando break no switch	91
3.6.2	Uso das chaves {} no case	94
<b>4</b>	<b>Comandos de Repetição</b>	<b>96</b>
4.1	Repetição por condição	96
4.1.1	Laço infinito	97
4.2	Comando while	98
4.3	Comando for	101
4.3.1	Omitindo uma clausula do comando for	104
4.3.2	Usando o operador de vírgula (,) no comando for	107
4.4	Comando do-while	109
4.5	Aninhamento de repetições	112
4.6	Comando break	113
4.7	Comando continue	115
4.8	Goto e label	116
<b>5</b>	<b>Vetores e matrizes - Arrays</b>	<b>119</b>
5.1	Exemplo de uso	119
5.2	Array com uma dimensão - vetor	120
5.3	Array com duas dimensões - matriz	124
5.4	Arrays multidimensionais	125
5.5	Inicialização de arrays	127
5.5.1	Inicialização sem tamanho	129
5.6	Exemplo de uso de arrays	130

<b>6</b>	<b>Arrays de caracteres - Strings</b>	<b>133</b>
6.1	Definição e declaração de uma string . . . . .	133
6.1.1	Inicializando uma string . . . . .	134
6.1.2	Acessando um elemento da string . . . . .	134
6.2	Trabalhando com strings . . . . .	135
6.2.1	Lendo uma string do teclado . . . . .	136
6.2.2	Escrevendo uma string na tela . . . . .	139
6.3	Funções para manipulação de strings . . . . .	140
6.3.1	Tamanho de uma string . . . . .	140
6.3.2	Copiando uma string . . . . .	141
6.3.3	Concatenando strings . . . . .	142
6.3.4	Comparando duas strings . . . . .	142
<b>7</b>	<b>Tipos definidos pelo programador</b>	<b>144</b>
7.1	Estruturas: struct . . . . .	144
7.1.1	Inicialização de estruturas . . . . .	149
7.1.2	Array de estruturas . . . . .	150
7.1.3	Atribuição entre estruturas . . . . .	152
7.1.4	Estruturas aninhadas . . . . .	153
7.2	Unões: union . . . . .	155
7.3	Enumerações: enum . . . . .	158
7.4	Comando typedef . . . . .	163
<b>8</b>	<b>Funções</b>	<b>167</b>
8.1	Definição e estrutura básica . . . . .	167
8.1.1	Declarando uma função . . . . .	168
8.1.2	Parâmetros de uma função . . . . .	171

8.1.3	Corpo da função . . . . .	173
8.1.4	Retorno da função . . . . .	176
8.2	Tipos de passagem de parâmetros . . . . .	181
8.2.1	Passagem por valor . . . . .	182
8.2.2	Passagem por referência . . . . .	183
8.2.3	Passagem de arrays como parâmetros . . . . .	186
8.2.4	Passagem de estruturas como parâmetros . . . . .	190
8.2.5	Operador Seta . . . . .	193
8.3	Recursão . . . . .	194
<b>9</b>	<b>Ponteiros</b>	<b>200</b>
9.1	Declaração . . . . .	201
9.2	Manipulando ponteiros . . . . .	202
9.2.1	Inicialização e atribuição . . . . .	202
9.2.2	Aritmética com ponteiros . . . . .	208
9.2.3	Operações relacionais com ponteiros . . . . .	211
9.3	Ponteiros genéricos . . . . .	213
9.4	Ponteiros e arrays . . . . .	215
9.4.1	Ponteiros e arrays multidimensionais . . . . .	219
9.4.2	Array de ponteiros . . . . .	220
9.5	Ponteiro para ponteiro . . . . .	221
<b>10</b>	<b>Alocação Dinâmica</b>	<b>225</b>
10.1	Funções para alocação de memória . . . . .	227
10.1.1	sizeof() . . . . .	227
10.1.2	malloc() . . . . .	228
10.1.3	calloc() . . . . .	231

10.1.4 realloc() . . . . .	233
10.1.5 free() . . . . .	236
10.2 Alocação de arrays multidimensionais . . . . .	238
10.2.1 Solução 1: usando array unidimensional . . . . .	238
10.2.2 Solução 2: usando ponteiro para ponteiro . . . . .	240
10.2.3 Solução 3: ponteiro para ponteiro para array . . . . .	244
<b>11 Arquivos</b>	<b>248</b>
11.1 Tipos de Arquivos . . . . .	248
11.2 Sobre escrita e leitura em arquivos . . . . .	250
11.3 Ponteiro para arquivo . . . . .	251
11.4 Abrindo e fechando um arquivo . . . . .	251
11.4.1 Abrindo um arquivo . . . . .	251
11.4.2 Fechando um arquivo . . . . .	256
11.5 Escrita e leitura em arquivos . . . . .	257
11.5.1 Escrita e leitura de caractere . . . . .	257
11.5.2 Fim do arquivo . . . . .	261
11.5.3 Arquivos pré-definidos . . . . .	262
11.5.4 Forçando a escrita dos dados do “buffer” . . . . .	263
11.5.5 Sabendo a posição atual dentro do arquivo . . . . .	264
11.5.6 Escrita e leitura de strings . . . . .	265
11.5.7 Escrita e leitura de blocos de bytes . . . . .	269
11.5.8 Escrita e leitura de dados formatados . . . . .	277
11.6 Movendo-se dentro do arquivo . . . . .	282
11.7 Excluindo um arquivo . . . . .	284
11.8 Erro ao acessar um arquivo . . . . .	285

<b>12 Avançado</b>	<b>287</b>
12.1 Diretivas de compilação . . . . .	287
12.1.1 O comando #include . . . . .	287
12.1.2 Definindo macros: #define e #undef . . . . .	287
12.1.3 Diretivas de Inclusão Condicional . . . . .	294
12.1.4 Controle de linha: #line . . . . .	297
12.1.5 Diretiva de erro: #error . . . . .	298
12.1.6 Diretiva #pragma . . . . .	298
12.1.7 Diretivas pré-definidas . . . . .	299
12.2 Trabalhando com Ponteiros . . . . .	299
12.2.1 Array de Ponteiros e Ponteiro para array . . . . .	299
12.2.2 Ponteiro para função . . . . .	300
12.3 Argumentos na linha de comando . . . . .	308
12.4 Recursos avançados da função printf() . . . . .	311
12.4.1 Os tipos de saída . . . . .	312
12.4.2 As “flags” para os tipos de saída . . . . .	317
12.4.3 O campo “largura” dos tipos de saída . . . . .	320
12.4.4 O campo “precisão” dos tipos de saída . . . . .	320
12.4.5 O campo “comprimento” dos tipos de saída . . . . .	323
12.4.6 Usando mais de uma linha na função printf() . . . . .	323
12.5 Recursos avançados da função scanf() . . . . .	324
12.5.1 Os tipos de entrada . . . . .	325
12.5.2 O campo asterisco “*” . . . . .	329
12.5.3 O campo “largura” dos tipos de entrada . . . . .	329
12.5.4 Os “modificadores” dos tipos de entrada . . . . .	330
12.5.5 Lendo e descartando caracteres . . . . .	331

12.5.6 Lendo apenas caracteres pré-determinados . . . . .	332
12.6 Classes de Armazenamento de Variáveis . . . . .	333
12.6.1 A Classe auto . . . . .	334
12.6.2 A Classe extern . . . . .	334
12.6.3 A Classe static . . . . .	335
12.6.4 A Classe register . . . . .	337
12.7 Trabalhando com campos de bits . . . . .	338
12.8 O Modificador de tipo “volatile” . . . . .	340
12.9 Funções com número de parâmetros variável . . . . .	342

# 1 INTRODUÇÃO

## 1.1 A LINGUAGEM C

A linguagem C é uma das mais bem sucedidas linguagens de alto nível já criadas e considerada uma das linguagens de programação mais utilizadas de todos os tempos. Define-se como linguagem de alto nível aquela que possui um nível de abstração relativamente elevado, que está mais próximo da linguagem humana do que do código de máquina. Ela foi criada em 1972 nos laboratórios Bell por Dennis Ritchie, sendo revisada e padronizada pela ANSI (Instituto Nacional Americano de Padrões, do inglês American National Standards Institute) em 1989.

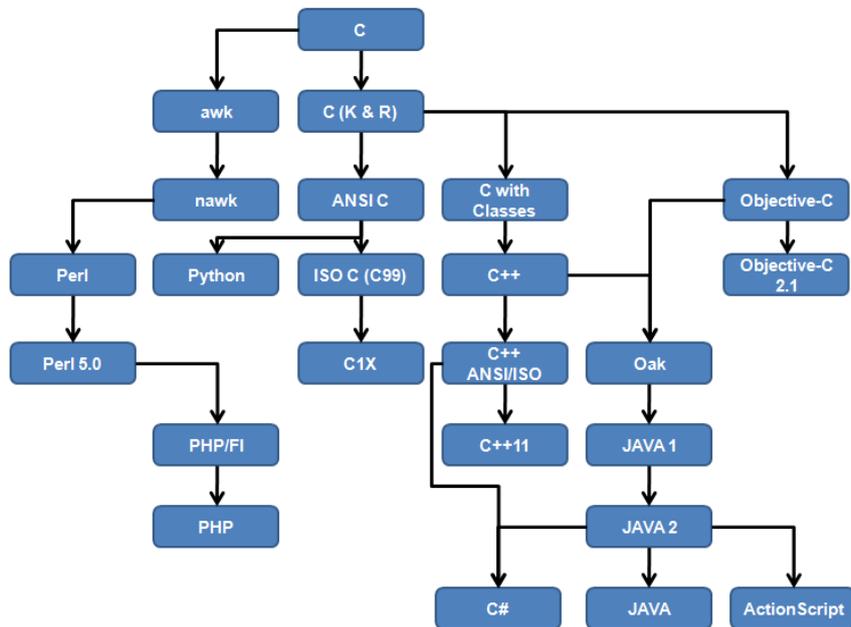
Trata-se de uma linguagem estruturalmente simples e de grande portabilidade. Poucas são as arquiteturas de computadores para que um compilador C não exista. Além disso, o compilador da linguagem gera códigos mais enxutos e velozes do que muitas outras linguagens.

A linguagem C é uma linguagem procedural, ou seja, ela permite que um problema complexo seja facilmente decomposto em módulos, onde cada módulo representa um problema mais simples. Além disso, ela fornece acesso de baixo nível à memória, o que permite o acesso e a programação direta do microprocessador. Ela também permite a implementação de programas utilizando instruções em Assembly, o que permite programar problemas onde a dependência do tempo é crítica.

Por fim, a linguagem C foi criada para incentivar a programação multi-plataforma, ou seja, programas escritos em C podem ser compilado para uma grande variedade de plataformas e sistemas operacionais com apenas pequenas alterações no seu código fonte.

### 1.1.1 INFLUÊNCIA DA LINGUAGEM C

A linguagem C tem influenciado, direta ou indiretamente, muitas linguagens desenvolvidas posteriormente, tais como C++, Java, C# e PHP. Na figura abaixo é possível ver uma bre história da evolução da linguagem C e de sua influência no desenvolvimentos de outras linguagens de programação:



Provavelmente, a influência mais marcante da linguagem foi a sua sintática: todas as linguagem mencionadas combinam a sintaxe de declaração e a sintaxe da expressão da linguagem C com sistemas de tipo, modelos de dados, etc. A figura abaixo mostra como um comando de impressão de números variando de 1 até 10 pode ser implementado em diferentes linguagens:

```

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

```

Java
-----
for(int i=1;i<=10;i++)
{
    System.out.println(i);
}
  
```

```

PHP
-----
for ($i = 1; $i <= 10; $i++)
{
    echo $i;
}
  
```

```

Perl
-----
for($i = 1; $i<=10; $i++)
{
    print $i;
}
  
```

## 1.2 UTILIZANDO O CODE::BLOCKS PARA PROGRAMAR EM C

Existem diversos ambientes de desenvolvimento integrado ou IDE's (do inglês, Integrated Development Environment) que podem ser utilizados para a programação em linguagem C. Um deles é o **Code::Blocks**, uma IDE de código aberto e multiplataforma que suporta múltiplos compiladores. O **Code::Blocks** pode ser baixado diretamente de seu site

[www.codeblocks.org](http://www.codeblocks.org)

ou pelo link

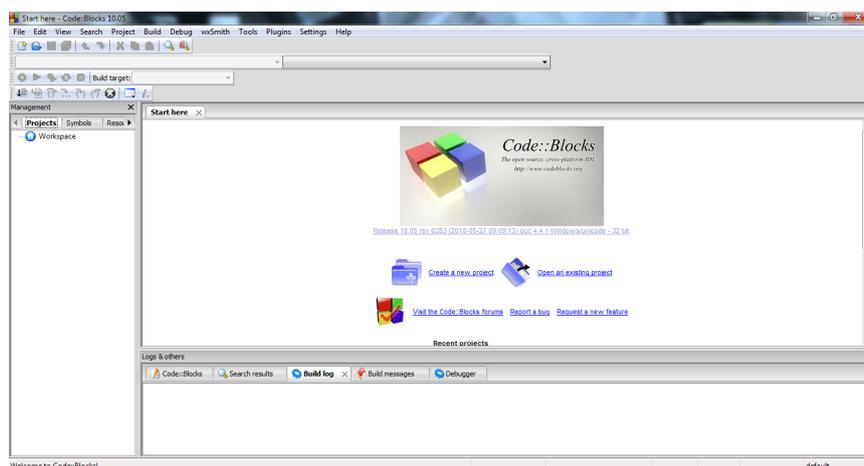
[prdownload.berlios.de/codeblocks/codeblocks-10.05mingw-setup.exe](http://prdownload.berlios.de/codeblocks/codeblocks-10.05mingw-setup.exe)

esse último inclui tanto a IDE do **Code::Blocks** como o compilador GCC e o debugger GDB da MinGW.

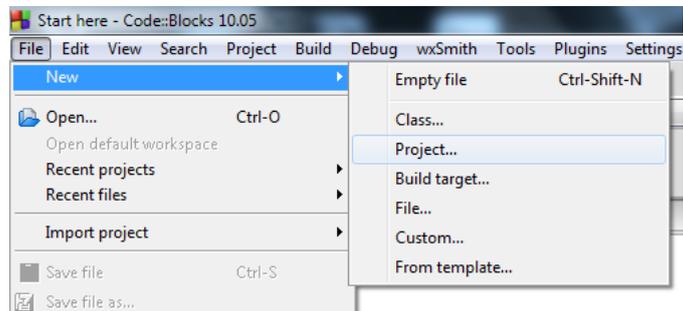
### 1.2.1 CRIANDO UM NOVO PROJETO NO CODE::BLOCKS

Para criar um novo projeto de um programa no software Code::Blocks, basta seguir os passos abaixo:

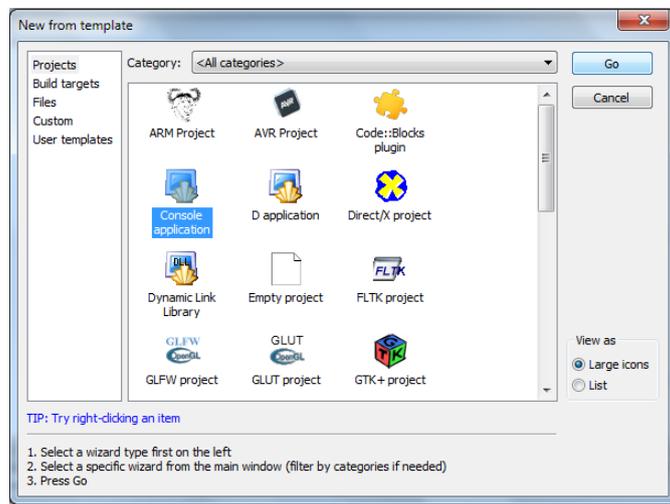
1. Primeiramente, inicie o software **Code::Blocks** (que já deve estar instalado no seu computador). A tela abaixo deverá aparecer;



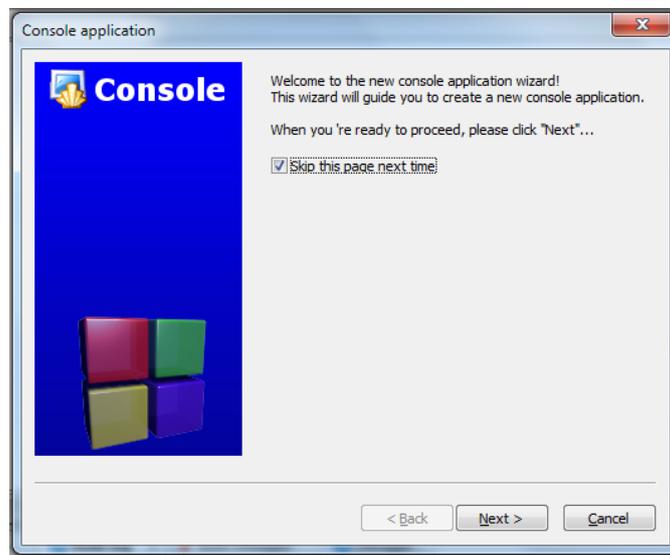
2. Em seguida clique em “**File**”, e escolha “**New**” e depois “**Project...**”;



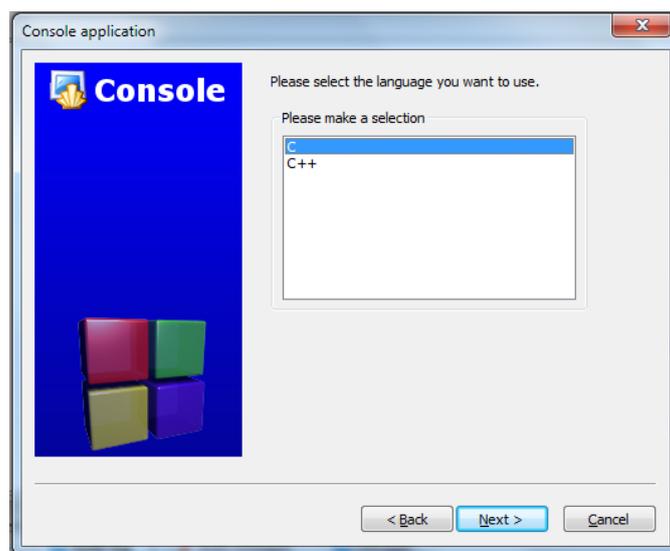
3. Uma lista de modelos (templates) de projetos irá aparecer. Escolha **“Console application”**;



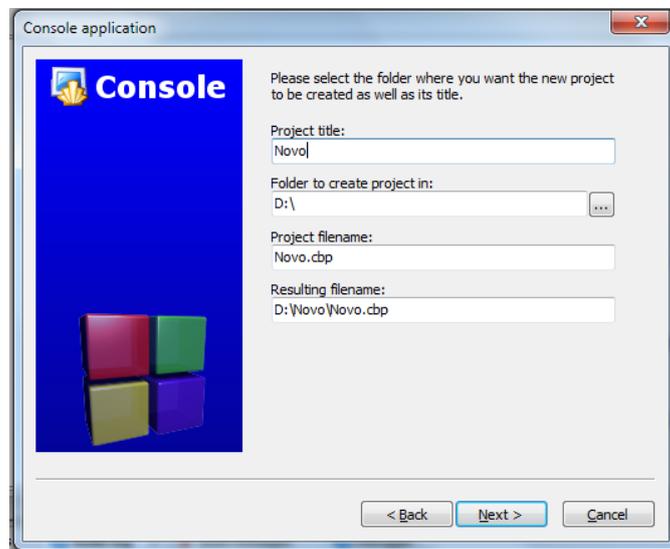
4. Caso esteja criando um projeto pela primeira vez, a tela abaixo irá aparecer. Se marcarmos a opção **“Skip this page next time”**, essa tela de boas vindas não será mais exibida da próxima vez que criarmos um projeto. Em seguida, clique em **“Next”**;



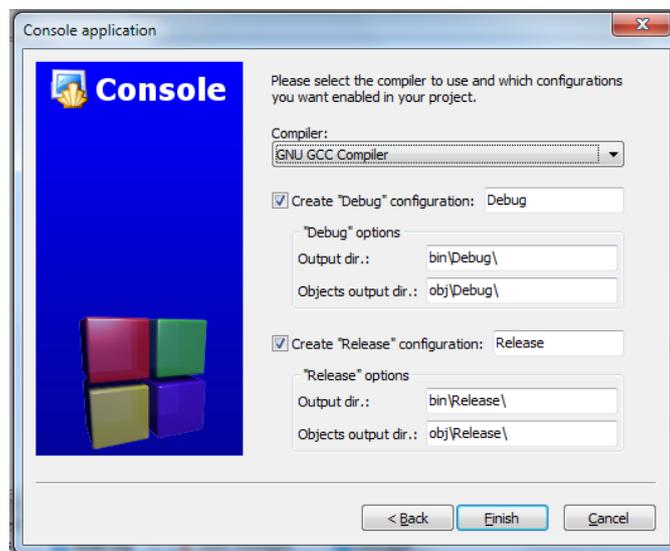
5. Escolha a opção “C” e clique em “Next”;



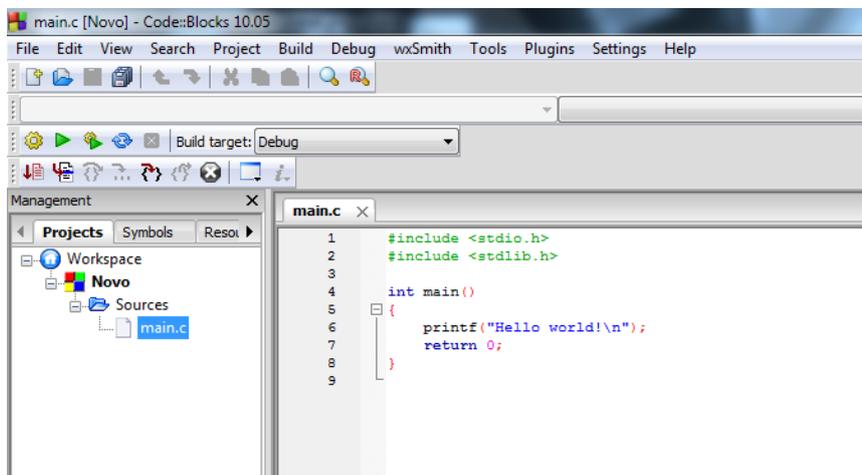
6. No campo “Project title”, coloque um nome para o seu projeto. No campo “Folder to create project in” é possível selecionar onde o projeto será salvo no computador. Clique em “Next” para continuar;



7. Na tela a seguir, algumas configurações do compilador podem ser modificados. No entanto, isso não será necessário. Basta clicar em **“Finish”**;



8. Ao fim desses passos, o esqueleto de um novo programa C terá sido criado, como mostra a figura abaixo:



9. Por fim, podemos utilizar as seguintes opções do menu “**Build**” para compilar e executar nosso programa:

- **Compile current file (Ctrl+Shift+F9)**: essa opção irá transformar seu arquivo de código fonte em instruções de máquina e gerar um arquivo do tipo objeto;
- **Build (Ctrl+F9)**: serão compilados todos os arquivos do seu projeto e fazer o processo de *linkagem* com tudo que é necessário para gerar o executável do seu programa;
- **Build and run (F9)**: além de gerar o executável, essa opção também executa o programa gerado.

## 1.2.2 UTILIZANDO O DEBUGGER DO CODE::BLOCKS

Com o passar do tempo, nosso conhecimento sobre programação cresce, assim como a complexidade de nossos programas. Surge então a necessidade de examinar o nosso programa a procura de erros ou defeitos no código fonte. Para realizar essa tarefa, contamos com a ajuda de um **depurador** ou **debugger**.

O **debugger** nada mais é do que um programa de computador usado para testar e depurar (limpar, purificar) outros programas. Dentre as principais funcionalidades de um debugger estão:

- a possibilidade de executar um programa **passo-a-passo**;
- pausar o programa em pontos pré-definidos, chamados pontos de parada ou **breakpoints**, para examinar o estado atual de suas variáveis.

Para utilizar o debugger do **Code::Blocks**, imagine o seguinte código abaixo:

#### Exemplo: código para o debugger

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int fatorial(int n){
4     int i,f = 1;
5     for (i = 1; i <= n; i++)
6         f = f * i;
7     return f;
8 }
9 int main(){
10    int x,y;
11    printf("Digite um valor inteiro: ");
12    scanf("%d",&x);
13    if (x > 0){
14        printf("X eh positivo\n");
15        y = fatorial(x);
16        printf("Fatorial de X eh %d\n",y);
17    }else{
18        if (x < 0)
19            printf("X eh negativo\n");
20        else
21            printf("X eh Zero\n");
22    }
23    printf("Fim do programa!\n");
24    system(pause);
25    return 0;
26 }
```

Todas as funcionalidades do debugger podem ser encontradas no menu **Debug**. Um programa pode ser facilmente depurado seguindo os passos abaixo:

1. Primeiramente, vamos colocar dois pontos de parada ou **breakpoints** no programa, nas linhas 13 e 23. Isso pode ser feito de duas maneiras: clicando do lado direito do número da linha, ou colocando-se o cursor do mouse na linha que se deseja adicionar o **breakpoint** e selecionar a opção **Toggle breakpoint (F5)**. Um **breakpoint** é identificado por uma bolinha vermelha na linha;

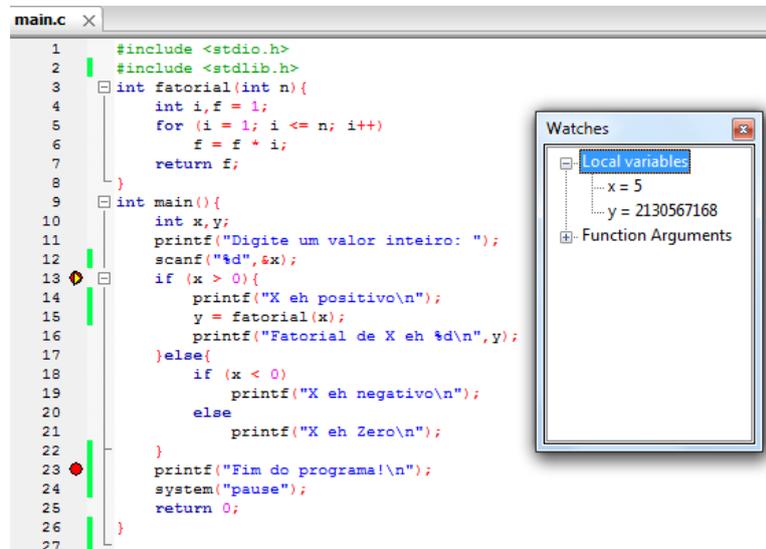
```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3  int fatorial(int n){
4      int i, f = 1;
5      for (i = 1; i <= n; i++)
6          f = f * i;
7      return f;
8  }
9  int main(){
10     int x, y;
11     printf("Digite um valor inteiro: ");
12     scanf("%d", &x);
13     if (x > 0){
14         printf("X eh positivo\n");
15         y = fatorial(x);
16         printf("Fatorial de X eh %d\n", y);
17     }else{
18         if (x < 0)
19             printf("X eh negativo\n");
20         else
21             printf("X eh Zero\n");
22     }
23     printf("Fim do programa!\n");
24     system("pause");
25     return 0;
26 }
27
```

2. Iniciamos o debugger com a opção **Start (F8)**. Isso fará com que o programa seja executado normalmente até encontrar um **breakpoint**. No nosso exemplo, o usuário deverá digitar, no console, o valor lido pelo comando **scanf()** e depois retornar para a tela do **Code::Blocks** onde o programa se encontra pausado. Note que existe um triângulo amarelo dentro do primeiro **breakpoint**. Esse triângulo indica em que parte do programa a pausa está;

```
main.c x
1  #include <stdio.h>
2  #include <stdlib.h>
3  int fatorial(int n){
4      int i, f = 1;
5      for (i = 1; i <= n; i++)
6          f = f * i;
7      return f;
8  }
9  int main(){
10     int x, y;
11     printf("Digite um valor inteiro: ");
12     scanf("%d", &x);
13     if (x > 0){
14         printf("X eh positivo\n");
15         y = fatorial(x);
16         printf("Fatorial de X eh %d\n", y);
17     }else{
18         if (x < 0)
19             printf("X eh negativo\n");
20         else
21             printf("X eh Zero\n");
22     }
23     printf("Fim do programa!\n");
24     system("pause");
25     return 0;
26 }
27
```

3. Dentro da opção **Debugging windows**, podemos habilitar a opção **Watches**. Essa opção irá abrir uma pequena janela que permite ver o valor atual das variáveis de um programa, assim como o valor pas-

sado para funções. Outra maneira de acessar a janela **Watches** é mudar a perspectiva do software para a opção **Debugging**, no menu **View, Perspectives**;



The screenshot shows a code editor window titled 'main.c' with the following C code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int fatorial(int n){
4     int i, f = 1;
5     for (i = 1; i <= n; i++)
6         f = f * i;
7     return f;
8 }
9
10 int main(){
11     int x, y;
12     printf("Digite um valor inteiro: ");
13     scanf("%d", &x);
14     if (x > 0){
15         printf("X eh positivo\n");
16         y = fatorial(x);
17         printf("Fatorial de X eh %d\n", y);
18     }else{
19         if (x < 0)
20             printf("X eh negativo\n");
21         else
22             printf("X eh Zero\n");
23     }
24     printf("Fim do programa!\n");
25     system("pause");
26     return 0;
27 }
```

The 'Watches' window is open, showing the following content:

```
Local variables
- x = 5
- y = 2130567168
Function Arguments
```

4. A partir de um determinado ponto de pausa do programa, podemos nos mover para a próxima linha do programa com a opção **Next line (F7)**. Essa opção faz com que o programa seja executado passo-a-passo, sempre avançando para a linha seguinte do escopo onde nos encontramos;
5. Frequentemente, pode haver uma chamada a uma função construída pelo programador em nosso código, como é o caso da função **fatorial()**. A opção **Next line (F7)** chama a função, mas não permite que a estudemos passo-a-passo. Para entrar dentro do código de uma função utilizamos a opção **Step into (Shift+F7)** na linha da chamada da função. Nesse caso, o triângulo amarelo que marca onde estamos no código vai para a primeira linha do código da função (linha 4);

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int fatorial(int n){
4      int i, f = 1;
5      for (i = 1; i <= n; i++)
6          f = f * i;
7      return f;
8  }
9  int main(){
10     int x, y;
11     printf("Digite um valor inteiro: ");
12     scanf("%d", &x);
13     if (x > 0){
14         printf("X eh positivo\n");
15         y = fatorial(x);
16         printf("Fatorial de X eh %d\n", y);
17     }else{
18         if (x < 0)
19             printf("X eh negativo\n");
20         else
21             printf("X eh Zero\n");
22     }
23     printf("Fim do programa!\n");
24     system("pause");
25     return 0;
26 }
27

```

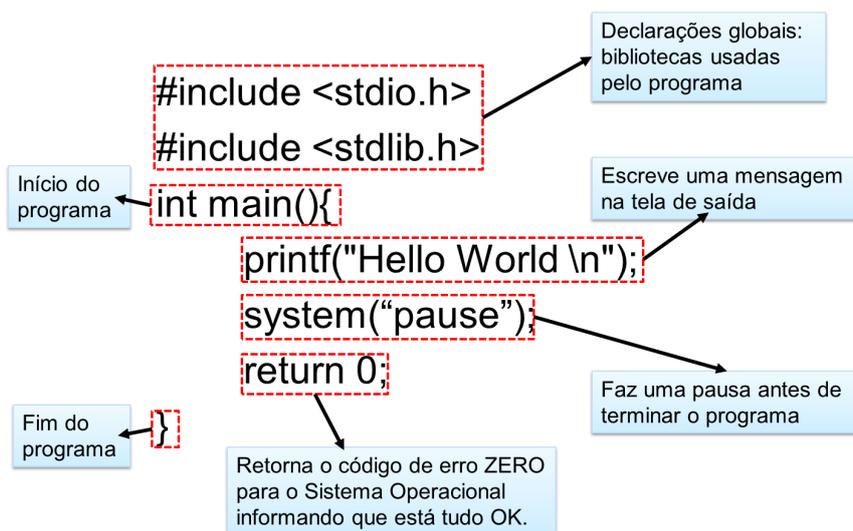
6. Uma vez dentro de uma função, podemos percorrê-la passo-a-passo com a opção **Next line (F7)**. Terminada a função, o debugger vai para a linha seguinte ao ponto do código que chamou a função (linha 16). Caso queiramos ignorar o resto da função e voltar para onde estávamos no código que chamou a função, basta clicar na opção **Step out (Shift+Ctrl+F7)**;
7. Para avançar todo o código e ir direto para o próximo **breakpoint**, podemos usar a opção **Continue (Ctrl+F7)**;
8. Por fim, para parar o debugger, basta clicar na opção **Stop debugger**.

### 1.3 ESQUELETO DE UM PROGRAMA EM LINGUAGEM C

Todo programa escrito em linguagem C que vier a ser desenvolvido deve possuir o seguinte esqueleto:

Primeiro programa em linguagem C	
1	<b>#include</b> <stdio.h>
2	<b>#include</b> <stdlib.h>
3	<b>int</b> main(){
4	printf(‘‘Hello World \n’’);
5	system(‘‘pause’’);
6	<b>return</b> 0;
7	}

A primeira vista este parece ser um programa fútil, já que sua única finalidade é mostrar na tela uma mensagem dizendo **Hello World**, fazer uma pausa, e terminar o programa. Porém, ele permite aprender alguns dos conceitos básicos da linguagem C, como mostra a figura abaixo:



Abaixo, é apresentada uma descrição mais detalhada do esqueleto do programa:

- Temos, no início do programa, a região onde são feitas as declarações globais do programa, ou seja, aquelas que são válidas para todo o programa. No exemplo, o comando **#include <nome\_da\_biblioteca>** é utilizado para declarar as bibliotecas que serão utilizadas pelo programa. Uma biblioteca é um conjunto de funções (pedaços de código) já implementados e que podem ser utilizados pelo programador. No exemplo anterior, duas bibliotecas foram adicionadas ao programa: **stdio.h** (que contém as funções de leitura do teclado e escrita em tela) e **stdlib.h**;
- Todo o programa em linguagem C deve conter a função **main()**. Esta função é responsável pelo início da execução do programa, e é dentro dela que iremos colocar os comandos que queremos que o programa execute.
- As chaves definem o início “{” e o fim “}” de um bloco de comandos / instruções. No exemplo, as chaves definem o início e o fim do programa;
- A função **main** foi definida como uma função **int** (ou seja, inteira), e por isso precisa devolver um valor inteiro. Temos então a necessi-

dade do comando **return 0**, apenas para informar que o programa chegou ao seu final e que está tudo OK;

- A função **printf()** está definida na biblioteca **stdio.h**. Ela serve para imprimir uma mensagem de texto na tela do computador (ou melhor, em uma janela MSDOS ou shell no Linux). O texto a ser escrito deve estar entre “aspas duplas”, e dentro dele podemos também colocar caracteres especiais, como o “\n”, que indica que é para mudar de linha antes de continuar a escrever na tela;
- O comando **system(“pause”)** serve para interromper a execução do programa (fazer uma pausa) para que você possa analisar a tela de saída, após o término da execução do programa. Ela está definida dentro da biblioteca **stdlib.h**;
- A declaração de um comando **quase sempre** termina com um ponto e vírgula “;”. Nas próximas seções, nós veremos quais os comandos que não terminam com um ponto e vírgula “;”;
- Os parênteses definem o início “(” e o fim “)” da lista de argumentos de uma função. Um argumento é a informação que será passada para a função agir. No exemplo, podemos ver que os comandos **main**, **printf** e **system**, são funções;

### 1.3.1 INDENTAÇÃO DO CÓDIGO

Outra coisa importante que devemos ter em mente quando escrevemos um programa é a **indentação do código**. Trata-se de uma convenção de escrita de códigos fonte que visa modificar a estética do programa para auxiliar a sua leitura e interpretação.



A indentação torna a leitura do código fonte muito mais fácil e facilita a sua modificação.

A indentação é o espaçamento (ou tabulação) colocado antes de começar a escrever o código na linha. Ele tem como objetivo indicar a hierarquia dos elementos. No nosso exemplo, os comandos **printf**, **system** e **return** possuem a mesma hierarquia (portanto o mesmo espaçamento) e estão todos contidos dentro do comando **main()** (daí o porquê do espaçamento).



O ideal é sempre criar um novo nível de indentação para um novo bloco de comandos.

A indentação é importante pois o nosso exemplo anterior poderia ser escrito em apenas três linhas, sem afetar o seu desempenho, mas com um alto grau de dificuldade de leitura para o programador:

#### Programa sem indentação

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){printf(‘‘Hello World \n’’);system(‘‘pause’’)
    ;return 0;}
```

### 1.3.2 COMENTÁRIOS

Um comentário, como seu próprio nome diz, é um trecho de texto incluído dentro do programa para descrever alguma coisa, por exemplo, o que aquele pedaço do programa faz. Os comentários não modificam o funcionamento do programa pois são ignorados pelo compilador e servem, portanto, apenas para ajudar o programador a organizar o seu código.

Um comentário pode ser adicionado em qualquer parte do código. Para tanto, a linguagem C permite fazer comentários de duas maneiras diferentes: por linha ou por bloco.

- Se o programador quiser comentar uma única linha do código, basta adicionar // na frente da linha. Tudo o que vier na linha depois do // será considerado como comentário e será ignorado pelo compilador.
- Se o programador quiser comentar mais de uma linha do código, isto é, um bloco de linhas, basta adicionar /\* no começo da primeira linha de comentário e \*/ no final da última linha de comentário. Tudo o que vier depois do símbolo de /\* e antes do \*/ será considerado como comentário e será ignorado pelo compilador.

Abaixo, tem-se alguns exemplos de comentários em um programa:

### Exemplo: comentários no programa

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     /*
5     Escreve
6     na
7     tela
8     */
9     printf(“Hello World \n”);
10    //faz uma pausa no programa
11    system(“pause”);
12    return 0;
13 }
```

Outro aspecto importante do uso dos comentários é que eles permitem fazer a documentação interna de um programa, ou seja, permitem descrever o que cada bloco de comandos daquele programa faz. A documentação é uma tarefa extremamente importante no desenvolvimento e manutenção de um programa, mas muitas vezes ignoradas.

Os comentários dentro de um código permitem que um programador entenda muito mais rapidamente um código que nunca tenha visto ou que ele lembre o que faz um trecho de código a muito tempo implementado por ele. Além disso, saber o que um determinado trecho de código realmente faz aumenta as possibilidades de reutilizá-lo em outras aplicações.

## 1.4 BIBLIOTECAS E FUNÇÕES ÚTEIS DA LINGUAGEM C

### 1.4.1 O COMANDO #INCLUDE

O comando **#include** é utilizado para declarar as bibliotecas que serão utilizadas pelo programa.



Uma biblioteca é um arquivo contendo um conjunto de funções (pedaços de código) já implementados e que podem ser utilizados pelo programador em seu programa.

Esse comando diz ao pré-processador para tratar o conteúdo de um arquivo especificado como se o seu conteúdo houvesse sido digitado no programa no ponto em que o comando **#include** aparece.

O comando **#include** permite duas sintaxes:

- **#include <nome\_da\_biblioteca>**: o pré-processador procurará pela biblioteca nos caminhos de procura pré-especificados do compilador. Usa-se essa sintaxe quando estamos incluindo uma biblioteca que é própria do sistema, como as bibliotecas **stdio.h** e **stdlib.h**;
- **#include "nome\_da\_biblioteca"**: o pré-processador procurará pela biblioteca no mesmo diretório onde se encontra o nosso programa. Podemos ainda optar por informar o nome do arquivo com o caminho completo, ou seja, em qual diretório ele se encontra e como chegar até lá.



De modo geral, os arquivos de bibliotecas na linguagem C são terminados com a extensão **.h**.

Abaixo temos dois exemplos do uso do comando **#include**:

```
#include <stdio.h>  
#include "D:\Programas\soma.h"
```

Na primeira linha, o comando **#include** é utilizado para adicionar uma biblioteca do sistema: **stdio.h** (que contém as funções de leitura do teclado e escrita em tela). Já na segunda linha, o comando é utilizado para adicionar uma biblioteca de nome **soma.h**, localizada no diretório "D:\Programas\".

## 1.4.2 FUNÇÕES DE ENTRADA E SAÍDA: STDIO.H

### Operações em arquivos

- **remove**: apaga o arquivo
- **rename**: renomeia o arquivo

### Acesso a arquivos

- **fclose**: fecha o arquivo
- **fflush**: limpa o buffer. Quaisquer dados não escritos no buffer de saída é gravada no arquivo

- **fopen**: abre o arquivo
- **setbuf**: controla o fluxo de armazenamento em buffer

#### **Entrada/saída formatadas**

- **fprintf**: grava uma saída formatada em arquivo
- **fscanf**: lê dados formatados a partir de arquivo
- **printf**: imprime dados formatados na saída padrão (monitor)
- **scanf**: lê dados formatados da entrada padrão (teclado)
- **sprintf**: grava dados formatados em uma string
- **sscanf**: lê dados formatados a partir de uma string

#### **Entrada/saída de caracteres**

- **fgetc**: lê um caractere do arquivo
- **fgets**: lê uma string do arquivo
- **fputc**: escreve um caractere em arquivo
- **fputs**: escreve uma string em arquivo
- **getc**: lê um caractere do arquivo
- **getchar**: lê um caractere da entrada padrão (teclado)
- **gets**: lê uma string da entrada padrão (teclado)
- **putc**: escreve um caractere na saída padrão (monitor)
- **putchar**: escreve um caractere na saída padrão (monitor)
- **puts**: escreve uma string na saída padrão (monitor)
- **ungetc**: retorna um caractere lido para o arquivo dele

#### **Entrada/saída direta**

- **fread**: lê um bloco de dados do arquivo
- **fwrite**: escreve um bloco de dados no arquivo

## Posicionamento no arquivo

- **fgetpos**: retorna a posição atual no arquivo
- **fseek**: reposiciona o indicador de posição do arquivo
- **fsetpos**: configura o indicador de posição do arquivo
- **ftell**: retorna a posição atual no arquivo
- **rewind**: reposiciona o indicador de posição do arquivo para o início do arquivo

## Tratamento de erros

- **clearerr**: limpa os indicadores de erro
- **feof**: indicador de fim-de-arquivo
- **ferror**: indicador de checagem de erro
- **perror**: impressão de mensagem de erro

## Tipos e macros

- **FILE**: tipo que contém as informações para controlar um arquivo
- **EOF**: constante que indica o fim-de-arquivo
- **NULL**: ponteiro nulo

### 1.4.3 FUNÇÕES DE UTILIDADE PADRÃO: STDLIB.H

## Conversão de strings

- **atof**: converte string para double
- **atoi**: converte string para inteiro
- **atol**: converte string para inteiro longo
- **strtod**: converte string para double e devolve um ponteiro para o próximo double contido na string
- **strtol**: converte string para inteiro longo e devolve um ponteiro para o próximo inteiro longo contido na string

- **strtoul**: converte string para inteiro longo sem sinal e devolve um ponteiro para o próximo inteiro longo sem sinal contido na string

### **Geração de seqüências pseudo-aleatórias**

- **rand**: gera número aleatório
- **srand**: inicializa o gerador de números aleatórios

### **Gerenciamento de memória dinâmica**

- **malloc**: aloca espaço para um array na memória
- **calloc**: aloca espaço para um array na memória e inicializa com zeros
- **free**: libera o espaço alocado na memória
- **realloc**: modifica o tamanho do espaço alocado na memória

### **Ambiente do programa**

- **abort**: abortar o processo atual
- **atexit**: define uma função a ser executada no término normal do programa
- **exit**: finaliza o programa
- **getenv**: retorna uma variável de ambiente
- **system**: executa um comando do sistema

### **Pesquisa e ordenação**

- **bsearch**: pesquisa binária em um array
- **qsort**: ordena os elementos do array

### **Aritmética de inteiro**

- **abs**: valor absoluto
- **div**: divisão inteira
- **labs**: valor absoluto de um inteiro longo
- **ldiv**: divisão inteira de um inteiro longo

#### 1.4.4 FUNÇÕES MATEMÁTICAS: MATH.H

##### Funções trigonométricas

- **cos**: calcula o cosseno de um ângulo em radianos
- **sin**: calcula o seno de um ângulo em radianos
- **tan**: calcula a tangente de um ângulo em radianos
- **acos**: calcula o arco cosseno
- **asin**: calcula o arco seno
- **atan**: calcula o arco tangente
- **atan2**: calcula o arco tangente com dois parâmetros

##### Funções hiperbólicas

- **cosh**: calcula o cosseno hiperbólico de um ângulo em radianos
- **sinh**: calcula o seno hiperbólico de um ângulo em radianos
- **tanh**: calcula a tangente hiperbólica de um ângulo em radianos

##### Funções exponenciais e logarítmicas

- **exp**: função exponencial
- **log**: logaritmo natural
- **log10**: logaritmo comum (base 10)
- **modf**: quebra um número em partes fracionárias e inteira

##### Funções de potência

- **pow**: retorna a base elevada ao expoente
- **sqrt**: raiz quadrada de um número

##### Funções de arredondamento, valor absoluto e outras

- **ceil**: arredonda para cima um número

- **fabs**: calcula o valor absoluto de um número
- **floor**: arredonda para baixo um número
- **fmod**: calcula o resto da divisão

#### 1.4.5 TESTES DE TIPOS DE CARACTERES: CTYPE.H

- **isalnum**: verifica se o caractere é alfanumérico
- **isalpha**: verifica se o caractere é alfabético
- **iscntrl**: verifica se o caractere é um caractere de controle
- **isdigit**: verifica se o caractere é um dígito decimal
- **islower**: verifica se o caractere é letra minúscula
- **isprint**: verifica se caractere é imprimível
- **ispunct**: verifica se é um caractere de pontuação
- **isspace**: verifica se caractere é um espaço em branco
- **isupper**: verifica se o caractere é letra maiúscula
- **isxdigit**: verifica se o caractere é dígito hexadecimal
- **tolower**: converte letra maiúscula para minúscula
- **toupper**: converte letra minúscula para maiúscula

#### 1.4.6 OPERAÇÕES EM STRING: STRING.H

##### Cópia

- **memcpy**: cópia de bloco de memória
- **memmove**: move bloco de memória
- **strcpy**: cópia de string
- **strncpy**: cópia de caracteres da string

##### Concatenação

- **strcat**: concatenação de strings
- **strncat**: adiciona “n” caracteres de uma string no final de outra string

### Comparação

- **memcmp**: compara dois blocos de memória
- **strcmp**: compara duas strings
- **strncmp**: compara os “n” caracteres de duas strings

### Busca

- **memchr**: localiza caractere em bloco de memória
- **strchr**: localiza primeira ocorrência de caractere em uma string
- **strcspn**: retorna o número de caracteres lidos de uma string antes da primeira ocorrência de uma segunda string
- **strpbrk**: retorna um ponteiro para a primeira ocorrência na string de qualquer um dos caracteres de uma segunda string
- **strrchr**: retorna um ponteiro para a última ocorrência do caractere na string
- **strspn**: retorna o comprimento da string que consiste só de caracteres que fazem parte de uma outra string
- **strtok**: divide uma string em sub-strings com base em um caractere

### Outras

- **memset**: preenche bloco de memória com valor especificado
- **strerror**: retorna o ponteiro para uma string de mensagem de erro
- **strlen**: comprimento da string

## 1.4.7 FUNÇÕES DE DATA E HORA: TIME.H

### Manipulação do tempo

- **clock**: retorna o número de pulsos de clock decorrido desde que o programa foi lançado
- **difftime**: retorna a diferença entre dois tempos
- **mktime**: converte uma estrutura tm para o tipo time\_t
- **time**: retorna o tempo atual do calendário como um time\_t

### Conversão

- **asctime**: converte uma estrutura tm para string
- **ctime**: converte um valor time\_t para string
- **gmtime**: converte um valor time\_t para estrutura tm como tempo UTC
- **localtime**: converte um valor time\_t para estrutura tm como hora local
- **strftime**: formata tempo para string

### Tipos e macros

- **clock\_t**: tipo capaz de representar as contagens clock e suportar operações aritméticas
- **size\_t**: tipo inteiro sem sinal
- **time\_t**: tipo capaz de representar os tempos e suportar operações aritméticas
- **struct tm**: estrutura contendo uma data e hora dividida em seus componentes
- **CLOCKS\_PER\_SEC**: número de pulsos de clock em um segundo

## 2 MANIPULANDO DADOS, VARIÁVEIS E EXPRESSÕES EM C

### 2.1 VARIÁVEIS

Na matemática, uma variável é uma entidade capaz de representar um valor ou expressão. Ela pode representar um número ou um conjunto de números, como na equação

$$x^2 + 2x + 1 = 0$$

ou na função

$$f(x) = x^2$$

Na computação, uma variável é uma posição de memória onde poderemos guardar um determinado dado ou valor e modificá-lo ao longo da execução do programa. Em linguagem C, a declaração de uma variável pelo programador segue a seguinte forma geral:

**tipo\_da\_variavel** nome\_da\_variavel;

O **tipo\_da\_variavel** determina o conjunto de valores e de operações que uma variável aceita, ou seja, que ela pode executar. Já o **nome\_da\_variavel** é como o programador identifica essa variável dentro do programa. Ao nome que da variável o computador associa o endereço do espaço que ele reservou na memória para guardar essa variável.



Depois declaração de uma variável é necessário colocar um ponto e vírgula (;).

Isso é necessário uma vez que o ponto e vírgula é utilizado para separar as instruções que compõem um programa de computador.

### DECLARANDO VARIÁVEIS

Uma variável do tipo inteiro pode ser declarada como apresentado a seguir:

**int** x;

Além disso, mais de uma variável pode ser declarada para um mesmo tipo. Para tanto, basta separar cada nome de variável por uma **vírgula** (,):

```
int x,y,z;
```



Uma variável deve ser declarada antes de ser usada no programa.

Lembre-se, apenas quando declaramos uma variável é que o computador reserva um espaço de memória para guardarmos nossos dados.



Antes de usar o conteúdo de uma variável, tenha certeza de que o mesmo foi atribuído antes.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int x;
5     printf('x = %d\n',x);
6     x = 5;
7     printf('x = %d\n',x);
8     system('pause');
9     return 0;
10 }
```

Saída     x = qualquer valor  
          x = 5

Quando falamos de memória do computador não existe o conceito de posição de memória “vazia”. A posição pode apenas não estar sendo utilizada. Cada posição de memória do computador está preenchida com um conjunto de 0's e 1's. Portanto, ao criarmos uma variável, ela automaticamente estará preenchida com um valor chamado de “lixo”.

### 2.1.1 NOMEANDO UMA VARIÁVEL

Quando criamos uma variável, o computador reserva um espaço de memória onde poderemos guardar o valor associado a essa variável. Ao nome que damos a essa variável o computador associa o endereço do espaço que

ele reservou na memória para guardar essa variável. De modo geral, interessa ao programador saber o nome das variáveis. Porém, existem algumas regras para a escolha dos nomes das variáveis na linguagem C.

- O nome de uma variável é um conjunto de caracteres que podem ser letras, números ou underscores "\_";
- O nome de uma variável deve sempre iniciar com uma letra ou o underscore "\_".



Na linguagem C, letras maiúsculas e minúsculas são consideradas diferentes.

A linguagem C é **case-sensitive**, ou seja, uma palavra escrita utilizando **caracteres maiúsculos** é diferente da mesma palavra escrita com **caracteres minúsculos**. Sendo assim, as palavras **Soma**, **soma** e **SOMA** são consideradas diferentes para a linguagem C e representam **TRÊS** variáveis distintas.



Palavras chaves não podem ser usadas como nomes de variáveis.

As palavras chaves são um conjunto de 38 palavras reservadas dentro da linguagem C. São elas que formam a sintaxe da linguagem de programação C. Essas palavras já possuem funções específicas dentro da linguagem de programação e, por esse motivo, elas não podem ser utilizadas para outro fim como, por exemplo, nomes de variáveis. Abaixo, tem-se uma lista com as 38 palavras reservadas da linguagem C.

#### lista de palavras chaves da linguagem C

auto	double	int	struct	break	else	long	switch
case	enum	if	typeof	continue	float	return	while
union	const	for	short	unsigned	char	extern	signed
void	default	do	sizeof	volatile	goto	register	static

O exemplo abaixo apresenta alguns nomes possíveis de variáveis e outros que fogem as regras estabelecidas:

Exemplo: nomeando variáveis							
comp!	.var	int	.var	1cont	-x	Va-123	 
cont	Cont	Va_123	_teste	int1	cont1	x&	

## 2.1.2 DEFININDO O TIPO DE UMA VARIÁVEL

Vimos anteriormente que o tipo de uma variável determina o conjunto de valores e de operações que uma variável aceita, ou seja, que ela pode executar. A linguagem C possui um total de cinco tipos de dados básicos. São eles:

Tipo	Bits	Intervalo de valores
<b>char</b>	8	-128 A 127
<b>int</b>	32	-2.147.483.648 A 2.147.483.647
<b>float</b>	32	1,175494E-038 A 3,402823E+038
<b>double</b>	64	2,225074E-308 A 1,797693E+308
<b>void</b>	8	sem valor

### O TIPO CHAR

Começemos pelo tipo **char**. Esse tipo de dados permite armazenar em um único byte (8 bits) um número inteiro muito pequeno ou o código de um caractere do conjunto de caracteres da tabela **ASCII**:

```
char c = 'a';
```

```
char n = 10;
```



Caracteres sempre ficam entre 'aspas simples'!

Lembre-se: uma única letra pode ser o nome de uma variável. As '**aspas simples**' permitem que o compilador saiba que estamos inicializando nossa variável com uma letra e não com o conteúdo de outra variável.

### O TIPO INT

O segundo tipo de dado é o tipo inteiro: **int**. Esse tipo de dados permite armazenar um número inteiro (**sem parte fracionária**). Seu tamanho depende do processador em que o programa está rodando, e é tipicamente 16 ou 32 bits:

```
int n = 1459;
```



Cuidado com a forma com que você inicializa as variáveis dos tipos **char** e **int**.

Na linguagem C, os tipos **char** e **int** podem ser especificados nas bases **decimal** (padrão), **octal** ou **hexadecimal**. A base **decimal** é a base padrão. Porém, se o valor inteiro for precedido por:

- “0”, ele será interpretado como octal. Nesse caso, o valor deve ser definido utilizando os dígitos de 0, 1, 2, 3, 4, 5, 6 e 7. Ex: **int** x = 044; Nesse caso, 044 equivale a 36 ( $4 * 8^1 + 4 * 8^0$ );
- “0x” ou “0X”, ele será interpretado como hexadecimal. Nesse caso, o valor deve ser definido utilizando os dígitos de 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9, e as letras A (10), B (11), C (12), D (13), E (14) e F (15). Ex: **int** y = 0x44; Nesse caso, 0x44 equivale a 68 ( $4 * 16^1 + 4 * 16^0$ );

## OS TIPOS FLOAT E DOUBLE

O terceiro e quarto tipos de dados são os tipos reais: **float** e **double**. Esses tipos de dados permitem armazenar um valor real (**com parte fracionária**), também conhecido como **ponto flutuante**. A diferença entre eles é de precisão:

- tipo **float**: precisão simples;
- tipo **double**: dupla precisão. São úteis quando queremos trabalhar com intervalos de números reais realmente grandes.



Em números reais, a parte decimal usa ponto e não vírgula!

A linguagem C usa o padrão numérico americano, ou seja, a parte decimal fica depois de um ponto. Veja os exemplos:

```
float f = 5.25;
```

```
double d = 15.673;
```



Pode-se escrever números dos tipos **float** e **double** usando *notação científica*.

A *notação científica* é uma forma de escrever números extremamente grandes ou extremamente pequenos. Nesse caso, o valor real é seguido por uma letra “e” ou “E” e um número inteiro (positivo ou negativo) que indica o expoente da base 10 (representado pela letra “e” ou “E” que multiplica o número):

```
double x = 5.0e10;
```

equivale a

```
double x = 50000000000;
```

## O TIPO VOID

Por fim, temos o tipo **void**. Esse tipo de dados permite declarar uma função que não retorna valor ou um ponteiro genérico, como será visto nas próximas seções.



A linguagem C não permite que se declare uma variável do tipo **void**. Esse tipo de dados só deve ser usado para declarar funções que não retornam valor ou ponteiros genéricos.

## OS MODIFICADORES DE TIPOS

Além desses cinco tipos básicos, a linguagem C possui quatro modificadores de tipos. Eles são aplicados precedendo os tipos básicos (com a exceção do tipo **void**), e eles permitem alterar o significado do tipo, de modo a adequá-lo às necessidades do nosso programa. São eles:

- **signed**: determina que a variável declarada dos tipos **char** ou **int** terá valores positivos ou negativos. Esse é o padrão da linguagem. Exemplo:

```
signed char x;
```

```
signed int y;
```

- **unsigned**: determina que a variável declarada dos tipos **char** ou **int** só terá valores positivos. Nesse caso, a variável perde o seu bit de sinal, o que aumenta a sua capacidade de armazenamento. Exemplo:

```
unsigned char x;  
unsigned int y;
```

- **short**: determina que a variável do tipo **int** terá 16 bits (*inteiro pequeno*), independente do processador. Exemplo:

```
short int i;
```

- **long**: determina que a variável do tipo **int** terá 32 bits (*inteiro grande*), independente do processador. Também determina que o tipo **double** possua maior precisão. Exemplo:

```
long int n;  
long double d;
```



A linguagem C permite que se utilize mais de um modificador de tipo sobre um mesmo tipo.

Desse modo, podemos declarar um inteiro grande (**long**) e sem sinal (**unsigned**), o que aumenta em muito o seu intervalo de valores possíveis:

```
unsigned long int m;
```

A tabela a seguir mostra todas as combinações permitidas dos tipos básicos e dos modificadores de tipo, o seu tamanhos em bits e seu intervalo de valores:

Tipo	Bits	Intervalo de valores
char	8	-128 A 127
unsigned char	8	0 A 255
signed char	8	-128 A 127
int	32	-2.147.483.648 A 2.147.483.647
unsigned int	32	0 A 4.294.967.295
signed int	32	-32.768 A 32.767
short int	16	-32.768 A 32.767
unsigned short int	16	0 A 65.535
signed short int	16	-32.768 A 32.767
long int	32	-2.147.483.648 A 2.147.483.647
unsigned long int	32	0 A 4.294.967.295
signed long int	32	-2.147.483.648 A 2.147.483.647
float	32	1,175494E-038 A 3,402823E+038
double	64	2,225074E-308 A 1,797693E+308
long double	96	3,4E-4932 A 3,4E+4932

## 2.2 LENDO E ESCRIVENDO DADOS

### 2.2.1 PRINTF

A função **printf()** é uma das funções de saída/escrita de dados da linguagem C. Seu nome vem da expressão em inglês *print formatted*, ou seja, escrita formatada. Basicamente, a função **printf()** escreve na saída de vídeo (tela) um conjunto de valores, caracteres e/ou sequência de caracteres de acordo com o formato especificado. A forma geral da função **printf()** é:

```
printf("tipos de saída", lista de variáveis)
```

A função **printf()** recebe 2 parâmetros de entrada

- **“tipos de saída”**: conjunto de caracteres que especifica o formato dos dados a serem escritos e/ou o texto a ser escrito;
- **lista de variáveis**: conjunto de nomes de variáveis, separados por vírgula, que serão escritos.

### ESCREVENDO UMA MENSAGEM DE TEXTO

A forma geral da função **printf()** especifica que ela sempre receberá uma lista de variáveis para formatar e escrever na tela. Isso nem sempre é

verdade. A função **printf()** pode ser usada quando queremos escrever apenas um texto simples na tela:

```
printf("texto");
```

### ESCREVENDO VALORES FORMATADOS

Quando queremos escrever dados formatados na tela usamos a forma geral da função, a qual possui os **tipos de saída**. Eles especificam o formato de saída dos dados que serão escritos pela função **printf()**. Cada tipo de saída é precedido por um sinal de % e um tipo de saída deve ser especificado para cada variável a ser escrita. Assim, se quisessemos escrever uma única expressão com o comando **printf()**, faríamos

```
printf("%tipo_de_saida", expressão);
```



Se fossem duas as expressões a serem escritas, faríamos

```
printf("%tipo1 %tipo2", expressão1, expressão2);
```



e assim por diante. Note que os formatos e as expressões a serem escritas com aquele formato devem ser especificados na mesma ordem, como mostram as setas.



O comando **printf()** não exige o símbolo & na frente do nome de cada variável.

Diferente do comando **scanf()**, o comando **printf()** não exige o símbolo & na frente do nome de uma variável que será escrita na tela. Se usado, ele possui outro significado (como será visto mais adiante) e não exibe o conteúdo da variável.

A função **printf()** pode ser usada para escrever virtualmente qualquer tipo de dado. A tabela abaixo mostra alguns dos tipos de saída suportados pela linguagem:

Alguns “tipos de saída”	
%c	escrita de um caractere
%d ou %i	escrita de números inteiros
%u	escrita de números inteiros sem sinal
%f	escrita de número reais
%s	escrita de vários caracteres
%p	escrita de um endereço de memória
%e ou %E	escrita em notação científica

Abaixo, tem-se alguns exemplos de escrita de dados utilizando o comando **printf()**. Nesse momento não se preocupe com o ‘\n’ que aparece dentro do comando **printf()**, pois ele serve apenas para ir para uma nova linha ao final do comando:

Exemplo: escrita de dados na linguagem C	
	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int x = 10; 5     //Escrita de um valor inteiro 6     printf(“%d\n”,x); 7     float y = 5.0; 8     //Escrita de um valor inteiro e outro real 9     printf(“%d%f\n”,x,y); 10    //Adicionando espaço entre os valores 11    printf(“%d %f\n”,x,y); 12    system(“pause”); 13    return 0; 14 } </pre>
Saída	<pre> 10 105.000000 10 5.000000 </pre>

No exemplo acima, os comandos

```
printf(“%d%f\n”,x,y);
```

e

```
printf(“%d %f\n”,x,y);
```

imprimem os mesmos dados, mas o segundo os separa com um espaço. Isso ocorre por que o comando **printf()** aceita textos junto aos tipos de saída. Pode-se adicionar texto antes, depois ou entre dois ou mais tipos de saída:

printf("texto **%tipo de saída** texto" **expressão**);



Junto ao tipo de saída, pode-se adicionar texto e não apenas espaços.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int x = 10;
5     printf("Total = %d\n", x);
6     printf("%d caixas\n", x);
7     printf("Total de %d caixas\n", x);
8     system("pause");
9     return 0;
10 }
```

Saída      Total = 10  
            10 caixas  
            Total de 10 caixas

Isso permite que o comando **printf()** seja usado para escrever não apenas dados, mas sentenças que façam sentido para o usuário do programa.

## 2.2.2 PUTCHAR

A função **putchar()** (*put character*) permite escrever um único caractere na tela. Sua forma geral é:

```
int putchar(int caractere)
```

A função **putchar()** recebe como parâmetro de entrada um único valor inteiro. Esse valor será convertido para caractere e mostrado na tela. A função retorna

- Se NÃO ocorrer um erro: o próprio caractere que foi escrito;
- Se ocorrer um erro: a constante **EOF** (definida na biblioteca `stdio.h`) é retornada.

### Exemplo: putchar()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     char c = 'a';
5     int x = 65;
6     putchar(c);
7     putchar('\n');
8     putchar(x);
9     putchar('\n');
10    system('pause');
11    return 0;
12 }
```

Saída    a  
          A

Perceba, no exemplo acima, que a conversão na linguagem C é direta no momento da impressão, ou seja, o valor 65 é convertido para o caractere ASCII correspondente, no caso, o caractere “A”. Além disso, o comando `putchar()` também aceita o uso de seqüências de escape como o caractere `'\n'` (nova linha).

### 2.2.3 SCANF

A função **scanf()** é uma das funções de entrada/leitura de dados da linguagem C. Seu nome vem da expressão em inglês *scan formatted*, ou seja, leitura formatada. Basicamente, a função **scanf()** lê do teclado um conjunto de valores, caracteres e/ou seqüência de caracteres de acordo com o formato especificado. A forma geral da função **scanf()** é:

```
scanf(“tipos de entrada”, lista de variáveis)
```

A função **scanf()** recebe 2 parâmetros de entrada

- **“tipos de entrada”**: conjunto de caracteres que especifica o formato dos dados a serem lidos;
- **lista de variáveis**: conjunto de nomes de variáveis que serão lidos e separados por vírgula, onde cada nome de variável é precedido pelo operador `&`.

Os **tipo de entrada** especificam o formato de entrada dos dados que serão lidos pela função **scanf()**. Cada tipo de entrada é precedido por um sinal de % e um tipo de entrada deve ser especificado para cada variável a ser lida. Assim, se quisessemos ler uma única variável com o comando **scanf()**, faríamos

```
scanf("%tipo_de_entrada",&variável);
```

Se fossem duas as variáveis a serem lidas, faríamos

```
scanf("%tipo1%tipo2",&var1, &var2);
```

e assim por diante. Note que os formatos e as variáveis que armazenarão o dado com aquele formato devem ser especificados na mesma ordem, como mostram as setas.



Na linguagem C, é necessário colocar o símbolo de & antes do nome de cada variável a ser lida pelo comando **scanf()**.

Trata-se de uma exigência da linguagem C. Todas as variáveis que receberão valores do teclado por meio da **scanf()** deverão ser passadas pelos seus endereços. Isso se faz colocando o operador de endereço "&" antes do nome da variável.

A função **scanf()** pode ser usada para ler virtualmente qualquer tipo de dado. No entanto, ela é usada com mais frequência para a leitura de números inteiros e/ou de ponto flutuante (números reais). A tabela abaixo mostra alguns dos tipos de entrada suportados pela linguagem:

Alguns "tipos de entrada"	
%c	leitura de um caractere
%d ou %i	leitura de números inteiros
%f	leitura de número reais
%s	leitura de vários caracteres

Abaixo, tem-se alguns exemplos de leitura de dados utilizando o comando **scanf()**:

### Exemplo: leitura de dados na linguagem C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int x,z;
5     float y;
6     //Leitura de um valor inteiro
7     scanf("%d",&x);
8     //Leitura de um valor real
9     scanf("%f",&y);
10    //Leitura de um valor inteiro e outro real
11    scanf("%d%f",&x,&y);
12    //Leitura de dois valores inteiros
13    scanf("%d%d",&x,&z);
14    //Leitura de dois valores inteiros com espaço
15    scanf("%d %d",&x,&z);
16    system('pause');
17    return 0;
18 }
```

No exemplo acima, os comandos

```
scanf("%d%d",&x,&z);
```

e

```
scanf("%d %d",&x,&z);
```

são equivalentes. Isso ocorre por que o comando **scanf()** ignora os espaços em branco entre os tipos de entrada. Além disso, quando o comando **scanf()** é usado para ler dois ou mais valores, podemos optar por duas formas de digitar os dados no teclado:

- Digitar um valor e, em seguida, pressionar a tecla **ENTER** para cada valor digitado;
- Digitar todos os valores separados por espaço e, por último, pressionar a tecla **ENTER**.



O comando **scanf()** ignora apenas os espaços em branco entre os tipos de entrada. Qualquer outro caractere inserido entre os tipos de dados deverá ser digitado pelo usuário, mas será descartado pelo programa.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int dia, mes, ano;
5     //Leitura de três valores inteiros
6     //com barras entre eles
7     scanf( '%d/%d/%d' , &dia, &mes, &ano );
8     system( 'pause' );
9     return 0;
10 }
```

Isso permite que o comando **scanf()** seja usado para receber dados formatados como, por exemplo, uma data: dia/mês/ano. No exemplo acima, o comando **scanf()** é usado para a entrada de três valores inteiros separados por uma barra “/” cada. Quando o usuário for digitar os três valores, ele será obrigado a digitar os três valores separados por barra (as barras serão descartadas e não interferem nos dados). Do contrário, o comando **scanf()** não irá ler corretamente os dados digitados.

## 2.2.4 GETCHAR

A função **getchar()** (*get character*) permite ler um único caractere do teclado. Sua forma geral é:

```
int putchar(void)
```

A função **getchar()** não recebe parâmetros de entrada. A função retorna

- Se NÃO ocorrer um erro: o código ASCII do caractere lido;
- Se ocorrer um erro: a constante **EOF** (definida na biblioteca `stdio.h`) é retornada.

### Exemplo: getchar()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     char c;
5     c = getchar();
6     printf( "Caractere: %c\n", c);
7     printf( "Codigo ASCII: %d\n", c);
8     system( "pause" );
9     return 0;
10 }
```

Perceba, no exemplo acima, que a conversão na linguagem C é direta no momento da leitura, ou seja, embora a função retorne um valor do tipo **int**, pode-se atribuir para uma variável do tipo **char** devido a conversão automática da linguagem C.

## 2.3 ESCOPO: TEMPO DE VIDA DA VARIÁVEL

Quando declararmos uma variável, vimos que é preciso sempre definir o seu **tipo** (conjunto de valores e de operações que uma variável aceita) e **nome** (como o programador identifica essa variável dentro do programa). Porém, além disso, é preciso definir o seu **escopo**.



O escopo é o conjunto de regras que determinam o uso e a validade das variáveis ao longo do programa.

Em outras palavras, escopo de uma variável define **onde** e **quando** a variável pode ser usada. Esse escopo está intimamente ligado com o local de declaração dessa variável e por esse motivo ele pode ser: **global** ou **local**.

### ESCOPO GLOBAL

Uma variável **global** é declarada fora de todas as funções do programa, ou seja, na área de declarações globais do programa (acima da cláusula **main**). Essas variáveis existem enquanto o programa estiver executando, ou seja, o tempo de vida de uma variável global é o tempo de execução do programa. Além disso, essas variáveis podem ser acessadas e alteradas em qualquer parte do programa.

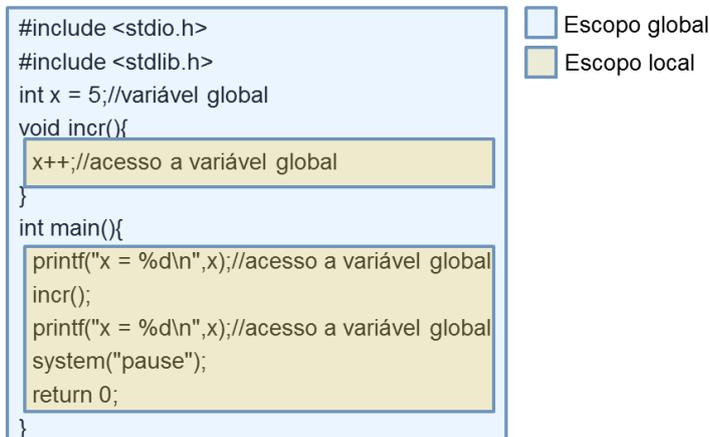


Variáveis globais podem ser acessadas e alteradas em qualquer parte do programa.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int x = 5; //variável global
4 void incr(){
5     x++; //acesso a variável global
6 }
7 int main(){
8     printf('x = %d\n', x); //acesso a variável
9         global
10    incr();
11    printf('x = %d\n', x); //acesso a variável
12        global
13    system('pause');
14    return 0;
15 }
```

Saída     x = 5  
          x = 6

Na figura abaixo, é possível ter uma boa representação de onde começa e termina cada escopo do código anterior:



Note, no exemplo acima, que a variável *x* é declarada junto com as bibliotecas do programa, portanto, trata-se de uma variável global (escopo global). Por esse motivo, ela pode ser acessada e ter seu valor alterado em qualquer parte do programa (ou seja, no escopo global e em qualquer escopo local).



De modo geral, evita-se o uso de variáveis globais em um programa.

As variáveis globais devem ser evitadas porque qualquer parte do programa pode alterá-la. Isso torna mais difícil a manutenção do programa, pois fica difícil saber onde ele é inicializada, para que serve, etc. Além disso, variáveis globais ocupam memória durante todo o tempo de execução do programa e não apenas quando elas são necessárias.

## ESCOPO LOCAL

Já uma variável **local** é declarada dentro de um bloco de comandos delimitado pelo operador de chaves {}(escopo local). Essas variáveis são visíveis apenas no interior do bloco de comandos onde ela foi declarada, ou seja, **dentro do seu escopo**.



Um bloco começa quando abrimos uma chave {e termina quando fechamos a chave }.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void func1 () {
4     int x; //variável local
5 }
6 void func2 () {
7     int x; //variável local
8 }
9 int main () {
10    int x;
11    scanf ("%d", &x);
12    if (x == 5) {
13        int y=1;
14        printf ("%d\n", y);
15    }
16    system ("pause");
17    return 0; }
```

Na figura abaixo, é possível ter uma boa representação de onde começa e termina cada escopo do código anterior:

```
#include <stdio.h>
#include <stdlib.h>
void func1(){
    int x;//variável local
}
void func2(){
    int x;//variável local
}
int main(){
    int x;
    scanf("%d",&x);
    if(x == 5){
        int y=1;
        printf("%d\n",y);
    }
    system("pause");
    return 0;
}
```

- Escopo global
- Escopo local
- Escopo local entro de outro escopo local

Note, no exemplo acima, que a variável *x* é declarada **três** vezes. Cada declaração dela está em um bloco de comandos distinto (ou seja, delimitado por um operador de chaves `{}`). Desse modo, apesar de possuíremos o mesmo nome, elas possuem escopos diferentes e, conseqüentemente, tempos de vida diferentes: **uma não existe enquanto a outra existe**. Já a variável *y* só existe dentro do bloco de comandos pertencente a instrução **if(x == 5)**, ou seja, outro escopo local.



Quando um bloco possui uma variável local com o mesmo nome de uma variável global, esse bloco dará preferência à variável local. O mesmo vale para duas variáveis locais em blocos diferentes: a declaração mais próxima tem maior precedência e oculta as demais variáveis com o mesmo nome.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int x = 5;
4 int main(){
5     printf('x = %d\n',x);
6     int x = 4;
7     printf('x = %d\n',x);
8     {
9         int x = 3;
10        printf('x = %d\n',x);
11    }
12    printf('x = %d\n',x);
13    system('pause');
14    return 0;
15 }
```

Saída

```
x = 5
x = 4
x = 3
x = 4
```

Na figura abaixo, é possível ter uma boa representação de onde começa e termina cada escopo do código anterior e como um escopo oculta os demais:

```

#include <stdio.h>
#include <stdlib.h>
int x = 5;
int main(){
    printf("x = %d\n",x);
    int x = 4;
    printf("x = %d\n",x);
    {
        int x = 3;
        printf("x = %d\n",x);
    }
    printf("x = %d\n",x);
    system("pause");
    return 0;
}

```

Escopo global

Escopo local

Escopo local entro de outro escopo local

Note, no exemplo acima, que a variável *x* é declarada **três** vezes. Cada declaração dela está em um escopo distinto: **uma é global e duas são locais**. Na primeira chamada do comando **printf()** (linha 5), a variável global *x* é acessada. Isso ocorre porque, apesar de estarmos em um escopo local, a segunda variável *x* ainda não foi criada e portanto não existe. Já na segunda chamada do comando **printf()** (linha 7), a segunda variável *x* já foi criada, ocultando a variável global de mesmo nome. Por isso, esse comando **printf()** imprime na tela de saída o valor **x = 4**. O mesmo acontece com a terceira chamada do comando **printf()** (linha 10): esse comando está dentro de um novo bloco de comandos, ou seja, delimitado por um operador de chaves **{}**. A declaração da terceira variável *x* oculta a declaração da segunda variável *x*. Por isso, esse comando **printf()** imprime na tela de saída o valor **x = 3**. No fim desse bloco de comandos, a terceira variável *x* é destruída, o que torna novamente visível a segunda variável *x*, a qual é impressa na tela pela quarta chamada do comando **printf()** (linha 12).



Como o escopo é um assunto delicado e que pode gerar muita confusão, evita-se o uso de variáveis com o mesmo nome.

## 2.4 CONSTANTES

Nós aprendemos que uma variável é uma posição de memória onde podemos guardar um determinado dado ou valor e modificá-lo ao longo da execução do programa. Já uma constante permite guardar um determi-

nado dado ou valor na memória do computador, mas com a certeza de que ele não se altera durante a execução do programa: será sempre o mesmo, portanto **constante**.



Para constantes é obrigatória a atribuição do valor no momento da declaração.

Isso ocorre por que após a declaração de uma constante, seu valor não poderá mais ser alterado: será **constante**. Na linguagem C existem duas maneiras para criar constantes: usando os comandos **#define** e **const**. Além disso, a própria linguagem C já possui algumas constantes pré-definidas, como as **sequências de escape**.

#### 2.4.1 COMANDO #DEFINE

Uma das maneiras de declarar uma constante é usando o comando **#define**, que segue a seguinte forma geral:

```
#define nome_da_constante valor_da_constante
```

O comando **#define** é uma diretiva de compilação que informa ao compilador que ele deve procurar por todas as ocorrências da palavra definida por **nome\_da\_constante** e substituir por **valor\_da\_constante** quando o programa for compilado. Por exemplo, uma constante que represente o valor de  $\pi$  pode ser declarada como apresentado a seguir:

```
#define PI 3.1415
```

#### 2.4.2 COMANDO CONST

Uma outra maneira de declarar uma constante é usando o comando **const**, que segue a seguinte forma geral:

```
const tipo_da_constante nome_da_constante = valor_da_constante;
```

Note que a forma geral do comando **const** se parece muito com a da declaração de uma variável. Na verdade, o prefixo **const** apenas informa

ao programa que a variável declarada não poderá ter seu valor alterado. E, sendo uma variável, esta constante está sujeita as mesmas regras que regem o uso das variáveis. Por exemplo, uma constante que represente o valor de  $\pi$  pode ser declarada como apresentado a seguir:

```
const float PI = 3.1415;
```

### 2.4.3 SEQÜÊNCIAS DE ESCAPE

A linguagem C possui algumas constantes pré-definidas, como as **sequências de escape** ou códigos de barra invertida. Essas constantes de escape permitem o envio de caracteres de controle não gráficos para dispositivos de saída.

A tabela abaixo apresenta uma relação das sequências de escape mais utilizadas em programação e seu significado:

Código	Comando
<code>\a</code>	bip
<code>\b</code>	retorcesso (backspace)
<code>\n</code>	nova linha (new line)
<code>\v</code>	tabulação vertical
<code>\t</code>	tabulação horizontal
<code>\r</code>	retorno de carro (carriage return)
<code>\'</code>	apóstrofe
<code>\"</code>	aspa
<code>\\</code>	barra invertida (backslash)
<code>\f</code>	alimentação de folha (form feed)

As sequências de escape permitem que o comando **printf()** imprima caracteres especiais na tela de saída, como tabulações e quebras de linha. Veja o exemplo abaixo:

### Exemplo: seqüências de escape

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     printf(‘‘Hello World\n’’);
5     printf(‘‘Hello\nWorld\n’’);
6     printf(‘‘Hello \\ World\n’’);
7     printf(‘‘\”Hello World\n’’);
8     system(‘‘pause’’);
9     return 0;
10 }
```

Saída

```
Hello World
Hello
World
Hello \World
“Hello World”
```

## 2.5 OPERADORES

### 2.5.1 OPERADOR DE ATRIBUIÇÃO: “=”

Uma das operações mais utilizadas em programação é a operação de atribuição “=”. Ela é responsável por armazenar um determinado valor em uma variável. Em linguagem C, o uso do operador de atribuição “=” segue a seguinte forma geral

`nome_da_variável = expressão;`

Por expressão, entende-se qualquer combinação de **valores**, **variáveis**, **constantes** ou **chamadas de funções** utilizando os operadores matemáticos +, -, \*, / e %, que resulte numa resposta do mesmo tipo da variável definida por **nome\_da\_variável**. Veja o exemplo abaixo:

### Exemplo: operador de atribuição "="

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 const int z = 9;
5 int main(){
6     float x;
7     //declara y e atribui um valor
8     float y = 3;
9     //atribui um valor a x
10    x = 5;
11    printf('x = %f\n',x);
12    //atribui uma constante a x
13    x = z;
14    printf('x = %f\n',x);
15    //atribui o resultado de uma
16    //expressão matemática a x
17    x = y + 5;
18    printf('x = %f\n',x);
19    //atribui o resultado de uma função a x
20    x = sqrt(9);
21    printf('x = %f\n',x);
22    system('pause');
23    return 0;
24 }
```

Saída

```
x = 5.000000
x = 9.000000
x = 8.000000
x = 3.000000
```

No exemplo acima, pode-se notar que o operador de atribuição também pode ser utilizado no momento da declaração da variável (linha8). Desse modo, a variável já é declarada possuindo um valor inicial.



O operador de atribuição “=” armazena o valor ou resultado de uma expressão contida a sua **direita** na variável especificada a sua **esquerda**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 const int z = 9;
5 int main(){
6     float x;
7     float y = 3;
8     //Correto
9     x = y + 5;
10    //ERRADO
11    y + 5 = x;
12    //Correto
13    x = 5;
14    //ERRADO
15    5 = x;
16    system( ' 'pause' ' );
17    return 0;
18 }
```

É importante ter sempre em mente que o operador de atribuição “=” calcula a expressão à direita do operador “=” e atribui esse valor à variável à esquerda do operador, nunca o contrário.



A linguagem C suporta múltiplas atribuições.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     float x, y, z;
5     x = y = z = 5;
6     printf( ' 'x = %f\n' ',x );
7     printf( ' 'y = %f\n' ',y );
8     printf( ' 'z = %f\n' ',z );
9     system( ' 'pause' ' );
10    return 0;
11 }
```

Saída    x = 5.000000  
         y = 5.000000  
         z = 5.000000

No exemplo acima, o valor 5 é copiado para a variável z. Lembre-se, o valor da **direita** é sempre armazenado na variável especificada a sua **esquerda**. Em seguida, o valor de z é copiado para a variável y e, na sequência, o valor de y é copiado para x.

A linguagem C também permite a atribuição entre tipos básicos diferentes. O compilador **converte** automaticamente o valor do lado direito para o tipo do lado esquerdo do comando de atribuição “=”. Durante a etapa de conversão de tipos, pode haver perda de informação.



Na conversão de tipos, durante a atribuição, pode haver perda de informação.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int x = 65;
5     char ch;
6     float f = 25.1;
7     //ch recebe 8 bits menos significativos de x
8     //converte para a tabela ASCII
9     ch = x;
10    printf('ch = %c\n',ch);
11    //x recebe parte apenas a parte inteira de f
12    x = f;
13    printf('x = %d\n',x);
14    //f recebe valor 8 bits convertido para real
15    f = ch;
16    printf('f = %f\n',f);
17    //f recebe o valor de x
18    f = x;
19    printf('f = %f\n',f);
20    system('pause');
21    return 0;
22 }
```

Saída

```
ch = A
x = 25
f = 65.000000
f = 25.000000
```

## 2.5.2 OPERADORES ARITMÉTICOS

Os operadores aritméticos são aqueles que operam sobre números (**valores**, **variáveis**, **constantes** ou **chamadas de funções**) e/ou expressões e tem

como resultado valores numéricos. A linguagem C possui um total de cinco operadores aritméticos, como mostra a tabela abaixo:

Operador	Significado	Exemplo
+	adição de dois valores	$z = x + y$
-	subtração de dois valores	$z = x - y$
*	multiplicação de dois valores	$z = x * y$
/	quociente de dois valores	$z = x / y$
%	resto de uma divisão	$z = x \% y$

Note que os operadores aritméticos são sempre usados em conjunto com o operador de atribuição. Afinal de contas, alguém precisa receber o resultado da expressão aritmética.



O operador de subtração também pode ser utilizado para inverter o sinal de um número.

De modo geral, os operadores aritméticos são operadores binários, ou seja, atuam sobre dois valores. Mas os operadores de adição e subtração também podem ser aplicados sobre um único valor. Nesse caso, eles são chamados de operadores unários. Por exemplo, na expressão:

$x = -y;$

a variável  $x$  receberá o valor de  $y$  multiplicado por  $-1$ , ou seja,  $x = (-1) * y;$



Numa operação utilizando o operador de quociente /, se ambos **numerador** e **denominador** forem números **inteiros**, por padrão o compilador irá retornar apenas a parte inteira da divisão.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     float x;
5     x = 5/4;
6     printf('x = %f\n', x);
7     x = 5/4.0;
8     printf('x = %f\n', x);
9     system('pause');
10    return 0;
11 }
```

Saída    x = 1.000000  
          x = 1.250000

No exemplo acima, a primeira divisão (linha 5) possui apenas operandos inteiros. Logo o resultado é um valor inteiro. Já a segunda divisão (linha 7), o número quatro é definido como real (4.0). Portanto, o compilador considera essa divisão como tendo resultado real.



O operador de resto da divisão (%) só é válido para valores inteiros (tipo **int**).

### 2.5.3 OPERADORES RELACIONAIS

Os operadores relacionais são aqueles que operam sobre dois valores (**valores**, **variáveis**, **constantes** ou **chamadas de funções**) e/ou expressões e verificam a magnitude (quem é maior ou menor) e/ou igualdade entre eles.



Os operadores relacionais são operadores de comparação de valores.

A linguagem C possui um total de seis operadores relacionais, como mostra a tabela abaixo:

Operador	Significado	Exemplo
>	Maior do que	$x > 5$
>=	Maior ou igual a	$x >= 10$
<	Menor do que	$x < 5$
<=	Menor ou igual a	$x <= 10$
==	Igual a	$x == 0$
!=	Diferente de	$x != 0$

Como resultado, esse tipo de operador retorna:

- o valor **UM** (1), se a expressão relacional for considerada **verdadeira**;
- o valor **ZERO** (0), se a expressão relacional for considerada **falsa**.



Não existem os operadores relacionais: “=<”, “=>” e “<>”.

Os símbolos “=<” e “=>” estão digitados na ordem invertida. O correto é “<=” (menor ou igual a) e “>=” (maior ou igual a). Já o símbolo “<>” é o operador de diferente da linguagem Pascal, não da linguagem C. O correto é “!=”.



Não confunda o operador de atribuição “=” com o operador de comparação “==”.

Esse é um erro bastante comum quando se está programando em linguagem C. O operador de atribuição é definido por **UM** símbolo de igual “=”, enquanto o operador de comparação é definido por **DOIS** símbolos de igual “==”. Se você tentar colocar o operador de comparação em uma operação de atribuição, o compilador acusará um erro. O mesmo não acontece se você acidentalmente colocar o operador de atribuição “=” no lugar do operador de comparação “==”.

O exemplo abaixo apresenta o resultado de algumas expressões relacionais:

### Exemplos de expressões relacionais

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4 int x = 5;
5 int y = 3;
6 printf ( ' 'Resultado: %d\n' , x > 4 ); // verdadeiro
   (1)
7 printf ( ' 'Resultado: %d\n' , x == 4 ); // falso (0)
8 printf ( ' 'Resultado: %d\n' , x != y ); // verdadeiro
   (1)
9 printf ( ' 'Resultado: %d\n' , x != y+2 ); // falso
   (0)
10 system ( ' 'pause' ' );
11 return 0;
12 }
```

Saída      Resultado: 1  
            Resultado: 0  
            Resultado: 1  
            Resultado: 0

### 2.5.4 OPERADORES LÓGICOS

Certas situações não podem ser modeladas apenas utilizando os operadores aritméticos e/ou relacionais. Um exemplo bastante simples disso é saber se uma determinada variável  $x$  está dentro de uma faixa de valores. Por exemplo, a expressão matemática

$$0 < x < 10$$

indica que o valor de  $x$  deve ser maior do que 0 (zero) e também menor do que 10.

Para modelar esse tipo de situação, a linguagem C possui um conjunto de 3 operadores lógicos, como mostra a tabela abaixo:

Operador	Significado	Exemplo
&&	Operador <b>E</b>	( $x \geq 0$ && $x \leq 9$ )
	Operador <b>OU</b>	( $a == 'F'$    $b != 32$ )
!	Operador <b>NEGAÇÃO</b>	!( $x == 10$ )

Esses operadores permitem representar situações lógicas, unindo duas ou mais expressões relacionais simples numa composta:

- Operador **E** (&&): a expressão resultante só é verdadeira se **ambas** as expressões unidas por esse operador também forem. Por exemplo, a expressão  $(x \geq 0 \ \&\& \ x \leq 9)$  será verdadeira somente se as expressões  $(x \geq 0)$  e  $(x \leq 9)$  forem verdadeiras;
- Operador **OU** (||): a expressão resultante é verdadeira se **alguma** das expressões unidas por esse operador também for. Por exemplo, a expressão  $(a == 'F' \ || \ b != 32)$  será verdadeira se uma de suas duas expressões,  $(a == 'F')$  ou  $(b != 32)$ , for verdadeira;
- Operador **NEGAÇÃO** (!): inverte o valor lógico da expressão a qual se aplica. Por exemplo, a expressão  $!(x == 10)$  se transforma em  $(x > 10 \ || \ x < 10)$ .

Os operadores lógicos atuam sobre valores lógicos e retornam um valor lógico:

- 1: se a expressão é verdadeira;
- 0: se a expressão é falsa.

Abaixo é apresentada a tabela verdade, onde os termos *a* e *b* representam duas expressões relacionais:

Tabela verdade					
a	b	!a	!b	a&&b	a  b
0	0	1	1	0	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	1	1

### 2.5.5 OPERADORES BIT-A-BIT

A linguagem C permite que se faça operações lógicas “bit-a-bit” em números. Na memória do computador um número é sempre representado por sua forma binária. Assim o número **44** é representado pelo seguinte conjunto de 0’s e 1’s na memória: **00101100**. Os operadores bit-a-bit permitem que o programador faça operações em cada bit do número.



Os operadores bit-a-bit ajudam os programadores que queiram trabalhar com o computador em “baixo nível”.

A linguagem C possui um total de seis operadores bit-a-bit, como mostra a tabela abaixo:

Operador	Significado	Exemplo
~	complemento bit-a-bit	~ x
&	E bit-a-bit	x & 167
	OU bit-a-bit	x   129
^	OU exclusivo	x ^ 167
<<	deslocamento de bits à esquerda	x << 2
>>	deslocamento de bits à direita	x >> 2

Na tabela acima, temos que os operadores ~, &, |, e ^ são operações lógicas que atuam em cada um dos bits do número (por isso, bit-a-bit). Já os operadores de deslocamento << e >> servem para rotacionar o conjunto de bits do número à esquerda ou à direita.



Os operadores bit-a-bit só podem ser usados nos tipos **char**, **int** e **long**.

Os operadores bit-a-bit não podem ser aplicados sobre valores dos tipos **float** e **double**. Em parte, isso se deve a maneira como um valor real, também conhecido como **ponto flutuante**, é representado nos computadores. A representação desses tipos segue a representação criada por Konrad Zuse, onde um número é dividido numa *mantissa* (**M**) e um *exponente* (**E**). O valor representado é obtido pelo produto:  $M2^E$ . Como se vê, a representação desses tipos é bem mais complexa: não se trata de apenas um conjunto de 0's e 1's na memória.

Voltemos ao número **44**, cuja representação binária é **00101100**. Abaixo são apresentados exemplos de operações bit-a-bit com esse valor:

### Exemplos de operadores bit-a-bit

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     unsigned char x, y;
5     x = 44;
6     printf('x = %d\n',x);
7     y = ~x;
8     printf('~x = %d\n',y);
9     y = x & 167;
10    printf('x & 167 = %d\n',y);
11    y = x | 129;
12    printf('x | 129 = %d\n',y);
13    y = x ^ 167;
14    printf('x ^ 167 = %d\n',y);
15    y = x << 2;
16    printf('x << 2 = %d\n',y);
17    y = x >> 2;
18    printf('x >> 2 = %d\n',y);
19    system('pause');
20    return 0;
21 }
```

Saída

```
x = 44
~x = 211
x & 167 = 36
x | 129 = 173
x ^ 167 = 139
x << 2 = 176
x >> 2 = 11
```

No exemplo acima, a primeira operação é a de complemento bit-a-bit “~”. Basicamente, essa operação inverte o valor dos 0’s e 1’s que compõem o número. Assim:

```
00101100 = x (44)
11010011 = ~x (211)
```

Já os operadores &, |, e ^ são as operações lógicas de E, OU e OU EXCLUSIVO realizadas bit-a-bit:

- Operador **E bit-a-bit (&)**: um bit terá valor 1 na expressão resultante somente se **ambas** as expressões unidas por esse operador também tiverem o valor 1 nos bits daquela posição:

00101100 = x (44)  
10100111 = 167  
00100100 = x & 167 (36)

- Operador **OU bit-a-bit** (|): um bit terá valor 1 na expressão resultante se **alguma** das expressões unidas por esse operador também tiverem o valor 1 nos bits daquela posição:

00101100 = x (44)  
10000001 = 129  
10101101 = x | 129 (173)

- Operador **OU EXCLUSIVO bit-a-bit** (^): um bit terá valor 1 na expressão resultante somente se **ambas** as expressões unidas por esse operador tiverem o valores de bits **diferentes** naquela posição:

00101100 = x (44)  
10100111 = 167  
10001011 = x ^ 167 (139)

Por fim, os operadores de deslocamento << e >> servem simplesmente para mover bits para a esquerda para a direita. Cada movimentação de bits equivale a multiplicar ou dividir (divisão inteira) por 2. Assim:

00101100 = x (44)  
10110000 = x << 2 (176)  
00001011 = x >> 2 (11)

## 2.5.6 OPERADORES DE ATRIBUIÇÃO SIMPLIFICADA

Como vimos anteriormente, muitos operadores são sempre usados em conjunto com o operador de atribuição. Para tornar essa tarefa mais simples, a linguagem C permite simplificar algumas expressões, como mostra a tabela abaixo:

Operador	Significado	Exemplo	
+=	soma e atribui	x += y	igual x = x + y
-=	subtrai e atribui	x -= y	igual x = x - y
*=	multiplica e atribui	x *= y	igual x = x * y
/=	divide e atribui quociente	x /= y	igual x = x / y
%=	divide e atribui resto	x %= y	igual x = x % y
&=	E bit-a-bit e atribui	x &= y	igual x = x & y
=	OU bit-a-bit e atribui	x  = y	igual x = x   y
^=	OU exclusivo e atribui	x ^= y	igual x = x ^ y
<<=	desloca à esquerda e atribui	x <<= y	igual x = x << y
>>=	desloca à direita e atribui	x >>= y	igual x = x >> y

Como se pode notar, esse tipo de operador é muito útil quando a variável que vai receber o resultado da expressão é também um dos operandos da expressão. Por exemplo, a expressão

```
x = x + 10 * y;
```

pode ser reescrita usando o operador simplificado como sendo

```
x += 10 * y;
```

## 2.5.7 OPERADORES DE PRÉ E PÓS-INCREMENTO

Além dos operadores simplificados, a linguagem C também possui operadores de **pré** e **pós**-incremento. Estes operadores podem ser utilizados sempre que for necessário necessário **incrementar** (somar uma unidade) ou **decrementar** (subtrair uma unidade) um determinado valor, como mostra a tabela abaixo:

Operador	Significado	Exemplo
++	pré ou pós incremento	++x ou x++
--	pré ou pós decremento	--x ou x--

Tanto o operador de **incremento** (++) quanto o de **decremento** (- -) já possui embutida uma operação de atribuição. Note, no entanto, que esse operador pode ser usado antes ou depois do nome da variável, com uma diferença significativa:

- **++x**: incrementa a variável x **antes** de utilizar seu valor;

- **x++**: incrementa a variável x **depois** de utilizar seu valor.

Essa diferença de sintaxe no uso do operador não tem importância se o operador for usado sozinho, como mostra o exemplo abaixo:

Exemplo de pós e pré incremento sozinho	
Pré incremento	Pós incremento
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int x = 10; 5     ++x; 6     printf('x = %d\n', 7           x); 8     system('pause'); 9     return 0; 10 } </pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int x = 10; 5     x++; 6     printf('x = %d\n', 7           x); 8     system('pause'); 9     return 0; 10 } </pre>
Saída x = 11	x = 11

Porém, se esse operador for utilizado dentro de uma expressão aritmética, como no exemplo abaixo, a diferença é entre os dois operadores é evidente:

Exemplo de pós e pré incremento numa expressão	
Pré incremento	Pós incremento
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int y,x = 10; 5     //incrementa, depois 6     //atribui 7     y = ++x; 8     printf('x = %d\n', 9           x); 10    printf('y = %d\n', 11          y); 12    system('pause'); 13    return 0; 14 } </pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int y,x = 10; 5     //atribui, depois 6     //incrementa 7     y = x++; 8     printf('x = %d\n', 9           x); 10    printf('y = %d\n', 11          y); 12    system('pause'); 13    return 0; 14 } </pre>
Saída x = 11 y = 11	x = 11 y = 10

Como se pode ver, no primeiro exemplo o operador de pré-incremento (**++x**) é a primeira coisa a ser realizada dentro da expressão. Somente depois de incrementado o valor de *x* é que o mesmo é atribuído a variável *y*. Nota-se, nesse caso, que a expressão

```
y = ++x;
```

é equivalente a fazer

```
x = x + 1;  
y = x;
```

Já no segundo exemplo, o operador de pós-incremento (**x++**) é a última coisa a ser realizada dentro da expressão. Primeiro atribui-se o o valor de *x* para a variável *y* para somente depois incrementar a variável *x*. Nota-se, nesse caso, que a expressão

```
y = x++;
```

é equivalente a fazer

```
y = x;  
x = x + 1;
```

### 2.5.8 MODELADORES DE TIPOS (CASTS)

Modeladores de tipos (também chamados de *type cast*) são uma forma explícita de conversão de tipo, onde o tipo a ser convertido é explicitamente definido dentro de um programa. Isso é diferente da conversão implícita, que ocorre naturalmente quando tentamos atribuir um número real para uma variável inteira. Em linguagem C, o uso de um modelador de tipo segue a seguinte forma geral:

**(nome\_do\_tipo)** expressão

Um modelador de tipo é definido pelo próprio **nome\_do\_tipo** entre parêntese. Ele é colocado a frente de uma expressão e tem como objetivo forçar o resultado da expressão a ser de um tipo especificado, como mostra o exemplo abaixo:

### Exemplos de modeladores de tipo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     float x,y,f = 65.5;
5     x = f/10.0;
6     y = (int) (f/10.0);
7     printf('x = %f\n',x);
8     printf('y = %f\n',y);
9     system('pause');
10    return 0;
11 }
```

Saída    x = 6.550000  
          y = 6.000000

No exemplo acima, tanto os valores de  $x$  quanto de  $y$  são obtidos utilizando a mesma expressão. Porém, no caso da variável  $y$  (linha 6), o resultado da expressão é convertido para o tipo inteiro (**int**), o que faz com que seu resultado perca as casas decimais.

#### 2.5.9 OPERADOR VÍRGULA “,”

Na linguagem C, o operador vírgula “,” pode ser utilizado de duas maneiras:

- Como pontuação. Por exemplo, para separar argumentos de uma função:

```
int minha_funcao(int a, float b)
```

- Determinar uma lista de expressões que devem ser executadas sequencialmente.

```
x = (y = 2, y + 3);
```

Nesse caso, as expressões são executadas da esquerda para a direita: o valor 2 é atribuído a  $y$ , o valor 3 é somado a  $y$  e o total (5) será atribuído à variável  $x$ . Pode-se encadear quantos operadores “,” forem necessários.



Na linguagem C, o operador “,” é um separador de comandos, enquanto o operador “;” é um terminador de comandos.

## 2.5.10 PRECEDÊNCIA DE OPERADORES

Como podemos ver, a linguagem C contém muitos operadores. Consequentemente, o uso de múltiplos operadores em uma única expressão pode tornar confusa a sua interpretação. Por esse motivo, a linguagem C possui uma série de regras de precedência de operadores. Isso permite que o compilador possa decidir corretamente qual a ordem em que os operadores deverão ser executados em uma expressão contendo vários operadores. As regras de precedência seguem basicamente as regras da matemática, onde a multiplicação e a divisão são executadas antes da soma e da subtração. Além disso, pode-se utilizar de parênteses para forçar o compilador a executar uma parte da expressão antes das demais.

A tabela abaixo mostra as regras de precedência dos operadores presentes na linguagem C. Quanto mais alto na tabela, maior o nível de precedência (prioridade) dos operadores em questão. Na primeira linha da tabela são apresentados os operadores executados em primeiro lugar, enquanto a última linha apresenta os operadores executados por último em uma expressão:

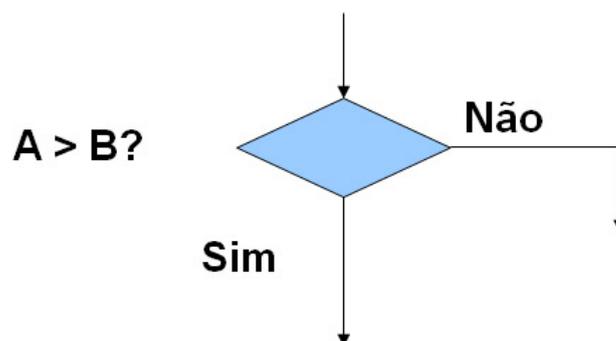
<b>MAIOR PRECEDÊNCIA</b>	
++ -	Pré incremento/decremento
()	Parênteses (chamada de função)
[ ]	Elemento de array
.	Elemento de struct
-i	Conteúdo de elemento de ponteiro para struct
++ -	Pós incremento/decremento
+ -	Adição e subtração unária
! ~	Não lógico e complemento bit-a-bit
(tipo)	Conversão de tipos ( <i>type cast</i> )
*	Acesso ao conteúdo de ponteiro
&	Endereço de memória do elemento
sizeof	Tamanho do elemento
* / %	Multiplicação, divisão, e módulo (resto)
+ -	Adição e subtração
<< >>	Deslocamento de bits à esquerda e à direita
< <=	“Menor do que” e “menor ou igual a”
> >=	“Maior do que” e “maior ou igual a”
== !=	“Igual a” e “Diferente de”
&	E bit-a-bit
^	OU exclusivo
	OU bit-a-bit
&&	E lógico
	OU lógico
?:	Operador ternário
=	Atribuição
+= -=	Atribuição por adição ou subtração
*= /= %=	Atribuição por multiplicação, divisão ou módulo (resto)
<<= >>=	Atribuição por deslocamento de bits
&= ^=  =	Atribuição por operações lógicas
,	Operador vírgula
<b>MENOR PRECEDÊNCIA</b>	

É possível notar que alguns operadores ainda são desconhecidos para nós, apesar de alguns possuírem o mesmo símbolo usado para outro operador (como é o caso do operador de *acesso ao conteúdo de ponteiro*, o qual possui o mesmo símbolo do operador de multiplicação “\*”). Esses operadores serão explicados ao longo da apostila, conforme surja a necessidade de utilizá-los.

### 3 COMANDOS DE CONTROLE CONDICIONAL

Os programas escritos até o momento são programas sequenciais: um comando é executado após o outro, do começo ao fim do programa, na ordem em que foram declarados no código fonte. Nenhum comando é ignorado.

Entretanto, há casos em que é preciso que um bloco de comandos seja executado somente se uma determinada condição for verdadeira. Para isso, precisamos de uma estrutura de seleção, ou um comando de controle condicional, que permita selecionar o conjunto de comandos a ser executado. Isso é muito similar ao que ocorre em um fluxograma, onde o símbolo do losango permitia escolher entre diferentes caminhos com base em uma condição do tipo verdadeiro/falso:



Nesta seção iremos ver como funcionam cada uma das estruturas de seleção presentes na linguagem C.

#### 3.1 DEFININDO UMA CONDIÇÃO

Por **condição**, entende-se qualquer expressão relacional (ou seja, que use os operadores  $>$ ,  $<$ ,  $>=$ ,  $<=$ ,  $==$  ou  $!=$ ) que resulte numa resposta do tipo **verdadeiro** ou **falso**. Por exemplo, para a condição  $x > 0$  temos que:

- Se o valor de  $x$  for um valor **POSITIVO**, a condição será considerada **verdadeira**;
- Se o valor de  $x$  igual a **ZERO** ou **NEGATIVO**, a condição será considerada **falsa**.

Já uma expressão condicional é qualquer expressão que resulte numa resposta do tipo **verdadeiro** ou **falso**. Ela pode ser construída utilizando operadores:

- Matemáticos : +, -, \*, /, %
- Relacionais: >, <, >=, <=, ==, !=
- Lógicos: &&, ||

Esses operadores permitem criar condições mais complexas, como mostra o exemplo abaixo, onde se deseja saber se a divisão de **x** por 2 é maior do que o valor de **y** menos 3:

$$x/2 > y - 3$$



Uma expressão condicional pode utilizar operadores dos tipos: matemáticos, relacionais e/ou lógicos.

x é maior ou igual a y?

`x >= y`

x é maior do que y+2?

`x > y+2`

x-5 é diferente de y+3?

`x-5 != y+3`

x é maior do que y e menor do que z?

`(x > y) && (x < z)`

Quando o compilador avalia uma condição, ele quer um valor de retorno (**verdadeiro** ou **falso**) para poder tomar a decisão. No entanto, esta expressão condicional não necessita ser uma expressão no sentido convencional.



Uma variável sozinha pode ser uma “expressão condicional” e retornar o seu próprio valor.

Para entender isso, é importante lembrar que o computador trabalha, internamente, em termos de 0's e 1's. Assim, se uma condição

- é considerada **FALSA**, o computador considera que a condição possui valor **ZERO**;
- é considerada **VERDADEIRA**, o computador considera que a condição possui valor **DIFERENTE DE ZERO**.

Isto significa que o valor de uma variável do tipo inteiro pode ser a resposta de uma expressão condicional:

- se o valor da variável for igual a **ZERO**, a condição é **FALSA**;
- se o valor da variável for **DIFERENTE DE ZERO**, a condição é **VERDADEIRA**.

Abaixo é possível ver algumas expressões que são consideradas equivalentes pelo compilador:

Se a variável possui valor **DIFERENTE DE ZERO**...

(num != 0)

...ela sozinha retorna um valor que é considerado **VERDADEIRO** pelo computador.

(num)

e

Se a variável possui valor igual a **ZERO**...

(num == 0)

...sua negação retorna um valor que é considerado **VERDADEIRO** pelo computador.

(!num)

## 3.2 COMANDO IF

Na linguagem C, o comando **if** é utilizado sempre que é necessário escolher entre dois caminhos dentro do programa, ou quando se deseja executar um ou mais comandos que estejam sujeitos ao resultado de um teste.

A forma geral de um comando **if** é:

```
if (condição) {  
    sequência de comandos;  
}
```

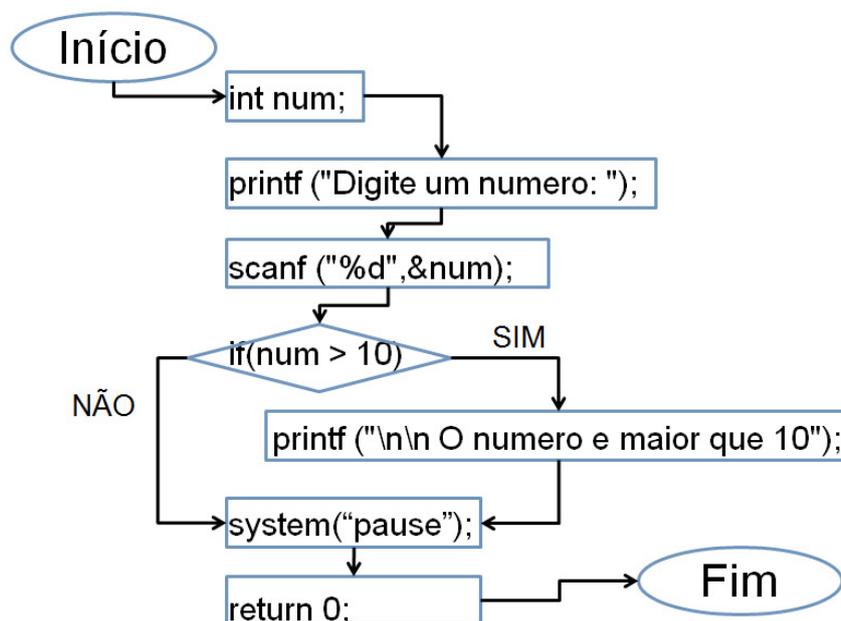
Na execução do comando **if** a condição será avaliada e:

- se a condição for **verdadeira** a sequência de comandos será executada;
- se a condição for **falsa** a sequência de comandos não será executada, e o programa irá continuar a partir do primeiro comando seguinte ao final do comando **if**.

Abaixo, tem-se um exemplo de um programa que lê um número inteiro digitado pelo usuário e informa se o mesmo é maior do que 10:

Exemplo: comando if	
1	<b>#include</b> <stdio.h>
2	<b>#include</b> <stdlib.h>
3	<b>int</b> main () {
4	<b>int</b> num;
5	printf ( ' ' Digite um numero: ' ' );
6	scanf ( ' ' %d ' ', &num );
7	<b>if</b> ( num > 10 )
8	printf ( ' ' O numero e maior do que 10 \n ' ' );
9	
10	system ( ' ' pause ' ' );
11	<b>return</b> 0;
12	}

No exemplo acima, a mensagem de que o número é maior do que 10 será exibida apenas se a condição for verdadeira. Se a condição for falsa, nenhuma mensagem será escrita na tela. Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



Diferente da maioria dos comandos, não se usa o ponto e vírgula (;) depois da condição do comando if.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int num;
5     printf ( ' ' Digite um numero: ' ' );
6     scanf ( ' ' %d ' ', &num );
7     if ( num > 10 ); //ERRADO
8         printf ( ' ' O numero e maior que 10\\n ' ' );
9     system ( ' ' pause ' ' );
10    return 0;
11 }
  
```

Na linguagem C, o operador ponto e vírgula (;) é utilizado para separar as instruções do programa. Colocá-lo logo após o comando if, como exemplificado acima, faz com que o compilador entenda que o comando if já terminou e trate o comando seguinte (**printf**) como se o mesmo estivesse fora do if. No exemplo acima, a mensagem de que o número é maior que 10 será exibida independente do valor do número.



O compilador não irá acusar um erro se colocarmos o operador ponto e vírgula (;) após o comando `if`, mas a lógica do programa poderá estar errada.

### 3.2.1 USO DAS CHAVES { }

No comando `if`, e em diversos outros comandos da linguagem C, usa-se os operadores de chaves { } para delimitar um bloco de instruções.



Por definição, comandos de condição (`if` e `else`) ou repetição (`while`, `for` e `do while`) atuam apenas sobre o comando seguinte a eles.

Desse modo, se o programador desejar que mais de uma instrução seja executada por aquele comando `if`, esse conjunto de instruções deve estar contido dentro de um bloco delimitado por chaves { }.

```
if (condição) {  
    comando 1;  
    comando 2;  
    ...  
    comando n;  
}
```



As chaves podem ser ignoradas se o comando contido dentro do `if` for único.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int num;
5     printf ( ' ' Digite um numero: ' ' );
6     scanf ( ' ' %d ' ', &num );
7     if ( num > 10 )
8         printf ( ' ' O numero e maior que 10 \n ' ' );
9
10    /*OU
11     if ( num > 10 ) {
12         printf ( ' ' O numero e maior que 10 \n ' ' );
13     }
14    */
15    system ( ' ' pause ' ' );
16    return 0;
17 }
```

### 3.3 COMANDO ELSE

O comando **else** pode ser entendido como sendo um complemento do comando **if**. Ele auxilia o comando **if** na tarefa de escolher dentre os vários caminhos a ser seguido dentro do programa.



O comando *else* é opcional e sua sequência de comandos somente será executada se o valor da condição que está sendo testada pelo comando **if** for **FALSA**.

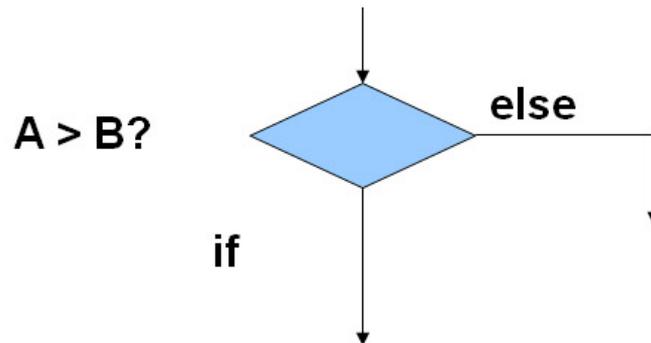
A forma geral de um comando `else` é:

```
if (condição) {
    primeira sequência de comandos;
}
else {
    segunda sequência de comandos;
}
```



Se o comando **if** diz o que fazer quando a condição é verdadeira, o comando **else** trata da condição quando ela é falsa.

Isso fica bem claro quando olhamos a representação do comando **else** em um fluxograma:



Antes, na execução do comando **if**, a condição era avaliada e:

- se a condição fosse **verdadeira**, a **primeira** sequência de comandos era executada;
- se a condição fosse **falsa**, a sequência de comandos não era executada e o programa seguia o seu fluxo padrão.

Com o comando **else**, temos agora que:

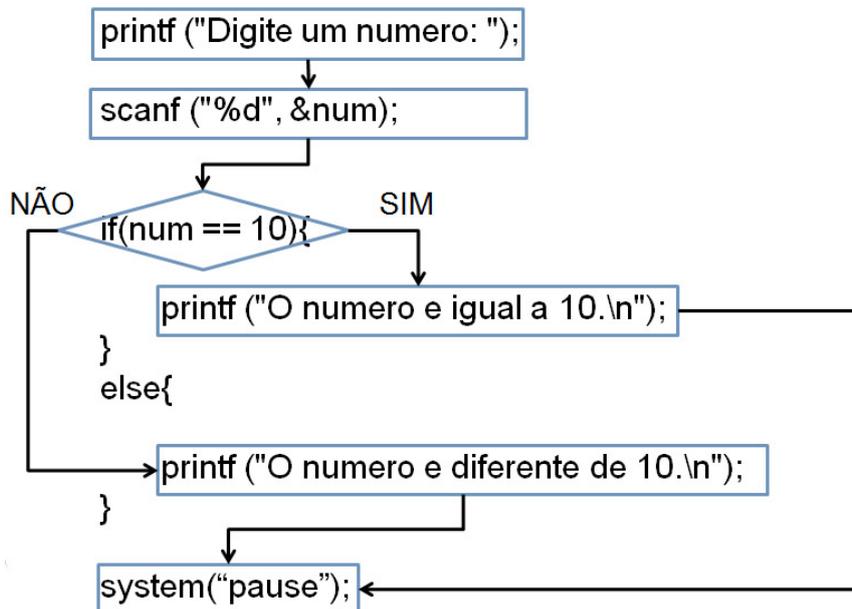
- se a condição for **verdadeira**, a **primeira** sequência de comandos (bloco **if**) será executada;
- se a condição for **falsa**, a **segunda** sequência de comandos (bloco **else**) será executada.

Abaixo, tem-se um exemplo de um programa que lê um número inteiro digitado pelo usuário e informa se o mesmo é ou não igual a 10:

### Exemplo: comando if-else

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     int num;
5     printf ("Digite um numero: ");
6     scanf ("%d", &num);
7     if (num == 10){
8         printf ("O numero e igual a 10.\n");
9     } else{
10        printf ("O numero e diferente de 10.\n");
11    }
12    system ("pause");
13    return 0;
14 }
```

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:





O comando **else** não tem condição. Ele é o caso contrário da condição do **if**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     int num;
5     printf (‘‘Digite um numero: ’’);
6     scanf (‘‘%d’’, &num);
7     if(num == 10){
8         printf (‘‘O numero e igual a 10.\n’’);
9     } else(num != 10){//ERRO
10        printf (‘‘O numero e diferente de 10.\n’’);
11    }
12    system(‘‘pause’’);
13    return 0;
14 }
```

O comando **else** deve ser entendido como sendo um complemento do comando **if**. Ele diz quais comandos se deve executar se a condição do comando **if** for **falsa**. Portanto, não é necessário estabelecer uma condição para o comando **else**: ele é o oposto do **if**.



Como no caso do **if**, não se usa o ponto e vírgula (;) depois do comando **else**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     int num;
5     printf (‘‘Digite um numero: ’’);
6     scanf (‘‘%d’’, &num);
7     if(num == 10){
8         printf (‘‘O numero e igual a 10.\n’’);
9     } else;{//ERRADO
10        printf (‘‘O numero e diferente de 10.\n’’);
11    }
12    system(‘‘pause’’);
13    return 0;
14 }
```

Como no caso do **if**, colocar o operador de ponto e vírgula (;) logo após o comando **else**, faz com que o compilador entenda que o comando **else** já

terminou e trate o comando seguinte (**printf**) como se o mesmo estivesse fora do **else**. No exemplo acima, a mensagem de que o número é diferente de 10 será exibida independente do valor do número.



A seqüência de comandos do **if** é independente da seqüência de comandos do **else**. Cada comando tem o seu próprio conjunto de chaves **{}**.

Se o comando **if** for executado em um programa, o seu comando **else** não será executado. Portanto, não faz sentido usar o mesmo conjunto de chaves **{}** para definir os dois conjuntos de comandos.

Uso das chaves no comando if-else	
Certo	Errado
<pre>1 if (condicao){ 2   seqüência de comandos; 3 } 4 else{ 5   seqüência de comandos; 6 }</pre>	<pre>1 if (condicao){ 2   seqüência de comandos; 3 else 4   seqüência de comandos; 5 }</pre>



Como no caso do comando **if**, as chaves podem ser ignoradas se o comando contido dentro do **else** for único.

### 3.4 ANINHAMENTO DE IF

Um if aninhado é simplesmente um comando if utilizado dentro do bloco de comandos de um outro if (ou else) mais externo. Basicamente, é um comando if dentro de outro.

A forma geral de um comando if aninhado é:

```
if(condição 1) {
    seqüência de comandos;
    if(condição 2) {
        seqüência de comandos;
        if...
```

```

    }
    else{
        seqüência de comandos;
        if...
    }
} else{
    seqüência de comandos;
}

```

Em um aninhamento de if's, o programa começa a testar as condições começando pela **condição 1**. Se o resultado dessa condição for diferente de zero (verdadeiro), o programa executará o bloco de comando associados a ela. Do contrário, irá executar o bloco de comando associados ao comando else correspondente, se ele existir. Esse processo se repete para cada comando if que o programa encontrar dentro do bloco de comando que ele executar.

O aninhamento de if's é muito útil quando se tem mais do que dois caminhos para executar dentro de um programa. Por exemplo, o comando if é suficiente para dizer se um número é maior do que outro número ou não. Porém, ele sozinho é incapaz de dizer se esse mesmo número é maior, menor ou igual ao outro como mostra o exemplo abaixo:

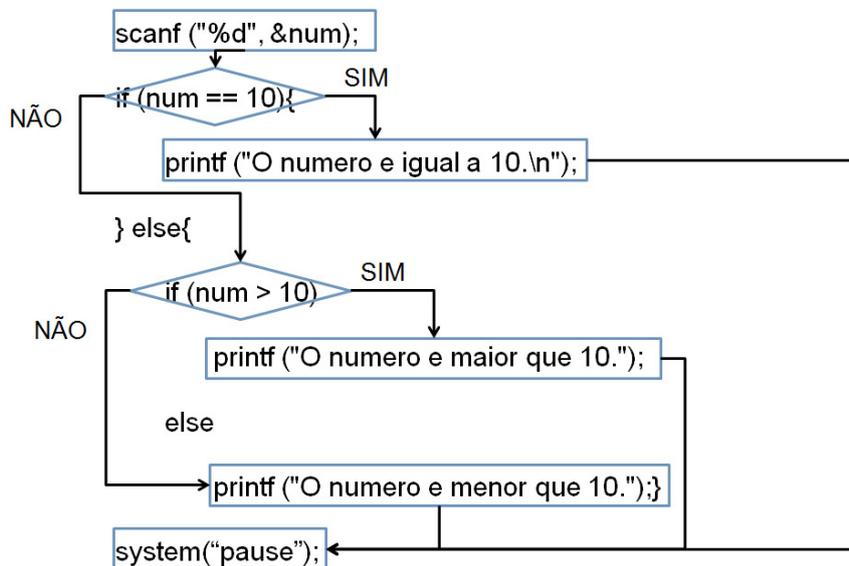
#### Exemplo: aninhamento de if

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     int num;
5     printf( "Digite um numero: " );
6     scanf( "%d", &num );
7     if ( num == 10 ){
8         printf( "O numero e igual a 10.\n" );
9     } else{
10        if ( num > 10 )
11            printf( "O numero e maior que 10.\n" );
12        else
13            printf( "O numero e menor que 10.\n" );
14    }
15    system( "pause" );
16    return 0;
17 }

```

Isso fica bem claro quando olhamos a representação do aninhamento de if's em um fluxograma:



O único cuidado que devemos ter no aninhamento de if's é o de saber exatamente a qual if um determinado else está ligado.

Esse cuidado fica claro no exemplo abaixo: apesar do comando else estar alinhado com o primeiro comando if, ele está na verdade associado ao segundo if. Isso acontece porque o comando else é sempre associado ao primeiro comando if encontrado antes dele dentro de um bloco de comandos.

```
if (cond1)
    if (cond2)
        seqüência de comandos;
else
    seqüência de comandos;
```

No exemplo anterior, para fazer com que o comando else fique associado ao primeiro comando if é necessário definir um novo bloco de comandos (usando os operadores de chaves { }) para isolar o comando if mais interno.

```

if (cond1) {
    if (cond2)
        seqüência de comandos;
    } else
        seqüência de comandos;

```



Não existe aninhamento de else's.

O comando `else` é o caso contrário da condição do comando `if`. Assim, para cada `else` deve existir um `if` anterior, porém nem todo `if` precisa ter um `else`.

```

if (cond1)
    seqüência de comandos;
else
    seqüência de comandos;
else //ERRO!
    seqüência de comandos;

```

### 3.5 OPERADOR ?

O operador `?` é também conhecido como *operador ternário*. Trata-se de uma simplificação do comando `if-else` na sua forma mais simples, ou seja, com apenas um comando e não blocos de comandos.

A forma geral do operador `?` é:

```

expressão condicional ? expressão1 : expressão2;

```

O funcionamento do operador `?` é idêntico ao do comando `if-else`: primeiramente, a *expressão condicional* será avaliada e

- se essa condição for **verdadeira**, o valor da *expressão1* será o resultado da *expressão condicional*;
- se essa condição for **falsa**, o valor da *expressão2* será o resultado da *expressão condicional*;



O operador ? é tipicamente utilizado para atribuições condicionais.

O exemplo abaixo mostra como uma expressão de atribuição pode ser simplificada utilizando o operador ternário:

Usando if-else	Usando o operador ternário
<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main () { 4     int x,y,z; 5     printf("Digite x:"); 6     scanf("%d",&amp;x); 7     printf("Digite y:"); 8     scanf("%d",&amp;y); 9     if (x &gt; y) 10        z = x; 11    else 12        z = y; 13    printf("Maior = %d", 14           z); 14    system("pause"); 15    return 0; 16 }</pre>	<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main () { 4     int x,y,z; 5     printf("Digite x:"); 6     scanf("%d",&amp;x); 7     printf("Digite y:"); 8     scanf("%d",&amp;y); 9     z = x &gt; y ? x : y; 10    printf("Maior = %d", 11           z); 11    system("pause"); 12    return 0; 13 }</pre>

O operador ? é limitado e por isso não atende a uma gama muito grande de casos que o comando if-else atenderia. Porém, ele pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular, onde uma variável é incrementada até um valor máximo e, sempre que atinge esse valor, a variável é zerada.

```
index = (index== 3) ? 0: ++index;
```



Apesar de limitado, o operador `?` não é restrito a atribuições apenas.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int num;
5     printf("Digite um numero: ");
6     scanf("%d", &num);
7     (num == 10)? printf("O numero e igual a 10.\n") : printf("O numero e diferente de 10.\n");
8     system("pause");
9     return 0;
10 }
```

### 3.6 COMANDO SWITCH

Além dos comandos `if` e `else`, a linguagem C possui um comando de seleção múltipla chamado **switch**. Esse comando é muito parecido com o aninhamento de comandos `if-else-if`.



O comando `switch` é muito mais limitado que o comando `if-else`: enquanto o comando `if` pode testar expressões lógicas ou relacionais, o comando `switch` somente verifica se uma variável (do tipo **int** ou **char**) é ou não igual a um certo valor constante.

A forma geral do comando switch é:

```
switch (variável) {  
    case valor1:  
        seqüência de comandos;  
        break;  
    case valor2:  
        seqüência de comandos;  
        break;  
    ...  
    case valorN:  
        seqüência de comandos;  
        break;  
    default:  
        seqüência de comandos; }
```



O comando switch é indicado quando se deseja testar uma variável em relação a diversos valores pré-estabelecidos.

Na execução do comando switch, o valor da *variável* é comparado, na ordem, com cada um dos valores definidos pelo comando **case**. Se um desses valores for igual ao valor da variável, a seqüência de comandos daquele comando case é executado pelo programa.

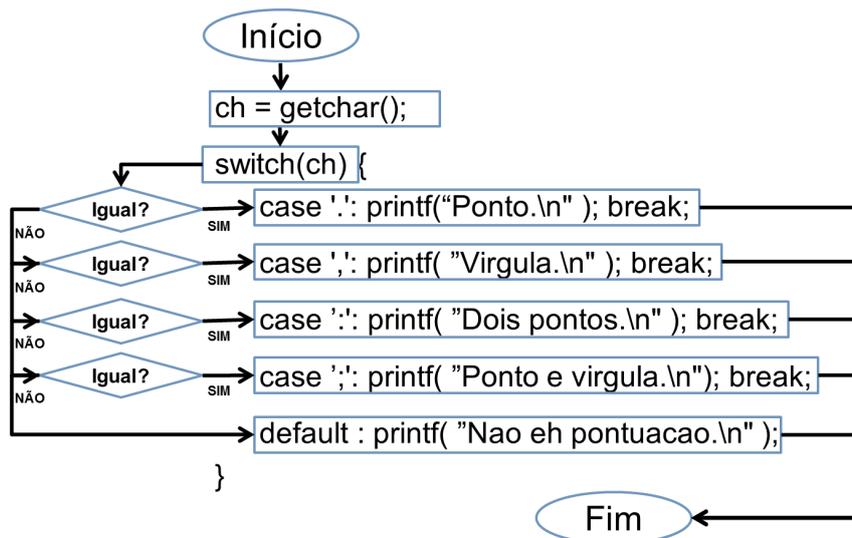
Abaixo, tem-se um exemplo de um programa que lê um caractere digitado pelo usuário e informa se o mesmo é um símbolo de pontuação:

### Exemplo: comando switch

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char ch;
5     printf("Digite um simbolo de pontuacao: ");
6     ch = getchar();
7     switch( ch ) {
8         case '.': printf("Ponto.\n"); break;
9         case ',': printf("Virgula.\n"); break;
10        case ':': printf("Dois pontos.\n"); break;
11        case ';': printf("Ponto e virgula.\n"); break;
12        default : printf("Nao eh pontuacao.\n");
13    }
14    system("pause");
15    return 0;
16 }
```

No exemplo acima, será pedido ao usuário que digite um caractere. O valor desse caractere será comparado com um conjunto de possíveis símbolos de pontuação, cada qual identificado em um comando **case**. Note que, se o caractere digitado pelo usuário não for um símbolo de pontuação, a seqüência de comandos dentro do comando default será executada.

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:





O comando *default* é opcional e sua seqüência de comandos somente será executada se o valor da variável que está sendo testada pelo comando **switch** não for igual a nenhum dos valores dos comandos **case**.

O exemplo anterior do comando **switch** poderia facilmente ser reescrito com o aninhamento de comandos **if-else-if** como se nota abaixo:

#### Exemplo: simulando o comando **switch** com **if-else-if**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char ch;
5     printf( "Digite um simbolo de pontuacao: " );
6     ch = getchar();
7     if (ch == '.')
8         printf( "Ponto.\n" );
9     else
10        if (ch == ',')
11            printf( "Virgula.\n" );
12        else
13            if (ch == ':')
14                printf( "Dois pontos.\n" );
15            else
16                if (ch == ';')
17                    printf( "Ponto e virgula.\n" );
18                else
19                    printf( "Nao eh pontuacao.\n" );
20    system( "pause" );
21    return 0;
22 }
```

Como se pode notar, o comando **switch** apresenta uma solução muito mais elegante que o aninhamento de comandos **if-else-if** quando se necessita comparar o valor de uma variável.

#### 3.6.1 USO DO COMANDO **BREAK** NO **SWITCH**

Apesar das semelhanças entre os dois comandos, o comando **switch** e o aninhamento de comandos **if-else-if**, existe uma diferença muito importante entre esses dois comandos: o comando **break**.



Quando o valor associado a um comando `case` é igual ao valor da variável do `switch` a respectiva seqüência de comandos é executada até encontrar um comando `break`. Caso o comando `break` não exista, a seqüência de comandos do `case` seguinte também será executada e assim por diante

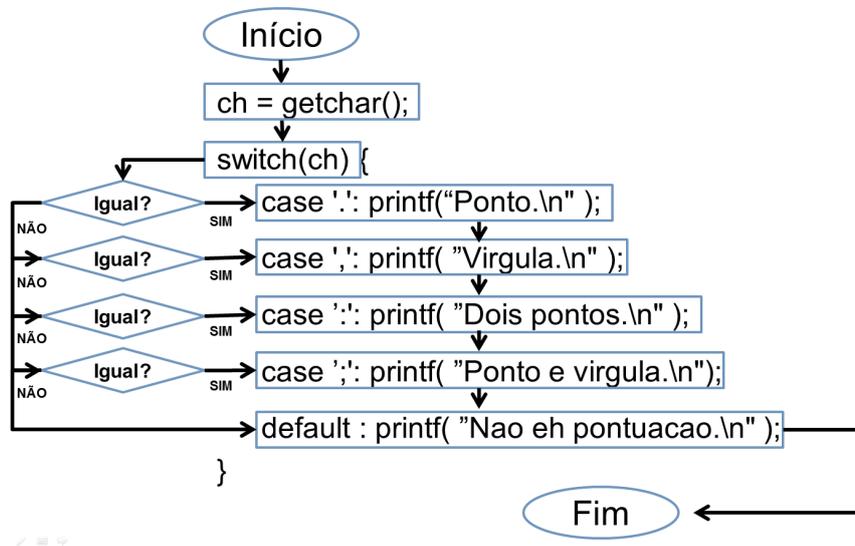
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char ch;
5     printf( "Digite um simbolo de pontuacao: " );
6     ch = getchar();
7     switch( ch ) {
8         case '.': printf( "Ponto.\n" );
9         case ',': printf( "Virgula.\n" );
10        case ':': printf( "Dois pontos.\n" );
11        case ';': printf( "Ponto e virgula.\n" );
12        default : printf( "Nao eh pontuacao.\n" );
13    }
14    system( "pause" );
15    return 0;
16 }
```

Note, no exemplo acima, que caso o usuário digite o símbolo de ponto (.) todas as mensagens serão escritas na tela de saída.



O comando **break** é **opcional** e faz com que o comando **switch** seja interrompido assim que uma das seqüência de comandos seja executada.

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



De modo geral, é quase certo que se venha a usar o comando `break` dentro do `switch`. Porém a sua ausência pode ser muito útil em algumas situações. Por exemplo, quando queremos que uma ou mais seqüências de comandos sejam executadas a depender do valor da variável do **switch**, como mostra o exemplo abaixo:

#### Exemplo: comando switch sem break

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int num;
5     printf( 'Digite um numero inteiro de 0 a 9: ' );
6     scanf( '%d', &num );
7     switch( num ) {
8         case 9: printf( 'Nove\n' );
9         case 8: printf( 'Oito\n' );
10        case 7: printf( 'Sete\n' );
11        case 6: printf( 'Seis\n' );
12        case 5: printf( 'Cinco\n' );
13        case 4: printf( 'Quatro\n' );
14        case 3: printf( 'Tres\n' );
15        case 2: printf( 'Dois\n' );
16        case 1: printf( 'Um\n' );
17        case 0: printf( 'Zero\n' );
18    }
19    system( 'pause' );
20    return 0;
21 }
  
```

Note, no exemplo acima, que caso o usuário digite o valor 9, todas as mensagens serão escritas na tela de saída. Caso o usuário digite o valor 5, apenas as mensagens desse **case** e as abaixo dele serão escritas na tela de saída.

### 3.6.2 USO DAS CHAVES {} NO CASE

De modo geral, a sequência de comandos do **case** não precisam estar entre chaves {}.



Porém, se o primeiro comando dentro de um **case** for a declaração de uma variável, será necessário colocar todos os comandos desse **case** dentro de um par de chaves {}.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char ch;
5     int a,b;
6     printf( "Digite uma operacao matematica: " );
7     ch = getchar();
8     printf( "Digite dois numeros inteiros: " );
9     scanf( "%d%d", &a,&b );
10    switch( ch ) {
11        case '+': {
12            int c = a + b;
13            printf( "Soma: %d\n", c );
14            break;
15        case '-': {
16            int d = a - b;
17            printf( "Subtracao: %d\n", d );
18            break;
19        case '*': {
20            int e = a * b;
21            printf( "Produto: %d\n", e );
22            break;
23        case '/': {
24            int f = a / b;
25            printf( "Divisao: %d\n", f );
26            break;
27        default : printf( "Nao eh operacao.\n" );
28    }
29    system( "pause" );
30    return 0;
31 }
```

A explicação para esse comportamento do **switch** se deve a uma regra da linguagem, que especifica que um salto condicional não pode pular uma declaração de variável no mesmo escopo. Quando colocamos as chaves {} depois do comando **case** e antes do comando **break**, estamos criando um novo escopo, ou seja, a variável declarada existe apenas dentro desse par de chaves. Portanto, ela pode ser “pulada” por um salto condicional.

## 4 COMANDOS DE REPETIÇÃO

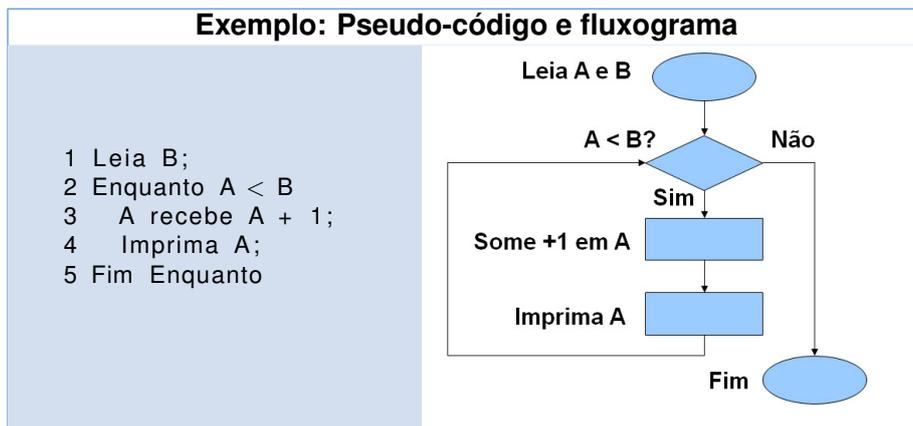
### 4.1 REPETIÇÃO POR CONDIÇÃO

Na seção anterior, vimos como realizar desvios condicionais em um programa. Desse modo, criamos programas em que um bloco de comandos é executado somente se uma determinada condição é verdadeira.

Entretanto, há casos em que é preciso que um bloco de comandos seja executado mais de uma vez se uma determinada condição for verdadeira:

```
enquanto condição faça
    sequência de comandos;
fim enquanto
```

Para isso, precisamos de uma estrutura de repetição que permita executar um conjunto de comandos quantas vezes forem necessárias. Isso é muito similar ao que ocorre em um fluxograma, onde o símbolo do losango permitia escolher entre diferentes caminhos com base em uma condição do tipo verdadeiro/falso, com a diferença de que agora o fluxo do programa é desviado novamente para a condição ao final da sequência de comandos:



De acordo com a condição, os comandos serão repetidos zero (se falsa) ou mais vezes (enquanto a condição for verdadeira). Essa estrutura normalmente é denominada laço ou loop.

Note que a sequência de comandos a ser repetida está subordinada a uma condição. Por **condição**, entende-se qualquer expressão relacional (ou seja, que use os operadores >, <, >=, <=, == ou !=) que resulte numa resposta do tipo **verdadeiro** ou **falso**. A condição pode ainda ser uma expressão que utiliza operadores:

- Matemáticos : +, -, \*, /, %
- Relacionais: >, <, >=, <=, ==, !=
- Lógicos: &&, ||

Na execução do comando enquanto, a condição será avaliada e:

- se a condição for considerada **verdadeira**, a sequência de comandos será executada. Ao final da sequência de comandos, o fluxo do programa é desviado novamente para o teste da condição;
- se a condição for considerada **falsa**, a sequência de comandos não será executada.



Como no caso do comando **if**, uma variável sozinha pode ser uma “expressão condicional” e retornar o seu próprio valor para um comando de repetição.

#### 4.1.1 LAÇO INFINITO

Um laço infinito (ou loop infinito) é uma sequência de comandos em um programa de computador que sempre se repete, ou seja, infinitamente. Isso geralmente ocorre por algum erro de programação, quando

- não definimos uma condição de parada;
- a condição de parada existe, mas nunca é atingida.

Basicamente, um laço infinito ocorre quando cometemos algum erro ao especificar a condição (ou expressão condicional) que controla a repetição, como é o caso do exemplo abaixo. Note que nesse exemplo, o valor de **X** é sempre diminuído em uma unidade, ou seja, fica mais negativo a cada passo. Portanto, a repetição nunca atinge a condição de parada:

### Exemplo 1: loop infinito

```
1 X recebe 4;  
2 enquanto (X < 5) faça  
3   X recebe X - 1;  
4   Imprima X;  
5 fim enquanto
```

Outro erro comum que produz um laço infinito é o de esquecer de algum comando dentro da sequência de comandos da repetição, como mostra o exemplo abaixo. Note que nesse exemplo, o valor de **X** nunca é modificado dentro da repetição. Portanto a condição é sempre verdadeira, e a repetição nunca termina:

### Exemplo 2: loop infinito

```
1 X recebe 4;  
2 enquanto (X < 5) faça  
3   Imprima X;  
4 fim enquanto
```

## 4.2 COMANDO WHILE

O comando `while` equivale ao comando “enquanto” utilizado nos pseudocódigos apresentados até agora.

A forma geral de um comando `while` é:

```
while (condição){  
    sequência de comandos;  
}
```

Na execução do comando **while**, a condição será avaliada e:

- se a condição for considerada **verdadeira** (ou possuir valor **diferente** de zero), a sequência de comandos será executada. Ao final da sequência de comandos, o fluxo do programa é desviado novamente para o teste da condição;

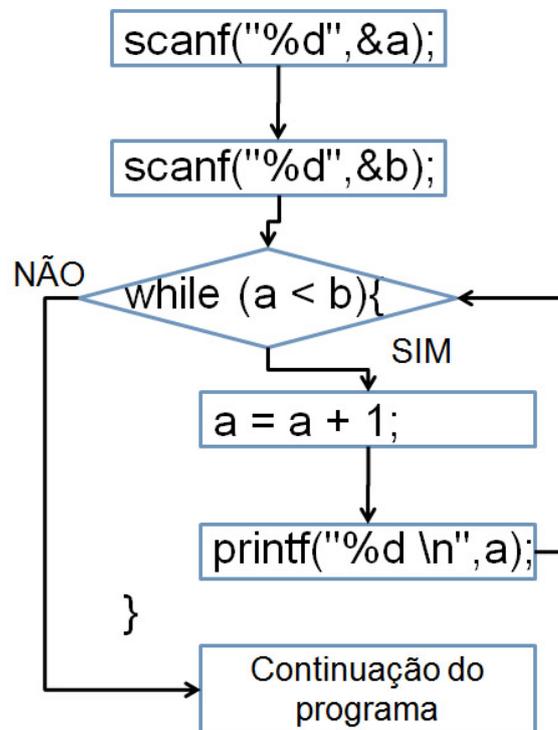
- se a condição for considerada **falsa** (ou possuir valor **igual** a zero), a sequência de comandos não será executada.

Abaixo, tem-se um exemplo de um programa que lê dois números inteiros *a* e *b* digitados pelo usuário e imprime na tela todos os números inteiros entre *a* e *b*:

#### Exemplo: comando while

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b;
5     printf( "Digite o valor de a: " );
6     scanf( "%d",&a );
7     printf( "Digite o valor de b: " );
8     scanf( "%d",&b );
9     while ( a < b ){
10        a = a + 1;
11        printf( "%d \n",a );
12    }
13    system( "pause" );
14    return 0;
15 }
```

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



O comando **while** segue todas as recomendações definidas para o comando **if** quanto ao uso das chaves e definição da condição usada.

Isso significa que a condição pode ser qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero), e que utiliza operadores dos tipos *matemáticos*, *relacionais* e/ou *lógicos*.

Como nos comandos condicionais, o comando **while** atua apenas sobre o comando seguinte a ele. Se quisermos que ele execute uma sequência de comandos, é preciso definir essa sequência de comandos dentro de chaves `{}`.



Como no comando if-else, não se usa o ponto e vírgula (;) depois da condição do comando while.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     while (a < b);{ //ERRADO!
10        a = a + 1;
11        printf("%d \n",a);
12    }
13    system("pause");
14    return 0;
15 }
```

Como no caso dos comandos condicionais, colocar o operador de ponto e vírgula (;) logo após o comando while, faz com que o compilador entenda que o comando while já terminou e trate o comando seguinte ( $a = a + 1$ ) como se o mesmo estivesse fora do while. No exemplo acima, temos um laço infinito (o valor de  $a$  e  $b$  nunca mudam, portanto a condição de parada nunca é atingida).



É responsabilidade do programador modificar o valor de algum dos elementos usados na condição para evitar que ocorra um laço infinito.

### 4.3 COMANDO FOR

O comando for é muito similar ao comando while visto anteriormente. Basicamente, o comando for é usado para repetir um comando, ou uma sequência de comandos, diversas vezes.

A forma geral de um comando for é:

```
for (inicialização; condição; incremento) {
    sequência de comandos;
}
```

Na execução do comando for, a seguinte sequência de passo é realizada:

- a cláusula inicialização é executada: nela as variáveis recebem um valor inicial para usar dentro do for.
- a condição é testada:
  - se a condição for considerada **verdadeira** (ou possuir valor **distinto** de zero), a sequência de comandos será executada. Ao final da sequência de comandos, o fluxo do programa é desviado para o incremento;
  - se a condição for considerada **falsa** (ou possuir valor **igual** a zero), a sequência de comandos não será executada (fim do comando for).
- incremento: terminada a execução da sequência de comandos, ocorre a etapa de incremento das variáveis usadas no for. Ao final dessa etapa, o fluxo do programa é novamente desviado para a condição.

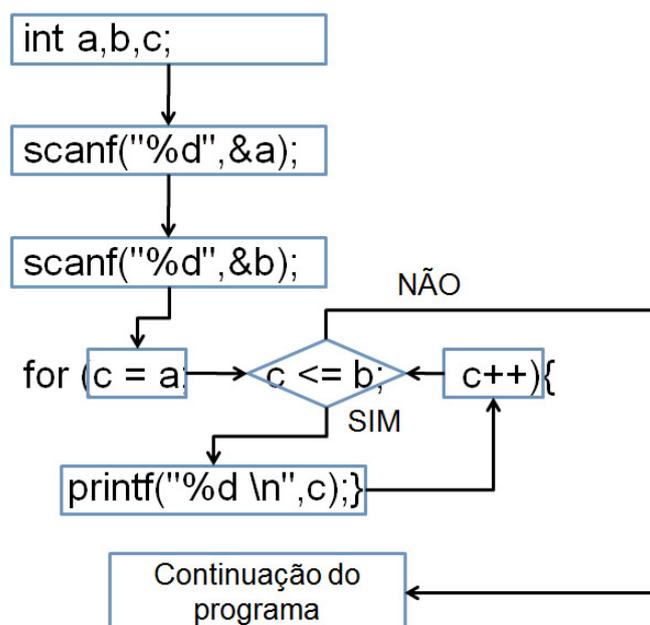
Abaixo, tem-se um exemplo de um programa que lê dois números inteiros *a* e *b* digitados pelo usuário e imprime na tela todos os números inteiros entre *a* e *b* (incluindo *a* e *b*):

#### Exemplo: comando for

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     for (c = a; c <= b; c++){
10         printf("%d \n",c);
11     }
12     system("pause");
13     return 0;
14 }
```

No exemplo acima, a variável *c* é inicializada como valor de *a* ( $c = a$ ). Em seguida, o valor de *c* é comparado com o valor de *b* ( $c \leq b$ ). Por fim, se a sequência de comandos foi executada, o valor da variável *c* será incrementado em uma unidade ( $c++$ ).

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



O comando for segue todas as recomendações definidas para o comando if e while quanto ao uso das chaves e definição da condição usada.

Isso significa que a condição pode ser qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero), e que utiliza operadores dos tipos *matemáticos*, *relacionais* e/ou *lógicos*.

Como nos comandos condicionais, o comando while atua apenas sobre o comando seguinte a ele. Se quisermos que ele execute uma sequência de comandos, é preciso definir essa sequência de comandos dentro de chaves {}.

Exemplo: for versus while	
for	while
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int i, soma = 0; 5     for (i = 1; i &lt;= 10; i       ++){ 6         soma = soma + i; 7     } 8     printf('Soma = %d \n',           soma); 9     system('pause'); 10    return 0; 11 }</pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int i, soma = 0; 5     i = 1 6     while (i &lt;= 10){ 7         soma = soma + i; 8         i++; 9     } 10    printf('Soma = %d \n',           soma); 11    system('pause'); 12    return 0; 13 }</pre>

#### 4.3.1 OMITINDO UMA CLAUSULA DO COMANDO FOR

Dependendo da situação em que o comando for é utilizado, podemos omitir qualquer uma de suas cláusulas:

- inicialização;
- condição;
- incremento.



Independente de qual cláusula é omitida, o comando for exige que se coloque os dois operadores de ponto e vírgula (;).

O comando for exige que se coloque os dois operadores de ponto e vírgula (;) pois é este operador que indica a separação entre as cláusulas de inicialização, condição e incremento. Sem elas, o compilador não tem certeza de qual cláusula foi omitida.

Abaixo, são apresentados três exemplos de comando for onde, em cada um deles, uma das cláusulas é omitida.

#### COMANDO FOR SEM INICIALIZAÇÃO

No exemplo abaixo, a variável *a* é utilizada nas cláusulas de condição e incremento do comando for. Como a variável *a* teve seu valor inicial definido

através de um comando de leitura do teclado (**scanf**), não é necessário a etapa de inicialização do comando for para definir o seu valor.

#### Exemplo: comando for sem inicialização

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b,c;
5     printf( "Digite o valor de a: " );
6     scanf( "%d",&a);
7     printf( "Digite o valor de b: " );
8     scanf( "%d",&b);
9     for ( ; a <= b; a++){
10         printf( "%d \n",a);
11     }
12     system( "pause" );
13     return 0;
14 }
```

#### COMANDO FOR SEM CONDIÇÃO



Ao omitir a condição do comando for, criamos um laço infinito.

Para o comando for, a ausência da cláusula de condição é considerada como uma condição que é sempre verdadeira. Sendo a condição sempre verdadeira, não existe condição de parada para o comando for, o qual vai ser executado infinitamente.

### Exemplo: comando for sem condição

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     //o comando for abaixo é um laço infinito
10    for (c = a; ; c++){
11        printf("%d \n",c);
12    }
13    system("pause");
14    return 0;
15 }
```

### COMANDO FOR SEM INCREMENTO

Por último, temos um exemplo de comando for sem a cláusula de incremento. Nessa etapa do comando for, um novo valor é atribuído para uma (ou mais) variáveis utilizadas.

### Exemplo: comando for sem incremento

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     for (c = a; c <= b; ){
10        printf("%d \n",c);
11        c++;
12    }
13    system("pause");
14    return 0;
15 }
```

No exemplo acima, a cláusula de incremento foi omitida da declaração do comando for. Para evitar a criação de uma laço infinito (onde a condição de parada existe, mas nunca é atingida), foi colocado um comando de incremento (c++) dentro da sequência de comandos do for. Perceba que,

desse modo, o comando `for` fica mais parecido com o comando `while`, já que agora se pode definir em qual momento o incremento vai ser executado, e não apenas no final.



A cláusula de incremento é utilizada para **atribuir** um novo valor a uma ou mais variáveis durante o comando `for`. Essa atribuição não está restrita a apenas o operador de incremento (`++`).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9
10    //incremento de duas unidades
11    for (c = a; c <= b; c=c+2){
12        printf("%d \n",c);
13    }
14
15    //novo valor é lido do teclado
16    for (c = a; c <= b; scanf("%d",&c)){
17        printf("%d \n",c);
18    }
19    system("pause");
20    return 0;
21 }
```

Nesse exemplo, fica claro que a cláusula de incremento pode conter qualquer comando que altere o valor de uma das variáveis utilizadas pelo comando `for`.

#### 4.3.2 USANDO O OPERADOR DE VÍRGULA (,) NO COMANDO FOR

Na linguagem C, o operador “,” é um separador de comandos. Ele permite determinar uma lista de expressões que devem ser executadas sequencialmente, inclusive dentro do comando **for**.



O operador de vírgula (,) pode ser usado em qualquer uma das cláusulas.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i,j;
5     for (i = 0, j = 100; i < j; i++, j--){
6         printf('i = %d e j = %d \n',i,j);
7     }
8     system('pause');
9     return 0;
10 }
```

No exemplo acima, foram definidos dois comandos para a cláusula de inicialização:  $i = 0$  e  $j = 100$ . Cada comando na inicialização é separado pelo operador de vírgula (,). A cláusula de inicialização só termina quando o operador de ponto e vírgula (;) é encontrado. Na fase de incremento, novamente o valor das duas variáveis é modificado: o valor de  $i$  é incrementado ( $i++$ ) enquanto o de  $j$  é decrementado ( $j--$ ). Novamente, cada comando na cláusula de incremento é separado pelo operador de vírgula (,).



A variável utilizada no laço **for** não precisa ser necessariamente do tipo **int**. Pode-se, por exemplo, usar uma variável do tipo **char** para imprimir uma sequência de caracteres.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char c;
5     for (c = 'A'; c <= 'Z'; c++){
6         printf('Letra = %c\n',c);
7     }
8     system('pause');
9     return 0;
10 }
```

Nesse exemplo, utilizamos uma variável do tipo **char** para controle do laço. Essa variável se inicia com o caractere letra "A" e o laço é executado até que a variável do laço possua como valor o caractere "Z".

## 4.4 COMANDO DO-WHILE

O comando do-while é bastante semelhante ao comando while visto anteriormente. Sua principal diferença é com relação a avaliação da condição: enquanto o comando while avalia a condição para depois executar uma sequência de comandos, o comando do-while executa uma sequência de comandos para depois testar a condição.

A forma geral de um comando do-while é:

```
do{  
    sequência de comandos;  
} while(condição);
```

Na execução do comando do-while, a seguinte ordem de passos é executada:

- a sequência de comandos é executada;
- a condição é avaliada:
  - se a condição for considerada **verdadeira** (ou possuir valor **diferente** de zero), o fluxo do programa é desviado novamente para o comando **do**, de modo que a sequência de comandos seja executada novamente;
  - se a condição for considerada **falsa** (ou possuir valor **igual** a zero) o laço termina (fim do comando do-while).



O comando do-while é utilizado sempre que se desejar que a sequência de comandos seja executada pelo menos uma vez.

No comando while, a condição é sempre avaliada antes da sequência de comandos. Isso significa que a condição pode ser falsa logo na primeira repetição do comando while, o que faria com que a sequência de comandos não fosse executada nenhuma vez. Portanto, o comando while pode repetir uma sequência de comandos **zero** ou mais vezes.

Já no comando do-while, a sequência de comandos é executada primeiro. Mesmo que a condição seja falsa logo na primeira repetição do comando do-while, a sequência de comandos terá sido executada pelo menos uma vez. Portanto, o comando do-while pode repetir uma sequência de comandos **uma** ou mais vezes.



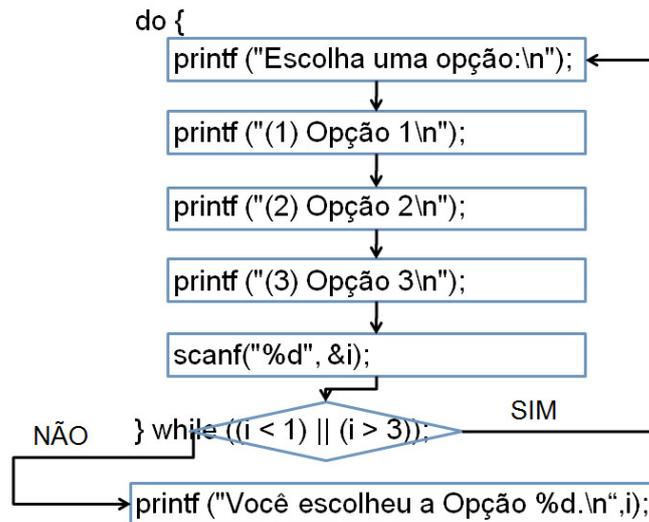
O comando do-while segue todas as recomendações definidas para o comando if quanto ao uso das chaves e definição da condição usada.

Abaixo, tem-se um exemplo de um programa que exibe um menu de opções para o usuário e espera que ele digite uma das suas opções:

#### Exemplo: comando do-while

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i;
5     do {
6         printf ("Escolha uma opção:\n");
7         printf ("(1) Opção 1\n");
8         printf ("(2) Opção 2\n");
9         printf ("(3) Opção 3\n");
10        scanf("%d", &i);
11    } while ((i < 1) || (i > 3));
12    printf ("Você escolheu a Opção %d.\n", i);
13    system("pause");
14    return 0;
15 }
```

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



Diferente do comando if-else, é necessário colocar um ponto e vírgula (;) depois da condição do comando do-while.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i = 0;
5     do{
6         printf('Valor %d\n', i);
7         i++;
8     }while(i < 10); //Esse ponto e vírgula é
                       necessário!
9     system('pause');
10    return 0;
11 }

```

No comando do-while, a sequência de comandos é definida antes do teste da condição, diferente dos outros comando condicionais e de repetição. Isso significa que o teste da condição é o último comando da repetição do-while. Sendo assim, o compilador entende que a definição do comando do-while já terminou e exige que se coloque o operador de ponto e vírgula (;) após a condição.



É responsabilidade do programador modificar o valor de algum dos elementos usados na condição para evitar que ocorra um laço infinito.

## 4.5 ANINHAMENTO DE REPETIÇÕES

Uma repetição aninhada é simplesmente um comando de repetição utilizado dentro do bloco de comandos de um outro comando de repetições. Basicamente, é um comando de repetição dentro de outro, semelhante ao que é feito com o comando **if**.

A forma geral de um comando de repetição aninhado é:

```
repetição(condição 1) {  
    sequência de comandos;  
    repetição(condição 2) {  
        sequência de comandos;  
        repetição...  
    }  
}
```

onde **repetição** representa um dos três possíveis comandos de repetição da linguagem C: **while**, **for** e **do-while**.

Em um aninhamento de repetições, o programa começa a testar as condições começando pela **condição 1** da primeira repetição. Se o resultado dessa condição for diferente de zero (verdadeiro), o programa executará o bloco de comando associados a ela, aí incluído o segundo comando de repetição. Note que os comando da segunda repetição só serão executados se a condição da primeira for verdadeira. Esse processo se repete para cada comando de repetição que o programa encontrar dentro do bloco de comando que ele executar.

O aninhamento de comandos de repetição é muito útil quando se tem que percorrer dois conjuntos de valores que estão relacionados dentro de um programa. Por exemplo, para imprimir uma matriz identidade (composta apenas de 0's e 1's na diagonal principal) de tamanho  $4 \times 4$  é preciso percorrer as quatro linhas da matriz e, para cada linha, percorrer as suas quatro colunas. Um único comando de repetição não é suficiente para realizar essa tarefa, como mostra o exemplo abaixo:

Exemplo: comandos de repetição aninhados	
com for	com while
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int i,j; 5     for (i=1;i&lt;5;i++){ 6         for (j=1; j&lt;5; j++){ 7             if (i==j) 8                 printf("1 "); 9             else 10                printf("0 "); 11        } 12        printf("\n"); 13    } 14    system("pause"); 15    return 0; 16 }</pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int i=1,j; 5     while(i&lt;5){ 6         j = 1; 7         while(j&lt;5){ 8             if (i==j) 9                 printf("1 "); 10            else 11                printf("0 "); 12            j++; 13        } 14        printf("\n"); 15        i++; 16    } 17    system("pause"); 18    return 0; 19 }</pre>

Note, no exemplo anterior, que a impressão de uma matriz identidade pode ser feita com dois comandos **for** ou dois comandos **while**. É possível ainda fazê-lo usando um comando de cada tipo.



A linguagem C não proíbe que se misture comandos de repetições de tipos diferentes no aninhamento de repetições.

## 4.6 COMANDO BREAK

Vimos, anteriormente, que o comando **break** pode ser utilizado em conjunto com o comando **switch**. Basicamente, sua função era interromper o comando **switch** assim que uma das sequências de comandos da cláusula **case** fosse executada. Caso o comando **break** não existisse, a sequência de comandos do **case** seguinte também seria executada e assim por diante.

Na verdade, o comando **break** serve para quebrar a execução de um comando (como no caso do **switch**) ou interromper a execução de qualquer comando de laço (**for**, **while** ou **do-while**). O **break** faz com que a execução do programa continue na primeira linha seguinte ao laço ou bloco que está sendo interrompido.



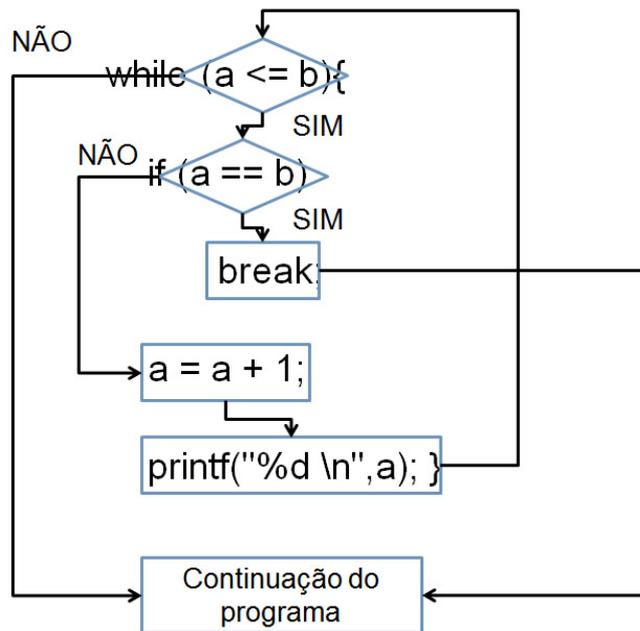
O comando `break` é utilizado para terminar abruptamente uma repetição. Por exemplo, se estivermos em uma repetição e um determinado resultado ocorrer, o programa deverá sair da iteração.

Abaixo, tem-se um exemplo de um programa que lê dois números inteiros  $a$  e  $b$  digitados pelo usuário e imprime na tela todos os números inteiros entre  $a$  e  $b$ . Note que no momento em que o valor de  $a$  atinge o valor de  $b$ , o comando `break` é chamado e o laço terminado:

#### Exemplo: comando `break`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     while (a <= b){
10         if (a == b)
11             break;
12         a = a + 1;
13         printf("%d \n",a);
14     }
15     system("pause");
16     return 0;
17 }
```

Relembrando o conceito de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados pelo programa:



#### 4.7 COMANDO CONTINUE

O comando continue é muito parecido com o comando break. Tanto o comando break quanto o comando continue ignoram o restante da sequência de comandos da repetição que os sucedem. A diferença é que, enquanto o comando break termina o laço de repetição, o comando break interrompe apenas aquela repetição e passa para a próxima repetição do laço (se ela existir).

Por esse mesmo motivo, o comando continue só pode ser utilizado dentro de um laço.



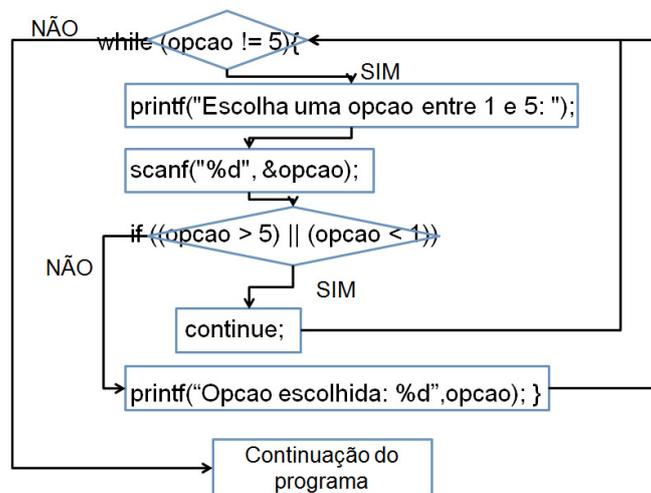
Os comandos que sucedem o comando continue no bloco não são executados.

Abaixo, tem-se um exemplo de um programa que lê, repetidamente, um número inteiro do usuário e a imprime apenas se ela for maior ou igual a 1 e menor ou igual a 5. Caso o número não esteja nesse intervalo, essa repetição do laço é desconsiderada e reiniciada:

### Exemplo: comando continue

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int opcao = 0;
5     while (opcao != 5){
6         printf("Escolha uma opcao entre 1 e 5: ");
7         scanf("%d", &opcao);
8         if ((opcao > 5) || (opcao < 1))
9             continue;
10        printf("Opcao escolhida: %d",opcao);
11    }
12    system("pause");
13    return 0;
14 }
```

Relembrando o conceito de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados pelo programa:



## 4.8 GOTO E LABEL

O comando goto é um salto condicional para um local especificado por uma palavra chave no código. A forma geral de um comando goto é:

**destino:**  
goto **destino**;

Na sintaxe acima, o comando **goto** (do inglês go to, literalmente “ir para”) muda o fluxo do programa para um local previamente especificado pela expressão **destino**, onde **destino** é uma palavra definida pelo programador. Este local pode ser a frente ou atrás no programa, mas deve ser dentro da mesma função.

O teorema da programação estruturada prova que a instrução **goto** não é necessária para escrever programas; alguma combinação das três construções de programação (comandos sequenciais, condicionais e de repetição) são suficientes para executar qualquer cálculo. Além disso, o uso de **goto** pode deixar o programa muitas vezes ilegível.

Exemplo: goto versus for	
goto	for
<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int i = 0; 5     inicio : 6     if (i &lt; 5){ 7         printf('Numero %d\n',i); 8         i++; 9         goto inicio; 10    } 11    system('pause'); 12    return 0; 13 }</pre>	<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int i; 5     for(i = 0; i &lt; 5; i++) 6         printf('Numero %d\n',i); 7 8     system('pause'); 9     return 0; 10 }</pre>

Como se nota no exemplo acima, o mesmo programa feito com o comando **for** é muito mais fácil de entender do que o mesmo programa feito com o comando **goto**.



Apesar de banido da prática de programação, o comando goto pode ser útil em determinadas circunstâncias. Ex: sair de dentro de laços aninhados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i,j,k;
5     for(i = 0; i < 5; i++)
6         for(j = 0; j < 5; j++)
7             for(k = 0; k < 5; k++)
8                 if(i == 2 && j == 3 && k == 1)
9                     goto fim;
10                else
11                    printf(“Posicao [%d,%d,%d]\n”,i,j,k);
12
13
14 fim:
15 printf(“Fim do programa\n”);
16
17 system(“pause”);
18 return 0;
19 }
```

## 5 VETORES E MATRIZES - ARRAYS

### 5.1 EXEMPLO DE USO

Um array ou “vetor” é a forma mais comum de dados estruturados da linguagem C. Um array é simplesmente um conjunto de variáveis do mesmo tipo, igualmente acessíveis por um índice.



Imagine o seguinte problema: dada uma relação de 5 estudantes, imprimir o nome de cada estudante, cuja nota é maior do que a média da classe.

Um algoritmo simples para resolver esse problema poderia ser o pseudo-código apresentado abaixo:

```
Leia(nome1, nome2, nome3, nome4, nome5);  
Leia(nota1, nota2, nota3, nota4, nota5);  
media = (nota1+nota2+nota3+nota4+nota5) / 5,0;  
Se nota1 > media então escreva (nome1)  
Se nota2 > media então escreva (nome2)  
Se nota3 > media então escreva (nome3)  
Se nota4 > media então escreva (nome4)  
Se nota5 > media então escreva (nome5)
```

O algoritmo anterior representa uma solução possível para o problema. O grande inconveniente dessa solução é a grande quantidade de variáveis para gerenciarmos e o uso repetido de comandos praticamente idênticos.



Essa solução é inviável para uma lista de 100 alunos.

Expandir o algoritmo anterior para trabalhar com um total de 100 alunos significaria, basicamente, aumentar o número de variáveis para guardar os dados de cada aluno e repetir, ainda mais, um conjunto de comandos praticamente idênticos. Desse modo, teríamos:

- Uma variável para armazenar cada nome de aluno: 100 variáveis;
- Uma variável para armazenar a nota de cada aluno: 100 variáveis;
- Um comando de teste e impressão na tela para cada aluno: 100 testes.

O pseudo-código abaixo representa o algoritmo anterior expandido para poder trabalhar com 100 alunos:

```

Leia(nome1, nome2, ..., nome100);
Leia(nota1, nota2,..., nota100);
media = (nota1+nota2+...+nota100) / 100,0;
Se nota1 > media então escreva (nome1)
Se nota2 > media então escreva (nome2)
...
Se nota100 > media então escreva (nome100)

```

Como se pode notar, temos uma solução extremamente engessada para o nosso problema. Modificar o número de alunos usado pelo algoritmo implica em reescrever todo o código, repetindo comandos praticamente idênticos. Além disso, temos uma grande quantidade de variáveis para gerenciar, cada uma com o seu próprio nome, o que torna essa tarefa ainda mais difícil de ser realizada sem a ocorrência de erros.



Como estes dados têm uma relação entre si, podemos declará-los usando um ÚNICO nome para todos os 100 elementos.

Surge então a necessidade de usar um array.

## 5.2 ARRAY COM UMA DIMENSÃO - VETOR

A idéia de um array ou “vetor” é bastante simples: criar um conjunto de variáveis do mesmo tipo utilizando apenas um nome.

Relembrando o exemplo anterior, onde as variáveis que guardam as notas dos 100 alunos são todas do mesmo tipo, essa solução permitiria usar apenas um nome (**notas**, por exemplo) de variável para representar todas as notas dos alunos, ao invés de um nome para cada variável.

## DECLARANDO UM VETOR

Em linguagem C, a declaração de um array segue a seguinte forma geral:

```
tipo_dado nome_array[tamanho];
```

O comando acima define um array de nome *nome\_array* contendo *tamanho* elementos adjacentes na memória. Cada elemento do array é do tipo *tipo\_dado*. Pensando no exemplo anterior, poderíamos usar um array de inteiros contendo 100 elementos para guardar as notas dos 100 alunos. Ele seria declarado com o mostrado abaixo:

```
int notas[100];
```

## ACESSANDO UM ELEMENTO DO VETOR

Como a variável que armazena a nota de um aluno possui agora o mesmo nome que as demais notas dos outros alunos, o acesso ao valor de cada nota é feito utilizando um índice, como mostra a figura abaixo:

0	1	...	99
81	55		72

**notas**

Note que na posição “0” do array está armazenado o valor “81”, na posição “1” está armazenado o valor “55”, e assim por diante.



Para indicar qual índice do array queremos acessar, utiliza-se o operador de colchetes [ ]: notas[índice].

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int notas[100];
5     int i;
6     for (i = 0; i < 100; i++){
7         printf(“ Digite a nota do aluno %d”,i);
8         scanf(“%d”,&notas[i]);
9     }
10    system(“pause”);
11    return 0;
12 }
```

No exemplo acima, percebe-se que cada posição do array possui todas as características de uma variável. Isso significa que ela pode aparecer em comandos de entrada e saída de dados, expressões e atribuições. Por exemplo:

```
scanf("%d",&notas[5]);  
notas[0] = 10;  
notas[1] = notas[5] + notas[0];
```



O tempo para acessar qualquer uma das posições do array é o mesmo.

Lembre-se, cada posição do array é uma variável. Portanto, todas as posições do array são igualmente acessíveis, isto é, o tempo e o tipo de procedimento para acessar qualquer uma das posições do array são iguais ao de qualquer outra variável.



Na linguagem C a numeração começa sempre do ZERO e termina em N-1, onde N é o número de elementos do array.

Isto significa que, no exemplo anterior, as notas dos alunos serão indexadas de 0 a 99:

```
notas[0]  
notas[1]  
...  
notas[99]
```

Isso acontece pelo seguinte motivo: um array é um agrupamento de dados, do mesmo tipo, adjacentes na memória. O nome do array indica onde esses dados começam na memória. O índice do array indica quantas posições se deve pular para acessar uma determinada posição. A figura abaixo exemplifica como o array está na memória:

Memória			
#	var	conteúdo	
123	notas	81	notas[0]
124		55	notas[1]
125		63	notas[2]
126		67	notas[3]
127		90	notas[4]
128		...	
129			
▪ ▪ ▪			



Num array de 100 elementos, índices menores do que 0 e maiores do que 99 também podem ser acessados. Porém, isto pode resultar nos mais variados erros durante a execução do programa.

Como foi explicado, um array é um agrupamento de dados adjacentes na memória e o seu índice apenas indica quantas posições se deve pular para acessar uma determinada posição. Isso significa que se tentarmos acessar o índice 100, o programa tentará acessar a centésima posição a partir da posição inicial (que é o nome do array). O mesmo vale para a posição de índice -1. Nesse caso o programa tentará acessar uma posição anterior ao local onde o array começa na memória. O problema é que, apesar dessas posições existirem na memória e serem acessíveis, elas não pertencem ao array. Pior ainda, elas podem pertencer a outras variáveis do programa, e a alteração de seus valores pode resultar nos mais variados erros durante a execução do programa.



É função do programador garantir que os limites do array estão sendo respeitados.

Deve-se tomar cuidado ao se rabalhar com arrays. Principalmente ao se usar a operação de atribuição (=).



Não se pode fazer atribuição de arrays.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int v[5] = {1,2,3,4,5};
5     int v1[5];
6     v1 = v; //ERRADO!
7
8     system( ' 'pause' ' );
9     return 0;
10 }
```

Isso ocorre porque a linguagem C não suporta a atribuição de um array para outro. Para atribuir o conteúdo de um array a outro array, o correto é copiar seus valores elemento por elemento para o outro array.

### 5.3 ARRAY COM DUAS DIMENSÕES - MATRIZ

Os arrays declarados até o momento possuem apenas uma dimensão. Há casos, em que uma estrutura com mais de uma dimensão é mais útil. Por exemplo, quando trabalhamos com matrizes, onde os valores são organizados em uma estrutura de linhas e colunas.

#### DECLARANDO UM MATRIZ

Em linguagem C, a declaração de uma matriz segue a seguinte forma geral:

```
tipo_dado nome_array[nro_linhas][nro_colunas];
```

O comando acima define um array de nome *nome\_array* contendo *nro\_linhas* × *nro\_colunas* elementos adjacentes na memória. Cada elemento do array é do tipo *tipo\_dado*.

Por exemplo, para criar um array de inteiros que possua 100 linhas e 50 colunas, isto é, uma matriz de inteiros de tamanho 100×50, usa-se a declaração abaixo:

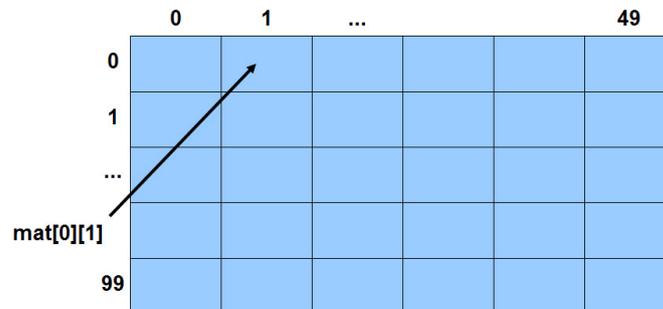
```
int mat[100][50];
```

## ACESSANDO UM ELEMENTO DA MATRIZ

Como no caso dos arrays de uma única dimensão, cada posição da matriz possui todas as características de uma variável. Isso significa que ela pode aparecer em comandos de entrada e saída de dados, expressões e atribuições:

```
scanf("%d",&mat[5][0]);  
mat[0][0] = 10;  
mat[1][2] = mat[5][0] + mat[0][0];
```

Perceba, no entanto, que o acesso ao valor de uma posição da matriz é feito agora utilizando dois índices: um para a linha e outro para a coluna.



Lembre-se, cada posição do array é uma variável. Portanto, todas as posições do array são igualmente acessíveis, isto é, o tempo e o tipo de procedimento para acessar qualquer uma das posições do array são iguais ao de qualquer outra variável.

## 5.4 ARRAYS MULTIDIMENSIONAIS

Vimos até agora como criar arrays com uma ou duas dimensões. A linguagem C permite que se crie arrays com mais de duas dimensões de maneira fácil.



Na linguagem C, cada conjunto de colchetes [ ] representa uma dimensão do array.

Cada par de colchetes adicionado ao nome de uma variável durante a sua declaração adiciona uma nova dimensão àquela variável, independente do seu tipo:

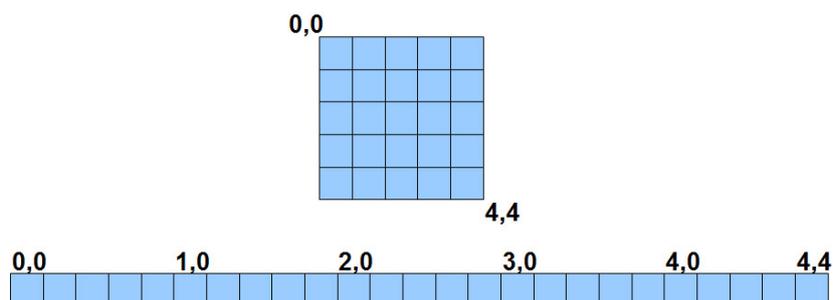
```
int vet[5]; // 1 dimensão
float mat[5][5]; // 2 dimensões
double cub[5][5][5]; // 3 dimensões
int X[5][5][5][5]; // 4 dimensões
```



O acesso ao valor de uma posição de um array multidimensional é feito utilizando um índice para cada dimensão do array.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int cub[5][5][5];
5     int i,j,k;
6     //preenche o array de 3 dimensões com zeros
7     for (i=0; i < 5; i++){
8         for (j=0; j < 5; j++){
9             for (k=0; k < 5; k++){
10                cub[i][j][k] = 0;
11            }
12        }
13    }
14    system('pause');
15    return 0;
16 }
```

Apesar de terem o comportamento de estruturas com mais de uma dimensão, os dados dos arrays multidimensionais são armazenados linearmente na memória. É o uso dos colchetes que cria a impressão de estarmos trabalhando com mais de uma dimensão.



Por esse motivo, é importante ter em mente qual a dimensão que se move mais rapidamente na memória: sempre a mais a direita, independente do tipo ou número de dimensões do array, como se pode ver abaixo marcado em vermelho:

```
int vet[5]; // 1 dimensão
float mat[5][5]; // 2 dimensões
double cub[5][5][5]; // 3 dimensões
int X[5][5][5][5]; // 4 dimensões
```



Basicamente, um array multidimensional funciona como qualquer outro array. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

## 5.5 INICIALIZAÇÃO DE ARRAYS

Um array pode ser inicializado com certos valores durante sua declaração. Isso pode ser feito com qualquer array independente do tipo ou número de dimensões do array.

A forma geral de inicialização de um array é:

```
tipo_dado nome_array[tam1][tam2]...[tamN] = {dados };
```

Na declaração acima, dados é uma lista de valores (do mesmo tipo do array) separados por vírgula e delimitado pelo operador de chaves {}. Esses valores devem ser colocados na mesma ordem em que serão colocados dentro do array.



A inicialização de uma array utilizando o operador de chaves {} só pode ser feita durante sua declaração.

A inicialização de uma array consiste em atribuir um valor inicial a cada posição do array. O operador de chaves apenas facilita essa tarefa, como mostra o exemplo abaixo:

Exemplo 1: inicializando um array	
Com o operador de {}	Sem o operador de {}
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int vet[5] =         {15,12,91,35}; 5 6     system('pause'); 7     return 0; 8 }</pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int vet[5]; 5     vet[0] = 15; 6     vet[1] = 12; 7     vet[2] = 9; 8     vet[3] = 1; 9     vet[4] = 35; 10 11     system('pause'); 12     return 0; 13 }</pre>

Abaixo são apresentados alguns exemplos de inicialização de arrays de diferentes tipos e número de dimensões:

Exemplo 2: inicializando um array
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main(){ 4     int matriz1 [3][4] = {1,2,3,4,5,6,7,8,9,10,11,12}; 5     int matriz2 [3][4] =         {{1,2,3,4},{5,6,7,8},{9,10,11,12}}; 6 7     char str1 [10] = {'J','o','a','o','\0' }; 8     char str2 [10] = "Joao"; 9 10    char str_matriz [3][10] = {"Joao", "Maria", "Jose"         '}; 11 12    system('pause'); 13    return 0; 14 }</pre>

Note no exemplo acima que a inicialização de um array de 2 dimensões pode ser feita de duas formas distintas. Na primeira matriz (**matriz1**) os valores iniciais da matriz são definidos utilizando um único conjunto de chaves {}, igual ao que é feito com vetores. Nesse caso, os valores são atribuídos para todas as colunas da primeira linha da matriz, para depois passar para as colunas da segunda linha e assim por diante. Lembre-se,

a dimensão que se move mais rapidamente na memória é sempre a mais a direita, independente do tipo ou número de dimensões do array. Já na segunda matriz (**matriz2**) usa-se mais de um conjunto de chaves {} para definir cada uma das dimensões da matriz.

Para a inicialização de um array de caracteres, pode-se usar o mesmo princípio definido na inicialização de vetores (**str1**). Percebe-se que essa forma de inicialização não é muito prática. Por isso, a inicialização de um array de caracteres também pode ser feita por meio de “aspas duplas”, como mostrado na inicialização de **str2**. O mesmo princípio é válido para iniciar um array de caracteres de mais de uma dimensão.



Na inicialização de um array de caracteres não é necessário definir todos os seus elementos.

### 5.5.1 INICIALIZAÇÃO SEM TAMANHO

A linguagem C também permite inicializar um array sem que tenhamos definido o seu tamanho. Nesse caso, simplesmente não se coloca o valor do tamanho entre os colchetes durante a declaração do array:

```
tipo_dado nome_array[ ] = {dados };
```

Nesse tipo de inicialização, o compilador da linguagem C vai considerar o tamanho do dado declarado como sendo o tamanho do array. Isto ocorre durante a compilação do programa. Depois disso, o tamanho do array não poderá mais ser modificado durante o programa.

Abaixo são apresentados alguns exemplos de inicialização de arrays sem tamanhos:

### Exemplos: inicializando um array sem tamanho

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //A string texto terá tamanho 13
5     //(12 caracteres + o caractere '\0')
6     char texto [] = "Linguagem C.";
7
8     // O número de posições do vetor será 10.
9     int vetor [] = {1,2,3,4,5,6,7,8,9,10};
10
11     //O número de linhas de matriz será 5.
12     int matriz [][] = {1,2,3,4,5,6,7,8,9,10};
13
14     system("pause");
15     return 0;
16 }
```

Note no exemplo acima que foram utilizados 12 caracteres para iniciar o array de char “texto”. Porém, o seu tamanho final será 13. Isso ocorre por que arrays de caracteres sempre possuem o elemento seguinte ao último caractere como sendo o caractere ‘\0’. Mais detalhes sobre isso podem ser vistos na seção seguinte.



Esse tipo de inicialização é muito útil quando não queremos contar quantos caracteres serão necessários para inicializarmos uma string (array de caracteres).

No caso da inicialização de arrays de mais de uma dimensão, é necessário sempre definir as demais dimensões. Apenas a primeira dimensão pode ficar sem tamanho definido.

## 5.6 EXEMPLO DE USO DE ARRAYS

Nesta seção são apresentados alguns exemplos de operações básicas de manipulação de vetores e matrizes em C.

### Somar os elementos de um vetor de 5 inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i, lista[5] = {3,51,18,2,45};
5     int soma = 0;
6     for(i=0; i < 5; i++)
7         soma = soma + lista[i];
8     printf('Soma = %d',soma);
9     system('pause');
10    return 0;
11 }
```

### Encontrar o maior valor contido em um vetor de 5 inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i, lista[5] = {3,18,2,51,45};
5     int Maior = lista[0];
6     for(i=1; i<5; i++){
7         if (Maior < lista[i])
8             Maior = lista[i];
9     }
10    printf('Maior = %d', Maior);
11    system('pause');
12    return 0;
13 }
```

### Calcular a média dos elementos de um vetor de 5 inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i, lista[5] = {3,51,18,2,45};
5     int soma = 0;
6     for(i=0; i < 5; i++)
7         soma = soma + lista[i];
8     float media = soma / 5.0;
9     printf('Media = %f',media);
10    system('pause');
11    return 0;
12 }
```

### Somar os elementos de uma matriz de inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int mat[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
5     int i, j, soma = 0;
6     for(i=0; i < 3; i++)
7         for(j=0; j < 3; j++)
8             soma = soma + mat[i][j];
9     printf("Soma = %d",soma);
10    system("pause");
11    return 0;
12 }
```

### Imprimir linha por linha uma matriz

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int mat[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
5     int i, j;
6     for(i=0; i < 3; i++){
7         for(j=0; j < 3; j++)
8             printf("%d ",mat[i][j]);
9         printf("\n");
10    }
11    system("pause");
12    return 0;
13 }
```

## 6 ARRAYS DE CARACTERES - STRINGS

### 6.1 DEFINIÇÃO E DECLARAÇÃO DE UMA STRING

**String** é o nome que usamos para definir uma sequência de caracteres adjacentes na memória do computador. Essa sequência de caracteres, que pode ser uma palavra ou frase, é armazenada na memória do computador na forma de um array do tipo **char**.

Sendo a string um array de caracteres, sua declaração segue as mesmas regras da declaração de um array convencional:

```
char str[6];
```

A declaração acima cria na memória do computador uma string (array de caracteres) de nome **str** e tamanho igual a **6**. No entanto, apesar de ser um array, devemos ficar atentos para o fato de que as strings têm no elemento seguinte a última letra da palavra/frase armazenada um caractere `'\0'`.



O caractere `'\0'` indica o fim da sequência de caracteres.

Isso ocorre por que podemos definir uma string com um tamanho maior do que a palavra armazenada. Imagine uma string definida com um tamanho de 50 caracteres, mas utilizada apenas para armazenar a palavra "oi". Nesse caso, temos 48 posições não utilizadas e que estão preenchidas com **lixo de memória** (um valor qualquer). Obviamente, não queremos que todo esse lixo seja considerado quando essa string for exibida na tela. Assim, o caractere `'\0'` indica o fim da sequência de caracteres e o início das posições restantes da nossa string que não estão sendo utilizadas nesse momento:



Ao definir o tamanho de uma string, devemos considerar o caractere `'\0'`.

Como o caractere ‘\0’ indica o final de nossa string, isso significa que numa string definida com um tamanho de 50 caracteres, apenas 49 estarão disponíveis para armazenar o texto digitado pelo usuário.

### 6.1.1 INICIALIZANDO UMA STRING

Uma string pode ser lida do teclado ou já ser definida com um valor inicial. Para sua inicialização, pode-se usar o mesmo princípio definido na inicialização de vetores e matrizes:

```
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
```

Percebe-se que essa forma de inicialização não é muito prática. Por isso, a inicialização de strings também pode ser feita por meio de “aspas duplas”:

```
char str [10] = "Joao";
```

Essa forma de inicialização possui a vantagem de já inserir o caractere ‘\0’ no final da string.

### 6.1.2 ACESSANDO UM ELEMENTO DA STRING

Outro ponto importante na manipulação de strings é que, por se tratar de um array, cada caractere pode ser acessado individualmente por indexação como em qualquer outro vetor ou matriz:

```
char str[6] = "Teste";  
str[0] = 'L';
```

T	e	s	t	e	\0
---	---	---	---	---	----

L	e	s	t	e	\0
---	---	---	---	---	----



Na atribuição de strings usa-se “aspas duplas”, enquanto que na de caracteres, usa-se ‘aspas simples’.

## 6.2 TRABALHANDO COM STRINGS

O primeiro cuidado que temos que tomar ao se trabalhar com strings é na operação de atribuição.



Strings são arrays. Portanto, não se pode fazer atribuição de strings.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char str1[20] = "Hello World";
5     char str2[20];
6
7     str1 = str2; //ERRADO!
8
9     system("pause");
10    return 0;
11 }
```

Isso ocorre porque uma string é um array e a linguagem C não suporta a atribuição de um array para outro. Para atribuir o conteúdo de uma string a outra, o correto é copiar a string elemento por elemento para a outra string.

### Exemplo: Copiando uma string

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int count;
5     char str1[20] = "Hello World", str2[20];
6     for (count = 0; str1[count]!='\0'; count++)
7         str2[count] = str1[count];
8     str2[count] = '\0';
9     system("pause");
10    return 0;
11 }
```

O exemplo acima permite copiar uma string elemento por elemento para outra string. Note que foi utilizada a mesma forma de indexação que seria feita com um array de qualquer outro tipo (**int**, **float**, etc). Infelizmente, esse tipo de manipulação de arrays não é muito prática quando estamos trabalhando com palavras.



Felizmente, a biblioteca padrão da linguagem C possui funções especialmente desenvolvidas para a manipulação de strings na biblioteca `<string.h>`.

A seguir, serão apresentadas algumas das funções mais utilizadas para a leitura, escrita e manipulação de strings.

## 6.2.1 LENDO UMA STRING DO TECLADO

### USANDO A FUNÇÃO SCANF()

Existem várias maneiras de se fazer a leitura de uma sequência de caracteres do teclado. Uma delas é utilizando a já conhecida função **scanf()** com o formato de dados `"%s"`:

```
char str[20];
scanf("%s",str);
```



Quando usamos a função **scanf()** para ler uma string, o símbolo de `&` antes do nome da variável não é utilizado. Os colchetes também não são utilizados pois queremos ler a string toda e não apenas uma letra.

Infelizmente, para muitos casos, a função **scanf()** não é a melhor opção para se ler uma string do teclado.



A função **scanf()** lê apenas strings digitadas sem espaços, ou seja, palavras.

No caso de ter sido digitada uma frase (uma sequência de caracteres contendo espaços), apenas os caracteres digitados antes do primeiro espaço encontrado serão armazenados na string se a sua leitura for feita com a função **scanf()**.

### USANDO A FUNÇÃO GETS()

Uma alternativa mais eficiente para a leitura de uma string é a função **gets()**, a qual faz a leitura do teclado considerando todos os caracteres digitados (incluindo os espaços) até encontrar uma tecla enter:

```
char str[20];  
gets(str);
```

### USANDO A FUNÇÃO FGETS()

Basicamente, para se ler uma string do teclado utilizamos a função **gets()**. No entanto, existe outra função que, utilizada de forma adequada, também permite a leitura de strings do teclado. Essa função é a **fgets()**, cujo protótipo é:

```
char *fgets (char *str, int tamanho, FILE *fp);
```

A função **fgets()** recebe 3 parâmetros de entrada

- str: a string a ser lida;
- tamanho: o limite máximo de caracteres a serem lidos;
- fp: a variável que está associado ao *arquivo* de onde a string será lida.

e retorna

- **NULL**: no caso de erro ou fim do arquivo;

- O ponteiro para o primeiro caractere da string recuperada em `str`.

Note que a função **fgets()** utiliza uma variável FILE `*fp`, que está associado ao arquivo de onde a string será lida.



Para ler do teclado, basta substituir **FILE \*fp** por **stdin**, o qual representa o dispositivo de entrada padrão (geralmente o teclado).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char nome[30];
5     printf('Digite um nome: ');
6     fgets (nome, 30, stdin);
7     printf('O nome digitado foi: %s', nome);
8     system('pause');
9     return 0;
10 }
```

Como a função **gets()**, a função **fgets()** lê a string do teclado até que um caractere de nova linha (ENTER) seja lido. Apesar de parecerem iguais, a função `fgets` possui algumas diferenças e vantagens sobre a **gets()**.



Na função **fgets()**, o caractere de nova linha (`'\n'`) fará parte da string, o que não acontecia com **gets()**.

A função **gets()** armazena tudo que for digitado até o comando de enter. Já a função **fgets()** armazena tudo que for digitado, incluindo o comando de enter (`'\n'`).



A função **fgets()** especifica o tamanho máximo da string de entrada.

Diferente da função **gets()**, a função **fgets()** lê a string até que um caractere de nova linha seja lido ou “tamanho-1” caracteres tenham sido lidos. Isso evita o estouro do buffer, que ocorre quando se tenta ler algo maior do que pode ser armazenado na string.

#### LIMPANDO O BUFFER DO TECLADO

Às vezes, podem ocorrer erros durante a leitura de caracteres ou strings do teclado. Para resolver esse pequenos erros, podemos limpar o buffer do teclado (entrada padrão) usando a função **setbuf(stdin, NULL)** antes de realizar a leitura de caracteres ou strings:

Exemplo: limpando o buffer do teclado	
leitura de caracteres	leitura de strings
<pre>1 char ch; 2 setbuf(stdin, NULL); 3 scanf('%c', &amp;ch);</pre>	<pre>1 char str[10]; 2 setbuf(stdin, NULL); 3 gets(str);</pre>

Basicamente, a função **setbuf()** preenche um buffer (primeiro parâmetro) com um determinado valor (segundo parâmetro). No exemplo acima, o buffer da entrada padrão (**stdin**), ou seja, o teclado, é preenchido com o valor vazio (**NULL**). Na linguagem C a palavra **NULL** é uma constante padrão que significa um valor nulo. Um buffer preenchido com **NULL** é considerado limpo/vazio.

## 6.2.2 ESCRIVENDO UMA STRING NA TELA

### USANDO A FUNÇÃO PRINTF()

Basicamente, para se escrever uma string na tela utilizamos a função **printf()** com o formato de dados “%s”:

```
char str[20] = "Hello World";  
printf("%s",str);
```



Para escrever uma string, utilizamos o tipo de saída “%s”. Os colchetes não são utilizados pois queremos escrever a string toda e não apenas uma letra.

### USANDO A FUNÇÃO FPUTS()

No entanto, existe uma outra função que, utilizada de forma adequada, também permite a escrita de strings. Essa função é a **fputs()**, cujo protótipo é:

```
int fputs (char *str,FILE *fp);
```

A função **fputs()** recebe 2 parâmetros de entrada

- str: a string (array de caracteres) a ser escrita na tela;
- fp: a variável que está associado ao *arquivo* onde a string será escrita.

e retorna

- a constante **EOF** (em geral, -1), se houver erro na escrita;
- um valor diferente de ZERO, se o texto for escrito com sucesso.

Note que a função **fputs()** utiliza uma variável FILE \*fp, que está associado ao arquivo onde a string será escrita.



Para escrever no monitor, basta substituir *FILE \*fp* por *stdout*, o qual representa o dispositivo de saída padrão (geralmente a tela do monitor).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char texto[30] = "Hello World\n";
5     fputs(texto, stdout);
6     system("pause");
7     return 0;
8 }
```

## 6.3 FUNÇÕES PARA MANIPULAÇÃO DE STRINGS

A biblioteca padrão da linguagem C possui funções especialmente desenvolvidas para a manipulação de strings na biblioteca <string.h>. A seguir são apresentadas algumas das mais utilizadas.

### 6.3.1 TAMANHO DE UMA STRING

Para se obter o tamanho de uma string, usa-se a função **strlen()**:

```
char str[15] = "teste";  
printf("%d",strlen(str));
```

Neste caso, a função retornará 5, que é o número de caracteres na palavra "teste" e não 15, que é o tamanho do array de caracteres.



A função `strlen()` retorna o número de caracteres até o caractere '\0', e não o tamanho do array onde a string está armazenada.

### 6.3.2 COPIANDO UMA STRING

Vimos que uma string é um array e que a linguagem C não suporta a atribuição de um array para outro. Nesse sentido, a única maneira de atribuir o conteúdo de uma string a outra é a cópia, elemento por elemento, de uma string para outra. A linguagem C possui uma função que realiza essa tarefa para nós: a função **`strcpy()`**:

```
strcpy(char *destino, char *origem)
```

Basicamente, a função **`strcpy()`** copia a sequência de caracteres contida em *origem* para o array de caracteres *destino*:

#### Exemplo: `strcpy()`

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 int main(){  
4     char str1[100], str2[100];  
5     printf("Entre com uma string: ");  
6     gets(str1);  
7     strcpy(str2, str1);  
8     system("pause");  
9     return 0;  
10 }
```



Para evitar estouro de buffer, o tamanho do array *destino* deve ser longo o suficiente para conter a sequência de caracteres contida em *origem*.

### 6.3.3 CONCATENANDO STRINGS

A operação de concatenação é outra tarefa bastante comum ao se trabalhar com strings. Basicamente, essa operação consistem em copiar uma string para o final de outra string. Na linguagem C, para se fazer a concatenação de duas strings, usa-se a função **strcat()**:

```
strcat(char *destino, char *origem)
```

Basicamente, a função **strcat()** copia a sequência de caracteres contida em *origem* para o final da string *destino*. O primeiro caractere da string contida em *origem* é colocado no lugar do caractere '\0' da string *destino*:

#### Exemplo: strcat()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char str1[15] = 'bom ';
5     char str2[15] = 'dia';
6     strcat(str1, str2);
7     printf('%s', str1);
8     system('pause');
9     return 0;
10 }
```



Para evitar estouro de buffer, o tamanho do array *destino* deve ser longo o suficiente para conter a sequência de caracteres contida em ambas as strings: *origem* e *destino*.

### 6.3.4 COMPARANDO DUAS STRINGS

Da mesma maneira como o operador de atribuição não funciona para strings, o mesmo ocorre com operadores relacionais usados para comparar duas strings. Desse modo, para saber se duas strings são iguais usa-se a função **strcmp()**:

```
int strcmp(char *str1, char *str2)
```

A função **strcmp()** compara posição a posição as duas strings (str1 e str2) e retorna um valor inteiro igual a zero no caso das duas strings serem iguais. Um valor de retorno diferente de zero significa que as strings são diferentes:

#### Exemplo: strcmp()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char str1[100], str2[100];
5     printf("Entre com uma string: ");
6     gets(str1);
7     printf("Entre com outra string: ");
8     gets(str2);
9     if (strcmp(str1, str2) == 0)
10        printf("Strings iguais\n");
11    else
12        printf("Strings diferentes\n");
13    system("pause");
14    return 0;
15 }
```



A função **strcmp()** é case-sensitive. Isso significa que letras maiúsculas e minúsculas tornam as strings diferentes.

## 7 TIPOS DEFINIDOS PELO PROGRAMADOR

Os tipos de variáveis vistos até agora podem ser classificados em duas categorias:

- tipos básicos: char, int, float, double e void;
- tipos compostos homogêneos: array.

Dependendo da situação que desejamos modelar em nosso programa, esses tipos existentes podem não ser suficientes. Por esse motivo, a linguagem C permite criar novos tipos de dados a partir dos tipos básicos. Para criar um novo tipo de dado, um dos seguintes comandos pode ser utilizado:

- Estruturas: comando **struct**
- Uniões: comando **union**
- Enumerações: comando **enum**
- Renomear um tipo existente: comando **typedef**

Nas seções seguintes, cada um desses comandos será apresentado em detalhes.

### 7.1 ESTRUTURAS: STRUCT

Uma estrutura pode ser vista como um conjunto de variáveis sob um mesmo nome, sendo que cada uma delas pode ter qualquer tipo (ou o mesmo tipo). A idéia básica por trás da estrutura é criar apenas um tipo de dado que contenha vários membros, que nada mais são do que outras variáveis. Em outras palavras, estamos criando uma variável que contém dentro de si outras variáveis.

#### DECLARANDO UMA ESTRUTURA

A forma geral da definição de uma nova estrutura é utilizando o comando **struct**:

```
struct nome_struct{  
    tipo1 campo1;
```

```
tipo2 campo2;  
...  
tipon campoN;  
};
```

A principal vantagem do uso de estruturas é que agora podemos agrupar de forma organizada vários tipos de dados diferentes dentro de uma única variável.



As estruturas podem ser declaradas em qualquer escopo do programa (global ou local).

Apesar disso, a maioria das estruturas são declaradas no escopo global. Por se tratar de um novo tipo de dado, muitas vezes é interessante que todo o programa tenha acesso a estrutura. Daí a necessidade de usar o escopo global.

Abaixo, tem-se um exemplo de uma estrutura declarada para representar o cadastro de uma pessoa:

Exemplo de estrutura.					
<pre>1 struct cadastro{ 2   char nome[50]; 3   int idade; 4   char rua[50]; 5   int numero; 6 };</pre>	<table border="1"><tr><td>char nome[50];</td></tr><tr><td>int idade;</td></tr><tr><td>char rua[50];</td></tr><tr><td>int numero;</td></tr></table> <p>cadastro</p>	char nome[50];	int idade;	char rua[50];	int numero;
char nome[50];					
int idade;					
char rua[50];					
int numero;					

Note que os campos da estrutura são definidos da mesma forma que variáveis. Como na declaração de variáveis, os nomes dos membros de uma estrutura devem ser diferentes um do outro. Porém, estruturas diferentes podem ter membros com nomes iguais:

```

struct cadastro{
    char nome[50];
    int idade;
    char rua[50];
    int numero; };
struct aluno{
    char nome[50];
    int matricula
    float nota1,nota2,nota3;
};

```



Depois do símbolo de fecha chaves (}) da estrutura é necessário colocar um ponto e vírgula (;).

Isso é necessário uma vez que a estrutura pode ser também declarada no escopo local. Por questões de simplificações, e por se tratar de um novo tipo, é possível logo na definição da struct definir algumas variáveis desse tipo. Para isso, basta colocar os nomes das variáveis declaradas após o comando de fecha chaves (}) da estrutura e antes do ponto e vírgula (;):

```

struct cadastro{
    char nome[50];
    int idade;
    char rua[50];
    int numero;
} cad1, cad2;

```

No exemplo acima, duas variáveis (*cad1* e *cad2*) são declaradas junto com a definição da estrutura.

#### DECLARANDO UMA VARIÁVEL DO TIPO DA ESTRUTURA

Uma vez definida a estrutura, uma variável pode ser declarada de modo similar aos tipos já existente:

```

struct cadastro c;

```



Por ser um tipo definido pelo programador, usa-se a palavra *struct* antes do tipo da nova variável declarada.

O uso de estruturas facilita muito a vida do programador na manipulação dos dados do programa. Imagine ter que declarar 4 cadastros, para 4 pessoas diferentes:

```
char nome1[50], nome2[50], nome3[50], nome4[50];  
int idade1, idade2, idade3, idade4;  
char rua1[50], rua2[50], rua3[50], rua4[50];  
int numero1, numero2, numero3, numero4;
```

Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:

```
struct cadastro c1, c2, c3, c4;
```

## ACESSANDO OS CAMPOS DE UMA ESTRUTURA

Uma vez definida uma variável do tipo da estrutura, é preciso poder acessar seus campos (ou variáveis) para se trabalhar.



Cada campo (variável) da estrutura pode ser acessada usando o operador “.” (ponto).

O operador de acesso aos campos da estrutura é o ponto (.). Ele é usado para referenciar os campos de uma estrutura. O exemplo abaixo mostra como os campos da estrutura *cadastro*, definida anteriormente, podem ser facilmente acessados:

### Exemplo: acessando as variáveis de dentro da estrutura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main (){
10     struct cadastro c;
11     //Atribui a string "Carlos" para o campo nome
12     strcpy(c.nome, "Carlos");
13
14     //Atribui o valor 18 para o campo idade
15     c.idade = 18;
16
17     //Atribui a string "Avenida Brasil" para o campo
18     //rua
19     strcpy(c.rua, "Avenida Brasil");
20
21     //Atribui o valor 1082 para o campo numero
22     c.numero = 1082;
23
24     system("pause");
25     return 0;
26 }
```

Como se pode ver, cada campo da estrutura é tratado levando em consideração o tipo que foi usado para declará-la. Como os campos *nome* e *rua* são **strings**, foi preciso usar a função **strcpy()** para copiar o valor para esses campos.



E se quiséssemos ler os valores dos campos da estrutura do teclado?

Nesse caso, basta ler cada variável da estrutura independentemente, respeitando seus tipos, como é mostrado no exemplo abaixo:

### Exemplo: lendo do teclado as variáveis da estrutura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main (){
10 struct cadastro c;
11 //Lê do teclado uma string e armazena no campo nome
12 gets(c.nome);
13
14 //Lê do teclado um valor inteiro e armazena no campo
    idade
15 scanf("%d",&c.idade);
16
17 //Lê do teclado uma string e armazena no campo rua
18 gets(c.rua);
19
20 //Lê do teclado um valor inteiro e armazena no campo
    numero
21 scanf("%d",&c.numero);
22 system('pause');
23 return 0;
24 }
```

Note que cada variável dentro da estrutura pode ser acessada como se apenas ela existisse, não sofrendo nenhuma interferência das outras.



Lembre-se: uma estrutura pode ser vista como um simples agrupamento de dados.

Como cada campo é independente um do outro, outros operadores podem ser aplicados a cada campo. Por exemplo, pode se comparar a idade de dois cadastros.

#### 7.1.1 INICIALIZAÇÃO DE ESTRUTURAS

Assim como nos arrays, uma estrutura também pode ser inicializada, independente do tipo das variáveis contidas nela. Para tanto, na declaração da variável do tipo da estrutura, basta definir uma lista de valores separados por vírgula e delimitado pelo operador de chaves {}.

```
struct cadastro c = {"Carlos",18,"Avenida Brasil",1082 };
```

Nesse caso, como nos arrays, a ordem é mantida. Isso significa que o primeiro valor da inicialização será atribuído a primeira variável membro (*nome*) da estrutura e assim por diante.

Elementos omitidos durante a inicialização são inicializados com 0. Se for uma string, a mesma será inicializada com uma string vazia ("").

```
struct cadastro c = {"Carlos",18 };
```

No exemplo acima, o campo *rua* é inicializado com "" e *numero* com zero.

### 7.1.2 ARRAY DE ESTRUTURAS

Voltemos ao problema do cadastro de pessoas. Vimos que o uso de estruturas facilita muito a vida do programador na manipulação dos dados do programa. Imagine ter que declarar 4 cadastros, para 4 pessoas diferentes:

```
char nome1[50], nome2[50], nome3[50], nome4[50];  
int idade1, idade2, idade3, idade4;  
char rua1[50], rua2[50], rua3[50], rua4[50];  
int numero1, numero2, numero3, numero4;
```

Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:

```
struct cadastro c1, c2, c3, c4;
```

A representação desses 4 cadastros pode ser ainda mais simplificada se utilizarmos o conceito de arrays:

```
struct cadastro c[4];
```

Desse modo, cria-se um array de estruturas, onde cada posição do array é uma estrutura do tipo cadastro.



A declaração de uma array de estruturas é similar a declaração de uma array de um tipo básico.

A combinação de arrays e estruturas permite que se manipule de modo muito mais prático várias variáveis de estrutura. Como vimos no uso de arrays, o uso de um índice permite que usemos comando de repetição para executar uma mesma tarefa para diferentes posições do array. Agora, os quatro cadastros anteriores podem ser lidos com o auxílio de um comando de repetição:

#### Exemplo: lendo um array de estruturas do teclado

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main (){
10    struct cadastro c[4];
11    int i;
12    for (i=0; i<4; i++){
13        gets(c[i].nome);
14        scanf("%d",&c[i].idade);
15        gets(c[i].rua);
16        scanf("%d",&c[i].numero);
17    }
18    system("pause");
19    return 0;
20 }
```



Em um array de estruturas, o operador de ponto (.) vem depois dos colchetes [ ] do índice do array.

Essa ordem deve ser respeitada pois o índice do array é quem indica qual posição do array queremos acessar, onde cada posição do array é uma estrutura. Somente depois de definida qual das estruturas contidas dentro do array nós queremos acessar é que podemos acessar os seus campos.

### 7.1.3 ATRIBUIÇÃO ENTRE ESTRUTURAS

As únicas operações possíveis em um estrutura são as de acesso aos membros da estrutura, por meio do operador ponto (.), e as de cópia ou atribuição (=). A atribuição entre duas variáveis de estrutura faz com que os conteúdos das variáveis contidas dentro de uma estrutura sejam copiado para outra estrutura.



Atribuições entre estruturas só podem ser feitas quando as estruturas são **AS MESMAS**, ou seja, possuem o mesmo nome!

#### Exemplo: atribuição entre estruturas

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     int x;
6     int y;
7 };
8
9 struct novo_ponto {
10    int x;
11    int y;
12 };
13
14 int main () {
15     struct ponto p1, p2= {1,2};
16     struct novo_ponto p3= {3,4};
17
18     p1 = p2;
19     printf( 'p1 = %d e %d', p1.x,p1.y);
20
21     //ERRO! TIPOS DIFERENTES
22     p1 = p3;
23     printf( 'p1 = %d e %d', p1.x,p1.y);
24
25     system( 'pause' );
26     return 0;
27 }
```

No exemplo acima, *p2* é atribuído a *p1*. Essa operação está correta pois ambas as variáveis são do tipo **ponto**. Sendo assim, o valor de *p2.x* é copiado para *p1.x* e o valor de *p2.y* é copiado para *p1.y*.

Já na segunda atribuição ( $p1 = p3$ ;) ocorre um erro. Isso por que os tipos das estruturas das variáveis são diferentes: uma pertence ao tipo *struct ponto* enquanto a outra pertence ao tipo *struct novo\_ponto*. Note que o mais importante é o nome do tipo da estrutura, e não as variáveis dentro dela.



No caso de estarmos trabalhando com arrays de estruturas, a atribuição entre diferentes elementos do array também é válida.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main(){
10     struct cadastro c[10];
11     ...
12     c[1] = c[2]; //CORRETO
13
14     system(' 'pause' ');
15     return 0;
16 }
```

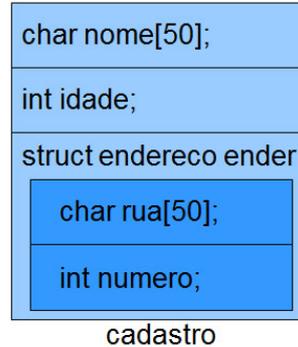
Um array ou “vetor” é um conjunto de variáveis do mesmo tipo utilizando apenas um nome. Como todos os elementos do array são do mesmo tipo, a atribuição entre elas é possível, mesmo que o tipo do array seja uma estrutura.

#### 7.1.4 ESTRUTURAS ANINHADAS

Uma estrutura pode agrupar um número arbitrário de variáveis de tipos diferentes. Uma estrutura também é um tipo de dado, com a diferença de se trata de um tipo de dado criado pelo programador. Sendo assim, podemos declarar uma estrutura que possua uma variável do tipo de outra estrutura previamente definida. A uma estrutura que contenha outra estrutura dentro dela damos o nome de **estruturas aninhadas**. O exemplo abaixo exemplifica bem isso:

### Exemplo: struct aninhada.

```
1 struct endereco{
2   char rua[50]
3   int numero;
4 };
5 struct cadastro{
6   char nome[50];
7   int idade;
8   struct endereco
9     ender;
9 };
```



No exemplo acima, temos duas estruturas: uma chamada **endereco** e outra chamada de **cadastro**. Note que a estrutura **cadastro** possui uma variável *ender* do tipo **struct endereco**. Trata-se de uma estrutura aninhada dentro de outra.



No caso da estrutura **cadastro**, o acesso aos dados da variável do tipo **struct endereco** é feito utilizando-se novamente o operador "." (ponto).

Lembre-se, cada campo (variável) da estrutura pode ser acessada usando o operador "." (ponto). Assim, para acessar a variável *ender* é preciso usar o operador ponto (.). No entanto, a variável *ender* também é uma estrutura. Sendo assim, o operador ponto (.) é novamente utilizado para acessar as variáveis dentro dessa estrutura. Esse processo se repete sempre que houver uma nova estrutura aninhada. O exemplo abaixo mostra como a estrutura aninhada **cadastro** poderia ser facilmente lida do teclado:

### Exemplo: lendo do teclado as variáveis da estrutura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct endereco{
4     char rua[50]
5     int numero;
6 };
7 struct cadastro{
8     char nome[50];
9     int idade;
10    struct endereco ender;
11 };
12 int main (){
13     struct cadastro c;
14     //Lê do teclado uma string e armazena no campo nome
15     gets(c.nome);
16
17     //Lê do teclado um valor inteiro e armazena no campo
18     //idade
19     scanf("%d",&c.idade);
20
21     //Lê do teclado uma string
22     //e armazena no campo rua da variável ender
23     gets(c.ender.rua);
24
25     //Lê do teclado um valor inteiro
26     //e armazena no campo numero da variável ender
27     scanf("%d",&c.ender.numero);
28
29     system('pause');
30     return 0;
31 }
```

## 7.2 UNIÕES: UNION

Uma união pode ser vista como uma lista de variáveis, sendo que cada uma delas pode ter qualquer tipo. A idéia básica por trás da união é similar a da estrutura: criar apenas um tipo de dado que contenha vários membros, que nada mais são do que outras variáveis.



Tanto a declaração quanto o acesso aos elementos de uma união são similares aos de uma estrutura.

### DECLARANDO UMA UNIÃO

A forma geral da definição de uma união é utilizando o comando **union**:

```
union nome_union{  
    tipo1 campo1;  
    tipo2 campo2;  
    ...  
    tipon campoN;  
};
```

### DIFERENÇA ENTRE ESTRUTURA E UNIÃO

Até aqui, uma união se parece muito com uma estrutura. No entanto, diferente das estruturas, todos os elementos contidos na união ocupam o mesmo espaço físico na memória. Uma estrutura reserva espaço de memória para todos os seus elementos, enquanto que numa **union** reserva espaço de memória para o seu maior elemento e compartilha essa memória com os demais elementos.



Numa **struct** é alocado espaço suficiente para armazenar todos os seus elementos, enquanto que numa **union** é alocado espaço para armazenar o maior dos elementos que a compõem.

Tome como exemplo a seguinte declaração de união:

```
union tipo{  
    short int x;  
    unsigned char c; };
```

Essa união possui o nome **tipo** e duas variáveis: *x*, do tipo **short int** (2 bytes), e *c*, do tipo **unsigned char** (1 byte). Assim, uma variável declarada desse tipo

```
union tipo t;
```

ocupará 2 (DOIS) bytes na memória, que é o tamanho do maior dos elementos da união (**short int**).



Em uma união, apenas um membro poderá ser armazenado de cada vez.

Isso acontece por que o espaço de memória é compartilhado. Portanto, é de total responsabilidade do programador saber qual o dado foi mais recentemente armazenado em uma união.



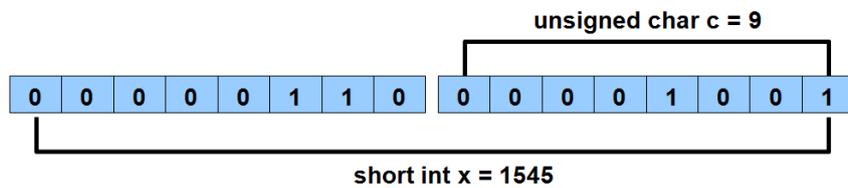
Como todos os elementos de uma união se referem a um mesmo local na memória, a modificação de um dos elementos afetará o valor de todos os demais. Numa união é impossível armazenar valores independentes.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 union tipo{
4     short int x;
5     unsigned char c;
6 };
7 int main(){
8     union tipo t;
9     t.x = 1545;
10    printf('x = %d\n', t.x);
11    printf('c = %d\n', t.c);
12    t.c = 69;
13    printf('x = %d\n', t.x);
14    printf('c = %d\n', t.c);
15    system('pause');
16    return 0;
17 }
```

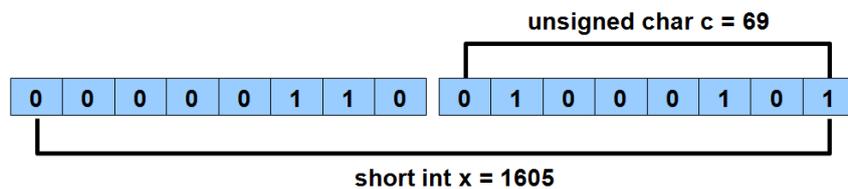
Saída

```
x = 1545
c = 9
x = 1605
c = 69
```

No exemplo acima, a variável *x* é do tipo **short int** e ocupa 16 bits (2 bytes) de memória. Já a variável *c* é do tipo **unsigned char** e ocupa os 8 (OITO) primeiros bits (1 byte) de *x*. Quando atribuímos o valor 1545 a variável *x*, a variável *c* receberá a porção de *x* equivalente ao número 9:



Do mesmo modo, se modificarmos o valor da variável *c* para 69, estaremos automaticamente modificando o valor da variável *x* para 1605:

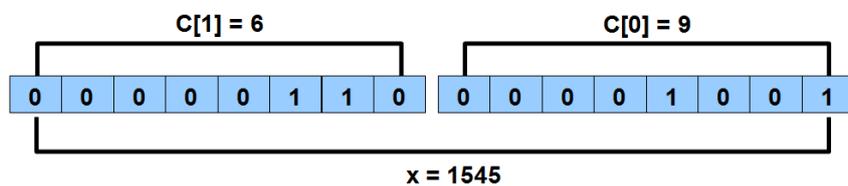


Um dos usos mais comuns de uma união é unir um tipo básico a um array de tipos menores.

Tome como exemplo a seguinte declaração de união:

```
union tipo{
    short int x;
    unsigned char c[2]; };
```

Sabemos que a variável *x* ocupa 2 bytes na memória. Como cada posição da variável *c* ocupa apenas 1 byte, podemos acessar facilmente cada uma das partes da variável *x*, sem precisar recorrer a operações de manipulação de bits (operações lógicas e de deslocamento de bits):



### 7.3 ENUMERAÇÕES: ENUM

Uma enumeração pode ser vista como uma lista de constantes, onde cada constante possui um nome significativo. A ideia básica por trás da enumeração

é criar apenas um tipo de dado que contenha várias constante, sendo que uma variável desse tipo só poderá receber como valor uma dessas constantes.

## DECLARANDO UMA ENUMERAÇÃO

A forma geral da definição de uma enumeração é utilizando o comando **enum**:

```
enum nome_enum {lista_de_identificadores };
```

Na declaração acima, **lista\_de\_identificadores** é uma lista de palavras separadas por vírgula e delimitadas pelo operador de chaves `{}`. Essas palavras constituem as constantes definidas pela enumeração. Por exemplo, o comando

```
enum semana {Domingo, Segunda, Terca, Quarta, Quinta, Sexta,  
Sabado };
```

cria uma enumeração de nome **semana**, onde seus valores constantes são os nomes dos dias da semana.



As estruturas podem ser declaradas em qualquer escopo do programa (global ou local).

Apesar disso, a maioria das enumerações são declaradas no escopo global. Por se tratar de um novo tipo de dado, muitas vezes é interessante que todo o programa tenha acesso a enumeração. Daí a necessidade de usar o escopo global.



Depois do símbolo de fecha chaves `}` da enumeração é necessário colocar um ponto e vírgula `;`.

Isso é necessário uma vez que a enumeração pode ser também declarada no escopo local. Por questões de simplificações, e por se tratar de um novo tipo, é possível logo na definição da enumeração definir algumas variáveis desse tipo. Para isso, basta colocar os nomes das variáveis declaradas após o comando de fecha chaves `}` da enumeração e antes do ponto e vírgula `;`:

```
enum semana {Domingo, Segunda, Terca, Quarta, Quinta, Sexta,  
Sabado }s1, s2;
```

No exemplo acima, duas variáveis (*s1* e *s2*) são declaradas junto com a definição da enumeração.

### DECLARANDO UMA VARIÁVEL DO TIPO DA ENUMERAÇÃO

Uma vez definida a enumeração, uma variável pode ser declarada de modo similar aos tipos já existente

```
enum semana s;
```

e inicializada como qualquer outra variável, usando, para isso, uma das constantes da enumeração

```
s = Segunda;
```



Por ser um tipo definido pelo programador, usa-se a palavra *enum* antes do tipo da nova variável declarada.

### ENUMERAÇÕES E CONSTANTES

Para o programador, uma enumeração pode ser vista como uma lista de constantes, onde cada constante possui um nome significativo. Porém, para o compilador, cada uma das constantes é representada por um valor inteiro, sendo que o valor da primeira constante da enumeração é 0 (ZERO). Desse modo, uma enumeração pode ser usada em qualquer expressão válida com inteiros, como mostra o exemplo abaixo:

### Exemplo: enumeração e inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 enum semana {Domingo, Segunda, Terca, Quarta,
4   Sexta, Sabado};
5 int main(){
6   enum semana s1, s2, s3;
7   s1 = Segunda;
8   s2 = Terca;
9   s3 = s1 + s2;
10  printf( "Domingo = %d\n", Domingo);
11  printf( "s1 = %d\n", s1);
12  printf( "s2 = %d\n", s2);
13  printf( "s3 = %d\n", s3);
14  system( "pause");
15  return 0;
16 }
```

Saída

```
Domingo = 0
s1 = 1
s2 = 2
s3 = 3
```

No exemplo acima, a constante **Domingo**, **Segunda** e **Terca**, possuem, respectivamente, os valores 0 (ZERO), 1 (UM) e 2 (DOIS). Como o compilador trata cada uma das constantes internamente como um valor inteiro, é possível somar as enumerações, ainda que isso não faça muito sentido.



Na definição da enumeração, pode-se definir qual valor aquela constante possuirá.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 enum semana {Domingo = 1, Segunda, Terca, Quarta
              =7, Quinta, Sexta, Sabado};
4 int main(){
5     printf( "Domingo = %d\n", Domingo);
6     printf( "Segunda = %d\n", Segunda);
7     printf( "Terca = %d\n", Terca);
8     printf( "Quarta = %d\n", Quarta);
9     printf( "Quinta = %d\n", Quinta);
10    printf( "Sexta = %d\n", Sexta);
11    printf( "Sabado = %d\n", Sabado);
12    system( "pause");
13    return 0;
14 }
```

Saída

```
Domingo = 1
Segunda = 2
Terca = 3
Quarta = 7
Quinta = 8
Sexta = 9
Sabado = 10
```

No exemplo acima, a constante **Domingo** foi inicializada com o valor 1 (UM). As constantes da enumeração que não possuem valor definido são definidas automaticamente como o valor do elemento anterior acrescidos de um. Assim, **Segunda** é inicializada com 2 (DOIS) e **Terca** com 3 (TRÊS). Para a constante **Quarta** foi definido o valor 7 (SETE). Assim, as constantes definidas na sequência após a constante **Quarta** possuirão os valores 8 (OITO), 9 (NOVE) e 10 (DEZ).



Na definição da enumeração, pode-se também atribuir valores da tabela ASCII para as constante.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 enum escapes {retrocesso='\b', tabulacao='\t',
4               novalinha='\n'};
5 int main(){
6     enum escapes e = novalinha;
7     printf("Teste %c de %c escrita\n",e,e);
8     e = tabulacao;
9     printf("Teste %c de %c escrita\n",e,e);
10    system("pause");
11    return 0;
}
```

Saída	Teste de escrita Teste de escrita
-------	--

## 7.4 COMANDO TYPEDEF

A linguagem C permite que o programador defina os seus próprios tipos baseados em outros tipos de dados existentes. Para isso, utiliza-se o comando **typedef**, cuja forma geral é:

```
typedef tipo_existente novo_nome;
```

onde **tipo\_existente** é um tipo básico ou definido pelo programador (por exemplo, uma **struct**) e **novo\_nome** é o nome para o novo tipo estamos definindo.



O comando **typedef** NÃO cria um novo tipo. Ele apenas permite que você defina um sinônimo para um tipo já existente.

Pegue como exemplo o seguinte comando:

```
typedef int inteiro;
```

O comando **typedef** não cria um novo tipo chamado *inteiro*. Ele apenas cria um sinônimo (*inteiro*) para o tipo **int**. Esse novo nome torna-se equivalente ao tipo já existente.



No comando **typedef**, o sinônimo e o tipo existente são equivalentes.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef int inteiro;
4 int main(){
5     int x = 10;
6     inteiro y = 20;
7     y = y + x;
8     printf( 'Soma = %d\n' ,y );
9     system( 'pause' );
10    return 0;
11 }
```

No exemplo acima, as variáveis do tipo **int** e **inteiro** são usadas de maneira conjunta. Isso ocorre pois elas são, na verdade, do mesmo tipo (**int**). O comando **typedef** apenas disse ao compilador para reconhecer **inteiro** como um outro nome para o tipo **int**.



O comando **typedef** pode ser usado para simplificar a declaração de um tipo definido pelo programador (**struct**, **union**, etc) ou de um ponteiro.

Imagine a seguinte declaração de uma **struct**:

```
struct cadastro{
    char nome[50];
    int idade;
    char rua[50];
    int numero; };
```

Para declarar uma variável deste tipo na linguagem C a palavra-chave **struct** é necessária. Assim, a declaração de uma variável *c* dessa estrutura seria:

```
struct cadastro c;
```



O comando **typedef** tem como objetivo atribuir nomes alternativos aos tipos já existentes, na maioria das vezes aqueles cujo padrão de declaração é pesado e potencialmente confuso.

O comando **typedef** pode ser usado para eliminar a necessidade da palavra-chave **struct** na declaração de variáveis. Por exemplo, usando o comando:

```
typedef struct cadastro cad;
```

Podemos agora declarar uma variável deste tipo usando apenas a palavra **cad**:

```
cad c;
```



O comando **typedef** pode ser combinado com a declaração de um tipo definido pelo programador (**struct**, **union**, etc) em uma única instrução.

Tome como exemplo a **struct** cadastro declarada anteriormente:

```
typedef struct cadastro{  
    char nome[50];  
    int idade;  
    char rua[50];  
    int numero; } cad;
```

Note que a definição da estrutura está inserida no meio do comando do **typedef** formando, portanto, uma única instrução. Além disso, como estamos associando um novo nome a nossa **struct**, seu nome original pode ser omitido da declaração da **struct**:

```
typedef struct {  
    char nome[50];  
    int idade;  
    char rua[50];  
    int numero; } cad;
```



O comando **typedef** deve ser usado com cuidado pois ele pode produzir declarações confusas.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 typedef unsigned int positivos[5];
4 int main(){
5     positivos v;
6     int i;
7     for (i = 0; i < 5; i++){
8         printf("Digite o valor de v[%d]:",i);
9         scanf("%d",&v[i]);
10    }
11
12    for (i = 0; i < 5; i++)
13        printf("Valor de v[%d]: %d\n",i,v[i]);
14
15    system("pause");
16    return 0;
17 }
```

No exemplo acima, o comando **typedef** é usado para criar um sinônimo (**positivos**) para o tipo “array de 5 inteiros positivos” (**unsigned int [5]**). Apesar de válida, essa declaração é um tanto confusa já que o novo nome (**positivos**) não dá nenhum indicativo de que a variável declarada (**v**) seja um array e nem seu tamanho.

## 8 FUNÇÕES

Uma função nada mais é do que um bloco de código (ou seja, declarações e outros comandos) que podem ser nomeado e chamado de dentro de um programa. Em outras palavras, uma função é uma seqüência de comandos que recebe um nome e pode ser chamada de qualquer parte do programa, quantas vezes forem necessárias, durante a execução do programa.

A linguagem C possui muitas funções já implementadas e nós temos utilizadas elas constantemente. Um exemplo delas são as funções básicas de entrada e saída: **scanf()** e **printf()**. O programador não precisa saber qual o código contido dentro das funções de entrada e saída para utilizá-las. Basta saber seu nome e como utilizá-la.

A seguir, serão apresentados os conceitos e detalhes necessários para um programador criar suas próprias funções.

### 8.1 DEFINIÇÃO E ESTRUTURA BÁSICA

Duas são as principais razões para o uso de funções:

- **estruturação dos programas;**
- **reutilização de código.**

Por **estruturação dos programas** entende-se que agora o programa será construído a partir de pequenos blocos de código (isto é, funções) cada um deles com uma tarefa específica e bem definida. Isso facilita a compreensão do programa.



Programas grandes e complexos são construídos bloco a bloco com a ajuda de funções.

Já por **reutilização de código** entende-se que uma função é escrita para realizar uma determinada tarefa. Pode-se definir, por exemplo, uma função para calcular o fatorial de um determinado número. O código para essa função irá aparecer uma única vez em todo o programa, mas a função que calcula o fatorial poderá ser utilizadas diversas vezes e em pontos diferentes do programa.



O uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros.

### 8.1.1 DECLARANDO UMA FUNÇÃO

Em linguagem C, a declaração de uma função pelo programador segue a seguinte forma geral:

```
tipo_retornado nome_função (lista_de_parâmetros){  
    sequência de declarações e comandos  
}
```

O **nome\_função** é como aquele trecho de código será conhecido dentro do programa. Para definir esse nome, valem, basicamente, as mesmas regras para se definir uma variável.

### LOCAL DE DECLARAÇÃO DE UMA FUNÇÃO

Com relação ao local de declaração de uma função, ela deve ser definida ou declarada antes de ser utilizada, ou seja, antes da cláusula **main**, como mostra o exemplo abaixo:



Exemplo: função declarada **antes** da cláusula *main*.

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int Square (int a){  
5     return (a*a);  
6 }  
7  
8 int main (){  
9     int n1,n2;  
10    printf(“Entre com um numero: ”);  
11    scanf(“%d”, &n1);  
12    n2 = Square(n1);  
13    printf(“O seu quadrado vale: %d\n”, n2);  
14    system(“pause”);  
15    return 0;  
16 }
```

Pode-se também declarar uma função depois da cláusula **main**. Nesse caso, é preciso declarar antes o protótipo da função:

```
tipo_retornado nome_função (lista_de_parâmetros);
```

O protótipo de uma função, é uma declaração de função que omite o corpo mas especifica o seu nome, tipo de retorno e lista de parâmetros, como mostra o exemplo abaixo:



Exemplo: função declarada **depois** da cláusula *main*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 //protótipo da função
4 int Square (int a);
5
6 int main () {
7     int n1, n2;
8     printf("Entre com um numero: ");
9     scanf("%d", &n1);
10    n2 = Square(n1);
11    printf("O seu quadrado vale: %d\n", n2);
12    system("pause");
13    return 0;
14 }
15
16 int Square (int a) {
17     return (a*a);
18 }
```



O protótipo de uma função não precisa incluir os nomes das variáveis passadas como parâmetros. Apenas os seus tipos já são suficientes.

A inclusão de nome de cada parâmetro no protótipo de uma função é uma tarefa opcional. Podemos declarar o seu protótipo apenas com os tipos dos parâmetros que serão passados para a função. Os nomes dos parâmetros são importantes apenas na implementação da função. Assim, ambos os protótipos abaixo são válidos para uma mesma função:

```
int Square (int a);
```

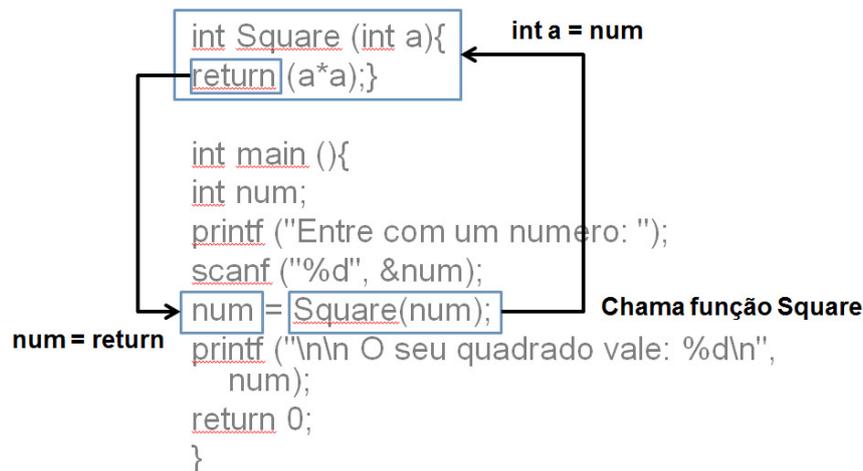
```
int Square (int );
```

## FUNCIONAMENTO DE UMA FUNÇÃO

Independente de onde uma função seja declarada, seu funcionamento é basicamente o mesmo:

- o código do programa é executado até encontrar uma chamada de função;
- o programa é então interrompido temporariamente, e o fluxo do programa passa para a função chamada;
- se houver parâmetros na função, os valores da chamada da função são copiados para os parâmetros no código da função;
- os comandos da função são executados;
- quando a função termina (seus comandos acabaram ou o comando **return** foi encontrado), o programa volta ao ponto onde foi interrompido para continuar sua execução normal;
- se houver um comando **return**, o valor dele será copiado para a variável que foi escolhida para receber o retorno da função.

Na figura abaixo, é possível ter uma boa representação de como uma chamada de função ocorre:



Nas seções seguintes, cada um dos itens que definem uma função serão apresentados em detalhes.

## 8.1.2 PARÂMETROS DE UMA FUNÇÃO

Os parâmetros de uma função são o que o programador utiliza para passar a informação de um trecho de código para dentro da função. Basicamente, os parâmetros de uma função são uma lista de variáveis, separadas por vírgula, onde é especificado o tipo e o nome de cada variável passada para a função.



Por exemplo, a função **sqrt** possui a seguinte lista de parâmetros: `float sqrt(float x)`;

## DECLARANDO OS PARÂMETROS DE UMA FUNÇÃO

Em linguagem C, a declaração dos parâmetros de uma função segue a seguinte forma geral:

```
tipo_retornado nome_função (tipo nome1, tipo nome2, ... ,  
tipo nomeN){  
    sequência de declarações e comandos  
}
```



Diferente do que acontece na declaração de variáveis, onde muitas variáveis podem ser declaradas com o mesmo especificador de tipo, na declaração de parâmetros de uma função é necessário especificar o tipo de cada variável.

```
1 //Declaração CORRETA de parâmetros  
2 int soma(int x, int y){  
3     return x + y;  
4 }  
5  
6 //Declaração ERRADA de parâmetros  
7 int soma(int x, y){  
8     return x + y;  
9 }
```

## FUNÇÕES SEM LISTA DE PARÂMETROS

Dependendo da função, ela pode possuir nenhum parâmetro. Nesse caso, pode-se optar por duas soluções:

- Deixar a lista de parâmetros vazia: void imprime ();
- Colocar void entre parênteses: void imprime (void).



Mesmo se não houver parâmetros na função, os parênteses ainda são necessários.

Apesar das duas declarações estarem corretas, existe uma diferença entre elas. Na primeira declaração, não é especificado nenhum parâmetro, portanto a função pode ser chamada passando-se valores para ela. O compilador não irá verificar se a função é realmente chamada sem argumentos e a função não conseguirá ter acesso a esses parâmetros. Já na segunda declaração, nenhum parâmetro é esperado. Nesse caso, o programa acusará um erro se o programador tentar passar um valor para essa função.



Colocar void na lista de parâmetros é diferente de se colocar nenhum parâmetro.

O exemplo abaixo ilustra bem essa situação:

Exemplo: função sem parâmetros	
Sem void	Com void
1 #include <stdio.h>	1 #include <stdio.h>
2 #include <stdlib.h>	2 #include <stdlib.h>
3	3
4 void imprime(){	4 void imprime(void){
5     printf(''Teste de	5     printf(''Teste de
funcao\n'');	funcao\n'');
6 }	6 }
7	7
8 int main (){	8 int main (){
9     imprime();	9     imprime();
10    imprime(5);	10    imprime(5); //ERRO
11    imprime(5, 'a');	11    imprime(5, 'a'); //ERRO
12	12
13    system(''pause'');	13    system(''pause'');
14    return 0;	14    return 0;
15 }	15 }

Os parâmetros das funções também estão sujeitos ao escopo das variáveis. O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa.



O parâmetro de uma função é uma variável local da função e portanto, só pode ser acessado dentro da função.

### 8.1.3 CORPO DA FUNÇÃO

Pode-se dizer que o corpo de uma função é a sua alma. É no corpo de uma função que se define qual a tarefa que a função irá realizar quando for chamada.

Basicamente, o corpo da função é formado por:

- sequência de declarações: variáveis, constantes, arrays, etc;
- sequência de comandos: comandos condicionais, de repetição, chamada de outras funções, etc.

Para melhor entender o corpo da função, considere que todo programa possui ao menos uma função: a função **main**. A função mais é a função “principal” do programa, o “corpo” do programa. Note que nos exemplo usados até agora, a função main é sempre do tipo int, e sempre retorna o valor 0:

```
int main () {  
    sequência de declarações e comandos  
    return 0;  
}
```

Basicamente, é no corpo da função que as entradas (parâmetros) são processadas, as saídas são geradas ou outras ações são feitas. Além disso, a função main se encarrega de realizar a comunicação com o usuário, ou seja, é ela quem realiza as operações de entrada e saída de dados (comandos **scanf** e **printf**). Desse modo, tudo o que temos feito dentro de uma função main pode ser feito em uma função desenvolvida pelo programador.



Tudo o que temos feito dentro da função main pode ser feito em uma função desenvolvida pelo programador.

Uma função é construída com o intuito de realizar uma tarefa específica e bem definida. Por exemplo, uma função para calcular o fatorial deve ser construída de modo a receber um determinado número como parâmetro e retornar (usando o comando `return`) o valor calculado. As operações de entrada e saída de dados (comandos **scanf** e **printf**) devem ser feitas em quem chamou a função (por exemplo, na **main**). Isso garante que a função construída possa ser utilizada nas mais diversas aplicações, garantindo a sua generalidade.



De modo geral, evita-se fazer operações de leitura e escrita dentro de uma função.

Os exemplos abaixo ilustram bem essa situação. No primeiro exemplo temos o cálculo do fatorial realizado dentro da função main:

#### Exemplo: cálculo do fatorial dentro da função main

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main () {
5     printf( "Digite um numero inteiro positivo: " );
6     int x;
7     scanf( "%d", &x );
8     int i, f = 1;
9     for ( i=1; i<=x; i++)
10         f = f * i;
11
12     printf( "O fatorial de %d eh: %d\n", x, f );
13     system( "pause" );
14     return 0;
15 }
```

Perceba que no exemplo acima, não foi feito nada de diferente do que temos feito até o momento. Já no exemplo abaixo, uma função específica para o cálculo do fatorial foi construída:

### Exemplo: cálculo do fatorial em uma função própria

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int fatorial (int n){
5     int i, f = 1;
6     for (i=1; i<=n; i++)
7         f = f * i;
8
9     return f;
10 }
11
12 int main (){
13     printf("Digite um numero inteiro positivo:");
14     int x;
15     scanf("%d",&x);
16     int fat = fatorial(x);
17     printf("O fatorial de %d eh: %d\n",x,fat);
18
19     system("pause");
20     return 0;
21 }
```

Note que dentro da função responsável pelo cálculo do fatorial, apenas o trecho do código responsável pelo cálculo do fatorial está presente. As operações de entrada e saída de dados (comandos **scanf** e **printf**) são feitos em quem chamou a função **fatorial**, ou seja, na função **main**.



Operações de leitura e escrita não são proibidas dentro de uma função. Apenas não devem ser usadas se esse não for o foco da função.

Uma função deve conter apenas o trecho de código responsável por fazer aquilo que é o objetivo da função. Isso não impede que operações de leitura e escrita sejam utilizadas dentro da função. Elas só não devem ser usadas quando os valores podem ser passados para a função por meio dos parâmetros.

Abaixo temos um exemplo de função que realiza operações de leitura e escrita:

### Exemplo: função contendo operações de leitura e escrita.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int menu(){
4     int i;
5     do {
6         printf(‘‘Escolha uma opção:\n’’);
7         printf(‘‘(1) Opcao 1\n’’);
8         printf(‘‘(2) Opcao 2\n’’);
9         printf(‘‘(3) Opcao 3\n’’);
10        scanf(‘‘%d’’, &i);
11    } while ((i < 1) || (i > 3));
12
13    return i;
14 }
15
16 int main(){
17     int op = menu();
18     printf(‘‘Vc escolheu a Opcao %d.\n’’,op);
19     system(‘‘pause’’);
20     return 0;
21 }
```

Na função acima, um menu de opções é apresentado ao usuário que tem de escolher dentre uma delas. A função se encarrega de verificar se a opção digitada é válida e, caso não seja, solicitar uma nova opção ao usuário.

#### 8.1.4 RETORNO DA FUNÇÃO

O retorno da função é a maneira como uma função devolve o resultado (se ele existir) da sua execução para quem a chamou. Nas seções anteriores vimos que uma função segue a seguinte forma geral:

```
tipo_retornado nome_função (lista_de_parâmetros){
    sequência de declarações e comandos
}
```

A expressão **tipo\_retornado** estabelece o tipo de valor que a função irá devolver para quem chamá-la. Uma função pode retornar qualquer tipo válido em na linguagem C:

- tipos básicos pré-definidos: int, char, float, double, void e ponteiros;

- tipos definidos pelo programador: struct, array (indiretamente), etc.

## FUNÇÕES SEM RETORNO DE VALOR



Uma função também pode NÃO retornar um valor. Para isso, basta colocar o tipo **void** como valor retornado.

O tipo **void** é conhecido como o tipo *vazio*. Uma função declarada com o tipo **void** irá apenas executar um conjunto de comando e não irá devolver nenhum valor para quem a chamar. Veja o exemplo abaixo:

### Exemplo: função com tipo void

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void imprime(int n){
4     int i;
5     for (i=1; i<=n; i++)
6         printf("Linha %d \n", i);
7 }
8
9 int main(){
10    imprime(5);
11
12    system("pause");
13    return 0;
14 }
```

No exemplo acima, a função **imprime** irá apenas imprimir uma mensagem na tela **n** vezes. Não há o que devolver para a função **main**. Portanto, podemos declarar ela como **void**.



Para executar uma função do tipo **void**, basta colocar no código onde a função será chamada o nome da função e seus parâmetros.

## FUNÇÕES COM RETORNO DE VALOR

Se a função não for do tipo **void**, então ela deverá retornar um valor. O comando **return** é utilizado para retornar esse valor para o programa:

**return** *expressão*;



A expressão da cláusula **return** tem que ser compatível com o tipo de retorno declarado para a função.

A *expressão* do comando **return** consiste em qualquer constante, variável ou expressão aritmética que o programador deseje retornar para o trecho do programa que chamou a função. Essa expressão pode até mesmo ser uma outra função, como a função **sqrt()**:

**return** sqrt(x);



Para executar uma função que tenha o comando **return**, basta atribuir a chamada da função (nome da função e seus parâmetros) a uma variável compatível com o tipo do retorno.

O exemplo abaixo mostra uma função que recebe dois parâmetros inteiros e retorna a sua soma para a função **main**:

#### Exemplo: função com retorno

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int soma(int x, int y){
4     return x + y;
5 }
6
7 int main(){
8     int a,b,c;
9     printf("Digite a: ");
10    scanf("%d", &a);
11    printf("Digite b: ");
12    scanf("%d", &b);
13    printf("Soma = %d\n",soma(a,b));
14    system("pause");
15    return 0;
16 }
```

Note, no exemplo acima, que a chamada da função foi feita dentro do comando **printf**. Isso é possível pois a função retorna um valor inteiro (**x+y**) e o comando **printf** espera imprimir um valor inteiro (**%d**).



Uma função pode ter mais de uma declaração **return**.

O uso de vários comandos **return** é útil quando o retorno da função está relacionado a uma determinada condição dentro da função. Veja o exemplo abaixo:

#### Exemplo: função com vários return

```
1 int maior(int x, int y){
2   if (x > y)
3     return x;
4   else
5     return y;
6 }
```

No exemplo acima, a função será executada e dependendo dos valores de  $x$  e  $y$ , uma das cláusulas **return** será executada. No entanto, é conveniente limitar as funções a usar somente um comando **return**. O uso de vários comandos **return**, especialmente em função grandes e complexas, aumenta a dificuldade de se compreender o que realmente está sendo feito pela função. Na maioria dos casos, pode-se reescrever uma função para que ela use somente um comando **return**, como é mostrado abaixo:

#### Exemplo: substituindo os vários return da função

```
1 int maior(int x, int y){
2   int z;
3   if (x > y)
4     z = x;
5   else
6     z = y;
7   return z;
8 }
```

No exemplo acima, os vários comando **return** foram substituídos por uma variável que será retornada no final da função.



Quando se chega a um comando **return**, a função é encerrada imediatamente.

O comando **return** é utilizado para retornar um valor para o programa. No entanto, esse comando também é usado para terminar a execução de uma função, similar ao comando **break** em um laço ou **switch**:

#### Exemplo: finalizando a função com return

```
1 int maior(int x, int y){
2     if (x > y)
3         return x;
4     else
5         return y;
6     printf("Fim da funcao\n");
7 }
```

No exemplo acima, a função irá terminar quando um dos comando **return** for executado. A mensagem "Fim da funcao" jamais será impressa na tela pois seu comando se encontra depois do comando **return**. Nesse caso, o comando **printf** será ignorado.



O comando **return** pode ser usado sem um valor associado a ele para terminar uma função do tipo **void**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 void imprime_log(float x){
5     if (x <= 0)
6         return; //termina a função
7     printf("Log = %f\n", log(x));
8 }
9 int main(){
10    float x;
11    printf("Digite x: ");
12    scanf("%f", &x);
13    imprime_log(x);
14    system("pause");
15    return 0;
16 }
```

Na função contida no exemplo acima, se o valor de  $x$  é negativo ou zero, o comando **return** faz com que a função termine antes que o comando **printf** seja executado, mas nenhum valor é retornado.



O valor retornado por uma função não pode ser um array.

Lembre-se: a linguagem C não suporta a atribuição de um array para outro. Por esse motivo, não se pode ter como retorno de uma função um array.



É possível retornar um array indiretamente, desde que ela faça parte de uma estrutura.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct vetor{
5     int v[5];
6 };
7
8 struct vetor retorna_array(){
9     struct vetor v = {1,2,3,4,5};
10    return v;
11 }
12
13 int main (){
14     int i;
15     struct vetor vet = retorna_array();
16     for (i=0; i<5; i++)
17         printf(“Valores: %d \n”,vet.v[i]);
18     system(“pause”);
19     return 0;
20 }
```

A linguagem C não suporta a atribuição de um array para outro. Mas ela permite a atribuição entre estruturas. A atribuição entre duas variáveis de estrutura faz com que os conteúdos das variáveis contidas dentro de uma estrutura sejam copiados para outra estrutura. Desse modo, é possível retornar um array desde que o mesmo esteja dentro de uma estrutura.

## 8.2 TIPOS DE PASSAGEM DE PARÂMETROS

Já vimos que, na linguagem C, os parâmetros de uma função é o mecanismo que o programador utiliza para passar a informação de um trecho de código para dentro da função. Mas existem dois tipos de passagem de parâmetro: passagem **por valor** e **por referência**.

Nas seções seguintes, cada um dos tipos de passagem de parâmetros será explicado em detalhes.

### 8.2.1 PASSAGEM POR VALOR

Na linguagem C, os argumentos para uma função são sempre passados por valor (by value), ou seja, uma cópia do dado é feita e passada para a função. Esse tipo de passagem de parâmetro é o padrão para todos os tipos básicos pré-definidos (**int**, **char**, **float** e **double**) e estruturas definidas pelo programador (**struct**).



Mesmo que o valor de uma variável mude dentro da função, nada acontece com o valor de fora da função.

```
1 include <stdio.h>
2 include <stdlib.h>
3
4 void soma_mais_um(int n){
5     n = n + 1;
6     printf("Dentro da funcao: x = %d\n",n);
7 }
8
9 int main (){
10     int x = 5;
11     printf("Antes da funcao: x = %d\n",x);
12     soma_mais_um(x);
13     printf("Depois da funcao: x = %d\n",x);
14     system("pause");
15     return 0;
16 }
```

Saída     Antes da funcao: x = 5  
           Dentro da funcao: x = 6  
           Depois da funcao: x = 5

No exemplo acima, no momento em que a função **soma\_mais\_um** é chamada, o valor de **x** é **copiado** para o parâmetro **n** da função. O parâmetro **n** é uma variável local da função. Então, tudo o que acontecer com ele (**n**) não se reflete no valor **original** da variável **x**. Quando a função termina, a variável **n** é destruída e seu valor é descartado. O fluxo do programa é devolvido ao ponto onde a função foi inicialmente chamada, onde a variável **x** mantém o seu valor **original**.



Na passagem de parâmetros por valor, quaisquer modificações que a função fizer nos parâmetros existem apenas dentro da própria função.

## 8.2.2 PASSAGEM POR REFERÊNCIA

Na passagem de parâmetros por valor, as funções não podem modificar o valor original de uma variável passada para a função. Mas existem casos em que é necessário que toda modificação feita nos valores dos parâmetros dentro da função sejam repassados para quem chamou a função. Um exemplo bastante simples disso é a função **scanf**: sempre que desejamos ler algo do teclado, passamos para a função **scanf** o nome da variável onde o dado será armazenado. Essa variável tem seu valor modificado dentro da função **scanf** e seu valor pode ser acessado no programa principal.



A função **scanf** é um exemplo bastante simples de função que altera o valor de uma variável e essa mudança se reflete fora da função.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int x = 5;
5     printf("Antes do scanf: x = %d\n",x);
6     printf("Digite um numero: ");
7     scanf("%d",&x);
8     printf("Depois do scanf: x = %d\n",x);
9     system("pause");
10    return 0;
11 }
```

Quando se quer que o valor da variável mude dentro da função e essa mudança se reflita fora da função, usa-se passagem de parâmetros **por referência**.



Na passagem de parâmetros por referência não se passa para a função os valores das variáveis, mas sim os *endereços das variáveis na memória*.

Na passagem de parâmetros por referência o que é enviado para a função é o endereço de memória onde a variável está armazenada, e não uma

simples cópia de seu valor. Assim, utilizando o endereço da variável na memória, qualquer alteração que a variável sofra dentro da função será também refletida fora da função.



Para passar um parâmetro por referência, usa-se o operador “\*” na frente do nome do parâmetro durante a declaração da função.

Para passar para a função um parâmetro por referência, a função precisa usar ponteiros. Um ponteiro é um tipo especial de variável que armazena um endereço de memória, da mesma maneira como uma variável armazena um valor. Mais detalhes sobre o uso de ponteiros serão apresentados no capítulo seguinte.

O exemplo abaixo mostra a mesma função declarada usando a passagem de parâmetro de valor e por referência:

Exemplo: passagem por valor e referência	
Por valor	Por referência
<pre>1 void soma_mais_um(int n)   { 2   n = n + 1; 3 }</pre>	<pre>1 void soma_mais_um(int *n)   { 2   *n = *n + 1; 3 }</pre>

Note, no exemplo acima, que a diferença entre os dois tipos de passagem de parâmetro é o uso do operador “\*” na passagem por referência. Consequentemente, toda vez que a variável passada por referência for usada dentro da função, o operador “\*” deverá ser usado na frente do nome da variável.



Na chamada da função é necessário utilizar o operador “&” na frente do nome da variável que será passada por referência.

Lembre-se do exemplo da função **scanf**. A função **scanf** é um exemplo de função que altera o valor de uma variável e essa mudança se reflete fora da função. Quando chamamos a função **scanf**, é necessário colocar o operador “&” na frente do nome da variável que será lida do teclado. O mesmo vale para outras funções que usam passagem de parâmetro por referência.



Na passagem de uma variável por referência é necessário usar o operador “\*” sempre que se desejar acessar o conteúdo da variável dentro da função.

```
1 include <stdio.h>
2 include <stdlib.h>
3
4 void soma_mais_um(int *n){
5     *n = *n + 1;
6     printf(“Dentro da funcao: x = %d\n”,*n);
7 }
8
9 int main (){
10    int x = 5;
11    printf(“Antes da funcao: x = %d\n”,x);
12    soma_mais_um(&x);
13    printf(“Depois da funcao: x = %d\n”,x);
14    system(“pause”);
15    return 0;
16 }
```

Saída     Antes da funcao: x = 5  
           Dentro da funcao: x = 6  
           Depois da funcao: x = 6

No exemplo acima, no momento em que a função **soma\_mais\_um** é chamada, o endereço de  $x$  ( $\&x$ ) é **copiado** para o parâmetro  $n$  da função. O parâmetro  $n$  é um ponteiro dentro da função que guarda o endereço de onde o valor de  $x$  está guardado fora da função. Sempre que alteramos o valor de  $*n$  (conteúdo da posição de memória guardada, ou seja,  $x$ ), o valor de  $x$  fora da função também é modificado.

Abaixo temos outro exemplo que mostra a mesma função declarada usando a passagem de parâmetro de valor e por referência:

Exemplo: passagem por valor e referência	
Por valor	Por referência
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 4 void Troca(int a, int b){ 5     int temp; 6     temp = a; 7     a = b; 8     b = temp; 9     printf("Dentro: %d e           %d\n", a, b); 10 } 11 12 int main(){ 13     int x = 2; 14     int y = 3; 15     printf("Antes: %d e %           d\n", x, y); 16     Troca(x, y); 17     printf("Depois: %d e           %d\n", x, y); 18     system("pause"); 19     return 0; 20 } </pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 4 void Troca(int *a, int *b){ 5     int temp; 6     temp = *a; 7     *a = *b; 8     *b = temp; 9     printf("Dentro: %d e           %d\n", *a, *b); 10 } 11 12 int main(){ 13     int x = 2; 14     int y = 3; 15     printf("Antes: %d e %           d\n", x, y); 16     Troca(&amp;x, &amp;y); 17     printf("Depois: %d e           %d\n", x, y); 18     system("pause"); 19     return 0; 20 } </pre>
<p><b>Saída</b>  Antes: 2 e 3  Dentro: 3 e 2  Depois: 2 e 3</p>	<p><b>Saída</b>  Antes: 2 e 3  Dentro: 3 e 2  Depois: 3 e 2</p>

### 8.2.3 PASSAGEM DE ARRAYS COMO PARÂMETROS

Para utilizar arrays como parâmetros de funções alguns cuidados simples são necessários. Além do parâmetro do array que será utilizado na função, é necessário declarar um segundo parâmetro (em geral uma variável inteira) para passar para a função o tamanho do array **separadamente**.



Arrays são sempre passados *por referência* para uma função.

Quando passamos um array por parâmetro, independente do seu tipo, o que é de fato passado para a função é o endereço do primeiro elemento do array.



A passagem de arrays por referência evita a cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função, o que afetaria o desempenho do programa.

Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime (int *m, int n);  
void imprime (int m[], int n);  
void imprime (int m[5], int n);
```



Mesmo especificando o tamanho de um array no parâmetro da função a semântica é a mesma das outras declarações, pois não existe checagem dos limites do array em tempo de compilação.

O exemplo abaixo mostra como um array de uma única dimensão pode ser passado como parâmetro para uma função:

### Exemplo: passagem de array como parâmetro

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 void imprime (int *n  
   , int m){  
5   int i;  
6   for (i=0; i<m;i++)  
7     printf("%d \n"  
   , n[i]);  
8 }  
9  
10 int main () {  
11   int v[5] =  
   {1,2,3,4,5};  
12   imprime(v,5);  
13   system('pause');  
14   return 0;  
15 }
```

Memória		
#	var	conteúdo
123	int *n	#125
124		
125		1
126		2
127		3
128		4
129		5

Note, no exemplo acima, que apenas o nome do array é passado para a função, sem colchetes. Isso significa que estamos passando o array inteiro.

Se usássemos o colchete, estaríamos passando o valor de uma posição do array e não o seu endereço, o que resultaria em um erro.



Na chamada da função, passamos para ela somente o nome do array, sem os colchetes: o programa “já sabe” que um array será enviado, pois isso já foi definido no protótipo da função.

Vimos que, para arrays, não é necessário especificar o número de elementos para a função no parâmetro do array:

```
void imprime (int *m, int n);
```

```
void imprime (int m[], int n);
```



Arrays com mais de uma dimensão (por exemplo, matrizes), precisam da informação do tamanho das dimensões extras.

Para arrays com mais de uma dimensão é necessário o tamanho de todas as dimensões, exceto a primeira. Sendo assim, uma declaração possível para uma matriz de 4 linhas e 5 colunas seria a apresentada abaixo:

```
void imprime (int m[][5], int n);
```

A declaração de arrays com uma dimensão e com mais de uma dimensão é diferente porque na passagem de um array para uma função o compilador precisar saber o tamanho de cada elemento, não o número de elementos.



Um array bidimensional poder ser entendido como um array de arrays.

Para a linguagem C, um array bidimensional poder ser entendido como um array de arrays. Sendo assim, o seguinte array

```
int m[4][5];
```

pode ser entendido como um array de 4 elementos, onde cada elemento é um array de 5 posições inteiras. Logo, o compilador precisa saber o tamanho de um dos elementos (por exemplo, o número de colunas da matriz) no momento da declaração da função:

```
void imprime (int m[][5], int n);
```

Na notação acima, informamos ao compilador que estamos passando um array, onde cada elemento dele é outro array de 5 posições inteiras. Nesse caso, o array terá sempre 5 colunas, mas poderá ter quantas linhas quiser (parâmetro **n**).

Isso é necessário para que o programa saiba que o array possui mais de uma dimensão e mantenha a notação de um conjunto de colchetes por dimensão.

O exemplo abaixo mostra como um array de duas dimensões pode ser passado como parâmetro para uma função:

#### Exemplo: passagem de matriz como parâmetro

```
1 #include <stdio .h>
2 #include <stdlib .h>
3
4 void imprime_matriz(int m[][2] , int n){
5     int i ,j ;
6     for (i=0; i<n; i++)
7         for (j=0; j< 2;j++)
8             printf(“%d \n” , m[i][j]);
9 }
10
11 int main (){
12     int mat[3][2] = {{1,2},{3,4},{5,6}};
13     imprime_matriz(mat,3);
14     system(“pause”);
15     return 0;
16 }
```

As notações abaixo funcionam para arrays com mais de uma dimensão. Mas o array é tratado como se tivesse apenas uma dimensão dentro da função

```
void imprime (int *m, int n);
```

```
void imprime (int m[], int n);
```

O exemplo abaixo mostra como um array de duas dimensões pode ser passado como um array de uma única dimensão para uma função:

### Exemplo: matriz como array de uma dimensão

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void imprime_matriz(int *m, int n){
5     int i;
6     for (i=0; i<n; i++)
7         printf("%d \n", m[i]);
8 }
9
10 int main (){
11     int mat[3][2] = {{1,2},{3,4},{5,6}};
12     imprime_matriz(&mat[0][0],6);
13     system('pause');
14     return 0;
15 }
```

Note que, nesse exemplo, ao invés de passarmos o nome do array nós passamos o endereço do primeiro elemento (&mat[0][0]). Isso faz com que percamos a notação de dois colchetes para a matriz, e ela seja tratada como se tivesse apenas uma dimensão.

### 8.2.4 PASSAGEM DE ESTRUTURAS COMO PARÂMETROS

Vimos anteriormente que uma estrutura pode ser vista como um conjunto de variáveis sob um mesmo nome ou, em outras palavras, a estrutura é uma variável que contém dentro de si outras variáveis. Sendo assim, uma estrutura pode ser passada como parâmetro para uma função de duas formas distintas:

- toda a estrutura;
- apenas determinados campos da estrutura.

### PASSAGEM DE ESTRUTURAS POR VALOR

Para passar uma estrutura como parâmetro de uma função, basta declarar na lista de parâmetros um parâmetro com o mesmo tipo da estrutura. Dessa forma, teremos acesso a todos os campos da estrutura dentro da função, como mostra o exemplo abaixo:

### Exemplo: estrutura como parâmetro da função

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct ponto {
4     int x, y;
5 };
6 void imprime(struct ponto p){
7     printf('x = %d\n',p.x);
8     printf('y = %d\n',p.y);
9 }
10 int main(){
11     struct ponto p1 = {10,20};
12     imprime(p1);
13     system('pause');
14     return 0;
15 }
```

Dependendo da aplicação, pode ser que não seja necessário passar todos os valores da estrutura para a função. Nesse caso, a função é declarada sem levar em conta a estrutura nos seus parâmetros. Mas é necessário que o parâmetro da função seja compatível com o campo da função que será passado como parâmetro, como mostra o exemplo abaixo:

### Exemplo: campo da estrutura como parâmetro da função

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct ponto {
4     int x, y;
5 };
6 void imprime_valor(int x){
7     printf('Valor = %d\n',p.x);
8 }
9 int main(){
10     struct ponto p1 = {10,20};
11     imprime_valor(p1.x);
12     imprime_valor(p1.y);
13     system('pause');
14     return 0;
15 }
```

## PASSAGEM DE ESTRUTURAS POR REFERÊNCIA

Vimos anteriormente que para passar um parâmetro por referência, usa-se o operador “\*” na frente do nome do parâmetro durante a declaração

da função. Isso também é válido para uma estrutura, mas alguns cuidados devem ser tomados ao acessar seus campos dentro da função. Para acessar o valor de um campo de uma estrutura passada por referência, devemos seguir o seguinte conjunto de passos:

1. utilizar o operador “\*” na frente do nome da variável que representa a estrutura;
2. colocar o operador “\*” e o nome da variável entre parênteses ();
3. por fim, acessar o campo da estrutura utilizando o operador ponto “.”.

O exemplo abaixo mostra como os campos de uma estrutura passada por referência devem ser acessado:

#### Exemplo: estrutura passada por referência

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct ponto {
4     int x, y;
5 };
6 void atribui(struct ponto *p){
7     (*p).x = 10;
8     (*p).y = 20;
9 }
10 int main(){
11     struct ponto p1;
12     atribui(&p1);
13     printf( 'x = %d\n' ,p1.x );
14     printf( 'y = %d\n' ,p1.y );
15     system( 'pause' );
16     return 0;
17 }
```

Note, no exemplo acima, que a função **atribui** recebe uma **struct ponto** por referência, “\*p”. Para acessar qualquer um dos seus campos (**x** ou **y**), é necessário utilizar o operador “\*” na frente do nome da variável que representa a estrutura, “\*p”, e em seguida colocar o operador “\*” e o nome da variável entre parênteses, “(\*p)”. Somente depois de feito isso é que podemos acessar um dos campos da estrutura com o operador ponto “.” (linhas 7 e 8).



Ao acessar uma estrutura passada por referência não podemos esquecer de colocar os parêntese antes de acessar o seu campo.

O uso dos parênteses serve para diferenciar que é que foi passado por referência de quem é ponteiro. Um ponteiro é um tipo especial de variável que armazena um endereço de memória, da mesma maneira como uma variável armazena um valor (mais detalhes sobre o uso de ponteiros serão apresentados no capítulo seguinte). A expressão

`(*p).x`

indica que a variável **p** é na verdade o ponteiro, ou melhor, a variável que foi passada por referência. Isso ocorre porque o asterisco está junto de **p**, e isolado de **x** por meio dos parênteses. Já nas notações abaixo são equivalentes

`*p.x`

`*(p.x)`

e ambas indicam que a variável **x** é na verdade o ponteiro, e não **p**. Isso ocorre pois o operador ponto “.” tem prioridade e é executado primeiro. Logo, o operador asterisco “\*” irá atuar sobre o campo da estrutura, e não sobre a variável da estrutura.

### 8.2.5 OPERADOR SETA

De modo geral, uma estrutura é sempre passada por valor para uma função. Mas ela também pode ser passada por referência sempre que desejarmos alterar algum dos valores de seus campos.

Durante o estudo dos tipos definidos pelo programador, vimos que o operador “.” (ponto) era utilizado para acessar os campos de uma estrutura. Se essa estrutura for passada por referência para uma função, será necessário usar ambos os operadores “\*” e “.” para acessar os valores originais dos campos da estrutura.

- operador “\*”: acessa o conteúdo da posição de memória (valor da variável fora da função) dentro da função;
- operador “.”: acessa os campos de uma estrutura.



O operador **seta** “->” substitui o uso conjunto dos operadores “\*” e “.” no acesso ao campo de uma estrutura passada por referência para uma função.

O operador **seta** “->” é utilizado quando uma referência para uma estrutura (struct) é passada para uma função. Ele permite acessar o valor do campo da estrutura fora da função sem utilizar o operador “\*”. O exemplo abaixo mostra como os campos de uma estrutura passada por referência podem ser acessado com ou sem o uso do operador seta “->”:

Exemplo: passagem por valor e referência	
Sem operador seta	Com operador seta
<pre> 1 struct ponto { 2     int x, y; 3 }; 4 5 void func(struct ponto *           p){ 6     (*p).x = 10; 7     (*p).y = 20; 8 } </pre>	<pre> 1 struct ponto { 2     int x, y; 3 }; 4 5 void func(struct ponto *           p){ 6     p-&gt;x = 10; 7     p-&gt;y = 20; 8 } </pre>

### 8.3 RECURSÃO

Na linguagem C, uma função pode chamar outra função. Um exemplo disso é quando chamamos qualquer uma das nossas funções implementadas na função **main**. Uma função pode, inclusive, chamar a si própria. Uma função assim é chamada de *função recursiva*.



A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo.

Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número. A função fatorial é definida como:

$$0! = 1$$

$$N! = N * (N - 1)!$$

A idéia básica da recursão é dividir um problema maior em um conjunto de problemas menores, que são então resolvidos de forma independente e depois combinados para gerar a solução final: dividir e conquistar.

Isso fica evidente no cálculo do fatorial. O fatorial de um número  $N$  é o produto de todos os números inteiros entre 1 e  $N$ . Por exemplo, o fatorial

de 3 é igual a  $1 * 2 * 3$ , ou seja, **6**. No entanto, o fatorial desse mesmo número 3 pode ser definido em termos do fatorial de 2, ou seja,  $3! = 3 * 2!$ . O exemplo abaixo apresenta as funções com e sem recursão para o cálculo do fatorial:

Exemplo: fatorial	
Com Recursão	Sem Recursão
<pre> 1 int fatorial (int n){ 2   if (n == 0) 3     return 1; 4   else 5     return n*fatorial(n 6       -1); </pre>	<pre> 1 int fatorial (int n){ 2   if (n == 0) 3     return 1; 4   else { 5     int i, f = 1; 6     for (i=2; i &lt;= n; i 7       ++ ) 8       f = f * i; 9     return f; 10  } </pre>

Em geral, as formas recursivas dos algoritmos são consideradas “mais enxutas” e “mais elegantes” do que suas formas iterativas. Isso facilita a interpretação do código. Porém, esses algoritmos apresentam maior dificuldade na detecção de erros e podem ser ineficientes.



Todo cuidado é pouco ao se fazer funções recursivas, pois duas coisas devem ficar bem estabelecidas: o *critério de parada* e o *parâmetro da chamada recursiva*.

Durante a implementação de uma função recursiva temos que ter em mente duas coisas: o **critério de parada** e o **parâmetro da chamada recursiva**:

- **Critério de parada:** determina quando a função deverá parar de chamar a si mesma. Se ele não existir, a função irá executar infinitamente. No cálculo de fatorial, o critério de parada ocorre quando tentamos calcular o fatorial de zero:  $0! = 1$ .
- **Parâmetro da chamada recursiva:** quando chamamos a função dentro dela mesmo, devemos sempre mudar o valor do parâmetro passado, de forma que a recursão chegue a um término. Se o valor do parâmetro for sempre o mesmo a função irá executar infinitamente. No cálculo de fatorial, a mudança no parâmetro da chamada recursiva ocorre quando definimos o fatorial de  $N$  em termos no fatorial de  $(N-1)$ :  $N! = N * (N - 1)!$ .

O exemplo abaixo deixa bem claro o **critério de parada** e o **parâmetro da chamada recursiva** na função recursiva implementada em linguagem C:

#### Exemplo: fatorial

```
1 int fatorial (int n){
2   if (n == 0) //critério de parada
3     return 1;
4   else //parâmetro do fatorial sempre muda
5     return n*fatorial(n-1);
6 }
```

Note que a implementação da função recursiva do fatorial em C segue exatamente o que foi definido matematicamente.



Algoritmos recursivos tendem a necessitar de mais tempo e/ou espaço do que algoritmos iterativos.

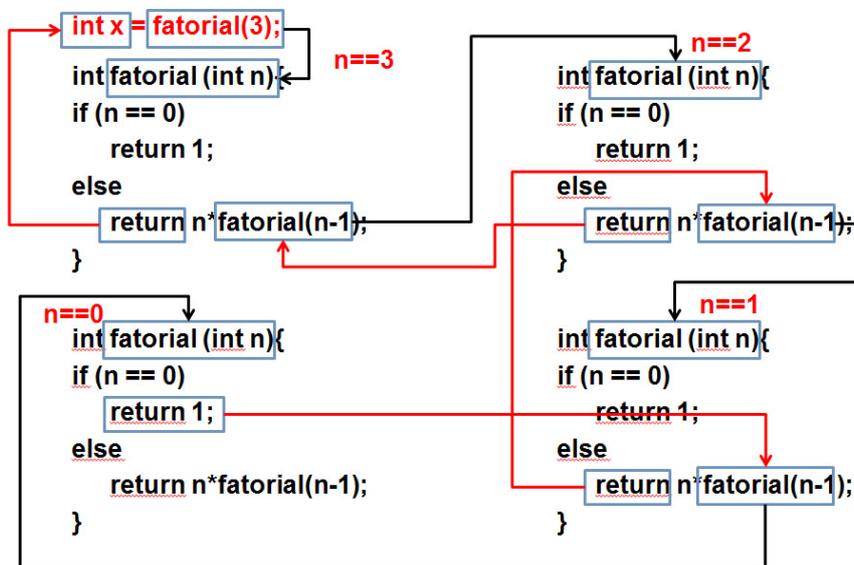
Sempre que chamamos uma função, é necessário um espaço de memória para armazenar os parâmetros, variáveis locais e endereço de retorno da função. Numa função recursiva, essas informações são armazenadas para cada chamada da recursão, sendo, portanto a memória necessária para armazená-las proporcional ao número de chamadas da recursão.

Além disso, todas essas tarefas de alocar e liberar memória, copiar informações, etc. envolvem tempo computacional, de modo que uma função recursiva gasta mais tempo que sua versão iterativa (sem recursão).



O que acontece quando chamamos a função fatorial com um valor como  $N = 3$ ?

Nesse caso, a função será chamada tantas vezes quantas forem necessárias. A cada chamada, a função irá verificar se o valor de  $N$  é igual a zero. Se não for, uma nova chamada da função será realizada. Esse processo, identificado pelas setas pretas, continua até que o valor de  $N$  seja decrementado para ZERO. Ao chegar nesse ponto, a função começa o processo inverso (identificado pelas setas vermelhas): ela passa a devolver para quem a chamou o valor do comando **return**. A figura abaixo mostra esse processo para  $N = 3$ :



Outro exemplo clássico de recursão é a seqüência de Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

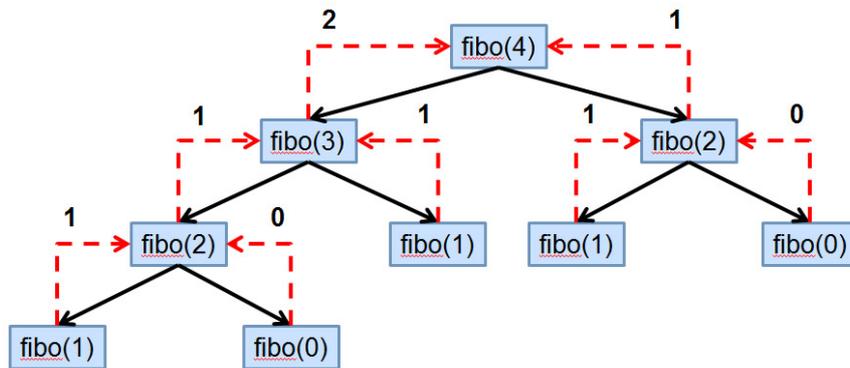
A seqüência de Fibonacci é definida como uma função recursiva utilizando a fórmula abaixo:

$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

O exemplo abaixo apresenta as funções com e sem recursão para o cálculo da seqüência de de Fibonacci:

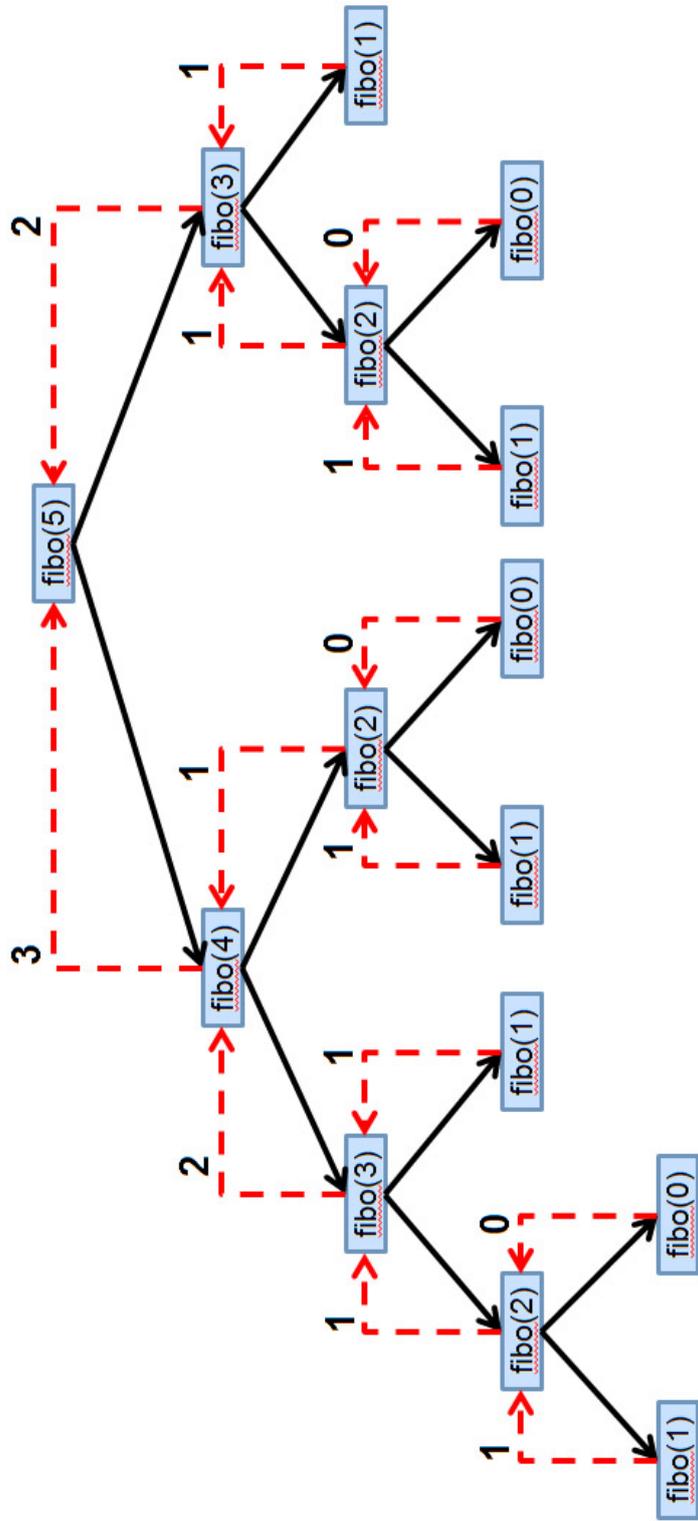
Exemplo: seqüência de Fibonacci	
Com Recursão	Sem Recursão
<pre> 1 int fibo(int n){ 2   if (n == 0    n == 1) 3     return n; 4   else 5     return fibo(n-1) + 6       fibo(n-2); 6 } </pre>	<pre> 1 int fibo(int n){ 2   int i,t,c,a=0, b=1; 3   for(i=0;i&lt;n;i++){ 4     c = a + b; 5     a = b; 6     b = c; 7   } 8   return a; 9 } </pre>

Como se nota, a solução recursiva para a seqüência de Fibonacci é muito elegante. Infelizmente, como se verifica na imagem abaixo, elegância não significa eficiência.



Na figura acima, as setas pretas indicam quando uma nova chamada da função é realizada, enquanto as setas vermelhas indicam o processo inverso, ou seja, quando a função passa a devolver para quem a chamou o valor do comando **return**. O maior problema da solução recursiva está nos quadrados marcados com pontilhados verde. Neles, fica claro que o mesmo cálculo é realizado duas vezes, *um desperdício de tempo e espaço!*

Se, ao invés de calcularmos **fibo(4)** quisermos calcular **fibo(5)**, teremos um desperdício ainda maior de tempo e espaço, como mostra a figura abaixo:



Toda informação que manipulamos dentro de um programa (esteja ela guardada em uma variável, array, estrutura, etc.) obrigatoriamente está armazenada na memória do computador. Quando criamos uma variável, o computador reserva um espaço de memória onde poderemos guardar o valor associado a essa variável. Ao nome que damos a essa variável o computador associa o endereço do espaço que ele reservou na memória para guardar essa variável. De modo geral, interessa ao programador saber o nome das variáveis. Já o computador precisa saber onde elas estão na memória, ou seja, precisa dos endereços das variáveis.



Ponteiros são um tipo especial de variáveis que permitem armazenar endereços de memória ao invés de dados numéricos (como os tipos **int**, **float** e **double**) ou caracteres (como o tipo **char**).

Por meio dos ponteiros, podemos acessar o endereço de uma variável e manipular o valor que está armazenado lá dentro. Eles são uma ferramenta extremamente útil dentro da linguagem C. Por exemplo, quando trabalhamos com arrays, nós estamos utilizando ponteiros.



Apesar de suas vantagens, muitos programadores tem medo, ou até mesmo aversão, ao uso dos ponteiros. Isso por que existem muitos perigos na utilização de ponteiros.

Isso ocorre por que os ponteiros permitem que um programa acesse objetos que não foram explicitamente declarados com antecedência e, consequentemente, permitem uma grande variedade de erros de programação. Outro grande problema dos ponteiros é que eles podem ser *apontados para endereços* (ou seja, **armazenar o endereço de uma posição de memória**) não utilizados, ou para dados dentro da memória que estão sendo usados para outros propósitos. Apesar desses perigos no uso de ponteiros, seu poder é tão grande que existem tarefas que são difíceis de serem implementadas sem a utilização de ponteiros.

A seguir, serão apresentados os conceitos e detalhes necessários para um programador utilizar com sabedoria um ponteiro.

## 9.1 DECLARAÇÃO

Ponteiros são um tipo especial de variáveis que permitem armazenar endereços de memória ao invés de dados numéricos (como os tipos **int**, **float** e **double**) ou caracteres (como o tipo **char**). É importante sempre lembrar:

- Variável: é um espaço reservado de memória usado para guardar um valor que pode ser modificado pelo programa;
- Ponteiro: é um espaço reservado de memória usado para guardar um endereço de memória.



Na linguagem C, um ponteiro pode ser declarado para qualquer tipo de variável (char, int, float, double, etc), inclusive para aquelas criadas pelo programador (struct, etc).

Em linguagem C, a declaração de um ponteiro pelo programador segue a seguinte forma geral:

```
tipo_do_ponteiro *nome_do_ponteiro;
```



É o operador *asterisco* (\*) que informa ao compilador que aquela variável não vai guardar um valor, mas sim um endereço de memória para aquele tipo especificado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //Declara um ponteiro para int
5     int *p;
6     //Declara um ponteiro para float
7     float *x;
8     //Declara um ponteiro para char
9     char *y;
10    //Declara uma variável do tipo int e um
        ponteiro para int
11    int soma, *p2,;
12
13    system('pause');
14    return 0;
15 }
```

Na linguagem C, quando declaramos um ponteiro nós informamos ao compilador para que tipo de variável nós vamos poder apontá-lo. Um ponteiro do tipo **int\*** só pode apontar para uma variável do tipo **int** (ou seja, esse ponteiro só poderá guardar o endereço de uma variável do tipo **int**)



Apesar de usarem o mesmo símbolo, o operador **\*** (**multiplicação**) não é o mesmo operador que o **\*** (**referência de ponteiros**).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int x = 3, y = 5, z;
5     z = y * x;
6     int *p;
7
8     system( ' 'pause' ' );
9     return 0;
10 }
```

No exemplo acima, o operador *asterisco* (\*) é usado de duas maneiras distintas:

- Na linha 5: trata-se de um operador binário, ou seja que atua sobre dois valores/variáveis (nesse caso, é a multiplicação das mesmas);
- Na linha 6: trata-se de um operador unário pré-fixado, ou seja atua sobre uma única variável (nesse caso, é a declaração de um ponteiro).



Lembre-se: o significado do operador *asterisco* (\*) depende de como ele é utilizado dentro do programa.

## 9.2 MANIPULANDO PONTEIROS

### 9.2.1 INICIALIZAÇÃO E ATRIBUIÇÃO

Ponteiros apontam para uma posição de memória. Sendo assim, a simples declaração de um ponteiro não faz dele útil para o programa. Precisamos indicar para que endereço de memória ele aponta.



Ponteiros não inicializados apontam para um lugar indefinido.

Quando um ponteiro é declarado, ele não possui um endereço associado. Qualquer tentativa de uso desse ponteiro causa um comportamento indefinido no programa.

Memória		
#	var	conteúdo
119		
120	int *p	????
121		

int \*p;

Isso ocorre por que seu valor não é um endereço válido ou porque sua utilização pode danificar partes diferentes do sistema. Por esse motivo, os ponteiros devem ser inicializados (apontado para algum lugar conhecido) antes de serem usados.

#### APONTANDO UM PONTEIRO PARA NENHUM LUGAR



Um ponteiro pode ter um valor especial **NULL**, que é o endereço de nenhum lugar.

A constante **NULL** está definida na biblioteca **stdlib.h**. Trata-se de um valor reservado que indica que aquele ponteiro aponta para uma posição de memória inexistente. O valor da constante **NULL** é **ZERO** na maioria dos computadores.

Memória		
#	var	conteúdo
119		
120	int *p	NULL
121		

int \*p = NULL;

nenhum lugar na memória



Não confunda um ponteiro apontando para **NULL** com um ponteiro não inicializado. O primeiro possui um valor fixo, enquanto um ponteiro não inicializado pode possuir qualquer valor.

## APONTANDO UM PONTEIRO PARA ALGUM LUGAR DA MEMÓRIA

Vimos que a constante **NULL** permite apontar um ponteiro para uma posição de memória inexistente. Mas como fazer para atribuir uma posição de memória válida para o ponteiro?

Basicamente, podemos fazer nosso ponteiro apontar para uma variável que já exista no nosso programa. Lembre-se, quando criamos uma variável, o computador reserva um espaço de memória. Ao nome que damos a essa variável o computador associa o endereço do espaço que ele reservou na memória para guardar essa variável.

Para saber o endereço onde uma variável está guardada na memória, usa-se o operador **&** na frente do nome da variável.



Para saber o endereço de uma variável do nosso programa na memória usa-se o operador **&**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //Declara uma variável int contendo o valor 10
5     int count = 10;
6     //Declara um ponteiro para int
7     int *p;
8     //Atribui ao ponteiro o endereço da variável
9     //int
10    p = &count;
11
12    system( ' 'pause' ' );
13    return 0;
14 }
```

No exemplo acima, são declarados uma variável tipo **int** (*count*) e um ponteiro para o mesmo tipo (*p*). Na linha 9, o ponteiro *p* é inicializado com o endereço da variável *count*. Note que usamos o **operador de endereçamento** (&) para a inicialização do ponteiro. Isso significa que o ponteiro *p* passa a conter o endereço de *count*, não o seu valor. Para melhor entender esse conceito, veja a figura abaixo:

Memória		
#	var	conteúdo
119		
120	int *p	#122
121		
122	int count	10
123		

Tendo um ponteiro armazenado um endereço de memória, como saber o valor guardado dentro dessa posição de memória? Simples, para acessar o conteúdo da posição de memória para a qual o ponteiro aponta, usa-se o operador *asterisco* (\*) na frente do nome do ponteiro.



Para acessar o valor guardado dentro de uma posição na memória apontada por um ponteiro, basta usar o operador *asterisco* (\*).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //Declara uma variável int contendo o valor 10
5     int count = 10;
6     //Declara um ponteiro para int
7     int *p;
8     //Atribui ao ponteiro o endereço da variável
9     int
10    p = &count;
11    printf('Conteudo apontado por p: %d \n',*p);
12    //Atribui um novo valor à posição de memória
13    apontada por p
14    *p = 12;
15    printf('Conteudo apontado por p: %d \n',*p);
16    printf('Conteudo de count: %d \n',count);
17
18    system('pause');
19    return 0;
20 }

```

Saída      Conteudo apontado por p: 10  
               Conteudo apontado por p: 12  
               Conteudo de count: 12

Note, no exemplo acima, que utilizamos o operador *asterisco* (\*) sempre que queremos acessar o valor contido na posição de memória apontada por *p*. Note também que, se alterarmos o valor contido nessa posição de memória (**linha 12**), alteramos o valor da variável *count*.

## OS OPERADORES “\*” E “&”

Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:

- acessar o endereço de memória de uma variável;
- acessar o conteúdo de um endereço de memória;

Para realizar essas tarefas, iremos sempre utilizar apenas dois operadores: o operador “\*” e o operador “&”.

Operador “*” versus operador “&”	
“*”	Declara um ponteiro: <b>int *x;</b> Conteúdo para onde o ponteiro aponta: <b>int y = *x;</b>
“&”	Endereço onde uma variável está guardada na memória: <b>&amp;y</b>

## ATRIBUIÇÃO ENTRE PONTEIROS

Devemos estar sempre atento a operação de atribuição quando estamos trabalhando com ponteiros. Não só com relação ao uso corretos dos operadores, mas também ao que estamos atribuindo ao ponteiro.



De modo geral, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

Isso ocorre por que diferentes tipos de variáveis ocupam um espaço de memória de tamanhos diferentes. Na verdade, nós podemos, por exemplo, atribuir a um ponteiro de inteiro (**int \***) o endereço de uma variável do tipo **float**. O compilador não irá acusar nenhum erro. No entanto, o compilador assume que qualquer endereço que esse ponteiro armazene **obrigatoriamente** apontará para uma variável do tipo **int**. Conseqüentemente, qualquer tentativa de uso desse ponteiro causa um comportamento indefinido no programa. Veja o exemplo abaixo:

### Exemplo: atribuição de ponteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p, *p1, x = 10;
5     float y = 20.0;
6     p = &x;
7     printf(‘‘Conteudo apontado por p: %d \n’’,*p);
8     p1 = p;
9     printf(‘‘Conteudo apontado por p1: %d \n’’,*p1
10    );
11    p = &y;
12    printf(‘‘Conteudo apontado por p: %d \n’’,*p);
13    printf(‘‘Conteudo apontado por p: %f \n’’,*((
14        float*)p));
15    system(‘‘pause’’);
16    return 0;
17 }
```

Saída

```
Conteudo apontado por p: 10
Conteudo apontado por p1: 10
Conteudo apontado por p: 1101004800
Conteudo apontado por p: 0.000000
Conteudo apontado por p: 20.000000
```

No exemplo acima, um endereço de uma variável do tipo **float** é atribuído a um ponteiro do tipo **int** (linha 10). Note que qualquer tentativa de acessar o seu conteúdo se mostra falha (linhas 11 e 12). Só conseguimos acessar corretamente o seu conteúdo quando utilizamos o operador de *typecast* sobre o ponteiro e antes de acessar o seu conteúdo (linha 13).



Um ponteiro pode receber o endereço apontado por outro ponteiro, se ambos forem do mesmo tipo.

Se dois ponteiros são do mesmo tipo, então eles podem guardar endereços de memória para o mesmo tipo de dado. Logo a atribuição entre eles é possível. Isso é mostrado no exemplo anterior (linhas 8 e 9).



Um ponteiro pode receber um valor hexadecimal representado um endereço de memória diretamente. Isso é muito útil quando se trabalha, por exemplo, com microcontroladores.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     //Endereço hexadecimal da porta serial
5     int *p = 0x3F8;
6     //O valor em decimal é convertido para seu
7     //valor hexadecimal: 0x5DC
8     int *p1 = 1500;
9     printf("Endereco em p: %p \n", p);
10    printf("Endereco em p1: %p \n", p1);
11    system("pause");
12    return 0;
}
```

Saída      Endereco em p: 000003F8  
            Endereco em p1: 000005DC

Na linguagem C, um valor hexadecimal deve começar com “0x” (um zero seguido de um x), seguido pelo valor em formato hexadecimal, que pode ser formado por:

- dígitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- letras: A, B, C, D, E, F.

Deve-se tomar muito cuidado com esse tipo de utilização de ponteiros, principalmente quando queremos acessar o conteúdo daquela posição de memória. Afinal de contas, o que existe na posição de memória **0x5DC**? Esse é um erro muito comum.

## 9.2.2 ARITMÉTICA COM PONTEIROS

As operações aritméticas utilizando ponteiros são bastante limitadas, o que facilita o seu uso. Basicamente, apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: adição e subtração.



Sobre o valor de endereço armazenado por um ponteiro podemos apenas **somar** e **subtrair** valores INTEIROS.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p = 0x5DC;
5     printf('p = Hexadecimal: %p Decimal: %d \n',
6           p,p);
7     //Incrementa p em uma posição
8     p++;
9     printf('p = Hexadecimal: %p Decimal: %d \n',
10          p,p);
11    //Incrementa p em 15 posições
12    p = p + 15;
13    printf('p = Hexadecimal: %p Decimal: %d \n',
14          p,p);
15    //Decrementa p em 2 posições
16    p = p - 2;
17    printf('p = Hexadecimal: %p Decimal: %d \n',
18          p,p);
19    system('pause');
20    return 0;
21 }
```

Saída

```
p = Hexadecimal: 000005DC Decimal: 1500
p = Hexadecimal: 000005E0 Decimal: 1504
p = Hexadecimal: 0000061C Decimal: 1564
p = Hexadecimal: 00000614 Decimal: 1556
```

As operações de adição e subtração no endereço permitem avançar ou retroceder nas posições de memória do computador. Esse tipo de operação é bastante útil quando trabalhamos com arrays, por exemplo. Lembre-se: um array nada mais é do que um conjunto de elementos adjacentes na memória.

Além disso, todas as operações de adição e subtração no endereço devem ser inteiras. Afinal de contas, não dá para andar apenas **MEIA** posição na memória.

No entanto, é possível notar no exemplo anterior que a operação de incremento **p++** (linha 7) não incrementou em uma posição o endereço, mas sim em quatro posições: ele foi da posição 1500 para a 1504. Isso aconteceu por que nosso ponteiro é do tipo inteiro (**int \***).



As operações de adição e subtração no endereço dependem do tipo de dado que o ponteiro aponta.

Suponha um ponteiro para inteiro, `int *p`. Esse ponteiro deverá receber um endereço de um valor inteiro. Quando declaramos uma variável inteira (`int x`), o computador reserva um espaço de *4 bytes* na memória para essa variável. Assim, nas operações de adição e subtração são adicionados/-subtraídos um total de *4 bytes* por incremento/decremento, pois este é o tamanho de um inteiro na memória e, portanto, é também o valor mínimo necessário para sair dessa posição reservada de memória. Se o ponteiro fosse para o tipo **double**, as operações de incremento/decremento mudariam a posição de memória em *8 bytes*.



Sobre o conteúdo apontado pelo ponteiro valem todas as operações aritméticas que o tipo do ponteiro suporta.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p, x = 10;
5     p = &x;
6     printf("Conteudo apontado por p: %d \n", *p);
7     *p = (*p)++;
8     printf("Conteudo apontado por p: %d \n", *p);
9     *p = (*p) * 10;
10    printf("Conteudo apontado por p: %d \n", *p);
11    system("pause");
12    return 0;
13 }
```

Saída      Conteudo apontado por p: 10  
             Conteudo apontado por p: 11  
             Conteudo apontado por p: 110

Quando utilizamos o operador *asterisco* (\*) na frente do nome do ponteiro estamos acessando o conteúdo da posição de memória para a qual o ponteiro aponta. Em resumo, estamos acessando o valor guardado na variável para qual o ponteiro aponta. Sobre esse valor, valem todas as operações que o tipo do ponteiro suporta.

### 9.2.3 OPERAÇÕES RELACIONAIS COM PONTEIROS

A linguagem C permite comparar os endereços de memória armazenados por dois ponteiros utilizando uma expressão relacional. Por exemplo, os operadores `==` e `!=` são usado para saber se dois ponteiros são iguais ou diferentes.



Dois ponteiros são considerados iguais se eles apontam para a mesma posição de memória.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p, *p1, x, y;
5     p = &x;
6     p1 = &y;
7     if (p == p1)
8         printf("Ponteiros iguais\n");
9     else
10        printf("Ponteiros diferentes\n");
11    system("pause");
12    return 0;
13 }
```

Já os operadores `>`, `<`, `>=` e `<=` são usado para saber se um ponteiro aponta para uma posição mais adiante na memória do que outro. Novamente, esse tipo de operação é bastante útil quando trabalhamos com arrays, por exemplo. Lembre-se: um array nada mais é do que um conjunto de elementos adjacentes na memória.

### Exemplo: comparando ponteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p, *p1, x, y;
5     p = &x;
6     p1 = &y;
7     if (p > p1)
8         printf('O ponteiro p aponta para uma posicao a
9             frente de p1\n');
10    else
11        printf('O ponteiro p NAO aponta para uma posicao a
12            frente de p1\n');
13    system('pause');
14    return 0;
15 }
```

Como no caso das operações aritméticas, quando utilizamos o operador *asterisco* (\*) na frente do nome do ponteiro estamos acessando o conteúdo da posição de memória para a qual o ponteiro aponta. Em resumo, estamos acessando o valor guardado na variável para qual o ponteiro aponta. Sobre esse valor, valem todas as operações relacionais que o tipo do ponteiro suporta.

### Exemplo: comparando o conteúdo dos ponteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p, *p1, x = 10, y = 20;
5     p = &x;
6     p1 = &y;
7     if (*p > *p1)
8         printf('O conteúdo de p e maior do que o conteúdo
9             de p1\n');
10    else
11        printf('O conteúdo de p NAO e maior do que o
12            conteúdo de p1\n');
13    system('pause');
14    return 0;
15 }
```

### 9.3 PONTEIROS GENÉRICOS

Normalmente, um ponteiro aponta para um tipo específico de dado. Porém, pode-se criar um ponteiro *genérico*. Esse tipo de ponteiro pode apontar para todos os tipos de dados existentes ou que ainda serão criados. Em linguagem C, a declaração de um ponteiro genérico segue a seguinte forma geral:

```
void *nome_do_ponteiro;
```



Um ponteiro genérico é um ponteiro que pode apontar para qualquer tipo de dado, inclusive para outro ponteiro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main () {
5     void *pp;
6     int *p1, p2 = 10;
7     p1 = &p2;
8     //recebe o endereço de um inteiro
9     pp = &p2;
10    printf(‘‘Endereco em pp: %p \n’’,pp);
11    //recebe o endereço de um ponteiro para
        inteiro
12    pp = &p1;
13    printf(‘‘Endereco em pp: %p \n’’,pp);
14    //recebe o endereço guardado em p1 (endereço
        de p2)
15    pp = p1;
16    printf(‘‘Endereco em pp: %p \n’’,pp);
17    system(‘‘pause’’);
18    return 0;
19 }
```

Note, no exemplo acima, que ponteiro genérico permite guardar o endereço de qualquer tipo de dado. Essa vantagem vem com uma desvantagem: sempre que tivermos que acessar o conteúdo de um ponteiro genérico será necessário utilizar o operador de *typecast* sobre o ponteiro e antes de acessar o seu conteúdo.



Sempre que se trabalhar com um ponteiro genérico é preciso converter o ponteiro genérico para o tipo de ponteiro com o qual se deseja trabalhar antes de acessar o seu conteúdo.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main () {
5     void *pp;
6     int p2 = 10;
7     // ponteiro genérico recebe o endereço de um
8     // inteiro
9     pp = &p2;
10    //enta acessar o conteúdo do ponteiro genérico
11    printf("Conteúdo: %d\n", *pp); //ERRO
12    //converte o ponteiro genérico pp para (int *)
13    // antes de acessar seu conteúdo.
14    printf("Conteúdo: %d\n", *(int*)pp); //
15    // CORRETO
16    system("pause");
17    return 0;
18 }
```

No exemplo acima, como o compilador não sabe qual o tipo do ponteiro genérico, acessar o seu conteúdo gera um tipo de erro. Somente é possível acessar o seu conteúdo depois de uma operação de *typecast*.

Outro cuidado que devemos ter com ponteiros genéricos: como o ponteiro genérico não possui tipo definido, deve-se tomar cuidado com ao se realizar operações aritméticas.



As operações aritméticas não funcionam em ponteiros genéricos da mesma forma como em ponteiros de tipos definidos. Elas são sempre realizadas com base em uma unidade de memória (*1 byte*).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     void *p = 0x5DC;
5     printf('p = Hexadecimal: %p Decimal: %d \n',
6           p,p);
7     //Incrementa p em uma posição
8     p++;
9     printf('p = Hexadecimal: %p Decimal: %d \n',
10          p,p);
11    //Incrementa p em 15 posições
12    p = p + 15;
13    printf('p = Hexadecimal: %p Decimal: %d \n',
14          p,p);
15    //Decrementa p em 2 posições
16    p = p - 2;
17    printf('p = Hexadecimal: %p Decimal: %d \n',
18          p,p);
19    system('pause');
20    return 0;
21 }
```

Saída p = Hexadecimal: 000005DC Decimal: 1500  
p = Hexadecimal: 000005DD Decimal: 1501  
p = Hexadecimal: 000005EC Decimal: 1516  
p = Hexadecimal: 000005EA Decimal: 1514

No exemplo acima, como o compilador não sabe qual o tipo do ponteiro genérico, nas operações de adição e subtração são adicionados/subtraídos um total de *1 byte* por incremento/decremento, pois este é o tamanho de uma unidade de memória. Portanto, se o endereço guardado for, por exemplo, de um inteiro, o incremento de uma posição no ponteiro genérico (*1 byte*) não irá levar ao próximo inteiro (*4 bytes*).

## 9.4 PONTEIROS E ARRAYS

Ponteiros e arrays possuem uma ligação muito forte dentro da linguagem C. Arrays são agrupamentos de dados do mesmo tipo na memória. Quando declaramos um array, informamos ao computador para reservar uma certa quantidade de memória para armazenar os elementos do array de forma **sequencial**. Como resultado dessa operação, o computador nos

devolve um ponteiro que aponta para o começo dessa sequência de *bytes* na memória.



O **nome** do array é apenas um ponteiro que aponta para o primeiro elemento do array.

Na linguagem C, o nome de um array **sem um índice** guarda o endereço para o começo do array na memória, ou seja, ele guarda o endereço do início de uma área de armazenamento dentro da memória. Isso significa que as operações envolvendo arrays podem ser feitas utilizando ponteiros e aritmética de ponteiros.

Exemplo: acessando arrays utilizando ponteiros.	
Usando Array	Usando Ponteiro
<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main (){ 4   int vet[5] =       {1,2,3,4,5}; 5   int *p = vet; 6   int i; 7   for (i = 0; i &lt; 5; i++) 8     printf("%d\n", p[i       ]); 9   system('pause'); 10  return 0; 11 }</pre>	<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main (){ 4   int vet[5] =       {1,2,3,4,5}; 5   int *p = vet; 6   int i; 7   for (i = 0; i &lt; 5; i++) 8     printf("%d\n", *(p+       i)); 9   system('pause'); 10  return 0; 11 }</pre>

No exemplo acima, temos o mesmo código utilizando a notação de colchetes e de aritmética de ponteiros para acessar os elementos de um array. Note que se para acessar o elemento na posição *i* do array podemos escrever **p[i]** ou **\*(p+i)**.

Quanto a atribuição do endereço do array para o ponteiro, podemos fazê-la de duas formas:

```
int *p = vet;
int *p = &vet[0];
```

Na primeira forma, o nome do array é utilizado para retornar o endereço onde ele começa na memória. Já na segunda forma, nós utilizamos o

operador de endereço (&) para retornar o endereço da primeira posição do array.



O operador **colchetes** [ ] substitui o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador “\*\*”) no acesso ao conteúdo de uma posição de um array.

Durante o estudo de ponteiros, vimos que o operador *asterisco* (\*) é utilizado para acessar o valor guardado dentro de uma posição na memória apontada por um ponteiro. Além disso, operações aritméticas podem ser usadas para avançar sequencialmente na memória. Lembre-se, um array é um agrupamento sequencial de dados do mesmo tipo na memória. Sendo assim, o operador **colchetes** apenas simplifica o uso conjunto de operações aritméticas e de acesso ao conteúdo (operador “\*\*”) no acesso ao conteúdo de uma posição de um array.

Abaixo temos uma lista mostrando as equivalências entre arrays e ponteiros:

### Exemplo: equivalências entre arrays e ponteiros.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int vet[5] = {1,2,3,4,5};
5     int *p, indice = 2;
6     p = vet;
7     //vet[0] é equivalente a *p;
8     printf(“%d\n”,*p);
9     printf(“%d\n”,vet[0]);
10    //vet[indice] é equivalente
11    //a *(p+indice);
12    printf(“%d\n”,vet[indice])
13    ;
14    printf(“%d\n”,*(p+indice))
15    ;
16    //vet é equivalente
17    //a &vet[0];
18    printf(“%d\n”,vet);
19    printf(“%d\n",&vet[0]);
20    //&vet[indice] é equivalente
21    //a (vet+indice);
22    printf(“%d\n",&vet[indice
23    ]);
24    printf(“%d\n”,(vet+indice)
25    );
26    system(“pause”);
27    return 0;
28 }
```

Memória		
#	var	conteúdo
123	int *p	#125
124		
125	int vet[5]	1
126		2
127		3
128		4
129		5
		.
		.
		.

No exemplo anterior, note que o valor entre colchetes é o deslocamento a partir da posição inicial. Nesse caso, **p[indice]** equivale a **\*(p+indice)**.



Um ponteiro também pode ser usado para acessar os dados de uma *string*.

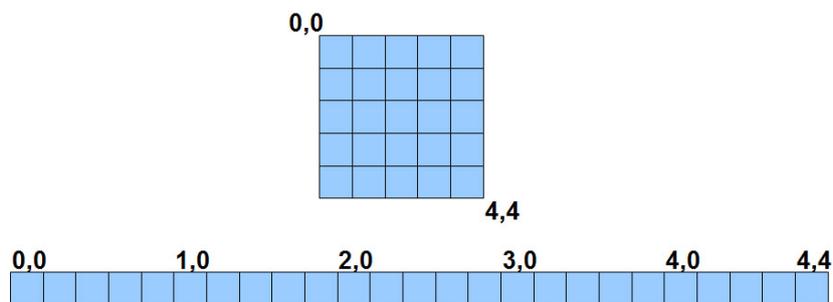
Lembre-se: *string* é o nome que usamos para definir uma seqüência de caracteres adjacentes na memória do computador. Essa seqüência de caracteres, que pode ser uma palavra ou frase, é armazenada na memória do computador na forma de um arrays do tipo **char**.

### 9.4.1 PONTEIROS E ARRAYS MULTIDIMENSIONAIS

Apesar de terem o comportamento de estruturas com mais de uma dimensão, os dados dos arrays multidimensionais são armazenados linearmente na memória. É o uso dos colchetes que cria a impressão de estarmos trabalhando com mais de uma dimensão. Por exemplo, a matriz

```
int mat[5][5];
```

apesar de ser bidimensional, ela é armazenada como um simples array na memória:



Nós podemos acessar os elementos de um array multidimensional usando a notação tradicional de colchetes ( $mat[linha][coluna]$ ) ou a notação por ponteiros:

```
*(*(mat + linha) + coluna)
```

Para entender melhor o que está acontecendo, vamos trocar

```
*(mat + linha)
```

por um valor  $X$ . Desse modo, a expressão fica

```
*(X + coluna)
```

É possível agora perceber que  $X$  é como um ponteiro, é que o seu conteúdo é o endereço de uma outra posição de memória. Em outras palavras, o valor de *linhas* é o deslocamento na memória do primeiro ponteiro (ou primeira dimensão da matriz), enquanto o valor de *colunas* é o deslocamento na memória do segundo ponteiro (ou segunda dimensão da matriz).



Ponteiros permitem percorrer as várias dimensões de um arrays multidimensional como se existisse apenas uma dimensão. As dimensões mais a direita mudam mais rápido.

Na primeira forma, o nome do array é utilizado para retornar o endereço onde ele começa na memória. Isso é muito útil quando queremos construir uma função que possa percorrer um array independente do número de dimensões que ele possua. Para realizar essa tarefa, nós utilizamos o operador de endereço (&) para retornar o endereço da primeira posição do array, como mostra o exemplo abaixo:

Acessando um array multidimensional utilizando ponteiros.	
Usando Array	Usando Ponteiro
<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main (){ 4     int mat[2][2] =       {{1,2},{3,4}}; 5     int i, j; 6     for (i=0; i&lt;2; i++) 7         for (j=0; j&lt;2; j++) 8             printf(“%d\n”,               mat[i][j]); 9     system(“pause”); 10    return 0; 11 }</pre>	<pre>1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 int main (){ 4     int mat[2][2] =       {{1,2},{3,4}}; 5     int * p = &amp;mat[0][0]; 6     int i; 7     for (i=0; i&lt;4; i++) 8         printf(“%d\n”,               *(p+i)); 9     system(“pause”); 10    return 0; 11 }</pre>

#### 9.4.2 ARRAY DE PONTEIROS

A linguagem C também permite que declaremos arrays de ponteiros como fazemos com qualquer outro tipo de dado. A declaração de um array de ponteiros segue a seguinte forma geral:

```
tipo_dado *nome_array[tamanho];
```

O comando acima define um array de nome *nome\_array* contendo *tamanho* elementos adjacentes na memória. Cada elemento do array é do tipo *tipo\_dado\**, ou seja, é um ponteiro para *tipo\_dado*. Assim, a declaração de um array de ponteiros para inteiros de tamanho 10 seria:

```
int *p[10];
```

Quanto ao seu uso, não existem diferenças de um array de ponteiros e um ponteiro. Basta lembrar que um array é sempre indexado. Assim, para atribuir o endereço de uma variável *x* a uma posição do array de ponteiros, escrevemos:

```
p[indice] = &x;
```

E para retornar o conteúdo guardado nessa posição de memória:

```
*p[indice]
```



Cada posição de um array de ponteiros pode armazenar o endereço de uma variável ou o endereço da posição inicial de um outro array.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int *pvet[2];
5     int x = 10, y[2] = {20,30};
6     pvet[0] = &x;
7     pvet[1] = y;
8     //imprime os endereços das variáveis
9     printf('Endereco pvet[0]: %p\n', pvet[0]);
10    printf('Endereco pvet[1]: %p\n', pvet[1]);
11    //imprime o conteúdo de uma variável
12    printf('Conteudo em pvet[0]: %d\n', *pvet[0]);
13    ;
14    //imprime uma posição do vetor
15    printf('Conteudo pvet[1][1]: %d\n', pvet
16           [1][1]);
17    system('pause');
18    return 0;
19 }
```

## 9.5 PONTEIRO PARA PONTEIRO

Ao longo dessa seção, vimos que toda informação que manipulamos dentro de um programa está obrigatoriamente armazenada na memória do computador e, portanto, possui um endereço de memória associado a ela. Ponteiros, como qualquer outra variável, também ocupam um espaço na memória do computador e também possuem o endereço desse espaço de memória associado ao seu nome. Como não existem diferenças entre a

maneira como uma variável e um ponteiro são guardados na memória, é possível criar um ponteiro que aponta para o endereço de outro ponteiro.



A linguagem C permite criar ponteiros com diferentes níveis de apontamento, isto é, ponteiros que apontam para outros ponteiros.

Em linguagem C, a declaração de um ponteiro para ponteiro pelo programador segue a seguinte forma geral:

```
tipo_do_ponteiro **nome_do_ponteiro;
```

Note que agora usamos dois *asteriscos* (\*) para informar ao compilador que aquela variável não vai guardar um valor, mas sim um endereço de memória para outro endereço de memória para aquele tipo especificado. Para ficar mais claro, veja o exemplo abaixo:

#### Exemplo: ponteiro para ponteiro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int x = 10;
5     int *p = &x;
6     int **p2 = &p;
7     //Endereço em p2
8     printf ( ' 'Endereco em p2: %
9             p\n' , p2);
10    //Conteúdo do endereço
11    printf ( ' 'Conteúdo em *p2:
12            %p\n' , *p2);
13    //Conteúdo do endereço do
14    endereço
15    printf ( ' 'Conteúdo em **p2:
16            %d\n' , **p2);
17    system ( ' 'pause' ' );
18    return 0;
19 }
```

Memória		
#	var	conteúdo
119		
120	int**p2	#122
121		
122	int*p	#124 ←
123		
124	int x	10
125		

No exemplo acima, foi declarado um ponteiro que aponta para outro ponteiro (*p2*). Nesse caso, esse ponteiro guarda o endereço de um segundo ponteiro (linha 8, endereço de *p*), que por sua vez guarda o endereço de uma variável. Assim, se tentarmos acessar o conteúdo do ponteiro (*\*p2*),

iremos acessar o endereço guardado dentro do ponteiro ( $p$ ), que nada mais é do que o endereço da variável  $x$  (linha 10). Como o  $p2$  é um ponteiro para ponteiro, isso significa que podemos acessar o seu conteúdo duas vezes. Afinal, seu conteúdo ( $*p2$ ) é um endereço. Assim, o comando  $**p2$  acessa o conteúdo do endereço do endereço apontado por ( $p2$ ), isto é, a variável  $x$  (linha 12).



Em um ponteiro para ponteiro, o primeiro ponteiro contém o endereço do segundo ponteiro que aponta para uma variável com o valor desejado.

A linguagem C permite ainda criar um ponteiro que aponte para outro ponteiro, que aponte para outro ponteiro, etc, criando assim diferentes níveis de apontamento ou endereçamento. Com isso, podemos criar um ponteiro para ponteiro, ou, um ponteiro para ponteiro para ponteiro, e assim por diante.



É a quantidade de *asteriscos* (\*) na declaração do ponteiro que indica o número de níveis de apontamento do ponteiro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     //variável inteira
5     int x;
6     //ponteiro para um inteiro (1 nível)
7     int *p1;
8     //ponteiro para ponteiro de inteiro (2 níveis)
9     int **p2;
10    //ponteiro para ponteiro para ponteiro de
        inteiro (3 níveis)
11    int ***p3;
12    system( ' 'pause' ' );
13    return 0;
14 }
```

Consequentemente, devemos respeitar a quantidade de *asteriscos* (\*) utilizados na declaração do ponteiro para acessar corretamente o seu conteúdo, como mostra o exemplo abaixo:

### Acessando o conteúdo de um ponteiro para ponteiro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     char letra='a';
5     char *ptrChar = &letra;
6     char **ptrPtrChar = &
7         ptrChar;
8     char ***ptrPtr = &
9         ptrPtrChar;
10    printf('Conteudo em *
11        ptrChar: %c\n', *
12        ptrChar);
13    printf('Conteudo em **
14        ptrPtrChar: %c\n', **
15        ptrPtrChar);
16    printf('Conteudo em ***
17        ptrPtr: %c\n', ***
18        ptrPtr);
19    system('pause');
20    return 0;
21 }
```

Memória		
#	var	conteúdo
119		
120	char ***ptrPtr	#122
121		
122	char **ptrPtrChar	#124
123		
124	char *ptrChar	#126
125		
126	char letra	'a'
127		

A linguagem C permite que se crie um ponteiro com um número infinito de níveis de apontamento. Porém, na prática, deve-se evitar trabalhar com muitos níveis de apontamento. Isso ocorre por que cada nova nível de apontamento adicionada aumenta a complexidade em lidar com aquele ponteiro e, conseqüentemente, dificulta a compreensão dos programas, causando assim confusão e facilitando o surgimento de erros.

## 10 ALOCAÇÃO DINÂMICA

Sempre que escrevemos um programa, é preciso reservar espaço para os dados que serão processados. Para isso usamos as variáveis.



Uma variável é uma posição de memória previamente reservada e que pode ser usada para armazenar algum dado.

Uma variável é uma posição de memória que armazena um dado que pode ser usado pelo programa. No entanto, por ser uma posição previamente reservada, uma variável deve ser declarada durante o desenvolvimento do programa.



Toda variável deve ser declarada antes de ser usada.

Infelizmente, nem sempre é possível saber o quanto de memória um programa irá precisar.



Imagine o seguinte problema: precisamos construir um programa que processe os valores dos salários dos funcionários de uma pequena empresa.

Uma solução simples para resolver esse problema poderia ser declarar um array do tipo **float** bem grande com, por exemplo, umas 1.000 posições:

```
float salarios[1000];
```

Esse array parece uma solução possível para o problema. Infelizmente, essa solução possui dois problemas:

- Se a empresa tiver **menos** de 1.000 funcionários: esse array será um exemplo de desperdício de memória. Um array de 1.000 posições é declarado quando não se sabe, de fato, se as 1.000 posições serão necessárias;
- Se a empresa tiver **mais** de 1.000 funcionários: esse array será insuficiente para lidar com os dados de todos os funcionários. O programa não atende as necessidades da empresa.

Na declaração de uma array, é dito para reservar uma certa quantidade de memória para armazenar os elementos do array. Porém, neste modo de declaração, a quantidade de memória reservada deve ser fixa. Surge então a necessidade de se utilizar ponteiros juntos com arrays.



Um ponteiro é uma variável que guarda o endereço de um dado na memória.

Além disso, é importante lembrar que arrays são agrupamentos **sequenciais** de dados de um mesmo tipo na memória.



O **nome** do array é apenas um **ponteiro** que aponta para o **primeiro** elemento do array.

A linguagem C permite alocar (reservar) dinamicamente (em tempo de execução) blocos de memórias utilizando ponteiros. A esse processo dá-se o nome de *alocação dinâmica*. A alocação dinâmica permite ao programador “criar” arrays em tempo de execução, ou seja, alocar memória para novos arrays quando o programa está sendo executado, e não apenas quando se está escrevendo o programa. Ela é utilizada quando não se sabe ao certo quanto de memória será necessário para armazenar os dados com que se quer trabalhar. Desse modo, pode-se definir o tamanho do array em tempo de execução, evitando assim o desperdício de memória.



A alocação dinâmica consiste em requisitar um espaço de memória ao computador, em tempo de execução, o qual devolve para o programa o endereço do início desse espaço alocado usando um ponteiro.



## 10.1 FUNÇÕES PARA ALOCAÇÃO DE MEMÓRIA

A linguagem C ANSI usa apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**. São elas:

- malloc
- calloc
- realloc
- free

Além dessas funções, existe também a função **sizeof** que auxilia as demais funções no processo de alocação de memória. A seguir, serão apresentados os detalhes necessários para um programador usar alocação dinâmica em seu programa.

### 10.1.1 SIZEOF()

No momento da alocação da memória, deve-se levar em conta o tamanho do dado alocado.



Alocar memória para um elemento do tipo **int** é diferente de alocar memória para um elemento do tipo **float**.

Isso ocorre pois tipos diferentes podem ter tamanhos diferentes na memória. O tipo **float**, por exemplo, ocupa mais espaço na memória que o tipo **int**.



A função *sizeof()* é usada para saber o número de bytes necessários para alocar **um único elemento** de um determinado tipo de dado.

A função **sizeof()** é usada para se saber o tamanho **em bytes** de variáveis ou de tipos. Ela pode ser usada de duas formas:

**sizeof** nome\_da\_variável

**sizeof** (nome\_do\_tipo)

O exemplo abaixo ilustra as duas formas de uso da função **sizeof**.

#### Exemplo: uso da função **sizeof**

```
1 #include <stdio .h>
2 #include <stdlib .h>
3 struct ponto{
4     int x,y;
5 };
6 int main(){
7     printf( 'Tamanho char: %d\n' , sizeof( char) );
8     printf( 'Tamanho int: %d\n' , sizeof( int) );
9     printf( 'Tamanho float: %d\n' , sizeof( float) );
10    printf( 'Tamanho double: %d\n' , sizeof( double) );
11    printf( 'Tamanho struct ponto: %d\n' , sizeof( struct
        ponto) );
12    int x;
13    double y;
14    printf( 'Tamanho da variavel x: %d\n' , sizeof x );
15    printf( 'Tamanho da variavel y: %d\n' , sizeof y );
16    system( 'pause' );
17    return 0;
18 }
```

#### 10.1.2 MALLOC()

A função **malloc()** serve para alocar memória durante a execução do programa. É ela quem faz o pedido de memória ao computador e retorna um ponteiro com o endereço do início do espaço de memória alocado. A função **malloc()** possui o seguinte protótipo:

```
void *malloc (unsigned int num);
```

A função **malloc()** recebe 1 parâmetros de entrada

- num: o tamanho do espaço de memória a ser alocado.

e retorna

- NULL: no caso de erro;
- O ponteiro para a primeira posição do array alocado.



Note que a função **malloc()** retorna um **ponteiro genérico** (*void\**). Esse ponteiro pode ser atribuído a qualquer tipo de ponteiro via *type cast*.

Existe uma razão para a função `malloc()` retornar um ponteiro genérico (*void\**): ela não sabe o que iremos fazer com a memória alocada. Veja o exemplo abaixo:

#### Exemplo: usando a função `malloc()`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int *p;
6     p = (int *) malloc(5*sizeof(int));
7     int i;
8     for (i=0; i<5; i++){
9         printf("Digite o valor da posicao %d: ", i);
10        scanf("%d", &p[i]);
11    }
12    system("pause");
13    return 0;
14 }
```

No exemplo acima:

- estamos alocando um array contendo 5 posições de inteiros: **5\*sizeof(int)**;
- a função **sizeof(int)** retorna 4 (número de bytes do tipo **int** na memória). Portanto, são alocados 20 bytes (5 \* 4 bytes);
- a função **malloc()** retorna um ponteiro genérico, o qual é convertido para o tipo do ponteiro via *type cast*: **(int\*)**;
- o ponteiro **p** passa a ser tratado como um array: **p[i]**.



Se não houver memória suficiente para alocar a memória requisitada, a função **malloc()** retorna um ponteiro nulo.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p;
5     p = (int *) malloc(5*sizeof(int));
6     if(p == NULL){
7         printf("Erro: Memoria Insuficiente!\n");
8         exit(1);
9     }
10    int i;
11    for (i=0; i<5; i++){
12        printf("Digite o valor da posicao %d: ",i)
13        ;
14        scanf("%d",&p[i]);
15    }
16    system("pause");
17    return 0;
18 }
```

É importante sempre testar se foi possível fazer a alocação de memória. A função **malloc()** retorna um ponteiro **NULL** para indicar que não há memória disponível no computador, ou que algum outro erro ocorreu que impediu a memória de ser alocada.



No momento da alocação da memória, deve-se levar em conta o tamanho do dado alocado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char *p;
5     //aloca espaço para 1.000 chars
6     p = (char *) malloc(1000);
7     int *p;
8     //aloca espaço para 250 inteiros
9     p = (int *) malloc(1000);
10    system("pause");
11    return 0;
12 }
```

Lembre-se: no momento da alocação da memória deve-se levar em conta o tamanho do dado alocado. Alocar 1000 bytes de memória equivale a um número de elementos diferente dependendo do tipo do elemento:

- 1.000 bytes para char: um array de 1.000 posições de caracteres;
- 1.000 bytes para int: um array de 250 posições de inteiros.

### 10.1.3 CALLOC()

Assim como a função **malloc()**, a função **calloc()** também serve para alocar memória durante a execução do programa. É ela quem faz o pedido de memória ao computador e retorna um ponteiro com o endereço do início do espaço de memória alocado. A função **malloc()** possui o seguinte protótipo:

```
void *calloc (unsigned int num, unsigned int size);
```

A função **malloc()** recebe 2 parâmetros de entrada

- num: o número de elementos no array a ser alocado;
- size: o tamanho de cada elemento do array.

e retorna

- NULL: no caso de erro;
- O ponteiro para a primeira posição do array alocado.

Basicamente, a função **calloc()** faz o mesmo que a função **malloc()**. A diferença é que agora passamos os valores da quantidade de elementos alocados e do tipo de dado alocado como parâmetros distintos da função.

### Exemplo: malloc() versus calloc()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //alocação com malloc
5     int *p;
6     p = (int *) malloc(50*sizeof(int));
7     if(p == NULL){
8         printf("Erro: Memória Insuficiente!\n");
9     }
10    //alocação com calloc
11    int *p1;
12    p1 = (int *) calloc(50, sizeof(int));
13    if(p1 == NULL){
14        printf("Erro: Memória Insuficiente!\n");
15    }
16    system("pause");
17    return 0;
18 }
```

Note, no exemplo acima, que enquanto a função **malloc()** multiplica o total de elementos do array pelo tamanho de cada elemento, a função **calloc()** recebe os dois valores como parâmetros distintos.



Existe uma outra diferença a função **calloc()** e a função **malloc()**: ambas servem para alocar memória, mas a função **calloc()** inicializa todos os **BITS** do espaço alocado com 0.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int i;
6     int *p, *p1;
7     p = (int *) malloc(5*sizeof(int));
8     p1 = (int *) calloc(5, sizeof(int));
9     printf("calloc \t\t malloc\n");
10    for (i=0; i<5; i++)
11        printf("p1[%d] = %d \t p[%d] = %d\n", i, p1[
12            i], i, p[i]);
13    system("pause");
14    return 0;
15 }
```

#### 10.1.4 REALLOC()

A função **realloc()** serve para alocar memória ou realocar blocos de memória previamente alocados pelas funções **malloc()**, **calloc()** ou **realloc()**. Essa função tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num);
```

A função **realloc()** recebe 2 parâmetros de entrada

- Um ponteiro para um bloco de memória previamente alocado;
- **num**: o tamanho em bytes do espaço de memória a ser alocado.

e retorna

- NULL: no caso de erro;
- O ponteiro para a primeira posição do array alocado/realocado.

Basicamente, a função **realloc()** modifica o tamanho da memória previamente alocada e apontada pelo ponteiro **ptr** para um novo valor especificado por **num**, sendo **num** o tamanho em bytes do bloco de memória solicitado (igual a função **malloc()**).



O novo valor de memória alocada (*num*) pode ser **maior** ou **menor** do que o tamanho previamente alocado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i;
5     int *p = malloc(5*sizeof(int));
6     for (i = 0; i < 5; i++){
7         p[i] = i+1;
8     }
9     for (i = 0; i < 5; i++){
10        printf("%d\n",p[i]);
11    }
12    printf("\n");
13    //Diminui o tamanho do array
14    p = realloc(p,3*sizeof(int));
15    for (i = 0; i < 3; i++){
16        printf("%d\n",p[i]);
17    }
18    printf("\n");
19    //Aumenta o tamanho do array
20    p = realloc(p,10*sizeof(int));
21    for (i = 0; i < 10; i++){
22        printf("%d\n",p[i]);
23    }
24    system('pause');
25    return 0;
26 }
```

A função **realloc()** retorna um ponteiro (**void \***) para o novo bloco alocado. Isso é necessário pois a função **realloc()** pode precisar mover o bloco antigo para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco, e nenhuma informação é perdida.



Se o novo tamanho é maior, o valor do bloco de memória recém-alocado é indeterminado.

Isso ocorre pois a função **realloc()** se comporta como a função **malloc()**. Ela não se preocupa em inicializar o espaço alocado.



Se o ponteiro para o bloco de memória previamente alocado for **NULL**, a função **realloc()** irá alocar memória da mesma forma como a função **malloc()** faz.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p;
5     p = (int *) realloc(NULL,50*sizeof(int));
6     for (i = 0; i < 5; i++){
7         p[i] = i+1;
8     }
9     for (i = 0; i < 5; i++){
10        printf("%d\n",p[i]);
11    }
12    system('pause');
13    return 0;
14 }
```

Se não houver memória suficiente para a realocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado.



Se o tamanho de memória solicitado (*num*) for igual a zero, a memória apontada por *\*ptr* **será liberada**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p;
5     p = (int *) malloc(50*sizeof(int));
6     for (i = 0; i < 5; i++){
7         p[i] = i+1;
8     }
9     for (i = 0; i < 5; i++){
10        printf("%d\n",p[i]);
11    }
12    //libera a memória alocada
13    p = (int *) realloc(p,0);
14    system('pause');
15    return 0;
16 }
```

No exemplo acima, a função **realloc()** funciona da mesma maneira que a função **free()** que veremos na próxima seção.

### 10.1.5 FREE()

Diferente das variáveis declarada durante o desenvolvimento do programa, as variáveis alocadas dinamicamente não são liberadas automaticamente pelo programa.



Sempre que alocamos memória de forma dinâmica (**malloc()**, **calloc()** ou **realloc()**) é necessário liberar essa memória quando ela não for mais necessária.

Desalocar, ou liberar, a memória previamente alocada faz com que ela se torne novamente disponível para futuras alocações. Para liberar um bloco de memória previamente alocado utilizamos a função **free()** cujo protótipo é:

```
void free (void *p);
```

A função **free()** recebe apenas um parâmetro de entrada: o ponteiro para o início do bloco de memória alocado.



Para liberar a memória alocada, basta passar para o parâmetro da função **free()** o ponteiro que aponta para o início do bloco de memória alocado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p,i;
5     p = (int *) malloc(50*sizeof(int));
6     if(p == NULL){
7         printf('Erro: Memoria Insuficiente!\n');
8         exit(1);
9     }
10    for (i = 0; i < 50; i++){
11        p[i] = i+1;
12    }
13    for (i = 0; i < 50; i++){
14        printf('%d\n',p[i]);
15    }
16    //libera a memória alocada
17    free(p);
18    system('pause');
19    return 0;
20 }
```

Como o programa sabe quantos bytes devem ser liberados? Quando se aloca a memória, o programa guarda o número de bytes alocados numa “tabela de alocação” interna.



Apenas libere a memória quando tiver certeza de que ela não será mais usada. Do contrário, um erro pode acontecer ou o programa poderá não funcionar como esperado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p,i;
5     p = (int *) malloc(50*sizeof(int));
6     if(p == NULL){
7         printf('Erro: Memoria Insuficiente!\n');
8         exit(1);
9     }
10    for (i = 0; i < 50; i++){
11        p[i] = i+1;
12    }
13    //libera a memória alocada
14    free(p);
15    //tenta imprimir o array
16    //cuja memória foi liberada
17    for (i = 0; i < 50; i++){
18        printf('%d\n',p[i]);
19    }
20    system('pause');
21    return 0;
22 }
```

No exemplo acima nenhum erro ocorre. Isso por que a função **free()** apenas libera a memória. O ponteiro **p** continua com o endereço para onde ela estava reservada. Sendo assim podemos tentar acessá-la. Como ela não nos pertence mais (foi liberada) não há garantias do que está guardado lá.



Sempre libere a memória que não for mais utilizar.

Além disso, convém não deixar ponteiros “soltos” (*dangling pointers*) no programa. Portanto, depois de chamar a função **free()**, atribua **NULL** ao ponteiro:

```
free(p);
```

p = NULL;

É conveniente fazer isso pois ponteiros “soltos” podem ser explorado por hackers para atacar o seu computador.

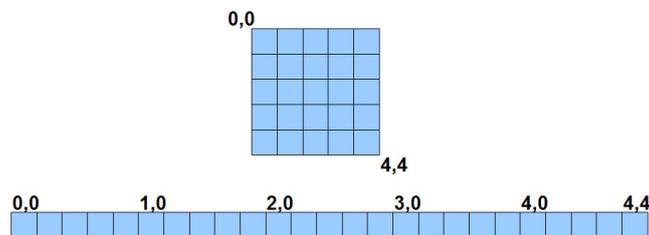
## 10.2 ALOCAÇÃO DE ARRAYS MULTIDIMENSIONAIS

Existem várias soluções na linguagem C para se alocar um array com mais de uma dimensão. A seguir apresentaremos algumas dessas soluções.

### 10.2.1 SOLUÇÃO 1: USANDO ARRAY UNIDIMENSIONAL

Apesar de terem o comportamento de estruturas com mais de uma dimensão, os dados dos arrays multidimensionais são armazenados linearmente na memória. É o uso dos colchetes que cria a impressão de estarmos trabalhando com mais de uma dimensão. Por exemplo:

```
int mat[5][5];
```



Sendo assim, uma solução trivial é simular um array bidimensional (ou com mais dimensões) utilizando um único array unidimensional alocado dinamicamente.



Podemos alocar um array de uma única dimensão e tratá-lo como se fosse uma matriz (2 dimensões).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p;
5     int i, j, Nlinhas = 2, Ncolunas = 2;
6     p = (int *) malloc(Nlinhas * Ncolunas * sizeof
7         (int));
8     for (i = 0; i < Nlinhas; i++){
9         for (j = 0; j < Ncolunas; j++){
10            p[i * Ncolunas + j] = i+j;
11        }
12    }
13    for (i = 0; i < Nlinhas; i++){
14        for (j = 0; j < Ncolunas; j++){
15            printf("%d ", p[i * Ncolunas + j]);
16            printf("\n");
17        }
18    }
19    free(p);
20    system('pause');
21    return 0;
22 }
```

O maior inconveniente dessa abordagem é que temos que abandonar a notação de colchetes para indicar a segunda dimensão da matriz. Como só possuímos uma única dimensão, é preciso calcular o deslocamento no array para simular a segunda dimensão. Isso é feito somando-se o índice da coluna que se quer acessar ao produto do índice da linha que se quer acessar pelo número total de colunas da “matriz”: **[i \* Ncolunas + j]**.



Ao simular uma matriz (2 dimensões) utilizando um array de uma única dimensão perdemos a notação de colchetes para indicar a segunda dimensão.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p;
5     int i, j, Nlinhas = 2, Ncolunas = 2;
6     p = (int *) malloc(Nlinhas * Ncolunas * sizeof
7         (int));
8     for (i = 0; i < Nlinhas; i++){
9         for (j = 0; j < Ncolunas; j++){
10            p[i * Ncolunas + j] = i+j; //CORRETO
11            p[i][j] = i+j; //ERRADO
12        }
13    }
14    free(p);
15    system('pause');
16    return 0;
17 }
```

## 10.2.2 SOLUÇÃO 2: USANDO PONTEIRO PARA PONTEIRO

Se quisermos alocar um array com mais de uma dimensão e manter a notação de colchetes para cada dimensão, precisamos utilizar o conceito de “**ponteiro para ponteiro**” aprendido anteriormente:

```
char ***ptrPtr;
```



A idéia de um ponteiro para ponteiro é similar a anotar o endereço de um papel que tem o endereço da casa do seu amigo.

O exemplo abaixo exemplifica como funciona o conceito de “**ponteiro para ponteiro**”.

### Exemplo: ponteiro para ponteiro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     char letra='a';
5     char *ptrChar;
6     char **ptrPtrChar;
7     char ***ptrPtr;
8     ptrChar = &letra;
9     ptrPtrChar = &ptrChar;
10    ptrPtr = &ptrPtrChar;
11    system('pause');
12    return 0;
13 }
```

Memória		
#	var	conteúdo
119		
120	char ***ptrPtr	#122
121		
122	char **ptrPtrChar	#124
123		
124	char *ptrChar	#126
125		
126	char letra	'a'
127		

Basicamente, para alocar uma matriz (array com 2 dimensões) utiliza-se um ponteiro com 2 níveis.



Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

Por exemplo, se quisermos um array com duas dimensões, precisaremos de um ponteiro com dois níveis (\*\*); Se queremos três dimensões, precisaremos de um ponteiro com três níveis (\*\*\*) e assim por diante.

O exemplo abaixo exemplifica como alocar cada nível de um “**ponteiro para ponteiro**” para criar uma matriz (array com duas dimensões).

**Exemplo: alocando cada nível de um ponteiro para ponteiro.**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     int **p; //2 '*' = 2
        níveis = 2 dimensões
5     int i, j, N = 2;
6     p = (int **) malloc(N*sizeof
        (int *));
7     for (i = 0; i < N; i++){
8         p[i] = (int *) malloc(N*
        sizeof(int));
9         for (j = 0; j < N; j++)
10            scanf("%d",&p[i][j]);
11     }
12     system('pause');
13     return 0;
14 }

```

Memória		
#	var	conteúdo
119	int **p;	#120
120	p[0]	#123
121	p[1]	#126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		

No exemplo acima, utilizando um ponteiro com 2 níveis (**int \*\*p**), nós alocamos no primeiro nível do ponteiro um **array de ponteiros** representando as **linhas da matriz**. Essa tarefa é realizada pela primeira chamada da função **malloc()**, a qual aloca o array usando o tamanho de um ponteiro para **int**:

`sizeof(int *)`

Em seguida, para cada posição desse **array de ponteiros**, nós alocamos um **array de inteiros**, o qual representa o espaço para as **colunas da matriz**, as quais irão efetivamente manter os dados. Essa tarefa é realizada pela segunda chamada da função **malloc()**, dentro do comando **for**, a qual aloca o array usando o tamanho de um **int**:

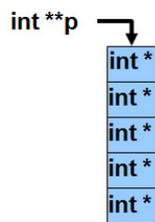
`sizeof(int)`



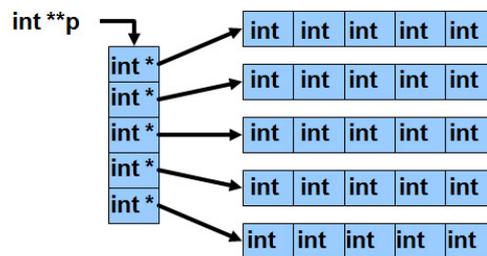
Note que desse modo é possível manter a notação de colchetes para representar cada uma das dimensões da matriz.

A figura abaixo exemplifica como funciona o processo de alocação de uma matriz usando o conceito de ponteiro para ponteiro:

1º malloc  
Cria as linhas da matriz



2º malloc  
Cria as colunas da matriz



Preste bastante atenção ao exemplo da figura acima. Note que sempre que se aloca memória, os dados alocados possuem um nível a menos que o do ponteiro usado na alocação. Assim, se tivermos um

- ponteiro para inteiro (**int \***), iremos alocar um array de inteiros (**int**);
- ponteiro para ponteiro para inteiro (**int \*\***), iremos alocar um array de ponteiros para inteiros (**int \***);
- ponteiro para ponteiro para ponteiro para inteiro (**int \*\*\***), iremos alocar um array de inteiros (**int \*\***);



Diferente dos arrays de uma dimensão, para liberar da memória um array com mais de uma dimensão, é preciso liberar a memória alocada em cada uma de suas dimensões, na ordem inversa da que foi alocada.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int **p; //2 '*' = 2 níveis = 2 dimensões
5     int i, j, N = 2;
6     p = (int **) malloc(N*sizeof(int *));
7     for (i = 0; i < N; i++){
8         p[i] = (int *) malloc(N*sizeof(int));
9         for (j = 0; j < N; j++)
10            scanf("%d",&p[i][j]);
11     }
12     for (i = 0; i < N; i++){
13         free(p[i]);
14     }
15     free(p);
16     system('pause');
17     return 0;
18 }
```

Para alocar nossa matriz, utilizamos duas chamadas da função **malloc()**: a primeira chamada faz a alocação das linhas, enquanto a segunda chamada faz a alocação das colunas. Na hora de liberar a matriz, devemos liberar a memória no sentido inverso da alocação: primeiro liberamos as colunas, para depois liberar as linhas da matriz. Essa ordem deve ser respeitada pois, se liberarmos primeiro as linhas, perdemos os ponteiros para onde estão alocadas as colunas e assim não poderemos liberá-las.



Esse tipo de alocação, usando ponteiro para ponteiro, permite criar matrizes que não sejam quadradas ou retangulares.

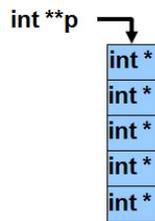
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int **p; //2 '*' = 2 níveis = 2 dimensões
5     int i, j, N = 3;
6     p = (int **) malloc(N*sizeof(int *));
7     for (i = 0; i < N; i++){
8         p[i] = (int *) malloc((i+1)*sizeof(int));
9         for (j = 0; j < (i+1); j++)
10            scanf("%d", &p[i][j]);
11     }
12     for (i = 0; i < N; i++){
13         free(p[i]);
14     }
15     free(p);
16     system('pause');
17     return 0;
18 }
```

Note, no exemplo acima, que a segunda chamada da função **malloc()** está condicionada ao valor de *i*: `malloc((i+1)*sizeof(int))`. Assim, as colunas de cada linha da matriz terão um número diferente de elementos. De fato, o código acima cria uma matriz triangular inferior, como fica claro pela figura abaixo:

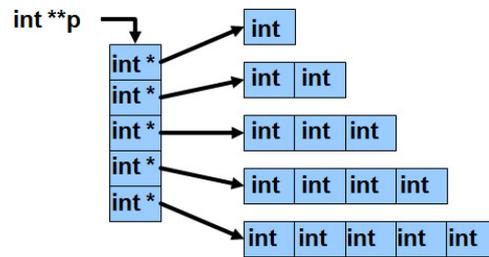
### 10.2.3 SOLUÇÃO 3: PONTEIRO PARA PONTEIRO PARA ARRAY

A terceira solução possível para alocar um array com mais de uma dimensão e manter a notação de colchetes para cada dimensão é um misto das duas soluções anteriores: simulamos um array bidimensional (ou com mais dimensões) utilizando:

1º malloc  
Cria as linhas da matriz



2º malloc  
Cria as colunas da matriz



- um array unidimensional alocado dinamicamente e contendo as posições de todos os elementos;
- um array de ponteiros unidimensional que irá simular as dimensões e assim manter a notação de colchetes.

O exemplo abaixo exemplifica como simular uma matriz utilizando um array de ponteiros e um array unidimensional contendo os dados:

### Exemplo: ponteiro para ponteiro e um array unidimensional

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main () {
5     int *v; //1 '*' = 1 nível = 1 dimensão
6     int **p; //2 '*' = 2 níveis = 2 dimensões
7     int i, j, Nlinhas = 2, Ncolunas = 2;
8     v = (int *) malloc(Nlinhas * Ncolunas * sizeof(int));
9     p = (int **) malloc(Nlinhas * sizeof(int *));
10
11     for (i = 0; i < Nlinhas; i++){
12         p[i] = v + i * Ncolunas;
13         for (j = 0; j < Ncolunas; j++)
14             scanf("%d",&p[i][j]);
15     }
16
17     for (i = 0; i < Nlinhas; i++){
18         for (j = 0; j < Ncolunas; j++)
19             printf("%d ",p[i][j]);
20         printf("\n");
21     }
22
23     free(v);
24     free(p);
25
26     system("pause");
27     return 0;
28 }
```

No exemplo acima, utilizando um ponteiro com 1 nível (**int \*v**), nós alocamos o total de elementos da matriz (**Nlinhas \* Ncolunas**). Essa tarefa é realizada pela primeira chamada da função **malloc()**, a qual aloca o array usando o tamanho de um tipo **int**:

**sizeof(int)**

Em seguida, utilizando um ponteiro com 2 níveis (**int \*\*p**), nós alocamos no primeiro nível do ponteiro um **array de ponteiros** representando as **linhas da matriz**. Essa tarefa é realizada pela primeira chamada da função **malloc()**, a qual aloca o array usando o tamanho de um ponteiro para **int**:

**sizeof(int \*)**

Por fim, utilizando de aritmética de ponteiros, nós associamos cada posição do array de ponteiros para uma porção do array de inteiros:

$p[i] = v + i * Ncolunas;$

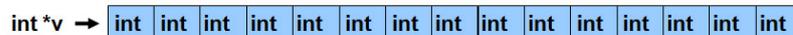


Note que, como tempos cada posição do array **p** associada a um porção de outro array (**v**), a notação de colchetes para mais de uma dimensão é mantida.

A figura abaixo exemplifica como funciona o processo de alocação de uma matriz usando o conceito de ponteiro para ponteiro e array unidimensional:

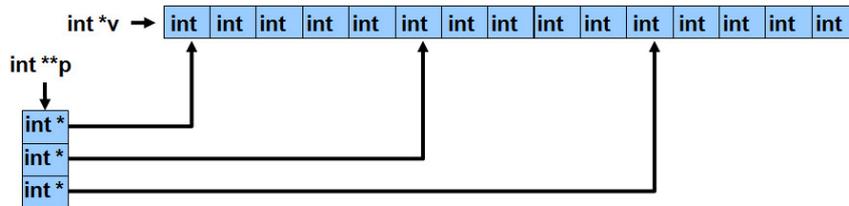
1º malloc

Cria todas as colunas da matriz com um único array



2º malloc

Cria as linhas da matriz e associa ao array de colunas



Do ponto de vista de alocação, essa solução é mais simples do que a anterior (Solução 2). Ela utiliza apenas duas chamadas da função **malloc()** para alocar toda a matriz. Conseqüentemente, apenas duas chamadas da função **free()** são necessárias para liberar a memória alocada.

Por outro lado, para arrays com mais de duas dimensões, essa solução pode se mostrar mais complicada de se trabalhar já que envolve aritmética de ponteiros no cálculo que associa as linhas com o array contendo os dados.

## 11 ARQUIVOS

Um arquivo, de modo abstrato, nada mais é do que uma coleção de bytes armazenados em um dispositivo de armazenamento secundário, que é geralmente um disco rígido, CD, DVD, etc. Essa coleção de bytes pode ser interpretada das mais variadas maneiras:

- caracteres, palavras, ou frases um documento de texto;
- campos e registros de uma tabela de banco de dados;
- pixels de uma imagem;
- etc.

O que define significado de um arquivo em particular é a maneira como as estruturas de dados estão organizadas e as operações usadas por um programa de processar (ler ou escrever) esse arquivo.

As vantagens de se usar arquivos são muitas:

- É geralmente baseado em algum tipo de armazenamento durável. Ou seja, seus dados permanecem disponíveis para uso dos programas mesmo que o programa que o gerou já tenha sido encerrado;
- Permitem armazenar uma grande quantidade de informação;
- O acesso aos dados pode ser ou não seqüencial;
- Acesso concorrente aos dados (ou seja, mais de um programa pode utilizá-lo ao mesmo tempo).

A linguagem C permite manipular arquivos das mais diversas formas. Ela possui um conjunto de funções que podem ser utilizadas pelo programador para criar e escrever em novos arquivos, ler o seu conteúdo, independente do tipo de dados que lá estejam armazenados. A seguir, serão apresentados os detalhes necessários para um programador poder rabalhar com arquivos em seu programa.

### 11.1 TIPOS DE ARQUIVOS

Basicamente, a linguagem C trabalha com apenas dois tipos de arquivos: **arquivos texto** e **arquivos binários**.



Um *arquivo texto* armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de textos simples como o Bloco de Notas.

Os dados gravados em um arquivo texto são gravados exatamente como seriam impressos na tela. Por isso eles podem ser modificados por um editor de textos simples como o Bloco de Notas. No entanto, para que isso ocorra, os dados são gravados como caracteres de 8 bits utilizando a tabela ASCII. Ou seja, durante a gravação dos dados existe uma etapa de “conversão” dos dados.



Essa “conversão” dos dados faz com que os arquivos texto sejam maiores. Além disso, suas operações de escrita e leitura consomem mais tempo em comparação as dos arquivos binários.

Para entender essa conversão dos dados em arquivos texto, imagine um número inteiro com 8 dígitos: 12345678. Esse número ocupa 32 bits na memória. Porém, quando for gravado em um arquivo texto, cada dígito dela será convertido para seu caractere ASCII, ou seja, 8 bits por dígito. Como resultado final, esse número ocupará 64 bits no arquivo, o dobro do seu tamanho na memória.



Dependendo do ambiente onde o aplicativo é executado, algumas conversões de caracteres especiais podem ocorrer na escrita/leitura de dados em arquivos texto.

Isso ocorre como uma forma de adaptar o arquivo ao formato de arquivo texto específico do sistema. No modo de arquivo texto, um caractere de **nova linha**, “\n”, pode ir a ser convertido pelo sistema para para o par de caracteres **retorno de carro + nova linha**, “\r\n”.



Um *arquivo binário* armazena uma seqüência de bits que está sujeita as convenções dos programas que o gerou.

Os dados gravados em um arquivo binário são gravados exatamente como estão organizados na memória do computador. Isso significa que não existe uma etapa de “conversão” dos dados. Portanto, suas operações de escrita e leitura são mais rápidas do que as realizadas em arquivos texto.

Voltemos ao nosso número inteiro com 8 dígitos: 12345678. Esse número ocupa 32 bits na memória. Quando for gravado em um arquivo binário, o conteúdo da memória será copiado diretamente para o arquivo, sem conversão. Como resultado final, esse número ocupará os mesmo 32 bits no arquivo.

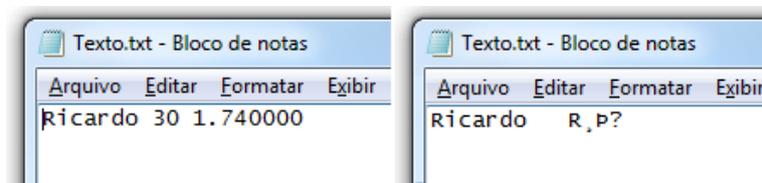


São exemplos de arquivos binários os arquivos executáveis, arquivos compactados, arquivos de registros, etc.

Para entender melhor a diferença entre esses dois tipos de arquivos, imagine os seguintes dados a serem gravados:

```
char nome[20] = "Ricardo";  
int i = 30;  
float a = 1.74;
```

A figura abaixo mostra como seria o resultado da gravação desses dados em um arquivo texto e em um arquivo binário. Note que os dados de um arquivo texto podem ser facilmente modificados por um editor de textos.



Arquivo Texto

Arquivo Binário



Caracteres são legíveis tanto em arquivos textos quanto binários.

## 11.2 SOBRE ESCRITA E LEITURA EM ARQUIVOS

Quanto as operações de escrita e leitura em arquivos, a linguagem C possui uma série de funções prontas para a manipulação de arquivos, cujos protótipos estão reunidos na biblioteca padrão de entrada e saída, **stdio.h**.



Diferente de outras linguagens, a linguagem C não possui funções que automaticamente leiam todas as informações de um arquivo.

Na linguagem C, as funções de escrita e leitura em arquivos se limitam a operações de abrir/fechar e ler/escrever caracteres e bytes. Fica a cargo do programador criar a função que irá ler ou escrever um arquivo de uma maneira específica.

### 11.3 PONTEIRO PARA ARQUIVO

A linguagem C usa um tipo especial de ponteiro para manipular arquivos. Quando o arquivo é aberto, esse ponteiro aponta para o registro 0 (o primeiro registro no arquivo). É esse ponteiro que controla qual o próximo byte a ser acessado por um comando de leitura. É ele também que indica quando chegamos ao final de um arquivo, entre outras tarefas.



Todas as funções de manipulação de arquivos trabalham com o conceito de “ponteiro de arquivo”.

Podemos declarar um ponteiro de arquivo da seguinte maneira:

```
FILE *p;
```

Nesse caso, **p** é o ponteiro que nos permitirá manipular arquivos na linguagem C. Um ponteiro de arquivo nada mais é do que um ponteiro para uma área na memória chamada de “buffer”. Nela se encontram vários dados sobre o arquivo aberto, tais como o nome do arquivo e posição atual.

### 11.4 ABRINDO E FECHANDO UM ARQUIVO

#### 11.4.1 ABRINDO UM ARQUIVO

A primeira coisa que devemos fazer ao se trabalhar com arquivos é abrí-lo. Para abrir um arquivo usa-se a função **fopen()**, cujo protótipo é:

```
FILE *fopen(char *nome_do_arquivo, char *modo)
```

A função **fopen()** recebe 2 parâmetros de entrada

- nome\_do\_arquivo: uma string contendo o nome do arquivo que deverá ser aberto;
- modo: uma string contendo o modo de abertura do arquivo.

e retorna

- NULL: no caso de erro;
- O ponteiro para o arquivo aberto.

### CAMINHO ABSOLUTO E RELATIVO PARA O ARQUIVO



No parâmetro **nome\_do\_arquivo** pode-se trabalhar com caminhos **absolutos** ou **relativos**.

Imagine que o arquivo com que desejamos trabalhar esteja no seguinte local:

“C:\Projetos\NovoProjeto\arquivo.txt”

O **caminho absoluto** de um arquivo é uma seqüência de diretórios separados pelo caractere barra ('\'), que se inicia no diretório raiz e termina com o nome do arquivo. Nesse caso, o **caminho absoluto** do arquivo é a string

“C:\Projetos\NovoProjeto\arquivo.txt”

Já o **caminho relativo**, como o próprio nome diz, é relativo ao local onde o programa se encontra. Nesse caso, o sistema inicia a pesquisa pelo nome do arquivo a partir do diretório do programa. Se tanto o programa quanto o arquivo estiverem no mesmo local, o caminho relativo até esse arquivo será

“.\arquivo.txt”

ou

“arquivo.txt”

Se o programa estivesse no diretório “C:\Projetos”, o **caminho relativo** até o arquivo seria

“.\NovoProjeto\arquivo.txt”



Ao se trabalhar com caminhos **absolutos** ou **relativos**, sempre usar duas barras ‘\\’ ao invés de uma ‘\’ para separar os diretórios.

Isso é necessário para evitar que alguma combinação de caractere e barra seja confundida com uma **seqüências de escape** que não seja a barra invertida. As duas barras ‘\\’ são a seqüências de escape da própria barra invertida. Assim, o **caminho absoluto** do arquivo anteriormente definido passa a ser

“C:\\Projetos\\NovoProjeto\\arquivo.txt”

## COMO POSSO ABRIR MEU ARQUIVO



O modo de abertura do arquivo determina que tipo de uso será feito do arquivo.

O modo de abertura do arquivo diz à função **fopen()** qual é o que tipo de uso que será feito do arquivo. Pode-se, por exemplo, querer escrever em um arquivo binário, ou ler um arquivo texto. A tabela a seguir mostra os modos válidos de abertura de um arquivo:

Modo	Arquivo	Função
“r”	Texto	Leitura. Arquivo deve existir.
“w”	Texto	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
“a”	Texto	Escrita. Os dados serão adicionados no fim do arquivo (“append”).
“rb”	Binário	Leitura. Arquivo deve existir.
“wb”	Binário	Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
“ab”	Binário	Escrita. Os dados serão adicionados no fim do arquivo (“append”).
“r+”	Texto	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
“w+”	Texto	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
“a+”	Texto	Leitura/Escrita. Os dados serão adicionados no fim do arquivo (“append”).
“r+b”	Binário	Leitura/Escrita. O arquivo deve existir e pode ser modificado.
“w+b”	Binário	Leitura/Escrita. Cria arquivo se não houver. Apaga o anterior se ele existir.
“a+b”	Binário	Leitura/Escrita. Os dados serão adicionados no fim do arquivo (“append”).

Note que para cada tipo de ação que o programador deseja realizar existe um modo de abertura de arquivo mais apropriado.



O arquivo deve sempre ser aberto em um modo que permita executar as operações desejadas.

Imagine que desejemos gravar uma informação em um arquivo texto. Obviamente, esse arquivo deve ser aberto em um modo que permita escrever nele. Já um arquivo aberto para leitura não irá permitir outra operação que não seja a leitura de dados.

### Exemplo: abrir um arquivo binário para escrita

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     FILE *fp;
5     fp = fopen( ' 'exemplo.bin' ', 'wb' );
6     if (fp == NULL)
7         printf ( ' 'Erro na abertura do arquivo.\n' );
8
9     fclose (fp);
10    system ( ' 'pause' ' );
11    return 0;
12 }
```

No exemplo anterior, o comando **fopen()** tenta abrir um arquivo de nome “exemplo.bin” no modo de escrita para arquivos binários, “wb”. Note que foi utilizado o **caminho relativo** do arquivo. Na sequência, a condição **if (fp == NULL)** testa se o arquivo foi aberto com sucesso. Isso é

### FINALIZANDO O PROGRAMA NO CASO DE ERRO



No caso de um erro, a função **fopen()** retorna um ponteiro nulo (**NULL**).

Caso o arquivo não tenha sido aberto com sucesso, provavelmente o programa não poderá continuar a executar. Nesse caso, utilizamos a função **exit()**, presente na biblioteca **stdlib.h**, para abortar o programa. Seu protótipo é:

```
void exit (int codigo_de_retorno)
```

A função **exit()** pode ser chamada de qualquer ponto do programa. Ela faz com que o programa termine e retorne, para o sistema operacional, o valor definido em **codigo\_de\_retorno**.



A convenção mais usada é que um programa retorne **zero** no caso de um término normal e retorne um número **não nulo** no caso de ter ocorrido um problema durante a sua execução.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     FILE *fp;
5     fp = fopen( ' 'exemplo.bin' ', 'wb' );
6     if (fp == NULL){
7         printf ( ' 'Erro na abertura do arquivo. Fim
8             de programa.\n' );
9         system( ' 'pause' ');
10        exit(1);
11    }
12    fclose (fp);
13    system( ' 'pause' ');
14    return 0;
15 }
```

#### 11.4.2 FECHANDO UM ARQUIVO

Sempre que terminamos de usar um arquivo, devemos fechá-lo. Para realizar essa tarefa, usa-se a função **fclose()**, cujo protótipo é:

```
int fclose (FILE *fp)
```

Basicamente, a função **fclose()** recebe como parâmetro o ponteiro **fp** que determina o arquivo a ser fechado. Como resultado, a função retorna um valor inteiro igual a zero no caso de sucesso no fechamento do arquivo. Um valor de retorno diferente de zero significa que houve um erro nessa tarefa.



Por que devemos fechar o arquivo?

Ao fechar um arquivo, todo caractere que tenha permanecido no “buffer” é gravado. O “buffer” é uma área intermediária entre o arquivo no disco e o programa em execução. Trata-se de uma região de memória que armazena temporariamente os caracteres a serem gravados em disco. Apenas quando o “buffer” está cheio é que seu conteúdo é escrito no disco.



Por que utilizar um “buffer” durante a escrita em um arquivo?

O uso de um “buffer” é uma questão de **eficiência**. Para ler e escrever arquivos no disco rígido é preciso posicionar a cabeça de gravação em um ponto específico do disco rígido. E isso consome tempo. Se tivéssemos que fazer isso para cada caractere lido ou escrito, as operações de leitura e escrita de um arquivo seriam extremamente lentas. Assim a gravação só é realizada quando há um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.



A função **exit()** fecha todos os arquivos que um programa tiver aberto.

## 11.5 ESCRITA E LEITURA EM ARQUIVOS

Uma vez aberto um arquivo, pode-se ler ou escrever nele. Para realizar essas tarefas, a linguagem C conta com uma série de funções de escrita e leitura que variam de funcionalidade de acordo com o tipo de dado que se deseja manipular. Desse modo, todas e as mais diversas aplicações do programador podem ser atendidas.

### 11.5.1 ESCRITA E LEITURA DE CARACTERE

#### ESCREVENDO UM CARACTERE

As funções mais básicas e fáceis de se trabalhar em um arquivo são as responsáveis pela escrita e leitura de um único caractere. Para se escrever um caractere em um arquivo usamos a função **fputc()**, cujo protótipo é:

```
int fputc(char c, FILE *fp);
```

A função **fputc()** recebe 2 parâmetros de entrada

- **c**: o caractere a ser escrito no arquivo. Note que o caractere é passado como seu valor inteiro;
- **fp**: a variável que está associada ao *arquivo* onde o caractere será escrito.

e retorna

- a constante **EOF** (em geral, -1), se houver erro na escrita;
- o próprio caractere, se ele foi escrito com sucesso.



Cada chamada da função **fputc()** grava um único caractere **c** no arquivo especificado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main(){
5     FILE *arq;
6     char string[100];
7     int i;
8     arq = fopen('arquivo.txt', 'w');
9     if(arq == NULL){
10        printf('Erro na abertura do arquivo');
11        system('pause');
12        exit(1);
13    }
14    printf('Entre com a string a ser gravada no
15        arquivo:');
16    gets(string);
17    //Grava a string, caractere a caractere
18    for(i = 0; i < strlen(string); i++)
19        fputc(string[i], arq);
20    fclose(arq);
21    system('pause');
22    return 0;
23 }
```

No exemplo anterior, a função **fputc()** é utilizada para escrever um caractere na posição atual do arquivo, como indicado pelo indicador de posição interna do arquivo. Em seguida, esse indicador de posição interna é avançado em um caractere de modo a ficar pronto para a escrita do próximo caractere.



A função **fputc()** também pode ser utilizada para escrever um caractere no dispositivo de saída padrão (geralmente a tela do monitor).

Para usar a função **fputc()** para escrever na tela, basta modificar o arquivo no qual se deseja escrever para a constante **stdout**. Essa constante

trata-se de um dos arquivos pré-definidos do sistema, um ponteiro para o dispositivo de saída padrão (geralmente o vídeo) das aplicações. Assim, o comando

```
fputc('*', stdout);
```

escreve um "\*" na tela do monitor (dispositivo de saída padrão) ao invés de em um arquivo no disco rígido.

### LENDO UM CARACTERE

Da mesma maneira que é possível gravar um único caractere em um arquivo, também é possível fazer a sua leitura. A função que corresponde a leitura de caracteres é a função **fgetc()**, cujo protótipo é:

```
int fgetc(FILE *fp);
```

A função **fgetc()** recebe como parâmetro de entrada apenas a variável que está associada ao *arquivo* de onde o caractere será lido. Essa função retorna

- a constante **EOF** (em geral, -1), se houver erro na leitura;
- o caractere lido do arquivo, na forma de seu valor inteiro, se o mesmo foi lido com sucesso.



Cada chamada da função **fgetc()** lê um único caractere do arquivo especificado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     FILE *arq;
5     char c;
6     arq = fopen("arquivo.txt", "r");
7     if(arq == NULL){
8         printf("Erro na abertura do arquivo");
9         system("pause");
10        exit(1);
11    }
12    int i;
13    for(i = 0; i < 5; i++){
14        c = fgetc(arq);
15        printf("%c", c);
16    }
17    fclose(arq);
18    system("pause");
19    return 0;
20 }
```

No exemplo anterior, a função **fgetc()** é utilizada para ler 5 caracteres de um arquivo. Note que a função **fgetc()** sempre retorna o caractere atualmente apontado pelo indicador de posição interna do arquivo especificado.



A cada operação de leitura, o indicador de posição interna do arquivo é avançado em um caractere para apontar para o próximo caractere a ser lido.

Similar ao que acontece com a função **fputc()**, a função **fgetc()** também pode ser utilizada para a leitura de caracteres do teclado. Para tanto, basta modificar o arquivo do qual se deseja ler para a constante **stdin**. Essa constante trata-se de um dos arquivos pré-definidos do sistema, um ponteiro para o dispositivo de entrada padrão (geralmente o teclado) das aplicações. Assim, o comando

```
char c = fgetc(stdin);
```

lê um caractere do teclado (dispositivo de entrada padrão) ao invés de um arquivo no disco rígido.



O que acontece quando a função **fgetc()** tenta ler um caractere de um arquivo que já acabou?

Neste caso, precisamos que a função retorne algo indicando que o arquivo acabou. Porém, todos os 256 caracteres da tabela ASCII são “válidos” em um arquivo. Para evitar esse tipo de situação, a função **fgetc()** não devolve um valor do tipo **char**, mas do tipo **int**. O conjunto de valores do tipo **char** está contido dentro do conjunto de valores do tipo **int**. Se o arquivo tiver acabado, a função **fgetc()** devolve um valor inteiro que não possa ser confundido com um valor do tipo **char**.



Quando atinge o final de um arquivo, a função **fgetc()** devolve a constante **EOF (End Of File)**, que está definida na biblioteca **stdio.h**. Em muitos computadores o valor de **EOF** é definido como -1.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     FILE *arq;
5     char c;
6     arq = fopen("arquivo.txt", "r");
7     if(arq == NULL){
8         printf("Erro na abertura do arquivo");
9         system("pause");
10        exit(1);
11    }
12    while((c = fgetc(arq)) != EOF)
13        printf("%c", c);
14    fclose(arq);
15    system("pause");
16    return 0;
17 }
```

No exemplo anterior, a função **fgetc()** é utilizada juntamente com a constante **EOF** para ler não apenas alguns caracteres, mas para continuar lendo caracteres enquanto não chegarmos ao final do arquivo.

## 11.5.2 FIM DO ARQUIVO

Como visto anteriormente, a constante **EOF** (“End of file”) indica o fim de um arquivo. Porém, quando manipulando dados binários, um valor inteiro

igual ao valor da constante **EOF** pode ser lido. Nesse caso, se utilizarmos a constante **EOF** para verificar se chegamos ao final do arquivo, vamos receber a confirmação de ter chegado ao final do arquivo, quando na verdade ainda não chegamos ao seu final. Para evitar este tipo de situação, a linguagem C inclui a função **feof()** que determina quando o final de um arquivo foi atingido. Seu protótipo é:

```
int feof(FILE *fp)
```

Basicamente, a função **feof()** recebe como parâmetro o ponteiro **fp** que determina o arquivo a ser verificado. Como resultado, a função retorna um valor inteiro igual a **ZERO** se ainda não tiver atingido o final do arquivo. Um valor de retorno diferente de zero significa que já foi atingido o final do arquivo.



Basicamente, a função **feof()** retorna um valor inteiro diferente de zero se o arquivo chegou ao fim, caso contrário, retorna **ZERO**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     FILE *fp;
5     char c;
6     fp = fopen('arquivo.txt','r');
7     if(!fp){
8         printf('Erro na abertura do arquivo\n');
9         system('pause');
10        exit(1);
11    }
12    while(!feof(fp)){
13        c = fgetc(fp);
14        printf('%c',c);
15    }
16    fclose(fp);
17    system('pause');
18    return 0;
19 }
```

### 11.5.3 ARQUIVOS PRÉ-DEFINIDOS

Como visto durante o aprendizado das funções **fputc()** e **fgetc()**, os ponteiros **stdin** e **stdout** podem ser utilizados para acessar os dispositivos

de entrada (geralmente o teclado) e saída (geralmente o vídeo) padrão. Porém, existem outros ponteiros que podem ser utilizados.



No início da execução de um programa, o sistema automaticamente abre alguns arquivos pré-definidos, entre eles **stdin** e **stdout**.

<b>stdin</b>	Dispositivo de entrada padrão (geralmente o teclado)
<b>stdout</b>	Dispositivo de saída padrão (geralmente o vídeo)
<b>stderr</b>	Dispositivo de saída de erro padrão (geralmente o vídeo)
<b>stdaux</b>	Dispositivo de saída auxiliar (em muitos sistemas, associado à porta serial)
<b>stdprn</b>	Dispositivo de impressão padrão (em muitos sistemas, associado à porta paralela)

#### 11.5.4 FORÇANDO A ESCRITA DOS DADOS DO “BUFFER”

Vimos anteriormente que os dados gravados em um arquivo são primeiramente gravados em um “buffer”, uma área intermediária entre o arquivo no disco e o programa em execução, e somente quando este “buffer” está cheio é que seu conteúdo é escrito no disco. Também vimos que o uso do “buffer” é uma questão de **eficiência**. Porém, a linguagem C permite que nós forcemos a gravação de qualquer dado contido no “buffer” no momento em que quisermos. Para realizar essa tarefa, usa-se a função **fflush()**, cujo protótipo é:

```
int fflush(FILE *fp)
```

Basicamente, a função **fflush()** recebe como parâmetro o ponteiro **fp** que determina o arquivo a ser manipulado. Como resultado, a função **fflush()** retorna

- o valor 0 (ZERO), se a operação foi realizada com sucesso;
- a constante **EOF** (em geral, -1), se houver algum erro.



O comportamento da função **fflush()** depende do modo como o arquivo foi aberto.

- Se o arquivo apontado por **fp** foi aberto para **escrita**, os dados contidos no “buffer de saída” são gravados no arquivo;

- Se o arquivo apontado por **fp** foi aberto para **leitura**, o comportamento depende da implementação da biblioteca. Em algumas implementações os dados contidos no “buffer de entrada” são apagados, mas esse não é um comportamento padrão;
- Se **fp** for um ponteiro nulo (**fp = NULL**), todos os arquivos abertos são liberados.

Abaixo, tem-se um exemplo de um programa que utiliza a função **fflush()** para forçar a gravação de dados no arquivo:

#### Exemplo: forçando a gravação de dados em um arquivo

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main() {
5     FILE *arq;
6     char string[100];
7     int i;
8     arq = fopen('arquivo.txt', 'w');
9     if(arq == NULL) {
10        printf('Erro na abertura do arquivo');
11        system('pause');
12        exit(1);
13    }
14    printf('Entre com a string a ser gravada no
arquivo:');
15    gets(string);
16    for(i = 0; i < strlen(string); i++)
17        fputc(string[i], arq);
18
19    fflush(arq);
20    fclose(arq);
21    system('pause');
22    return 0;
23 }

```

#### 11.5.5 SABENDO A POSIÇÃO ATUAL DENTRO DO ARQUIVO

Outra operação bastante comum é saber onde estamos dentro de um arquivo. Para realizar essa tarefa, usa-se a função **ftell()**, cujo protótipo é:

```
long int ftell(FILE *fp)
```

Basicamente, a função **ftell()** recebe como parâmetro o ponteiro **fp** que determina o arquivo a ser manipulado. Como resultado, a função **ftell()** retorna a posição atual dentro do fluxo de dados do arquivo:

- para arquivos **binário**, o valor retornado indica o número de bytes lidos a partir do início do arquivo;
- para arquivos **texto**, não existe garantia de que o valor retornado seja o número exato de bytes lidos a partir do início do arquivo;
- se um erro ocorrer, o valor -1 no formato **long** é retornado.

Abaixo, tem-se um exemplo de um programa que utiliza a função **ftell()** para descobrir o tamanho, em bytes, de um arquivo:

#### Exemplo: descobrindo o tamanho de um arquivo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main(){
5     FILE *arq;
6     arq = fopen( ' 'arquivo.bin' ', ' 'rb' ' );
7     if( arq == NULL){
8         printf( ' 'Erro na abertura do arquivo' ' );
9         system( ' 'pause' ' );
10        exit(1);
11    }
12    int tamanho;
13    fseek ( arq,0,SEEK_END);
14    tamanho = ftell( arq);
15    fclose(arq);
16    printf( ' 'Tamanho do arquivo em bytes: %d: ' ',tamanho
17           );
18    system( ' 'pause' ' );
19    return 0;
20 }
```

### 11.5.6 ESCRITA E LEITURA DE STRINGS

Até o momento, apenas caracteres únicos puderam ser escritos em um arquivo. Porém, existem funções na linguagem C que permitem escrever e ler uma sequência de caracteres, isto é, uma string, em um arquivo.

#### ESCREVENDO UMA STRING

Para se escrever uma string em um arquivo usamos a função **fputs()**, cujo protótipo é:

```
int fputs (char *str, FILE *fp);
```

A função **fputs()** recebe 2 parâmetros de entrada

- **str**: a string (array de caracteres) a ser escrita no arquivo;
- **fp**: a variável que está associada ao *arquivo* onde a string será escrita.

e retorna

- a constante **EOF** (em geral, -1), se houver erro na escrita;
- um valor diferente de ZERO, se o texto for escrito com sucesso.

#### Exemplo: escrevendo uma string em um arquivo com fputs()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char str[20] = "Hello World!";
5     int result;
6     FILE *arq;
7     arq = fopen("ArqGrav.txt", "w");
8     if (arq == NULL) {
9         printf("Problemas na CRIACAO do arquivo\n");
10        system("pause");
11        exit(1);
12    }
13    result = fputs(str, arq);
14    if (result == EOF)
15        printf("Erro na Gravacao\n");
16
17    fclose(arq);
18    system("pause");
19    return 0;
20 }
```

No exemplo anterior, o comando **fopen()** abre um arquivo de nome "Arq-Grav.txt" no modo de escrita para arquivos texto, "w". Na sequência, a string contida na variável **str** é escrita no arquivo por meio do comando **fputs(str,arq)**, sendo o resultado dessa operação devolvido na variável **result**.



A função **fputs()** não coloca o caracter de nova linha ‘\n’, nem nenhum outro tipo de caractere, no final da string escrita no arquivo. Essa tarefa pertence ao programador.

Imagine o seguinte conjunto de comandos:

```
fputs("Hello",arq);  
fputs("World",arq);
```

O resultado da execução desses dois comandos será a escrita da string **"HelloWorld"** no arquivo. Note que nem mesmo um espaço entre elas foi adicionado. A função **fputs()** simplesmente escreve no arquivo aquilo que o programador ordenou, e mais nada. Se o programador quisesse separá-las com um espaço, deve fazer como abaixo:

```
fputs("Hello ",arq);  
fputs("World",arq);
```

Note que agora existe um espaço ao final da string **"Hello "**. Portanto, o resultado no arquivo será a string **"Hello World"**. O mesmo vale para qualquer outro caractere, como a quebra de linha ‘\n’.



Como a função **fputc()**, a função **fputs()** também pode ser utilizada para escrever uma string no dispositivo de saída padrão (geralmente a tela do monitor).

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 int main(){  
4     char texto[30] = "Hello World\n";  
5     fputs(texto, stdout);  
6     system("pause");  
7     return 0;  
8 }
```

## LENDO UMA STRING

Da mesma maneira como é possível gravar uma string em um arquivo, também é possível fazer a sua leitura. A função utilizada para realizar essa tarefa é a função **fgets()**, cujo protótipo é:

```
char *fgets (char *str, int tamanho, FILE *fp);
```

A função **fgets()** recebe 3 parâmetros de entrada

- **str**: a string onde os caracteres lidos serão armazenados;
- **tamanho**: o limite máximo de caracteres a serem lidos;
- **fp**: a variável que está associada ao *arquivo* de onde a string será lida.

e retorna

- **NULL**: no caso de erro ou fim do arquivo;
- O ponteiro para o primeiro caractere da string recuperada em **str**.

#### Exemplo: lendo uma string de um arquivo com fgets()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char str[20];
5     int result;
6     FILE *arq;
7     arq = fopen('ArqGrav.txt', 'r');
8     if (arq == NULL) {
9         printf('Problemas na ABERTURA do arquivo\n');
10        system('pause');
11        exit(1);
12    }
13    result = fgets(str, 13, arq);
14    if (result == EOF)
15        printf('Erro na leitura\n');
16    else
17        printf('%s', str);
18
19    fclose(arq);
20    system('pause');
21    return 0;
22 }
```

No exemplo anterior, o comando **fopen()** abre um arquivo de nome “Arq-Grav.txt” no modo de leitura para arquivos texto, “r”. Na sequência, uma string de até 13 caracteres é lida do arquivo e armazenada na variável *str* por meio do comando **fgets(str,13,arq)**, sendo o resultado dessa operação devolvido na variável **result**.



A função **fgets()** lê uma string do arquivo até que um caractere de nova linha (`\n`) seja lido ou “tamanho-1” caracteres tenham sido lidos.

A string resultante de uma operação de leitura usando a função **fgets()** sempre terminará com a constante `'\0'` (por isto somente “tamanho-1” caracteres, no máximo, serão lidos). No caso do de um caractere de nova linha (`\n` ou ENTER) ser lido antes de “tamanho-1” caracteres, ele fará parte da string.



Como a função **gets()**, a função **fgets()**, também pode ser utilizada para ler uma string do dispositivo de entrada padrão (geralmente o teclado).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char nome[30];
5     printf(“ Digite um nome: ”);
6     fgets (nome, 30, stdin);
7     printf(“ O nome digitado foi: %s ”, nome);
8     system(“ pause ”);
9     return 0;
10 }
```

### 11.5.7 ESCRITA E LEITURA DE BLOCOS DE BYTES

Até esse momento, vimos como é possível escrever e ler em arquivos caracteres e sequências de caracteres, as strings. Isso significa que foi possível para nós apenas escrever e ler dados do tipo **char** em um arquivo. Felizmente, a linguagem C possui outras funções que permitem escrever e ler dados mais complexos, como os tipos **int**, **float**, **double**, **array**, ou mesmo um tipo definido pelo programador, como, por exemplo, a **struct**. São as funções de **escrita e leitura de blocos de bytes**.



As funções de escrita e leitura de blocos de bytes devem ser utilizadas preferencialmente com **arquivos binários**.

As funções de escrita e leitura de blocos de bytes trabalham com blocos de memória apontados por um **ponteiro**. Dentro de um bloco de memória,

qualquer tipo de dado pode existir: **int**, **float**, **double**, **array**, **struct**, etc. Dai a sua versatilidade. Além disso, como vamos escrever os dados como estão na memória, isso significa que não existe uma etapa de “conversão” dos dados. Mesmo que gravássemos esses dados em um arquivo texto, seus valores seriam ilegíveis. Dai a preferência por **arquivos binários**.

## ESCREVENDO BLOCOS DE BYTES

Iniciemos pela etapa de gravação. Para escrever em um arquivo um blocos de bytes usa-se a função **fwrite()**, cujo protótipo é:

```
int fwrite(void *buffer, int nro_de_bytes, int count, FILE *fp)
```

A função **fwrite()** recebe 4 parâmetros de entrada

- **buffer**: um ponteiro genérico para a região de memória que contém os dados que serão gravados no arquivo;
- **nro\_de\_bytes**: tamanho, em bytes, de cada unidade de dado a ser gravada;
- **count**: total de unidades de dados que devem ser gravadas.
- **fp**: o ponteiro para o arquivo que se deseja trabalhar;



Note que temos dois valores inteiros: **nro\_de\_bytes** e **count**. Isto significa que o número total de bytes gravados no arquivo será: **nro\_de\_bytes \* count**.

Como resultado, a função **fwrite()** retorna um valor inteiro que representa o número total de unidades de dados gravadas com sucesso. Esse número pode ser menor do que o número de itens esperado (**count**), indicando que houve erro parcial de escrita.



O valor do retorno da função **fwrite()** será igual ao valor de **count** a menos que ocorra algum erro na gravação dos dados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     FILE *arq;
5     arq = fopen( "ArqGrav.txt", "wb" );
6     if (arq == NULL) {
7         printf( "Problemas na CRIACAO do arquivo
8             \n" );
9         system( "pause" );
10        exit(1);
11    }
12    int total_gravado, v[5] = {1,2,3,4,5};
13    //grava todo o array no arquivo (5 posições)
14    total_gravado = fwrite(v, sizeof(int), 5, arq);
15    if (total_gravado != 5) {
16        printf( "Erro na escrita do arquivo!" );
17        system( "pause" );
18        exit(1);
19    }
20    fclose(arq);
21    system( "pause" );
22    return 0;
23 }
```

Note, que a função **sizeof()** foi usada aqui para determinar o tamanho, em bytes, de cada unidade de dado a ser gravada. Trata-se, basicamente do mesmo princípio utilizado na alocação dinâmica, onde alocávamos **N** posições de **sizeof()** bytes de tamanho cada. Nesse caso, como queríamos gravar um array de 5 inteiros, o **nro de bytes** de cada inteiro é obtido pela função **sizeof(int)**, e o total de unidades de dados que devem ser gravadas, **count**, é igual ao tamanho do array, ou seja, 5.

Abaixo, tem-se um exemplo de um programa que utiliza a função **fwrite()** para gravar os mais diversos tipos de dados:

### Exemplo: usando a função fwrite()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main () {
5     FILE *arq;
6     arq = fopen( "ArqGrav.txt" , "wb" );
7     if (arq == NULL) {
8         printf( "Problemas na CRIACAO do arquivo\n" );
9         system( "pause" );
10        exit(1);
11    }
12    char str[20] = "Hello World!";
13    float x = 5;
14    int v[5] = {1,2,3,4,5};
15    //grava a string toda no arquivo
16    fwrite( str , sizeof( char ) , strlen( str ) , arq );
17    //grava apenas os 5 primeiros caracteres da string
18    fwrite( str , sizeof( char ) , 5 , arq );
19    //grava o valor de x no arquivo
20    fwrite( &x , sizeof( float ) , 1 , arq );
21    //grava todo o array no arquivo (5 posições)
22    fwrite( v , sizeof( int ) , 5 , arq );
23    //grava apenas as 2 primeiras posições do array
24    fwrite( v , sizeof( int ) , 2 , arq );
25    fclose( arq );
26    system( "pause" );
27    return 0;
28 }
```

Note, nesse exemplo, que não é necessário gravar sempre o array por inteiro. Podemos gravar parcialmente um array. Para isso, basta modificar o valor do parâmetro **count**. As posições do array serão gravadas a partir da primeira. Então, se o valor de **count** for igual a 2 (**linha 22**), a função **fwrite()** irá gravar no arquivo apenas as 2 primeiras posições do array.



Note que ao gravar uma variável simples (**int**, **float**, **double**, etc.) e compostas (**struct**, etc) é preciso passar o endereço da variável. Para tanto, usa-se o operador **&** na frente do nome da variável. No caso de arrays, seu nome já é o próprio endereço, não sendo, portanto, necessário o operador de **&**.

### LENDO BLOCOS DE BYTES

Uma vez concluída a etapa de gravação de dados com a função **fwrite()**, é necessário agora ler eles do arquivo. Para ler de um arquivo um blocos de bytes usa-se a função **fread()**, cujo protótipo é:

```
int fread(void *buffer, int nro_de_bytes, int count, FILE *fp)
```

A função **fread()** recebe 4 parâmetros de entrada

- **buffer**: um ponteiro genérico para a região de memória que irá armazenar os dados que serão lidos do arquivo;
- **nro\_de\_bytes**: tamanho, em bytes, de cada unidade de dado a ser lida;
- **count**: total de unidades de dados que devem ser lidas.
- **fp**: o ponteiro para o arquivo que se deseja trabalhar;



Note que, como na função **fwrite()**, temos dois valores inteiros: **nro\_de\_bytes** e **count**. Isto significa que o número total de bytes lidos do arquivo será: **nro\_de\_bytes \* count**.

Como resultado, a função **fread()** retorna um valor inteiro que representa o número total de unidades de dados efetivamente lidas com sucesso. Esse número pode ser menor do que o número de itens esperado (**count**), indicando que houve erro parcial de leitura.



O valor do retorno da função **fread()** será igual ao valor de **count** a menos que ocorra algum erro na leitura dos dados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     FILE *arq;
5     arq = fopen( "ArqGrav.txt", "rb" );
6     if (arq == NULL) {
7         printf( "Problemas na ABERTURA do
8             arquivo\n" );
9         system( "pause" );
10        exit(1);
11    }
12    int i, total_lido, v[5];
13    //lê 5 posições inteiras do arquivos
14    total_lido = fread( v, sizeof( int ), 5, arq );
15    if (total_lido != 5) {
16        printf( "Erro na leitura do arquivo!" );
17        system( "pause" );
18        exit(1);
19    } else {
20        for ( i = 0; i < 5; i++)
21            printf( "v[%d] = %d\n", i, v[i] );
22    }
23    fclose( arq );
24    system( "pause" );
25    return 0;
26 }
```

Abaixo, tem-se um exemplo de um programa que utiliza a função **fread()** para ler os mais diversos tipos de dados:

### Exemplo: usando a função fread()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     FILE *arq;
5     arq = fopen( "ArqGrav.txt" , "rb" );
6     if (arq == NULL){
7         printf( "Problemas na ABERTURA do arquivo\n" );
8         system( "pause" );
9         exit(1);
10    }
11    char str1[20], str2[20];
12    float x;
13    int i, v1[5], v2[2];
14    //lê a string toda do arquivo
15    fread(str1, sizeof(char), 12, arq);
16    str1[12] = '\0';
17    printf( "%s\n" , str1 );
18    //lê apenas os 5 primeiros caracteres da string
19    fread(str2, sizeof(char), 5, arq);
20    str2[5] = '\0';
21    printf( "%s\n" , str2 );
22    //lê o valor de x do arquivo
23    fread(&x, sizeof(float), 1, arq);
24    printf( "%f\n" , x );
25    //lê todo o array do arquivo (5 posições)
26    fread(v1, sizeof(int), 5, arq);
27    for ( i = 0; i < 5; i++)
28        printf( "v1[%d] = %d\n" , i, v1[ i ] );
29    fread(v2, sizeof(int), 2, arq);
30    //lê apenas as 2 primeiras posições do array
31    for ( i = 0; i < 2; i++)
32        printf( "v2[%d] = %d\n" , i, v2[ i ] );
33    fclose( arq );
34    system( "pause" );
35    return 0;
36 }
```

Note, nesse exemplo, que após ler o conteúdo de uma string (**linhas 15 e 19**) é necessário atribuir o caractere '\0' para indicar o fim da sequência de caracteres e o início das posições restantes da nossa string que não estão sendo utilizadas nesse momento. Nesse exemplo nós sabíamos qual era o tamanho da string a ser lida. De modo geral, é sempre bom gravar no arquivo, **antes da string**, o seu tamanho. Isso facilita muito a sua leitura posterior.



Ao se trabalhar com strings ou arrays, é sempre bom gravar no arquivo, **antes da string ou array**, o seu tamanho. Isso facilita muito a sua leitura posterior.

O exemplo abaixo mostra como uma string pode ser gravada juntamente com seu tamanho:

#### Exemplo: gravando uma string e seu tamanho

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main () {
5     FILE *arq;
6     arq = fopen( "ArqGrav.txt", "wb" );
7     if (arq == NULL) {
8         printf( "Erro\n" );
9         system( "pause" );
10        exit(1);
11    }
12    char str[20] = "Hello World!";
13    int t = strlen( str );
14    fwrite( &t, sizeof( int ), 1, arq );
15    fwrite( str, sizeof( char ), t, arq );
16    fclose( arq );
17    system( "pause" );
18    return 0;
19 }
```

Já o exemplo seguinte mostra como uma string gravada juntamente com seu tamanho pode ser lida:

### Exemplo: lendo uma string e seu tamanho

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     FILE *arq;
5     arq = fopen( "ArqGrav.txt" , "rb" );
6     if (arq == NULL){
7         printf( "Erro\n" );
8         system( "pause" );
9         exit(1);
10    }
11    char str[20];
12    int t;
13    fread(&t , sizeof(int) , 1 , arq);
14    fread(str , sizeof(char) , t , arq);
15    str[t] = '\0';
16    printf( "%s\n" , str );
17    fclose( arq );
18    system( "pause" );
19    return 0;
20 }
```

## 11.5.8 ESCRITA E LEITURA DE DADOS FORMATADOS

As seções anteriores mostraram como é possível ler e escrever em arquivos caracteres, strings e até mesmo blocos de bytes. Porém, em nenhum momento foi mostrado como podemos escrever uma lista formatada de variáveis em um arquivo como o fazemos na tela do computador. Nem como podemos ler os dados desse mesmo arquivo, especificando a qual o tipo de dado a ser lido (**int**, **float**, **char** ou **double**).



As funções de escrita e leitura de dados formatados permitem ao programador escrever e ler em arquivos da mesma maneira como se escreve na tela e se lê do teclado.

### ESCREVENDO DADOS FORMATADOS

Começemos pela escrita. Para escrever em um arquivo um conjunto de dados formatados usa-se a função **fprintf()**, cujo protótipo é:

```
int fprintf(FILE *fp, "tipos de saída", lista de variáveis)
```

A função **fprintf()** recebe 3 parâmetros de entrada

- **fp**: o ponteiro para o arquivo que se deseja trabalhar;
- “**tipos de saída**”: conjunto de caracteres que especifica o formato dos dados a serem escritos e/ou o texto a ser escrito;
- **lista de variáveis**: conjunto de nomes de variáveis, separados por vírgula, que serão escritos.

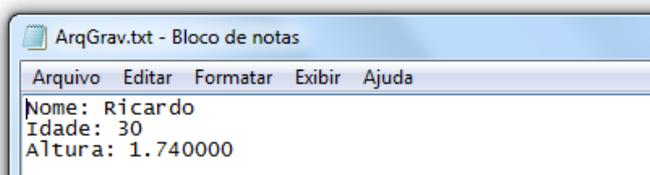
e retorna

- Em caso de sucesso, o número total de caracteres escritos no arquivo é retornado;
- Em caso de erro, um número negativo é retornado.

O exemplo abaixo apresenta um exemplo de uso da função **fprintf()**. Perceba que a função **fprintf()** funciona de maneira semelhante a função **printf()**. A diferença é que, ao invés de escrever na tela, a função **fprintf()** direciona os dados para o arquivo especificado.

### Exemplo: usando a função fprintf()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     FILE *arq;
5     char nome[20] = "Ricardo";
6     int i = 30;
7     float a = 1.74;
8     int result;
9     arq = fopen("ArqGrav.txt","w");
10    if (arq == NULL) {
11        printf("Problemas na ABERTURA do arquivo\n");
12        system("pause");
13        exit(1);
14    }
15    result = fprintf(arq, "Nome: %s\nIdade: %d\nAltura:
16    %f\n", nome, i, a);
17    if (result < 0)
18        printf("Erro na escrita\n");
19    fclose(arq);
20    system("pause");
21    return 0;
22 }
```



### LENDO DADOS FORMATADOS

Uma vez escritos os dados, é necessário agora ler eles do arquivo. Para ler um conjunto de dados formatados de um arquivo usa-se a função **fscanf()**, cujo protótipo é:

```
int fscanf(FILE *fp, "tipos de entrada", lista de variáveis)
```

A função **fscanf()** recebe 3 parâmetros de entrada

- **fp**: o ponteiro para o arquivo que se deseja trabalhar;
- **"tipos de entrada"**: conjunto de caracteres que especifica o formato dos dados a serem lidos;
- **lista de variáveis**: conjunto de nomes de variáveis separados por vírgula, onde cada nome de variável é precedido pelo operador &.

e retorna

- Em caso de sucesso, a função retorna o número de itens lidos com sucesso. Esse número pode ser menor do que o número de itens esperado, indicando que houve erro parcial de leitura.
- a constante **EOF**, indicando que nenhum item foi lido com sucesso.

O exemplo abaixo apresenta um exemplo de uso da função **fscanf()**. Perceba que a função **fscanf()** funciona de maneira semelhante a função **scanf()**. A diferença é que, ao invés de ler os dados do teclado, a função **scanf()** direciona a leitura dos dados para o arquivo especificado.

#### Exemplo: usando a função fscanf()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     FILE *arq;
5     char texto[20], nome[20];
6     int i;
7     float a;
8     int result;
9     arq = fopen("ArqGrav.txt", "r");
10    if (arq == NULL) {
11        printf("Problemas na ABERTURA do arquivo\n");
12        system("pause");
13        exit(1);
14    }
15    fscanf(arq, "%s%s", texto, nome);
16    printf("%s %s\n", texto, nome);
17    fscanf(arq, "%s %d", texto, &i);
18    printf("%s %d\n", texto, i);
19    fscanf(arq, "%s%f", texto, &a);
20    printf("%s %f\n", texto, a);
21
22    fclose(arq);
23    system("pause");
24    return 0;
25 }
```

Note, nesse exemplo, que foi preciso ler, em todos os comando **fscanf()**, o texto que acompanha os dados gravados no arquivo do exemplo do comando **fprintf()**.



A única diferença dos protótipos de **fprintf()** e **fscanf()** para os protótipos de **printf()** e **scanf()**, respectivamente, são a especificação do arquivo destino através do ponteiro **FILE**.

Embora as funções **fprintf()** e **fscanf()** sejam mais fáceis de escrever e ler dados em arquivos, nem sempre elas são as escolhas mais apropriadas. Tome como exemplo a função **fprint()**: os dados são gravados exatamente como seriam impressos na tela e podem ser modificados por um editor de textos simples como o Bloco de Notas. No entanto, para que isso ocorra, os dados são gravados como caracteres de 8 bits utilizando a tabela ASCII. Ou seja, durante a gravação dos dados existe uma etapa de “conversão” dos dados. Essa “conversão” dos dados faz com que os arquivos sejam maiores. Além disso, suas operações de escrita e leitura consomem mais tempo.



Se a intenção do programador é velocidade ou tamanho do arquivo, deve-se utilizar as funções **fwrite()** e **fread()** ao invés de **fprintf()** e **fscanf()**, respectivamente.

O exemplo abaixo mostra como uma matriz pode ser gravada dentro de um arquivo:

#### Exemplo: gravando uma matriz

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     FILE *arq;
5     arq = fopen('matriz.txt', 'w');
6     if (arq == NULL) {
7         printf('Erro\n');
8         system('pause');
9         exit(1);
10    }
11    int mat[2][2] = {{1,2},{3,4}};
12    int i,j;
13    for(i = 0; i < 2; i++)
14        for(j = 0; j < 2; j++)
15            fprintf(arq, "%d\n", mat[i][j]);
16    fclose(arq);
17    system('pause');
18    return 0;
19 }
```

O exemplo abaixo mostra como ler um conjunto de dados de um arquivo e somá-los:

#### Exemplo: lendo uma matriz

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     FILE *arq;
5     arq = fopen( "matriz.txt", "r" );
6     if (arq == NULL) {
7         printf( "Erro\n" );
8         system( "pause" );
9         exit(1);
10    }
11    int i, j, v, soma=0;
12    while (!feof(arq)) {
13        fscanf(arq, "%d", &v);
14        soma += v;
15    }
16    printf( "Soma = %d\n", soma );
17    fclose(arq);
18    system( "pause" );
19    return 0;
20 }
```

## 11.6 MOVENDO-SE DENTRO DO ARQUIVO

De modo geral, o acesso a um arquivo é quase sempre feito de modo seqüencial. Porém, a linguagem C permite realizar operações de leitura e escrita randômica. Para isso, usa-se a função **fseek()**, cujo protótipo é:

```
int fseek(FILE *fp, long numbytes, int origem)
```



Basicamente, a função **fseek()** move a posição atual de leitura ou escrita no arquivo para um byte específico, a partir de um ponto especificado.

A função **fseek()** recebe 3 parâmetros de entrada

- **fp**: o ponteiro para o arquivo que se deseja trabalhar;
- **numbytes**: é o total de bytes a partir de **origem** a ser pulado;

- **origem**: determina a partir de onde os **numbytes** de movimentação serão contados.

A função **fseek()** e retorna um valor inteiro igual a **ZERO** quando a movimentação dentro do arquivo for bem sucedida. Um valor de retorno diferente de zero significa que houve um erro durante a movimentação.

Os valores possíveis para o parâmetro **origem** são definidos por constante na biblioteca **stdio.h** e são:

Constante	Valor	Significado
<b>SEEK_SET</b>	0	Início do arquivo
<b>SEEK_CUR</b>	0	Ponto atual no arquivo
<b>SEEK_END</b>	0	Fim do arquivo

Portanto, para movermos **numbytes** a partir do início do arquivo, a origem deve ser **SEEK\_SET**. Se quisermos mover a partir da posição atual em que estamos no arquivo, devemos usar a constante **SEEK\_CUR**. E, por fim, se quisermos mover a partir do final do arquivo, a constante **SEEK\_END** deverá ser usada.

#### Exemplo: usando a função fseek()

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     FILE *arq;
5     arq = fopen( "ArqGrav.txt", "w" );
6     if (arq == NULL) {
7         printf( "Problemas na CRIAÇÃO do arquivo\n" );
8         system( "pause" );
9         exit(1);
10    }
11    fputs( "1234567890", arq );
12    fseek( arq, 5, SEEK_SET );
13    fputs( " abcde", arq );
14    fclose( arq );
15
16    system( "pause" );
17    return 0;
18 }

```

No exemplo anterior, o primeiro comando **fputs()** (linha 10) é utilizado para escrever uma sequência de 10 dígitos em um arquivo. Em seguida, o

ponteiro do arquivo é movido em 5 posições a partir do seu início (linha 11). Isso significa que os dados escritos pelo segundo comando **fputs()** (linha 12) serão escritos a partir do 6 byte do arquivo, sobrescrevendo o que já havia sido escrito.



O valor do parâmetro **numbytes** pode ser negativo dependendo do tipo de movimentação que formos realizar.

Por exemplo, se quisermos se mover no arquivo a partir do ponto atual (**SEEK\_CUR**) ou do seu final (**SEEK\_END**), um valor negativo de bytes é possível. Nesse caso, estaríamos voltando dentro do arquivo a partir daquele ponto.

## VOLTANDO AO COMEÇO DO ARQUIVO



A linguagem C também permite que se volte para o começo do arquivo. Para tanto, usa-se a função **rewind()**.

Outra opção de movimentação dentro arquivo é simplesmente retornar para o seu início. Para tanto, usa-se a função **rewind()**, cujo protótipo é:

```
void rewind(FILE *fp)
```

A função **rewind()** recebe como parâmetro de entrada apenas o ponteiro para o arquivo que se deseja retornar para o seu início.

## 11.7 EXCLUINDO UM ARQUIVO

Além de permitir manipular arquivos, a linguagem C também permite excluí-los do disco rígido. Isso pode ser feito facilmente utilizando a função **remove()**, cujo protótipo é:

```
int remove(char *nome_do_arquivo)
```

Diferente das funções vistas até aqui, a função **remove()** recebe como parâmetro de entrada o caminho e nome do arquivo a ser excluído do

disco rígido, e não um ponteiro para **FILE**. Como resultado, essa função retorna um valor inteiro igual a **ZERO** quando houver sucesso na exclusão do arquivo. Um valor de retorno diferente de zero significa que houve um erro durante a sua exclusão.



No parâmetro **nome\_do\_arquivo** pode-se trabalhar com caminhos **absolutos** ou **relativos**.

Abaixo, tem-se um exemplo de um programa que utiliza a função **remove()**:

#### Exemplo: usando a função remove()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int status;
5     status = remove( "ArqGrav.txt" );
6     if (status != 0) {
7         printf( "Erro na remocao do arquivo.\n" );
8         system( "pause" );
9         exit(1);
10    } else
11        printf( "Arquivo removido com sucesso.\n" );
12
13    system( "pause" );
14    return 0;
15 }
```

## 11.8 ERRO AO ACESSAR UM ARQUIVO

Ao se trabalhar com arquivos, diversos tipos de erros podem ocorrer: um comando de leitura pode falhar, pode não haver espaço suficiente em disco para gravar o arquivo, etc. Para determinar se uma operação realizada com o arquivo produziu algum erro existe a função **ferror()**, cujo protótipo é:

```
int ferror(FILE *fp)
```

Basicamente, a função **ferror()** recebe como parâmetro o ponteiro **fp** que determina o arquivo que se quer verificar. A função verifica se o indicador de erro associado ao arquivo está marcado, e retorna um valor igual a zero se nenhum erro ocorreu. Do contrário, a função retorna um número diferente de zero.



Como cada operação modifica a condição de erro do arquivo, a função **ferror()** deve ser chamada logo após cada operação realizada com o arquivo.

Uma outra função interessante para se utilizar em conjunto com a função **ferror()** é a função **perror()**. Seu nome vem da expressão em inglês *print error*, ou seja, impressão de erro, e seu protótipo é:

```
void perror(char *str)
```

A função **perror()** recebe como parâmetro uma string que irá preceder a mensagem de erro do sistema. Abaixo é apresentado um exemplo de uso das funções **ferror()** e **perror()**. Nele, um programador tenta acessar um arquivo que não existe. A abertura desse arquivo irá falhar e a seguinte mensagem será apresentada: “**O seguinte erro ocorreu : No such file or directory**”.

#### Exemplo: usando as funções **ferror()** e **perror()**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     FILE *arq;
5     arq = fopen( "NaoExiste.txt", "r" );
6     if (arq == NULL)
7         perror( "O seguinte erro ocorreu" );
8     else
9         fclose(arq);
10
11     system( "pause" );
12     return 0;
13 }
```

## 12.1 DIRETIVAS DE COMPILAÇÃO

As diretivas de compilação são instruções incluídas dentro do código fonte do programa, mas que não são compiladas. Sua função é fazer alterações no código fonte antes de enviá-lo para o compilador. Um exemplo dessas diretivas de compilação é o comando **#define**, que usamos para declarar uma constante na Seção 2.4.1. Basicamente, essa diretiva informa ao compilador que ele deve procurar por todas as ocorrências de uma determinada palavra e substituí-la por outra quando o programa for compilado.

As principais diretivas de compilação são:

lista de diretivas de compilação			
<b>#include</b>	<b>#define</b>	<b>#undef</b>	<b>#ifdef</b>
<b>#ifndef</b>	<b>#if</b>	<b>#endif</b>	<b>#else</b>
<b>#elif</b>	<b>#line</b>	<b>#error</b>	<b>#pragma</b>

Note que todas as diretivas de compilação se iniciam com o caractere **#**. Elas podem ser declaradas em qualquer parte do programa, porém, duas diretivas não podem ser colocadas na mesma linha.

### 12.1.1 O COMANDO #INCLUDE

O comando **#include** já foi visto em detalhes na Seção 1.4.1. Ele é utilizado para declarar as bibliotecas que serão utilizadas pelo programa. Basicamente, esse comando diz ao pré-processador para tratar o conteúdo de um arquivo especificado como se o seu conteúdo houvesse sido digitado no programa no ponto em que o comando **#include** aparece.

### 12.1.2 DEFININDO MACROS: #DEFINE E #UNDEF

Um exemplo dessas diretivas de compilação é o comando **#define**, que usamos para declarar uma constante na Seção 2.4.1. Basicamente, essa diretiva informa ao compilador que ele deve procurar por todas as ocorrências de uma determinada expressão e substituí-la por outra quando o programa for compilado.

O comando **#define** permite três sintaxes:

**#define** nome

**#define** nome\_da\_constante valor\_da\_constante

**#define** nome\_da\_macro(lista\_de\_parâmetros) expressão

## DEFININDO SÍMBOLOS COM #DEFINE

O primeiro uso possível do comando **#define** é simplesmente definir um **nome** que poderá ser testado mais tarde com os comandos de inclusão condicional, como mostra o exemplo abaixo:

Exemplo: inclusão condicional com #define	
Com #define	Sem #define
1 <b>#include</b> <stdio.h>	1 <b>#include</b> <stdio.h>
2 <b>#include</b> <stdlib.h>	2 <b>#include</b> <stdlib.h>
3 <b>#define</b> valor	3
4 <b>int</b> main(){	4 <b>int</b> main(){
5 <b>#ifdef</b> valor	5 <b>#ifdef</b> valor
6 <b>printf</b> (‘‘Valor definido\n’’);	6 <b>printf</b> (‘‘Valor definido\n’’);
7 <b>#else</b>	7 <b>#else</b>
8 <b>printf</b> (‘‘Valor NAO definido\n’’);	8 <b>printf</b> (‘‘Valor NAO definido\n’’);
9 <b>#endif</b>	9 <b>#endif</b>
10 <b>system</b> (‘‘pause’’);	10 <b>system</b> (‘‘pause’’);
11 <b>return</b> 0;	11 <b>return</b> 0;
12 }	12 }

No exemplo anterior, o código da esquerda irá exibir a mensagem “**Valor definido**” pois nós definimos o símbolo **valor** como comando **#define**. Já o código a direita irá exibir a mensagem “**Valor NAO definido**” pois em nenhum momento se definiu quem era o símbolo **valor**.

## DEFININDO CONSTANTES COM #DEFINE

A segunda forma de usar o comando **#define** já foi usada para declarar uma constante na Seção 2.4.1. Basicamente, essa diretiva informa ao compilador que ele deve procurar por todas as ocorrências de uma determinada expressão **nome\_da\_constante** e substituí-la por **valor\_da\_constante** quando o programa for compilado, como mostra o exemplo abaixo:

### Exemplo: constantes com #define

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define PI 3.1415
4 int main() {
5     printf('Valor de PI = %f\n',PI);
6     system('pause');
7     return 0;
8 }
```

O uso das diretivas de compilação **#define** permite declarar uma “constante” que possa ser utilizada como o tamanho dos arrays ao longo do programa, bastando mudar o valor da diretiva para redimensionar todos os arrays em uma nova compilação do programa:

```
#define TAMANHO 100
...
int VET[TAMANHO];
float mat[TAMANHO][TAMANHO];
```

### DEFININDO FUNÇÕES MACROS COM #DEFINE

A terceira e última forma de usar o comando **#define** serve para declarar **funções macros**: uma espécie de declaração de função onde são informados o nome e os parâmetros da função como sendo o nome da macro e o trecho de código equivalente a ser utilizado na substituição. Abaixo tem-se um exemplo:

Exemplo: criando uma macro com #define	
Com macro	Sem macro
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 #define maior(x,y) x&gt;y?x    :y 4 int main(){ 5     int a = 5; 6     int b = 8; 7     int c = maior(a,b); 8     printf('Maior valor =            %d\n',c); 9     system('pause'); 10    return 0; 11 }</pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 4 int main(){ 5     int a = 5; 6     int b = 8; 7     int c = a&gt;b?a:b; 8     printf('Maior valor =            %d\n',c); 9     system('pause'); 10    return 0; 11 }</pre>

No exemplo anterior, o código da esquerda irá substituir a expressão **maior(a,b)** pela macro **x>y?x:y**, trocando o valor de **x** por **a** e o valor de **y** por **b**, ou seja, **a>b?a:b**.



É aconselhável sempre colocar, na sequência de substituição, os parâmetros da macro entre parênteses. Isso serve para preservar a precedência dos operadores.

Considere o exemplo abaixo

Exemplo: macros com parênteses
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 #define prod1(x,y) x*y; 4 #define prod2(x,y) (x)*(y); 5 int main(){ 6     int a = 1, b = 2; 7     int c = prod1(a+2,b); 8     int d = prod2(a+2,b); 9     printf('Valor de c = %d\n',c); 10    printf('Valor de d = %d\n',d); 11    system('pause'); 12    return 0; 13 }</pre>

Quando as macros forem substituídas, as variáveis **c** e **d** serão preenchidas com os seguintes valores:

```
int a = 1, b = 2;
int c = a + 2 * b;
int d = (a+2) * (b);
```

Nesse exemplo, teremos que a variável **c = 5**, enquanto **d = 6**. Isso acontece por que **uma macro não é uma função**, e sim uma substituição de sequências de comandos. O valor de **a+2** não é calculado antes de ser chamada a macro, mas sim colocado no lugar do parâmetro **x**. Como a macro **prod1** não possui parênteses nos parâmetros, a multiplicação será executada antes da operação de soma. Já na macro **prod2**, os parênteses garantem que a soma seja feita antes da operação de multiplicação.



Dependendo da macro criada, pode ser necessário colocar a **expressão** entre chaves **{}**.

As macros permitem criar funções que podem ser utilizadas para qualquer tipo de dado. Isso é possível pois a macro permite que identifiquemos como um dos seus parâmetros o tipo das variáveis utilizadas. Se porventura tivermos que declarar uma variável para esse tipo dentro da expressão que substituirá a macro, o uso de chaves **{}** será necessário como mostra o exemplo abaixo:

#### Exemplo: criando uma macro com #define e chaves {}

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #define TROCA(a,b,c) {c t=a; a=b; b=t;}
4 int main(){
5     int x=10;
6     int y=20;
7     printf("%d %d\n", x, y);
8     TROCA(x, y, int);
9     printf("%d %d\n", x, y);
10    system('pause');
11    return 0;
12 }
```

No exemplo anterior, foi criada uma macro que troca os valores de duas variáveis de lugar. Para realizar essa tarefa, é necessário declarar uma terceira variável que dará suporte a essa operação. Essa é a variável **t** da macro, a qual é do tipo **c**. Para que não ocorram conflitos de nomes

de variáveis, essa variável **t** deve ser criada em um novo escopo, o qual é definido pelo par de `{}`. Desse modo, a variável **t** será criada para o tipo **c** (que será substituído por **int**) apenas para aquele escopo da macro, sendo destruída na sequência.

## FUNÇÕES MACRO COM MAIS DE UMA LINHA



De modo geral, uma função macro deve ser escrita toda em uma única linha. Porém, pode-se escrevê-la usando mais de uma linha adicionando uma barra `\` ao final de cada linha da macro.

Desse modo, a macro anteriormente criada

```
#define TROCA(a,b,c) {c t=a; a=b; b=t;}
```

pode ser reescrita como

```
#define TROCA(a,b,c) {c t=a; \  
a=b; \  
b=t;}
```

## OPERADORES ESPECIAIS: # E ##



Definições de funções macro aceitam dois operadores especiais (**#** e **##**) na sequência de substituição: **#** permite transformar um texto em **string**, enquanto o segundo concatena duas expressões.

Se o operador **#** é colocado antes de um parâmetro na sequência de substituição, isso significa que o parâmetro deverá ser interpretado como se o mesmo estivesse entre “aspas duplas”, ou seja, será considerado como uma **string** pelo compilador. Já o operador **##**, quando colocado entre dois parâmetro na sequência de substituição, fará com que os dois parâmetros da macro sejam concatenados ignorando os espaços em branco entre eles) e interpretados como um comando só. Veja os exemplos abaixo:

Exemplo: usando os operadores especiais # e ##	
operador #	operador ##
<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 #define str(x) #x 4 int main() { 5     printf(str( Teste!\n)); 6     system( ' 'pause' '); 7     return 0; 8 }</pre>	<pre> 1 #include &lt;stdio.h&gt; 2 #include &lt;stdlib.h&gt; 3 #define concatena(x,y) x    ## y 4 int main() { 5     concatena( print , f) ( ' '    Teste!\n' '); 6     system( ' 'pause' '); 7     return 0; 8 }</pre>

No exemplo anterior, o código da esquerda irá substituir a expressão **str( Teste!\n)** pela string **"Teste!\n"**. Já o código da direita irá substituir a expressão **concatena(print,f)** pela concatenação dos parâmetros, ou seja, o comando **printf**.

#### APAGANDO UMA DEFINIÇÃO: #UNDEF

Por fim, temos a diretiva **#undef**, que possui a seguinte forma geral:

**#undef** nome\_da\_macro

Basicamente, essa diretiva é utilizada sempre que desejarmos apagar a definição da macro **nome\_da\_macro** da tabela interna que guarda as macros. Em outras palavras, remove a definição de uma macro para que ela possa ser redefinida.



Enquanto a diretiva **#define** cria a definição de uma macro, a diretiva **#undef** remove a definição da macro para que ela não seja mais usada ou para que possa ser redefinida.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #define valor 10
4 int main() {
5     printf( ' 'Valor = %d\n' ', valor);
6     #undef valor
7     #define valor 20
8     printf( ' 'Novo valor = %d\n' ', valor);
9     system( ' 'pause' ');
10    return 0;
11 }
```

### 12.1.3 DIRETIVAS DE INCLUSÃO CONDICIONAL

O pré-processador da linguagem C também possui estruturas condicionais: são as diretivas de inclusão condicional. Elas permitem incluir ou descartar parte do código de um programa sempre que uma determinada condição é satisfeita.

#### DIRETIVAS #IFDEF E #IFNDEF

Começemos pelas diretivas **#ifdef** e **#ifndef**. Essas diretivas permitem verificar se uma determinada macro foi previamente definida (**#ifdef**) ou não (**#ifndef**). A sua forma geral é:

```
#ifdef nome_do_símbolo
    código
#endif
e
#ifndef nome_do_símbolo
    código
#endif
```

Abaixo é possível ver um exemplo para as diretivas **#ifdef** e **#ifndef**

Exemplo: usando as diretivas <b>#ifdef</b> e <b>#ifndef</b>	
com <b>#ifdef</b>	com <b>#ifndef</b>
1 <b>#include</b> <stdio.h>	1 <b>#include</b> <stdio.h>
2 <b>#include</b> <stdlib.h>	2 <b>#include</b> <stdlib.h>
3 <b>#define</b> TAMANHO 100	3
4 <b>int</b> main(){	4 <b>int</b> main(){
5 <b>#ifdef</b> TAMANHO	5 <b>#ifndef</b> TAMANHO
6	6 <b>#define</b> TAMANHO 100
7 <b>int</b> vetor[TAMANHO];	7 <b>int</b> vetor[TAMANHO];
8 <b>#endif</b>	8 <b>#endif</b>
9 system(' 'pause' ');	9 system(' 'pause' ');
10 <b>return</b> 0;	10 <b>return</b> 0;
11 }	11 }

No exemplo anterior, o código da esquerda irá verificar com a diretiva **#ifdef** se a macro **TAMANHO** foi definida. Como ela foi, o programa irá criar um array de inteiros com **TAMANHO** elementos. Já o código da direita não

possui a macro **TAMANHO** definida. Por isso usamos a diretiva **#ifndef** para verificar se a macro **TAMANHO** NÃO foi definida. Como ela NÃO foi, o programa irá executar a diretiva **#define** para definir a macro **TAMANHO** para somente em seguida criar um array de inteiros com **TAMANHO** elementos.



A diretiva **#endif** serve para indicar o fim de uma diretiva de inclusão condicional do tipo **#ifdef**, **#ifndef** e **#if**.

### DIRETIVAS #IF, #ELSE E #ELIF

As diretivas **#if**, **#else** e **#elif** são utilizadas para especificar algumas condições a serem cumpridas para que uma determinada parte do código seja compilado. As diretivas **#if** e **#else** são equivalentes aos comandos condicionais **if** e **else**. A forma geral dessas diretivas é

```
#if condição
    sequência de comandos
#else
    sequência de comandos
#endif
```



A diretiva **#else** é opcional quando usamos a diretiva **#if**. Exatamente como o comando **else** é opcional no uso do comando **if**.

Já a diretiva **#elif** serve para criar um aninhamento de diretivas **#if**. Ela é utilizada sempre que desejamos usar novamente a diretiva **#if** dentro de uma diretiva **#else**. A forma geral dessa diretiva é

```
#if condição1
    sequência de comandos
#elif condição2
    sequência de comandos
#else
    sequência de comandos
#endif
```



Como no caso da diretiva **#else**, a diretiva **#elif** também é opcional quando usamos a diretiva **#if**. Como a diretiva **#elif** testa um nova condição, ela também pode ser seguida pela diretiva **#else** ou outra **#elif**, ambas opcionais.

Abaixo é possível ver um exemplo do uso das diretivas **#if**, **#else** e **#elif**:

#### Exemplo: usando as diretivas **#if**, **#else** e **#elif**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define TAMANHO 55
4
5 #if TAMANHO > 100
6 #undef TAMANHO
7 #define TAMANHO 100
8 #elif TAMANHO < 50
9 #undef TAMANHO
10 #define TAMANHO 50
11 #else
12 #undef TAMANHO
13 #define TAMANHO 75
14 #endif
15 int main() {
16     printf( "Valor de TAMANHO = %d\n", TAMANHO );
17     system( "pause" );
18     return 0;
19 }
```

Para entender o exemplo anterior, imagine que a diretiva **#define** possa ser reescrita atribuindo diferentes valores para a macro **TAMANHO**. Se **TAMANHO** for maior do que 100, a diretiva **#if** será executada e um novo valor para **TAMANHO** será definido (100). Caso contrário, a condição da diretiva **#elif** será testada. Nesse caso, se **TAMANHO** for menor do que 50, a sequência de comandos da diretiva **#elif** será executada e um novo valor para **TAMANHO** será definido (50). Se a condição da diretiva **#elif** também for falsa, a sequência de comandos da diretiva **#else** será executada e um novo valor para **TAMANHO** será definido (75).



As diretivas **#if** e **#elif** só podem ser utilizadas para avaliar **expressões constantes**.

Como o código ainda não foi compilado, as diretivas **#if** e **#elif** não irão

resolver expressões matemáticas dentro da condição. Irão apenas fazer comparações de valores já definidos, ou seja, constantes.

#### 12.1.4 CONTROLE DE LINHA: #LINE

Sempre que ocorre um erro durante a compilação de um programa, o compilador mostra a mensagem relativa ao erro que ocorreu. Além dessa mensagem, o compilador também exibe o nome do arquivo onde o erro ocorreu e em qual linha desse arquivo. Isso facilita a busca de onde o erro se encontra no nosso programa.

A diretiva **#line**, cuja forma geral é

```
#line numero_da_linha nome_do_arquivo
```

permite controlar o número da linha (**numero\_da\_linha**) e o nome do arquivo (**nome\_do\_arquivo**) onde o erro ocorreu. O parâmetro **nome\_do\_arquivo** é opcional e se não for definido o compilador usará o próprio nome do arquivo. Veja o exemplo abaixo:

##### Exemplo: diretiva #line

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5   #line 5 'Erro de atribuicao'
6   float a=;
7   printf('Valor de a = %f\n',a);
8   printf('PI = %f\n',PI);
9   system('pause');
10  return 0;
11 }
```

Nesse exemplo, declaramos a diretiva **#line** logo acima a declaração de uma variável. Note que existe um erro de atribuição na variável *a* (linha 6). Durante o processo de compilação, o compilador irá acusar um erro, porém, ao invés de afirmar que o erro se encontra na linha 6, ele irá informar que o erro se encontra na linha 5. Além disso, ao invés de exibir o nome do arquivo onde o erro ocorreu, o compilador irá exibir a mensagem “**Erro de atribuicao**”.

### 12.1.5 DIRETIVA DE ERRO: #ERROR

A diretiva **#error** segue a seguinte forma geral:

**#error** texto

Basicamente, essa diretiva aborta o processo de compilação do programa sempre que ela for encontrada. Como resultado, ela gera a mensagem de erro especificada pelo parâmetro **texto**. Veja o exemplo abaixo:

#### Exemplo: diretiva #error

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #ifndef PI
5 #error O valor de PI nao foi definido
6 #endif
7
8 int main(){
9     printf(‘‘PI = %f\n’’,PI);
10    system(‘‘pause’’);
11    return 0;
12 }
```

Nesse exemplo, em nenhum momento a macro **PI** foi definida. Portanto o processo de compilação será abortado devido a falta da macro **PI** (linhas 4 a 6), e a mensagem de erro “**O valor de PI nao foi definido**” será exibida para o programador.

### 12.1.6 DIRETIVA #PRAGMA

A diretiva **#pragma** é comumente utilizada para especificar diversas opções do compilador.



A diretiva **#pragma** é específica do compilador. Se um argumento utilizado em conjunto com essa diretiva não for suportado pelo compilador, a diretiva será ignorada e nenhum erro será gerado.

Para poder utilizar de modo adequado e saber os possíveis parâmetros que você pode definir com a diretiva **#pragma**, consulte o manual de referência do seu compilador.

### 12.1.7 DIRETIVAS PRÉ-DEFINIDAS

A linguagem C possui algumas macros já pré-definidas, são elas:

- **\_\_LINE\_\_** : retorna um valor inteiro que representa a linha onde a macro foi chamada no arquivo de código fonte a ser compilado;
- **\_\_FILE\_\_** : retorna uma string contendo o caminho e nome do arquivo fonte a ser compilado.
- **\_\_DATE\_\_** : retorna uma string contendo a data de compilação do arquivo de código fonte no formato “Mmm dd yyyy”;
- **\_\_TIME\_\_** : retorna uma string contendo a hora de compilação do arquivo de código fonte no formato “hh:mm:ss”.

## 12.2 TRABALHANDO COM PONTEIROS

### 12.2.1 ARRAY DE PONTEIROS E PONTEIRO PARA ARRAY

Vimos na Seção 9.4 que ponteiros e arrays possuem uma ligação muito forte dentro da linguagem C. Arrays são agrupamentos **sequenciais** de dados do mesmo tipo na memória. O seu nome é apenas um ponteiro que aponta para o começo dessa sequência de *bytes* na memória.



O **nome** do array é apenas um ponteiro que aponta para o primeiro elemento do array.

De fato, podemos atribuir o endereço de um array para um ponteiro facilmente. A única restrição para essa operação é que o tipo do ponteiro seja o mesmo do array. E isso pode ser feito de duas formas distintas:

```
int vet[5] = { 1, 2, 3, 4, 5 }  
int *p1 = vet;  
int *p2 = &vet[0];
```

A linguagem C também permite o uso de arrays e ponteiros de forma conjunta na declaração de variáveis. Considere as seguintes declarações abaixo:

```
typedef vetor int[10];  
vetor p1;  
vetor *p2;  
int (*p3)[10];  
int *p4[10];
```

No exemplo acima, o comando **typedef** é usado para criar um sinônimo (**vetor**) para o tipo “array de 10 inteiros” (**int** [10]). Assim, a variável **p1**, que é do tipo **vetor**, é um “array de 10 inteiros”. Já a variável **p2** é um ponteiro para o tipo “array de 10 inteiros”.

Temos também a declaração da variável **p3**. Note que (**\*p3**) está dentro de um parênteses. Isso significa que estamos colocando ênfase na declaração do ponteiro. Na sequência existe também a definição do tamanho de um array. Como apenas o ponteiro está dentro de parênteses, isso significa para o compilador que estamos declarando um **ponteiro para um “array de 10 inteiros”**. Isso significa que a declaração das variáveis **p2** e **p3** são equivalentes.

Por fim, temos a declaração da variável **p4**. Apesar de semelhante a declaração de **p3**, note que não existem parênteses colocando ênfase na declaração do ponteiro. Isso significa para o compilador que estamos declarando um **array de 10 “ponteiros para inteiros”**.



Cuidado ao misturar ponteiros e arrays numa mesma declaração. Nas declarações **int (\*p3)[10];** e **int \*p4[10];**, **p3** é um ponteiro para um “array de 10 inteiros” enquanto **p4** é um array de 10 “ponteiros para inteiros”.

## 12.2.2 PONTEIRO PARA FUNÇÃO

Vimos nas seções anteriores, que as variáveis são espaços reservados da memória utilizados para guardar nossos dados. Já um programa é, na verdade, um conjunto de instruções armazenadas na memória, juntamente com seus dados. Vimos também que uma função nada mais é do que um bloco de código (ou seja, declarações e outros comandos) que podem ser nomeado e chamado de dentro de um programa.



Uma função também é um conjunto de instruções armazenadas na memória. Portanto, podemos acessar uma função por meio de um ponteiro que aponte para onde a função está na memória.

A principal vantagem de se declarar um ponteiro para uma função é a construção de códigos genéricos. Pense na ordenação de números: podemos definir um algoritmo que ordene números inteiros e querer reutilizar essa implementação para ordenar outros tipos de dados (por exemplo, strings). Ao invés de reescrever toda a função de ordenação, nós podemos passar para esta função o ponteiro da função de comparação que desejamos utilizar para cada tipo de dado.



Ponteiros permitem fazer uma chamada indireta à função e passá-la como parâmetro para outras funções. Isso é muito útil na implementação de algoritmos genéricos em C.

## DECLARANDO UM PONTEIRO PARA FUNÇÃO

Em linguagem C, a declaração de um ponteiro para uma função segue a seguinte forma geral:

```
tipo_retornado (*nome_do_ponteiro)(lista_de_tipos);
```



O **nome\_do\_ponteiro** deve sempre ser colocado entre parênteses juntamente com o **asterisco**: **(\*nome\_do\_ponteiro)**.

Isso é necessário para evitar confusões com a declaração de funções que retornem ponteiros. Por exemplo,

```
tipo_retornado *nome_da_função(lista_de_parâmetros);
```

é uma função que retorna um ponteiro do **tipo\_retornado**, enquanto

```
tipo_retornado (*nome_do_ponteiro)(lista_de_tipos);
```

é um ponteiro para funções que retornam **tipo\_retornado**.



Um ponteiro para funções só pode apontar para funções que possuam o mesmo protótipo.

Temos agora que **nome\_do\_ponteiro** é um ponteiro para funções. Mas não para qualquer função. Apenas para funções que possuam o mesmo protótipo definido para o ponteiro. Assim, se declararmos um ponteiro para funções como sendo

```
int (*ptr)(int, int);
```

ele poderá ser apontado para qualquer função que receba dois parâmetros inteiros (independente de seus nomes) e retorne um valor inteiro:

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

#### APONTANDO UM PONTEIRO PARA UMA FUNÇÃO



Ponteiros não inicializados apontam para um lugar indefinido.

Como qualquer outro ponteiro, quando um ponteiro de função é declarado ele não possui um endereço associado. Qualquer tentativa de uso desse ponteiro causa um comportamento indefinido no programa. O ponteiro para função também pode ser inicializado com a constante **NULL**, o que indica que aquele ponteiro aponta para uma posição de memória inexistente, ou seja, nenhuma função.



O **nome** de uma função é seu **endereço na memória**. Basta atribuí-lo ao ponteiro para que o ponteiro aponte para a função na memória. O operador de & não é necessário.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int max(int a, int b){
4     return (a > b) ? a : b;
5 }
6 int main(){
7     int x,y,z;
8     int (*p)(int, int);
9     printf("Digite 2 numeros: ");
10    scanf("%d %d",&x,&y);
11    //ponteiro recebe endereço da função
12    p = max;
13    z = p(x,y);
14    printf("Maior = %d\n",z);
15    system("pause");
16    return 0;
17 }
```

Lembre-se, quando criamos uma função, o computador a guarda em um espaço reservado de memória. Ao nome que damos a essa função o computador associa o endereço do espaço que ele reservou na memória para guardar essa função. Assim, basta atribuir o nome da função ao ponteiro para que o ponteiro passe a apontar para a função na memória (**linha 12 do exemplo anterior**).



Para usar a função apontada por um ponteiro, basta utilizar o nome do ponteiro como se ele fosse o nome da função.

Pode-se ver um exemplo de uso da função apontada na **linha 13 do exemplo anterior**. Nele, utilizamos o ponteiro **p** como se ele fosse um outro nome, ou um sinônimo, para a função **max()**. Abaixo é possível ver outro exemplo de uso de ponteiros para funções:

### Exemplo: ponteiro para função

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int soma(int a, int b){return a + b;}
4 int subtracao(int a, int b){return a - b;}
5 int produto(int a, int b){return a * b;}
6 int divisao(int a, int b){return a / b;}
7 int main(){
8     int x,y;
9     int (*p)(int ,int);
10    char ch;
11    printf("Digite uma operação matematica
12    (+, -, *, /): ");
13    ch = getchar();
14    printf("Digite 2 numeros: ");
15    scanf("%d %d",&x,&y);
16    switch(ch){
17        case '+': p = soma; break;
18        case '-': p = subtracao; break;
19        case '*': p = produto; break;
20        case '/': p = divisao; break;
21        default: p = NULL;
22    }
23    if(p!=NULL)
24        printf("Resultado = %d\n",p(x,y));
25    else
26        printf("Operacao invalida\n");
27    system("pause");
28    return 0;
29 }
```

### PASSANDO UM PONTEIRO PARA FUNÇÃO COMO PARÂMETRO

Como dito anteriormente, a principal vantagem de se declarar um ponteiro para uma função é que eles permitem a construção de códigos genéricos. Isso ocorre porque esses ponteiros permitem fazer uma chamada indireta à função, de modo que eles podem ser passados como parâmetro para outras funções. Vamos lembrar de como é a declaração de um ponteiro para uma função. A sua forma geral é

tipo\_retornado (\*nome\_do\_ponteiro)(lista\_de\_tipos);

Agora, se quisermos declarar uma função que possa receber um ponteiro para função como parâmetro, tudo o que devemos fazer é incorporar a declaração de um ponteiro para uma função dentro da declaração dos parâmetros da função. Considere o seguinte ponteiro para função:

```
int (*ptr)(int, int);
```

Se quisermos passar esse ponteiro para uma outra função, devemos declarar esse ponteiro na sua lista de parâmetros:

```
int executa(int (*ptr)(int, int), int x, int y);
```

Temos agora que a função **executa()** recebe três parâmetros:

- **ptr**: um ponteiro para uma função que receba dois parâmetros inteiros e retorne um valor inteiro;
- **x**: um valor inteiro;
- **y**: outro valor inteiro.

Abaixo podemos ver um exemplo de uso dessa função:

### Exemplo: Passando um ponteiro para função como parâmetro

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int soma(int a, int b){return a + b;}
4 int subtracao(int a, int b){return a - b;}
5 int produto(int a, int b){return a * b;}
6 int divisao(int a, int b){return a / b;}
7 int executa(int (*p)(int,int), int x, int y) {
    return p(x,y)}
8 int main(){
9     int x,y;
10    int (*p)(int,int);
11    char ch;
12    printf("Digite uma operação matematica
13          (+, -, *, /): ");
14    ch = getchar();
15    printf("Digite 2 numeros: ");
16    scanf("%d %d",&x,&y);
17    switch(ch){
18        case '+': p = soma; break;
19        case '-': p = subtracao; break;
20        case '*': p = produto; break;
21        case '/': p = divisao; break;
22        default: p = NULL;
23    }
24    if(p!=NULL)
25        printf("Resultado = %d\n",executa(p,x,y));
26    else
27        printf("Operacao invalida\n");
28    system("pause");
29    return 0;
30 }
```

### CRIANDO UM ARRAY DE PONTEIROS PARA FUNÇÃO

Vamos relembrar a declaração de arrays. Para declarar uma variável, a forma geral era

tipo nome;

Já para declarar um array, basta indicar, entre colchetes, o tamanho do array que queremos criar:

tipo nome[tamanho];



Para declarar um array de ponteiros para funções, o princípio é o mesmo usado na declaração de arrays dos tipos básicos: basta indicar na declaração o seu tamanho entre colchetes para transformar essa declaração na declaração de um array.

A declaração de arrays de ponteiros para funções funciona exatamente da mesma maneira que a declaração para outros tipos, ou seja, basta indicar na declaração do ponteiro para função o seu tamanho entre colchetes para criar um array:

```
//ponteiro para função  
tipo_retornado (*nome_do_ponteiro)(lista_de_tipos);  
//arrays de ponteiros para função com tamanho elementos tipo_retornado  
(*nome_do_ponteiro[tamanho])(lista_de_tipos);
```

Feito isso, cada posição do array pode agora apontar para uma função diferente, como mostra o exemplo abaixo:

### Exemplo: array de ponteiro para função

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int soma(int a, int b){return a + b;}
4 int subtracao(int a, int b){return a - b;}
5 int produto(int a, int b){return a * b;}
6 int divisao(int a, int b){return a / b;}
7 int main(){
8     int x,y,indice = -1;
9     int (*p[4])(int, int);
10    p[0] = soma;
11    p[1] = subtracao;
12    p[2] = produto;
13    p[3] = divisao;
14    char ch;
15    printf("Digite uma operação matematica
16          (+, -, *, /): ");
17    ch = getchar();
18    printf("Digite 2 numeros: ");
19    scanf("%d %d",&x,&y);
20    switch(ch){
21        case '+': indice = 0; break;
22        case '-': indice = 1; break;
23        case '*': indice = 2; break;
24        case '/': indice = 3; break;
25        default: indice = -1;
26    }
27    if(indice >= 0)
28        printf("Resultado = %d\n",p[indice](x,y));
29    else
30        printf("Operacao invalida\n");
31    system("pause");
32    return 0;
}
```

## 12.3 ARGUMENTOS NA LINHA DE COMANDO

Ao longo dos vários exemplos de programas criados, foi visto que a cláusula **main** indicava a função principal do programa. Ela era responsável pelo início da execução do programa, e era dentro dela que colocávamos os comandos que queríamos que o programa executasse. Sua forma geral era a seguinte:

```
int main() {
    sequência de comandos
```

```
}
```

Quando aprendemos a criar nossas próprias funções, vimos que era possível passar uma lista de parâmetros para a função criada sempre que ela fosse executada. Porém, a função **main** sempre foi utilizada sem parâmetros.



A cláusula **main** também é uma função. Portanto ela também pode receber uma lista de parâmetros no início da execução do programa.

A função **main** pode ser definida de tal maneira que o programa receba parâmetros que foram dados na linha de comando do sistema operacional. Para receber esses parâmetros, a função **main** adquire a seguinte forma:

```
int main(int argc, char *argv[]) {  
    sequência de comandos  
}
```

Note que agora a função **main** recebe dois parâmetros de entrada, são eles:

- **int argc**: trata-se de um valor inteiro que indica o número de parâmetros com os quais a função **main** foi chamada na linha de comando;



O valor de **argc** é sempre maior ou igual a 1. Esse parâmetro vale 1 se o programa foi chamado sem nenhum parâmetro (o nome do programa é contado como argumento da função), sendo somado +1 em **argc** para cada parâmetro passado para o programa.

- **char \*argv[]**: trata-se de um ponteiro para uma matriz de strings. Cada uma das string contidas nesta matriz é um dos parâmetros com os quais a função **main** foi chamada na linha de comando. Ao todo, existem **argc** strings guardadas em **argv**.



A string guardada em **argv[0]** sempre aponta para o nome do programa (lembre-se, o nome do programa é contado como argumento da função).

O exemplo abaixo apresenta um programa que recebe parâmetros da linha de comando:

#### Exemplo: parâmetros da linha de comando

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char* argv[]) {
4     if (argc == 1){
5         printf('Nenhum parametro passado para o programa
6             %s\n',argv[0]);
7     }else{
8         int i;
9         printf('Parametros passados para o programa %s:\n',argv[0]);
10        for (i=1; i<argc; i++)
11            printf('Parametro %d: %s\n',i,argv[i]);
12    }
13    system('pause');
14    return 0;
}
```

Para testar o exemplo acima, copie o programa e salve em uma pasta qualquer (por exemplo, C:\). Vamos considerar que o programa foi salvo com o nome “**prog.c**”. Gere o executável do programa (“**prog.exe**”). Agora abra o console (se estiver no Windows: iniciar, executar, cmd), vá para o diretório onde o programa foi salvo (C:\), e digite: **prog.exe**. Ao apertarmos a tecla **enter**, a seguinte mensagem irá aparecer:

**Nenhum parâmetro passado para o programa prog.exe**

Se, ao invés de digitar **prog.exe**, nós digitássemos **prog.exe par1 par2**, a mensagem impressa seria:

**Parametros passados para o programa prog.exe**

**Parametro 1: par1**

**Parametro 2: par2**

Abaixo, tem-se outro exemplo de programa que recebe parâmetros da linha de comando. No caso, esse programa recebe como parâmetros dois valores inteiros e os soma. Para isso, fazemos uso da função **atoi**, a qual converte uma string para o seu valor inteiro:

### Exemplo: soma dos parâmetros da linha de comando

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char* argv[]){
4     if (argc == 1){
5         printf('Nenhum parametro para ser somado\n');
6     }else{
7         int soma = 0, i;
8         printf('Somando os parametros passados para o
9             programa %s:\n',argv[0]);
10        for (i=1; i<argc; i++)
11            soma = soma + atoi(argv[i]);
12        printf('Soma = %d\n',soma);
13    }
14    system('pause');
15    return 0;
16 }
```

## 12.4 RECURSOS AVANÇADOS DA FUNÇÃO PRINTF()

Vimos na Seção 2.2.1 que a função **printf()** é uma das funções de saída/escrita de dados da linguagem C. Sua funcionalidade básica é escrever na saída de vídeo (tela) um conjunto de valores, caracteres e/ou sequência de caracteres de acordo com o formato especificado. Porém, essa função permite uma variedade muito maior de formatações do que as vistas até agora. Começemos pela sua definição. A forma geral da função **printf()** é:

```
int printf("tipos de saída", lista de variáveis)
```

A função **printf()** recebe 2 parâmetros de entrada

- **"tipos de saída"**: conjunto de caracteres que especifica o formato dos dados a serem escritos e/ou o texto a ser escrito;
- **lista de variáveis**: conjunto de nomes de variáveis, separados por vírgula, que serão escritos.

Note também que a função **printf()** retorna um valor inteiro, ignorado até o presente momento. Esse valor de retorno será

- o número total de caracteres escritos na tela, em caso de sucesso;
- um valor negativo, em caso de erro da função.



O valor de retorno da função **printf()** permite identificar o funcionamento adequado da função.

Vimos também que quando queremos escrever dados formatados na tela nós usamos os **tipos de saída** para especificar o formato de saída dos dados que serão escritos. E que cada tipo de saída é precedido por um sinal de % e um **tipo de saída** deve ser especificado para cada variável a ser escrita.



A string do **tipo de saída** permite especificar mais características dos dados além do formato. Essas características são **opcionais** e são: **flag**, **largura**, **precisão** e **comprimento**.

A ordem em que essas quatro características devem ser especificadas é a seguinte:

**%[flag][largura][.precisão][comprimento]tipo de saída**



Note que o campo **precisão** vem sempre começando com um caractere de **ponto** (.).

Como o **tipo de saída**, cada uma dessas características possui um conjunto de valores pré-definidos e suportados pela linguagem. Nas seções seguintes são apresentados todos os valores suportados para cada uma das características de formatação possíveis.

#### 12.4.1 OS TIPOS DE SAÍDA

A função **printf()** pode ser usada para escrever virtualmente qualquer tipo de dado. A tabela abaixo mostra todos os tipos de saída suportados pela linguagem C:

“tipo de saída”	Descrição
%c	Escrita de um caractere
%d ou %i	Escrita de números inteiros com sinal ( <b>signed</b> )
%u	Escrita de números inteiros sem sinal ( <b>unsigned</b> )
%f	Escrita de números reais ( <b>float</b> e <b>double</b> )
%s	Escrita de vários caracteres ( <b>string</b> )
%p	Escrita de um endereço de memória ( <b>ponteiro</b> )
%e ou %E	Escrita de número reais ( <b>float</b> e <b>double</b> ) em notação científica (usando caractere “e” ou “E”)
%x	Escrita de números inteiros sem sinal ( <b>unsigned</b> ) no formato hexadecimal (minúsculo)
%X	Escrita de números inteiros sem sinal ( <b>unsigned</b> ) no formato hexadecimal (Maiúsculo)
%o	Escrita de números inteiros sem sinal ( <b>unsigned</b> ) no formato octal ( <b>base 8</b> )
%g	Escrita de número reais ( <b>float</b> e <b>double</b> ). Compilador decide se é melhor usar %f ou %e
%G	Escrita de número reais ( <b>float</b> e <b>double</b> ). Compilador decide se é melhor usar %f ou %E
%%	Escrita do caractere %

A seguir, são apresentados alguns exemplos de como cada tipo de saída pode ser utilizado para escrever determinado dado na tela.

#### EXIBINDO OS TIPOS BÁSICOS

A linguagem C possui vários tipos de saída que podem ser utilizados com os tipos básicos, ou seja, **char** (“%c” e “%d”), **int** (“%d” e “%i”), **float** e **double** (“%f”), e por fim array de **char** ou **string** (“%s”). Note que o tipo **char** pode ser escrito na tela de saída por meio dos operadores “%c” e “%d”. Nesse caso, “%c” irá imprimir o caractere armazenado na variável, enquanto “%d” irá imprimir o seu valor na tabela ASCII. Abaixo, tem-se alguns exemplos de escrita dos tipos básicos:

### Exemplo: usando printf() para imprimir os tipos básicos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n = 125;
5     float f = 5.25;
6     double d = 10.53;
7     char letra = 'A';
8     char palavra[10] = "programa";
9     printf("Valor inteiro: %d\n",n);
10    printf("Valor inteiro: %i\n",n);
11    printf("Valor real: %f\n",f);
12    printf("Valor real: %f\n",d);
13    printf("Caractere: %c\n",letra);
14    printf("Valor numerico do caractere: %d\n",letra);
15    printf("Palavra: %s\n",palavra);
16    system("pause");
17    return 0;
18 }
```

### EXIBINDO VALORES NO FORMATO OCTAL OU HEXADECIMAL

O exemplo abaixo mostra como exibir um **valor inteiro** nos formatos **octal** (base 8) ou **hexadecimal** (base 16). Para isso, usamos os tipos de saída **“%o”** (sinal de porcento mais a letra “o”, não o zero “0”) para que a função **printf** exiba o valor em **octal**, e **“%x”** para **hexadecimal com letras minúsculas** e **“%X”** para **hexadecimal com letras maiúsculas**. Abaixo, podemos ver alguns exemplos:

### Exemplo: printf() com valores no formato octal e hexadecimal

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n = 125;
5     printf("Valor de n: %d\n",n);
6     printf("Valor em octal: %o\n",n);
7     printf("Valor em hexadecimal: %x\n",n);
8     printf("Valor em hexadecimal: %X\n",n);
9     system("pause");
10    return 0;
11 }
```

### EXIBINDO VALORES COMO NOTAÇÃO CIENTÍFICA

O exemplo abaixo mostra como exibir um **valor real** (também chamado **ponto flutuante**) no formato de **notação científica**. Para isso, usamos os tipos de saída “%e” ou “%E”, sendo que o primeiro usará a letra E minúscula enquanto o segundo usará ela maiúscula na saída.

Temos também os tipos de saída “%g” e “%G”. Esses tipos de saída, quando utilizados, deixam para o compilador decidir se é melhor usar “%f” ou “%e” (ou “%E”, se for utilizado “%G”). Nesse caso, o compilador usa “%e” (ou “%E”) para que números muito grandes ou muito pequenos sejam mostrados na forma de notação científica. Abaixo, podemos ver alguns exemplos:

#### Exemplo: imprimindo float e double como notação científica

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     float f = 0.00000025;
5     double d = 10.53;
6     printf( 'Valor real: %e\n', f );
7     printf( 'Valor real: %E\n', f );
8     printf( 'Valor real: %g\n', d );
9     printf( 'Valor real: %G\n', d );
10    system( 'pause' );
11    return 0;
12 }
```

#### EXIBINDO VALORES INTEIROS “SEM SINAL” E ENDEREÇOS

Para imprimir valores inteiros sem sinal, devemos utilizar o tipo de saída “%u” e evitar o uso do tipo “%d”. Isso ocorre por que o tipo “%u” trata o número inteiro como **unsigned** (sem sinal), enquanto “%d” o trata como **signed** (com sinal).

A primeira vista os dois tipos podem parecer iguais. Se o valor inteiro estiver entre 0 e **INT\_MAX** ( $2^{31} - 1$  em sistemas de **32 bits**), a saída será idêntica para os dois casos (“%d” e “%u”). Porém, se o valor inteiro for negativo (para entradas com sinal, **signed**) ou estiver entre **INT\_MAX** e **UINT\_MAX** (isto é, entre  $2^{31}$  e  $2^{32} - 1$  em sistemas de **32 bits**), os valores impressos pelos tipos “%d” e “%u” serão diferentes. Neste caso, o tipo “%d” irá imprimir um valor negativo, enquanto o tipo “%u” irá imprimir um valor positivo.

Já para imprimir o endereço de memória de uma variável ou ponteiro, podemos utilizar o tipo de saída “%p”. Esse tipo de saída irá imprimir o

endereço no formato hexadecimal, sendo que o valor impresso depende do compilador e da plataforma. O endereço de memória poderia ser também impresso por meio do tipo “%x” (ou “%X”), porém, esse tipo de saída pode gerar uma impressão incorreta do valor do endereço, principalmente em sistemas 64-bit. Abaixo, podemos ver alguns exemplos:

#### Exemplo: imprimindo valores inteiro sem sinal e endereços

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     unsigned int n = 2147483647;
5     printf(‘Valor real: %d\n’,n);
6     printf(‘Valor real: %u\n’,n);
7     n = n + 1;
8     printf(‘Valor real: %d\n’,n);
9     printf(‘Valor real: %u\n’,n);
10    printf(‘Endereco de n = %p\n’,&n);
11    system(‘pause’);
12    return 0;
13 }
```

#### EXIBINDO O SÍMBOLO DE “%”

O caractere “%” é normalmente utilizado dentro da função **printf()** para especificar o formato de saída em que um determinado dado será escrito. Porém, pode ser as vezes necessário imprimir o caractere “%” na tela de saída. Para realizar essa tarefa, basta colocar dois caracteres “%”, “%%”, para que ele seja impresso na tela de saída como mostra o exemplo abaixo:

#### Exemplo: imprimindo o símbolo de “%”

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     printf(‘Juros de 25%%\n’);
5     system(‘pause’);
6     return 0;
7 }
```

## 12.4.2 AS “FLAGS” PARA OS TIPOS DE SAÍDA

As “flags” permitem adicionar características extras a um determinado formato de saída utilizado com a função **printf()**. Elas vem logo em seguida ao sinal de % e antes do tipo de saída. A tabela abaixo mostra todas as “flags” suportadas pela linguagem C:

“flags”	Descrição
-	imprime o valor justificado à esquerda dentro da largura determinada pelo campo <i>largura</i> ; Por padrão, o valor é sempre justificado a direita.
+	imprime o símbolo de sinal (+ ou -) antes do valor impresso, mesmo para números positivos. Por padrão, apenas os números negativos são impressos com o sinal.
(espaço)	imprime o valor com espaços em branco à esquerda dentro da largura determinada pelo campo <i>largura</i> .
#	Se usado com os tipos “%o”, “%x” ou “%X”, o valor impresso é precedido de “0”, “0x” ou “0X”, respectivamente, para valores diferentes de zero. Se usado com valores do tipo <b>float</b> e <b>double</b> , imprime o ponto decimal mesmo se nenhum dígito vir em seguida. Por padrão, se nenhum dígito for especificado, nenhum ponto decimal é escrito.
0	imprime o valor com zeros (0) em vez de espaços à esquerda dentro da largura determinada pelo campo <i>largura</i>

### JUSTIFICANDO UM VALOR A ESQUERDA

O exemplo abaixo mostra o uso das “flags” para justificar os dados na tela de saída. Note que para justificar um valor e preciso definir o valor da **largura**, isto é, a quantidade mínima de caracteres que se poderá utilizar durante a impressão na tela de saída. No caso, definimos que a largura são 5 caracteres:

### Exemplo: justificando um valor a esquerda

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n = 5;
5     //justifica a direita
6     printf('n = %5d\n',n);
7     //justifica a esquerda
8     printf('n = %-5d\n',n);
9     system('pause');
10    return 0;
11 }
```

### FORÇAR A IMPRESSÃO DO SINAL DO NÚMERO

Por padrão, a função **printf()** imprime apenas os números negativos com o sinal. No entanto, pode-se forçar a impressão do sinal de positivo, como mostra o exemplo abaixo:

### Exemplo: imprimindo sempre o sinal do número

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n = 5;
5     //sem sinal
6     printf('n = %d\n',n);
7     //com sinal
8     printf('n = %+d\n',n);
9     system('pause');
10    return 0;
11 }
```

### IMPRIMINDO “ESPAÇOS” OU ZEROS ANTES DE UM NÚMERO

Quando definimos a **largura** do valor, estamos definindo a quantidade mínima de caracteres que será utilizada durante a impressão na tela de saída. Por padrão, a função **printf()** justifica os dados a direita e preenche o restante da largura com espaços. Porém, pode-se preencher o restante da largura com zeros, como mostra o exemplo abaixo:

### Exemplo: imprimindo espaços ou zeros antes do número

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n = 5;
5     //com espaços (padrão)
6     printf('n = % 5d\n',n);
7     //com zeros
8     printf('n = %05d\n',n);
9     system('pause');
10    return 0;
11 }
```

### IMPRIMINDO O PREFIXO HEXADECIMAL E OCTAL E O PONTO

Por padrão, a função **printf()** imprime valores no formato octal e hexadecimal sem os seus prefixo (0 e 0x, respectivamente). Já o ponto decimal dos valores em ponto flutuante é omitido caso não se tenha definido a precisão apesar de ter sido incluído na sua formatação o indicador de ponto (“.”). Felizmente, pode-se forçar a impressão do prefixo e do ponto, como mostra o exemplo abaixo:

### Exemplo: imprimindo o prefixo e o ponto decimal

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n = 125;
5     //octal e hexadecimal sem prefixo
6     printf('x = %o\n',n);
7     printf('x = %X\n',n);
8     //octal e hexadecimal com prefixo
9     printf('x = %#o\n',n);
10    printf('x = %#X\n',n);
11    float x = 5.00;
12    //float sem ponto
13    printf('x = %.f\n',x);
14    //float com ponto
15    printf('x = %#.f\n',x);
16    system('pause');
17    return 0;
18 }
```

### 12.4.3 O CAMPO “LARGURA” DOS TIPOS DE SAÍDA

O campo **largura** é comumente usado com outros campos (como visto com as “flags”). Ele na verdade especifica o número mínimo de caracteres a serem impressos na tela de saída. Ela pode ser definida de duas maneiras, como mostra a tabela abaixo:

“largura”	Descrição
“número”	Número mínimo de caracteres a serem impressos. Se a largura do valor a ser impresso é inferior a este número, espaços em branco serão acrescentados a esquerda.
*	Informa que a largura vai ser especificada por um valor inteiro passado como parâmetro para a função <b>printf()</b> .

Abaixo, podemos ver um exemplo de uso do campo largura:

#### Exemplo: definindo o campo largura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int n = 125;
5     int largura = 10;
6     //largura definida dentro do campo
7     printf('n = %10d\n', n);
8     //largura definida por uma variável inteira
9     printf('n = %*d\n', largura, n);
10    system('pause');
11    return 0;
12 }
```

### 12.4.4 O CAMPO “PRECISÃO” DOS TIPOS DE SAÍDA

O campo **precisão** é comumente usado com valores de ponto flutuante (tipos **float** e **double**). De modo geral, esse campo especifica o número de caracteres a serem impressos na tela de saída após o ponto decimal.



Note que o campo **precisão** vem sempre começando com um caractere de **ponto** (.).

Porém, o campo **precisão** pode ser utilizado com outros tipos, como mostra a tabela abaixo:

<b>“.precisão”</b>	<b>Descrição</b>
.número	Para os tipos “%d”, “%i”, “%u”, “%o”, “%x” e “%X”: número mínimo de caracteres a serem impressos. Se a largura do valor a ser impresso é inferior a este número, zeros serão acrescentados a esquerda Para os tipos “%f”, “%e” e “%E”: número de dígitos a serem impressos após o ponto decimal. Para os tipos “%g” e “%G”: número máximo de dígitos significativos a serem impressos. Para o tipo “%s”: número máximo de caracteres a serem impressos. Por padrão, todos os caracteres são impressos até que o caractere “\0” é encontrado. Para o tipo “%c”: sem efeito. Se nenhum valor for especificado para a precisão, a precisão é considerada 0 (padrão).
.*	Informa que a largura vai ser especificada por um valor inteiro passado como parâmetro para a função <b>printf()</b> .

### O CAMPO “PRECISÃO” PARA VALORES INTEIROS

O campo “precisão”, quando usado com valores inteiros (pode ser também no formato octal ou hexadecimal), funciona de modo semelhante a largura do campo, ou seja, especifica o número mínimo de caracteres a ser impressos, com a vantagem de já preencher o restante dessa largura com zeros, como mostra o exemplo abaixo:

#### Exemplo: a “precisão” para valores inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int n = 125;
5     printf('n = %.8d (decimal)\n', n);
6     printf('n = %.8o (octal)\n', n);
7     printf('n = %.8X (hexadecimal)\n', n);
8     system('pause');
9     return 0;
10 }
```

## O CAMPO “PRECISÃO” PARA VALORES REAIS

O campo “precisão”, quando usado com valores de ponto flutuante (tipos **float** e **double**), especifica o número de caracteres a serem impressos na tela de saída após o ponto decimal. A única exceção é com os tipos de saída “%g” e “%G”. Nesse caso, o campo “precisão” especifica o número máximo de caracteres a serem impressos. Abaixo é possível ver alguns exemplos:

### Exemplo: a “precisão” para valores reais

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     float n = 123.45678;
5     printf( 'n = %.3f\n', n);
6     printf( 'n = %.5f\n', n);
7     printf( 'n = %.5e\n', n);
8     printf( 'n = %.5g\n', n);
9     system( 'pause' );
10    return 0;
11 }
```

## O CAMPO “PRECISÃO” USADO COM STRINGS

O campo “precisão” também permite especificar o número máximo de caracteres a serem impressos de uma **string**. Por padrão, todos os caracteres da **string** são impressos até que o caractere “\0” é encontrado, como mostra o exemplo abaixo:

### Exemplo: a “precisão” para strings

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char texto[20] = 'Meu programa C';
5     printf( '%s\n', texto);
6     printf( '%.3s\n', texto);
7     printf( '%.12s\n', texto);
8     system( 'pause' );
9     return 0;
10 }
```

## O CAMPO “PRECISÃO” DEFINIDO POR UMA VARIÁVEL INTEIRA

Por fim, podemos informar que a “precisão” será especificada por um valor inteiro passado como parâmetro para a função **printf()**:

#### Exemplo: precisão como parâmetro

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     float n = 123.45678;
5     int precisao = 10;
6     //precisão definida por uma variável inteira
7     printf( 'n = %.*f\n', precisao, n);
8     system( 'pause' );
9     return 0;
10 }
```

#### 12.4.5 O CAMPO “COMPRIMENTO” DOS TIPOS DE SAÍDA

O campo “comprimento” é utilizado para imprimir valores que sejam do tipo **short int**, **long int** e **long double**, como mostra a tabela abaixo:

“comprimento”	Descrição
h	Para os tipos “%d”, “%i”, “%u”, “%o”, “%x” e “%X”: o valor é interpretado como <b>short int</b> ou <b>unsigned short int</b>
l	Para os tipos “%d”, “%i”, “%u”, “%o”, “%x” e “%X”: o valor é interpretado como <b>long int</b> ou <b>unsigned long int</b>
	Para os tipos “%c” e “%s”: permite imprimir caracteres e sequências de caracteres onde cada caractere possui mais do que 8-bits.
L	Para os tipos “%f”, “%e”, “%E”, “%g” e “%G”: o valor é interpretado como <b>long double</b>

Deve-se tomar cuidado com o campo “comprimento”, pois ele não funciona corretamente dependendo do compilador e da plataforma utilizada.

#### 12.4.6 USANDO MAIS DE UMA LINHA NA FUNÇÃO PRINTF()

Pode ocorrer de a linha que queiramos escrever na tela de saída seja muito grande. Isso faz com que a string dentro da função **printf()** não possa ser

visualizada toda de uma vez. Felizmente, a função permite que coloquemos um caractere de barra invertida “\” apenas para indicar que a **string** que estamos digitando continua na próxima linha:

#### Exemplo: a função printf() com mais de uma linha

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     printf( ' 'Esse texto que estou querendo escrever \
5     na tela de saída e muito grande. Por isso eu \
6     resolvi quebrar ele em varias linhas \n' );
7     system( ' 'pause' ' );
8     return 0;
9 }
```

## 12.5 RECURSOS AVANÇADOS DA FUNÇÃO SCANF()

Vimos na Seção 2.2.3 que a função **scanf()** é uma das funções de entrada/leitura de dados da linguagem C. Sua funcionalidade básica é ler no dispositivo de entrada de dados (teclado) um conjunto de valores, caracteres e/ou sequência de caracteres de acordo com o formato especificado. Porém, essa função permite uma variedade muito maior de formatações do que as vistas até agora. Começemos pela sua definição. A forma geral da função **scanf()** é:

```
int scanf(“tipos de entrada”, lista de variáveis)
```

A função **scanf()** recebe 2 parâmetros de entrada

- “**tipos de entrada**”: conjunto de caracteres que especifica o formato dos dados a serem lidos do teclado;
- **lista de variáveis**: conjunto de nomes de variáveis que serão lidos e separados por vírgula, onde cada nome de variável é precedido pelo operador &.

Note também que a função **scanf()** retorna um valor inteiro, ignorado até o presente momento. Esse valor de retorno será

- em caso de sucesso, o número total de itens lidos. Esse número pode ser igual ou menor do que o número esperado de itens a serem lidos;

- a constante **EOF**, em caso de erro da função.



O valor de retorno da função **scanf()** permite identificar o funcionamento adequado da função.

Vimos também que quando queremos ler dados formatados do teclado nós usamos os **tipos de entrada** para especificar o formato de entrada dos dados que serão lidos. E que cada tipo de entrada é precedido por um sinal de % e um **tipo de entrada** deve ser especificado para cada variável a ser lida.



A string do **tipo de entrada** permite especificar mais características dos dados lidos além do seu formato. Essas características são **opcionais** e são: \*, **largura** e **modificadores**.

A ordem em que essas quatro características devem ser especificadas é a seguinte:

**%[\*][largura][modificadores]tipo de entrada**

Como o **tipo de entrada**, cada uma dessas características possui um conjunto de valores pré-definidos e suportados pela linguagem. Nas seções seguintes são apresentados todos os valores suportados para cada uma das características de formatação possíveis.

### 12.5.1 OS TIPOS DE ENTRADA

A função **scanf()** pode ser usada para ler virtualmente qualquer tipo de dado. A tabela abaixo mostra todos os tipos de entrada suportados pela linguagem C:

“tipo de entrada”	Descrição
%c	Leitura de um caractere ( <b>char</b> ). Se uma largura diferente do valor 1 é especificada, a função lê o número de caracteres especificado na largura e os armazena em posições sucessivas de memória (array) do ponteiro para <b>char</b> passado como parâmetro. O caractere “\0” <b>não</b> é acrescentado no final.
%d	Leitura de um número inteiro ( <b>int</b> ). Ele pode ser precedido pelo símbolo de sinal (+ ou -)
%u	Leitura de um número inteiro sem sinal ( <b>unsigned int</b> )
%i	Leitura de um número inteiro ( <b>int</b> ). O valor pode estar precedido de “0x” ou “0X”) se for <b>hexadecimal</b> , ou pode ser precedido por zero (0) se for <b>octal</b> .
%f, %e, %E, %g, %G	Leitura de um número real ( <b>float</b> e <b>double</b> ). Ele pode ser precedido pelo símbolo de sinal (+ ou -), e/ou seguido pelos caracteres “e” ou “E” (notação científica) e/ou possuir o separador de ponto decimal
%o	Leitura de um número inteiro ( <b>int</b> ) no formato octal ( <b>base 8</b> ). O valor pode ou não estar precedido de zero (0).
%x ou %X	Leitura de um número inteiro ( <b>int</b> ) no formato hexadecimal. O valor pode ou não estar precedido de “0x” ou “0X”)
%s	Leitura de um sequência de caracteres ( <b>string</b> ) até um caractere de <b>nova linha</b> “\n” ou <b>espaço em branco</b> seja encontrado.

A seguir, são apresentados alguns exemplos de como cada tipo de entrada pode ser utilizado para ler determinado dado do teclado.

### LENDO OS TIPOS BÁSICOS

A linguagem C possui vários tipos de entrada que podem ser utilizados com os tipos básicos, ou seja, **char** (“%c” e “%d”), **int** (“%d” e “%i”), **float** e **double** (“%f”). Note que o tipo **char** pode ser lido do teclado por meio dos operadores “%c” e “%d”. Nesse caso, “%c” irá ler um caractere e armazenar na variável, enquanto “%d” irá ler um valor numérico e armazenar na variável o caractere correspondente da tabela ASCII. Abaixo, tem-se alguns exemplos de leitura dos tipos básicos:

### Exemplo: usando scanf() para ler os tipos básicos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n;
5     float f;
6     double d;
7     char letra;
8     //leitura de int
9     scanf("%d",&n);
10    scanf("%i",&n);
11    //leitura de char
12    scanf("%d",&letra);
13    scanf("%c",&letra);
14    //leitura de float e double
15    scanf("%f",&f);
16    scanf("%f",&d);
17    system("pause");
18    return 0;
19 }
```

### LENDO VALORES NO FORMATO OCTAL OU HEXADECIMAL

O exemplo abaixo mostra como ler um **valor inteiro** nos formatos **octal** (base 8) ou **hexadecimal** (base 16). Para isso, usamos os tipos de entrada **“%o”** (sinal de porcento mais a letra “o”, não o zero “0”) para que a função **scanf()** leia o valor em **octal**, e **“%x”** para ler um valor em **hexadecimal com letras minúsculas** e **“%X”** para **hexadecimal com letras maiúsculas**. Note que em ambos os casos, o valor lido pode ou não estar precedido de zero(0) se for octal ou **“%x”** (ou **“%X”**) se for hexadecimal. Abaixo, podemos ver alguns exemplos:

### Exemplo: scanf() com valores no formato octal e hexadecimal

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n;
5     //leitura no formato octal
6     scanf("%o",&n);
7     //leitura no formato hexadecimal
8     scanf("%x",&n);
9     scanf("%X",&n);
10    system("pause");
11    return 0;
12 }
```

## LENDO VALORES COMO NOTAÇÃO CIENTÍFICA

De modo geral, podemos ler um valor em notação científica com qualquer um dos tipos de entrada habilitados para lerem valores de ponto flutuante (**float** e **double**): “%f”, “%e”, “%E”, “%g” e “%G”. Na verdade, esses tipos de entrada não fazem distinção na forma como o valor em ponto flutuante é escrito, desde que seja ponto flutuante, como mostra o exemplo abaixo:

### Exemplo: lendo float e double como notação científica

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     float x;
5     scanf("%f",&x);
6     scanf("%e",&x);
7     scanf("%E",&x);
8     scanf("%g",&x);
9     scanf("%G",&x);
10    system("pause");
11    return 0;
12 }
```

## LENDO UMA STRING DO TECLADO

O exemplo abaixo mostra como ler uma **string** (ou array de caracteres, **char**) do teclado. Para isso, usamos o tipo de entrada “%s”. Note que quando usamos a função **scanf()** para ler uma string, o símbolo de & antes do nome da variável não é utilizado. Além disso, a função **scanf()** lê apenas **strings** digitadas sem espaços, ou seja, apenas palavras. No caso de ter sido digitada uma frase (uma sequência de caracteres contendo espaços) apenas os caracteres digitados antes do primeiro espaço encontrado serão armazenados na **string**.

### Exemplo: lendo uma string com scanf()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char texto[20];
5     printf("Digite algum texto: ");
6     scanf("%s",texto);
7     system("pause");
8     return 0;
9 }
```

## 12.5.2 O CAMPO ASTERISCO “\*\*”

O uso de um **asterisco** “\*\*” após o símbolo de % indica que os dados formatados devem ser lidos do teclado mas ignorados, ou seja, não devem ser armazenados em nenhuma variável. Abaixo é possível ver um exemplo de uso:

### Exemplo: ignorando dados digitados

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int x,y;
5     printf(‘‘Digite tres inteiros: ‘’);
6     scanf(‘‘%d %*d %d ‘’,&x,&y);
7     printf(‘‘Numeros lidos: %d e %d\n ‘’,x,y);
8     char nome[20], curso[20];
9     printf(‘‘Digite nome, idade e curso: ‘’);
10    scanf(‘‘%s %*d %s ‘’,nome,curso);
11    printf(‘‘Nome: %s\nCurso: %s\n ‘’,nome,curso);
12    system(‘‘pause ‘’);
13    return 0;
14 }
```

## 12.5.3 O CAMPO “LARGURA” DOS TIPOS DE ENTRADA

Basicamente, o campo **largura** é um valor inteiro que especifica o número máximo de caracteres que poderão ser lidos em uma operação de leitura para um determinado tipo de entrada. Isso é muito útil quando queremos limitar a quantidade de caracteres que serão lidos em uma **string** de modo a não ultrapassar o tamanho máximo de armazenamento dela, ou quando queremos limitar a quantidade de dígitos de uma valor como, por exemplo, no caso do valor de um dia do mês (dois dígitos).



Os caracteres que ultrapassam o tamanho da **largura** determinado são descartados pela função **scanf()**, mas continuam no **buffer** do teclado. Uma outra chamada da função **scanf()** irá considerar esses caracteres já contidos no **buffer** como parte do que será lido. Assim, para evitar confusões, é conveniente esvaziar o buffer do teclado com a função **fflush(stdin)** a cada nova leitura.

Abaixo é possível ver um exemplo de uso do campo **largura**:

### Exemplo: limitando a quantidade de caracteres

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int n;
5     printf("Digite um numero (2 digitos): ");
6     scanf("%2d",&n);
7     printf("Numero lido: %d\n",n);
8     fflush(stdin);
9     char texto[11];
10    printf("Digite uma palavra (max: 10 caracteres):");
11    scanf("%10s",texto);
12    printf("Palavra lida: %s\n",texto);
13    system("pause");
14    return 0;
15 }
```

### 12.5.4 OS “MODIFICADORES” DOS TIPOS DE ENTRADA

Os “modificadores” dos tipos de entrada se assemelham ao campo “comprimento” da função **printf()**. Eles são utilizados para ler valores que sejam do tipo **short int**, **long int** e **long double**, como mostra a tabela abaixo:

“modificadores”	Descrição
h	Para os tipos “%d” ou “%i”: o valor é interpretado como <b>short int</b> Para os tipos “%u”, “%o”, “%x” e “%X”: o valor é interpretado como <b>unsigned short int</b>
l	Para os tipos “%d” ou “%i”: o valor é interpretado como <b>long int</b> Para os tipos “%u”, “%o”, “%x” e “%X”: o valor é interpretado como <b>unsigned long int</b> Para os tipos “%f”, “%e”, “%E”, “%g” e “%G”: o valor é interpretado como <b>long double</b>
L	Para os tipos “%f”, “%e”, “%E”, “%g” e “%G”: o valor é interpretado como <b>long double</b>

Deve-se tomar cuidado com o uso desses “modificadores”, pois eles não funcionam corretamente dependendo do compilador e da plataforma utilizada.

## 12.5.5 LENDO E DESCARTANDO CARACTERES

Por definição, a função **scanf()** descarta qualquer espaço em branco que for digitado pelo usuário. Mesmo os espaços em branco adicionados junto ao tipo de entrada não possuem efeito, o que faz com que as duas chamadas da função abaixo possuam o mesmo efeito: ler dois valores de inteiro:

```
int x, y;
scanf("%d%d",&x,&y);
scanf("%d %d",&x,&y);
```

Porém, qualquer caractere que não seja um espaço em branco que for digitado junto ao tipo de entrada faz com que a função **scanf()** exija a leitura desse caractere e o descarte em seguida. Isso é interessante quando queremos que seja feita a entrada de dados em uma formatação pré-determinada, como uma data. Por exemplo, “%d / %d / %d” faz com que a função **scanf()** leia um inteiro, uma barra (que será descartada), outro valor inteiro, outra barra (que também será descartada) e , por fim, o terceiro último inteiro.



Se o caractere a ser lido e descartado não é encontrado, a função **scanf()** irá terminar, sendo os dados lidos de maneira incorreta.

Abaixo é possível ver esse exemplo em ação:

### Exemplo: lendo e descartando caracteres

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int d,m,a;
5     printf(“ Digite a data no formato dia/mes/ano: ”);
6     scanf(“ %d/%d/%d ”,&d,&m,&a);
7     printf(“ %d - %d - %d\n ”,d,m,a);
8     system(“ pause ”);
9     return 0;
10 }
```

## 12.5.6 LENDO APENAS CARACTERES PRÉ-DETERMINADOS

A função **scanf()** permite também que se defina uma lista de caracteres pré-determinados, chamada de **scanset**, que poderão ser lidos do teclado e armazenados em uma **string**. Essa lista é definida substituindo o tipo de entrada **%s**, normalmente utilizado para a leitura de uma **string**, por **%[]**, onde, **dentro dos colchetes**, é definida a lista de caracteres que poderão ser lidos pela função **scanf()**. Assim, se quiséssemos ler uma **string** contendo apenas vogais, a função **scanf()** seria usada como mostrado abaixo:

### Exemplo: lendo apenas caracteres pré-determinados

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char texto[20];
5     printf( "Digite algumas vogais: " );
6     scanf( "[%aeiou]", texto );
7     printf( "Texto: %s\n", texto );
8     system( "pause" );
9     return 0;
10 }
```

Note que dentro da lista de caracteres pré-determinados foram digitadas as vogais **minúsculas**. A linguagem C considera diferente letras maiúsculas e minúsculas.



Se um dos caracteres digitados não fizer parte da lista de caracteres pré-determinados (**scanset**) a leitura da **string** é terminada e a função **scanf()** passa para o próximo tipo de entrada, se houver.

## USANDO UM INTERVALO DE CARACTERES PRÉ-DETERMINADOS

Ao invés de definir uma lista de caracteres, pode-se definir um intervalo de caracteres, como por exemplo, todas as letras minúsculas, ou todos os dígitos numéricos. Para fazer isso, basta colocar o primeiro e o último caracteres do intervalo separados por um **hifen**. Assim, se quiséssemos ler apenas os caracteres de A a Z, a função **scanf()** ficaria como no exemplo abaixo:

### Exemplo: usando um intervalo de caracteres pré-determinados

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char texto[20];
5     printf(“Digite algumas letras: ”);
6     scanf(“%[A-Z]”, texto);
7     printf(“Texto: %s\n”, texto);
8     system(“pause”);
9     return 0;
10 }
```

Pode-se ainda especificar mais de um intervalo de caracteres pré-determinados. Para fazer isso, basta colocar o primeiro e o último caracteres do segundo intervalo, separados por um **hífen**, logo após definir o primeiro intervalo. Assim, se quiséssemos ler apenas os caracteres de A a Z e os dígitos de 0 a 9, a função **scanf()** ficaria como no exemplo abaixo:

### Exemplo: usando mais de um intervalo de caracteres

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char texto[20];
5     printf(“Digite letras e números: ”);
6     scanf(“%[A-Z0-9]”, texto);
7     printf(“Texto: %s\n”, texto);
8     system(“pause”);
9     return 0;
10 }
```

## 12.6 CLASSES DE ARMAZENAMENTO DE VARIÁVEIS

A linguagem C possui um conjunto de modificadores, chamados **classes de armazenamento**, que permitem alterar a maneira como o compilador vai armazenar uma variável.



As classes de armazenamento são utilizados para definir o escopo e tempo de vida das variáveis dentro do programa.

Basicamente, as classes de armazenamento definem a acessibilidade de uma variável dentro da linguagem C. Ao todo, existem quatro classes de armazenamento:

- **auto**
- **extern**
- **static**
- **register**

### 12.6.1 A CLASSE AUTO

A classe de armazenamento **auto** permite definir variáveis locais. Nele, as variáveis são automaticamente alocadas no início de uma função/bloco de comandos, e automaticamente liberadas quando essa função/bloco de comandos termina. Trata-se do **modo padrão** de definição de variáveis, por esse motivo ela raramente é usada. Por exemplo, as duas variáveis abaixo

```
int x;  
auto int y;
```

possuem a mesma classe de armazenamento (**auto**).



A classe **auto** só pode ser utilizada dentro de funções e blocos de comandos definidos por um conjunto de chaves {} (escopo local).

### 12.6.2 A CLASSE EXTERN

A classe de armazenamento **extern** permite definir variáveis globais que serão visíveis em mais de um arquivo do programa. Ao contrário dos programas escritos até aqui, podemos escrever programas que podem ser divididos em vários arquivos, os quais podem ser compilados separadamente.

Imagine que temos o seguinte trecho de código:

```
int soma = 0;
int main(){
    escreve();
    return 0;
}
```

Agora imagine que queiramos usar a variável global soma em um **segundo** arquivo do nosso programa. Para fazer isso, basta adicionar a palavra **extern** na declaração da variável para o compilador entender que ela já foi definida em outro arquivo:

```
extern int soma;
void escreve(){
    printf("Soma = %d ",soma);
}
```



Ao colocar a palavra **extern** antes da declaração da variável soma, não estamos declarando uma nova variável, mas apenas informando ao compilador que ela existe em outro local de armazenamento previamente definido. Por esse motivo, ela **NÃO** pode ser inicializada.

### 12.6.3 A CLASSE STATIC

O funcionamento da classe de armazenamento **static** depende de como ela é utilizada dentro do programa. A classe **static** é o **modo padrão** de definição de variáveis globais, ou seja, variáveis que existem durante todo o tempo de vida do programa. Por esse motivo ela raramente é usada na declaração de variáveis globais, como mostra o exemplo abaixo:

### Exemplo: variáveis globais com static

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int x = 20;
4 static y = 10;
5 int main() {
6     printf('x = %d\n', x);
7     printf('y = %d\n', y);
8     system('pause');
9     return 0;
10 }
```

No exemplo acima, ambas as variáveis **x** e **y** possuem a mesma classe de armazenamento (**static**).

A classe **static** também pode ser utilizada com variáveis locais, como as variáveis definidas dentro de uma função. Nesse caso, a variável é inicializada em tempo de compilação e o valor da inicialização deve ser uma constante. Uma variável local definida dessa maneira irá manter o seu valor entre as diferentes chamadas da função, portanto deve-se tomar muito cuidado com a sua utilização, como mostra o exemplo abaixo:

### Exemplo: variáveis locais com static

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 void imprime() {
4     static n = 0;
5     printf('%d\n', n++);
6 }
7 int main() {
8     int i;
9     for(i=1; i<=10; i++)
10        imprime();
11    system('pause');
12    return 0;
13 }
```

No exemplo acima, o valor da variável **n** será diferente para cada chamada da função **imprime()**.

Por fim, a classe **static** também pode ser utilizada para definir funções, como mostra o exemplo abaixo:

### Exemplo: funções com static

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 static void imprime(){
4     printf(“Executando funcao imprime()\n”);
5 }
6 int main(){
7     imprime();
8     system(“pause”);
9     return 0;
10 }
```

Uma função é, por padrão, da classe de armazenamento **extern**, ou seja, as funções são visíveis em mais de um arquivo do programa. Ao definirmos uma função como **static** estamos garantindo que ela seja visível apenas dentro daquele arquivo do programa, ou seja, apenas funções dentro daquele arquivo poderão ver uma função **static**.

#### 12.6.4 A CLASSE REGISTER

A classe de armazenamento **register** serve para especificar que uma variável será muito utilizada e que seria interessante armazená-la no registrador da CPU do computador. Isso por que o tempo de acesso aos registradores da CPU é muito menor que o tempo de acesso a memória RAM, onde as variáveis ficam normalmente armazenadas. Uma variável da classe **register** é declarada como mostrado abaixo:

```
register int y;
```

Algumas considerações são necessárias sobre a classe **register**:

- não se pode usar o operador de endereço &. Isso por que a variável está no registrador, e não mais na memória;
- o tamanho da variável é limitado pelo tamanho do registrador, portanto apenas variáveis de tipos pequenos (que ocupem poucos bytes) podem ser definidas como da classe **register**;



A classe de armazenamento **register** pode ser entendida como uma dica de armazenamento que damos para o compilador. O compilador é livre para decidir se vai ou não armazenar essa variável no registrador.

Se o compilador decidir ignorar classe **register**, a variável será definida como sendo da classe **auto**. Isso significa que não podemos definir uma variável global (**static**) como sendo da classe **register**.



A classe de armazenamento **register** é raramente utilizada. Os compiladores modernos fazem trabalhos de otimização na alocação de variáveis melhores que os programadores.

## 12.7 TRABALHANDO COM CAMPOS DE BITS

A linguagem C possui meios de acessar diretamente os bits, ou um único bit, dentro de um byte, sem fazer uso dos operadores bit-a-bit. Para isso, ela conta com um tipo especial de membro de estruturas e uniões chamado de **campo de bits**, ou **bitfield**. Seu uso é extremamente útil quando a quantidade de memória para armazenamento de dados é limitada. Nesse caso, várias informações podem ser armazenadas em um único byte, como as “flags” indicando se determinado item do sistema está ativo (1) ou não (0). Os campos de bits podem ainda ser utilizados para a leitura de arquivos externos, em especial, formatos não-padrão de arquivo como valores de tipos inteiros com 9 bits. Outro uso frequente dos campos de bits são para realizar a comunicação (entrada e saída de dados) com dispositivos de hardware.



Campos de bits só podem ser utilizados em variáveis que são membros de **structs** ou **unions**.

A forma geral de declaração de uma variável com campo de bit como membro de uma **struct** (ou **union**) segue o padrão abaixo:

tipo nome\_campo: **comprimento**;

Note que a declaração de uma variável com campo de bit é semelhante a declaração de uma variável membro de uma **struct/union**, possuindo apenas como informação extra o valor do **comprimento** (definido após os dois pontos), que nada mais é do que a quantidade de bits que o campo irá possuir. O valor do **comprimento** pode ser um número ou uma expressão constante. Note ainda que o **comprimento** de um campo de bits não deve exceder o número total de bits do tipo da variável utilizada na declaração.



Campos de bits só podem ser declarados como sendo do tipo **int**, sendo possível utilizar os modificadores **signed** e **unsigned**. Se ele for do tipo **int** ou **signed int**, seu comprimento deverá ser maior do que **um** (1).

Se a variável com campo de bit for do tipo **int** ou **signed int**, ela irá possuir, obrigatoriamente, um bit de sinal. Um campo de bit de comprimento um (1) não pode ter sinal sendo necessário, portanto, um comprimento mínimo de dois (2) bits. De qualquer modo, é aconselhável sempre utilizar campos de bits com o tipo **unsigned int**.

Abaixo é possível ver um exemplo de uma estrutura contendo variáveis com campo de bits:

```
struct status{  
    unsigned int ligado:1;  
    signed int valor:4;  
    unsigned int :3;  
};
```

Na estrutura acima, temos três campos de bits: **ligado** (1 bit), **valor** (4 bits), e um terceiro campo sem nome de tamanho 3 bits. Como o campo **ligado** possui apenas 1 bit, seus valores possíveis são 0 (desligado) ou 1 (ligado). Já o campo **valor** possui 4 bits, portanto, seus valores podem ir de -8 até 7. Por fim, temos um campo de bits sem nome e de tamanho 3 bits. Note que esses 3 bits servem apenas para completar um total de 8 bits na estrutura.



Campos de bits sem nome são úteis para preencher uma estrutura de modo a fazer com que ela esteja adequada a um layout de especificado.

Os membros de uma estrutura **que não são campos de bits** estão sempre alinhados aos limites dos bytes na memória. Os campos de bit **sem nome** permitem criar lacunas não identificadas no armazenamento da estrutura, completando os bytes e mantendo o alinhamento dos dados na memória. Por fim, campos de bits sem nome não pode ser acessados ou inicializado.



Campos de bits podem ter comprimento ZERO (0). Neste caso eles não podem possuir um nome. Sua função é a de alinhamento dos bits.

Um campo de bits de comprimento ZERO (0) faz com que o próximo campo de bits seja alinhado com o próximo byte de memória do mesmo tipo do campo de bits. Em outras palavras, um campo de bits de comprimento ZERO (0) indica que nenhum campo de bits adicional devem ser colocado dentro desse byte.



Os membros de uma estrutura com campos de bits não possuem endereços, e como tal não podem ser usados com o operador de endereço (&). Por esse motivo, não podemos ter ponteiros ou arrays deles. O operador **sizeof** também não pode ser aplicado a campos de bits.

Abaixo tem-se um exemplo de uso de campos de bits:

#### Exemplo: trabalhando com campos de bits

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct status{
4     unsigned int ligado:1;
5     signed int valor:4;
6     unsigned int :3;
7 };
8 void check_status(struct status s){
9     if (s.ligado == 1)
10        printf('LIGADO\n');
11     if (s.ligado == 0)
12        printf('DESLIGADO\n');
13 }
14 int main(){
15     struct status ESTADO;
16     ESTADO.ligado = 1;
17     check_status(ESTADO);
18     system('pause');
19     return 0;
20 }
```

## 12.8 O MODIFICADOR DE TIPO “VOLATILE”

A linguagem C possui mais um modificador de tipos de variáveis. Trata-se do modificador **volatile**. Sua forma geral de uso é

**volatile tipo\_variável** nome\_variável;



O modificador **volatile** pode ser aplicado a qualquer declaração de variável, incluindo as estruturas, uniões e enumerações.

O modificador **volatile** informa ao compilador que aquela variável poderá ser alterada por outros meios e, por esse motivo, ela **NÃO** deve ser otimizada. O principal motivo para o seu uso tem a ver com problemas que trabalham com sistemas dinâmicos, em tempo real ou com comunicação com algum dispositivo de hardware que esteja mapeado na memória.



O modificador **volatile** diz ao compilador para não otimizar qualquer coisa relacionada àquela variável.

Para entender melhor esse modificador, considere o seguinte trecho de código:

```
int reposta;  
void espera(){  
    reposta = 0;  
    while(reposta != 255);//laço infinito  
}
```

Um compilador que seja otimizado irá notar que nenhum outro código pode modificar o valor da variável **reposta** dentro da função **espera()**. Assim, o compilador irá assumir que o valor armazenado em **reposta** é sempre ZERO e, como nunca é modificado, esse laço é infinito. Por ser otimizado, o compilador poderá substituir a condição do comando **while** por UM (1), indicando assim também um laço infinito, mas economizando a comparação da variável **reposta**, como mostra o trecho de código abaixo:

```
int reposta;  
void espera(){  
    reposta = 0;  
    while(1);//laço infinito  
}
```

No entanto, vamos supor que a variável **reposta** possa ser modificada, a qualquer momento, por um dispositivo de hardware mapeado na memória RAM. Nesse caso, o valor da variável pode ser modificado enquanto ela estiver sendo testada no comando **while**, finalizando o laço. Portanto, não é interessante para o programa que esse laço seja otimizado e considerado sempre como um laço infinito. Para impedir que o compilador faça esse tipo de otimização, utilizamos o modificador **volatile**:

```
volatile int reposta;//variável não otimizada
void espera(){
    reposta = 0;
    while(reposta != 255);//laço pode não ser infinito
}
```

Com o modificador **volatile** a condição do laço não será otimizada, e o sistema irá detectar qualquer alteração nela quando está ocorrer. Porém, pode ser um exagero marcar uma variável como **volatile**. Isso porque esse modificador desativa qualquer otimização na variável. Uma alternativa muito mais eficiente é utilizar de **type cast** sempre que não quisermos, e apenas onde é necessário, otimizar a variável:

```
int reposta;
void espera(){
    reposta = 0;
    while(*(volatile int *)&reposta != 255);//laço pode não ser infinito
}
```

## 12.9 FUNÇÕES COM NÚMERO DE PARÂMETROS VARIÁVEL

Vimos na Seção 8 como criar nossas próprias funções. Vimos também que é por meio dos parâmetros de uma função que o programador pode passar a informação de um trecho de código para dentro da função. Esses parâmetros são uma lista de variáveis, separadas por vírgula, onde é especificado o tipo e o nome de cada variável passada para a função.

No entanto, algumas funções, como a função **printf()**, podem ser utilizadas com um, dois, três, ou até mais parâmetros, como mostra o exemplo abaixo:

### Exemplo: printf() com vários parâmetros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int x = 1, y =2;
5     float z = 3;
6     printf("Um parametro: texto\n");
7     printf("Dois parametros: texto e %d\n",x);
8     printf("Tres parametros: texto, %d e %d\n",x,y);
9     printf("Quatro parametros: texto, %d, %d e %f\n",x,
10          y,z);
11     system("pause");
12     return 0;
13 }
```

A linguagem C permite escrever funções que aceitam uma quantidade variável de parâmetros, onde esses parâmetros podem ser de diversos tipos, como é o caso das funções **printf()** e **scanf()**. A declaração, pelo programador, de uma função com uma quantidade variável de parâmetros segue a seguinte forma geral:

```
tipo_retornado nome_função (nome_tipo nome_parâmetros, ...){
    sequência de declarações e comandos
}
```



Para declarar uma função com uma quantidade variável de parâmetros basta colocar “...” como sendo o último parâmetro na declaração da função.

São os “...” declarados nos parâmetros da função que informam ao compilador que aquela função aceita uma quantidade variável de parâmetros.



Uma função com uma quantidade variável de parâmetros deve possuir pelo menos um parâmetro “**normal**” antes dos “...”, ou seja, antes da parte variável.

Isso é necessário pois a função agora não sabe quantos parâmetros serão passados para ela, nem os seus tipos. Portanto, o primeiro parâmetro deve ser usado para informar isso dentro da função. Daí a necessidade da função possuir pelo menos um parâmetro. A função **printf()**, por exemplo,

sabe quantos parâmetros ela irá receber, e os seus tipos, por meio dos **tipos de saída** presentes dentro do primeiro parâmetro: **%c** para **char**, **%d** para **int**, etc.



Uma vez declarada uma função com uma quantidade de parâmetros variável, é necessário acessar esse parâmetros. Para isso, usamos a biblioteca **stdarg.h**.

A biblioteca **stdarg.h** possui as definições de tipos e macros necessárias para acessar a lista de parâmetros da função. São eles:

- **va\_list**: este tipo é usado como um parâmetro para as macros definidas na biblioteca **stdarg.h** para recuperar os parâmetros adicionais da função;
- **va\_start(lista, ultimo\_parametro)**: esta macro inicializa uma variável **lista**, do tipo **va\_list**, com as informações necessárias para recuperar os parâmetros adicionais, sendo **ultimo\_parametro** o último parâmetro declarado na função antes do "...";
- **va\_arg(lista, tipo\_dado)**: esta macro retorna o parâmetro atual contido na variável **lista**, do tipo **va\_list**, sob a forma do tipo informado em **tipo\_dado**. Em seguida, a macro move a variável **lista** para o próximo parâmetro, se este existir. Assim,  $x = va\_arg(lista, float)$  irá retornar para a variável  $x$  o valor do parâmetro atual em lista formatado para o tipo **float**;
- **va\_end(lista)**: esta macro deve ser executada antes da finalização da função (ou antes do comando **return**, se este existir). Seu objetivo é destruir a variável **lista**, do tipo **va\_list**, de modo apropriado.



Funções com uma quantidade variável de parâmetros devem ser usadas com moderação.

Não devemos utilizar constantemente esse tipo de função pois existe um potencial muito grande para que uma função projetada para se trabalhar com um tipo, seja usada com outro. Isso ocorre por que não existe definição de tipos na lista de parâmetros variável, apenas dentro da função na macro **va\_arg()**.



Funções com uma quantidade variável de parâmetros podem expor o programa a uma série de problemas de segurança baseada em tipo (**type-safety**).

Isso ocorre pois esse tipo de função não possui segurança baseada em tipo (**type-safety**). A função permite que se tente recuperar mais parâmetros do que foram passados, corrompendo assim o funcionamento do programa que poderá apresentar um comportamento inesperado. A função **printf()**, por exemplo, pode ser usada para ataques. Um usuário mal-intencionado pode usar os tipos de saída **%o** e **%x**, entre outros, para imprimir os dados de outras posições da memória

O exemplo abaixo apresenta uma função que retorna a soma de uma quantidade variável de parâmetros inteiros. Note que o primeiro parâmetro, **n**, é o número de parâmetros que virão em seguida:

#### Exemplo: soma de uma quantidade variável de parâmetros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdarg.h>
4 int soma_int(int n,...) {
5     va_list lista;
6     int i, s = 0;
7     va_start(lista, n);
8     for(i = 1; i <= n; i++)
9         s = s + va_arg(lista, int);
10    va_end(lista);
11    return s;
12 }
13 int main(){
14     int soma;
15     soma = soma_int(2,4,5);
16     printf("Soma 2 parametros: %d\n",soma);
17     soma = soma_int(3,4,5,6);
18     printf("Soma 3 parametros: %d\n",soma);
19     soma = soma_int(4,4,5,6,10);
20     printf("Soma 4 parametros: %d\n",soma);
21     system('pause');
22     return 0;
23 }
```