

MINIAULA DE ALGORITMOS

VARIÁVEIS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Variáveis: o que são?

- Uma variável representa uma **posição de memória**, possui um nome (**identificador**) para sua representação, **que é utilizada para guardar um valor** que pode ser modificado pelo programa.

- Exemplos:

pontuacao = 198273

vidas = 2

media = 3.8

status = 'V'

Variáveis: identificadores

- Os identificadores são os nomes das variáveis, programas, constantes, rotinas de um programa.
- De forma geral podemos dizer que um identificador está associado a uma posição de memória ou a um trecho do código.
- Todas as variáveis precisam de um identificador, o que auxilia o programador e facilita sua manipulação ao longo do programa.

Variáveis: identificadores

- Um identificador pode conter os seguintes elementos em sua formação:
 - *Caracteres, incluindo letras maiúsculas e minúsculas, números;*
 - *Caractere sublinhado;*
- O primeiro caractere deve ser sempre uma letra ou o caractere sublinhado;
- Não são permitidos espaços em branco e caracteres especiais;
- Não devem ser utilizadas palavras reservadas da linguagem nos identificadores, ou seja, palavras que pertençam a uma linguagem de programação.
 - *Exemplos: if, else, for, do, while, etc.*

Variáveis: tipo de dados

- Computadores possuem tabelas de alocação que contêm o nome da variável, seu tipo (para saber quantos bytes ocupará) e seu endereço inicial de armazenamento.
- Os tipos de dados mais comuns são:
 - *Numéricos: inteiros (int, long) ou reais (float, double)*
 - *Lógicos: falso ou verdadeiro (0 ou 1)*
 - *Literais: caracteres (char)*

Variáveis: tipos de dados e seus tamanhos

Tipo	Faixa de Valores	Tamanho (bytes)
char	-128 a 127	1 (8 bits)
int	-2.147.483.648 a 2.147.483.647	4 (32 bits)
long	-9223372036854775808 a 9223372036854775807	8 (64 bits)
float	3.4×10^{-38} a 3.4×10^{38}	4 (32 bits)
double	1.7×10^{-308} a 1.7×10^{308}	8 (64 bits)

Variáveis: declaração em linguagem C

- A forma geral de declaração é:

<tipo de dados> <nome do identificador>;

- Exemplos:

int numero;

float n2;

double num3;

char c;

Variáveis: exemplo completo

```
1. #include <stdio.h>
2. int main () {
3.     int a, b, c;
4.     a = 5;
5.     b = 5;
6.     c = a + b;
7.     printf ("A soma de %d + %d = %d !\n", a, b, c);
8.     return 0;
9. }
```


MINIAULA DE ALGORITMOS MEU PEQUENO PROGRAMA EM C

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Programas em C: estrutura básica

- Observe a seguinte estrutura de código em C:

<code>#include <stdio.h></code>	<<< Declaração de bibliotecas
<code>#include <math.h></code>	
<code>int main () {</code>	<<< Função principal
<code>...</code>	<<< Demais instruções, incluindo:
<code>...</code>	declarações de variáveis, entradas,
<code>...</code>	processamento, saídas
<code>return 0;</code>	<<< retorno da função principal
<code>}</code>	<<< Fechamento do bloco de comandos

Programas em C: bibliotecas

- São conjuntos de funções pré-definidas para auxiliar o programador!
- A biblioteca **stdio.h** contém as declarações das principais funções de entrada e saída, como o **printf()** e o **scanf()** !
- Um programa em C pode conter quantas bibliotecas forem necessárias. DICA: Use uma declaração **#include<...>** por linha!
- A biblioteca **math.h** também contém funções úteis, como:
 - *Cálculo de raiz quadrada: **sqrt()***
 - *Cálculo de potência: **pow()***
 - *Existem ainda funções que trabalham com trigonometria e logaritmos*

Programas em C: função **main()**

- Uma função é um trecho de código que, em geral, recebe dados, processa e devolve alguma informação baseada nos dados recebidos.
- Um programa em C pode conter uma ou mais funções!
- A função **main()**, também chamada de principal, é **obrigatória** para todos os programas em C.
- Essa função é chamada pelo sistema operacional quando se inicia a execução do programa.
- **DICA:** todas as funções possuem um par de parênteses associadas ao seu nome! Não se esqueça!

Programas em C: blocos de código

- Os blocos de código devem ser limitados por símbolos
- Em C, o delimitador são os caracteres { }
- Todo bloco está ligado à uma estrutura de código, como por exemplo uma função ou uma estrutura condicional.

```
int main() {  
    ...  
    if (x != 0) {  
        ...  
    }  
    ...  
}
```

MINIAULA DE ALGORITMOS

ENTRADAS E SAÍDAS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Entradas e Saídas: o que são?

- São os comandos que nos permitem enviar e receber dados de um determinado programa.
- Comandos de entrada: conjunto de comandos que permitem enviar dados ao programa. Podem ser dados lidos do teclado, de um arquivo, ou até mesmo de um sensor.
- Comandos de saída: conjunto de comandos que nos permitem receber informações do programa. Podem ser dados apresentados no terminal de comandos ou uma janela do monitor, dados enviados para um arquivo, para uma impressora ou até mesmo um atuador.


Saídas: declaração em linguagem C

- O comando de saída mais utilizado em C é o **printf**.

- Pode ser utilizado para mostrar mensagens simples:

```
printf("Olá Mundo!\n");
```

- Também pode ser utilizado para mostrar o conteúdo de variáveis:



The diagram illustrates the mapping of variables to format specifiers in the printf statement. Three colored arrows (yellow, blue, and red) originate from the variables 'a', 'b', and 'c' at the end of the code line and point to the corresponding '%d' placeholders in the format string. The yellow arrow connects 'a' to the first '%d', the blue arrow connects 'b' to the second '%d', and the red arrow connects 'c' to the third '%d'.

```
printf("A soma de %d + %d = %d !\n", a, b, c);
```



Atenção: a ordem das variáveis é importante no printf!

Saídas: caracteres de controle

- São máscaras de formatação que determinam o tipo de dados que será aplicado a uma mensagem, conforme a necessidade:

%d – para números inteiros (**int**);

%ld – para números inteiros (**long**);

%f – para números reais (**float**);

%lf – para números reais (**double**);

%c – um único caractere (**char**);

Entradas: declaração em linguagem C

- O comando de entrada mais utilizado em C é o **scanf**.
- É utilizado para armazenar digitados em variáveis. Na maioria das vezes está nesse formato:

```
scanf("<character de controle>", &<variável>);
```

- Exemplos:

```
int n1; float n2;
```



```
scanf("%d", &n1); // o & é importante, pois
```

```
scanf("%f", &n2); // indica o endereço da var!
```

Entradas e Saídas: exemplo completo

```
01. #include <stdio.h>
02. int main () {
03.     int a, b, c;
04.     printf("Primeiro número: ");
05.     scanf("%d", &a);
06.     printf("Segundo número: ");
07.     scanf("%d", &b);
08.     c = a + b;
09.     printf("A soma de %d + %d = %d !\n", a, b, c);
10.     return 0;
11. }
```

MINIAULA DE ALGORITMOS OPERADORES E COMANDOS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Operadores

- Símbolos utilizados com uma função pré-determinada:

Operador	Exemplo	Descrição
=	$x = y;$	O conteúdo de y é atribuído a x
+	$z = x + y$	O resultado da soma x + y é atribuído a z
-	$z = x - y;$	O resultado da subtração x - y é atribuído a z
*	$z = x * y;$	O resultado da multiplicação x * y é atribuído a z
/	$z = x / y;$	O resultado da divisão x / y é atribuído a z
%	$z = x \% y;$	O resto da divisão inteira x % y é atribuído a z

Operadores: divisão inteira

- O resultado de uma divisão depende dos envolvidos!



Se os dois operandos são inteiros, o resultado é inteiro!

- Exemplo:

```
int x = 3, y = 2;
```

```
float z;
```

```
z = x / y;
```

- Qual o valor armazenado em z?
 - A resposta é **1** (pois ocorre uma divisão inteira)

Operadores: divisão completa

- Caso queira uma divisão completa, pelo menos um dos dois operandos deve ser real (float ou double)
- Exemplo:

```
float x = 3, y = 2, z;
```

```
z = x / y;
```

- O valor de z agora é 1.5 (pois todos os números são reais)



```
int x = 3, y = 2;
```

```
float z = (float)x / y;
```

- O valor de z também é 1.5 (pois x foi convertido para número real)

Operadores compostos!

Operador	Exemplo	Descrição
+=	<code>x += y;</code>	Equivale a <code>x = x + y;</code>
-=	<code>x -= y;</code>	Equivale a <code>x = x - y;</code>
*=	<code>x *= y;</code>	Equivale a <code>x = x * y;</code>
/=	<code>x /= y;</code>	Equivale a <code>x = x / y;</code>
%=	<code>x %= y;</code>	Equivale a <code>x = x % y;</code>
++	<code>x++;</code>	Equivale a <code>x = x + 1;</code>
--	<code>y--;</code>	Equivale a <code>y = y - 1;</code>



Os operadores ++ e --

- Atuam de acordo com a posição onde são colocados!

++	y = x++;	Equivale a y = x; e depois x = x + 1;
++	y = ++x;	Equivale a x = x + 1; e depois y=x;

- Dados os códigos abaixo:

```
int x=2;  
y = x++;  
printf("%d e %d\n", y, x);
```

```
int x=2;  
y = ++x;  
printf("%d e %d\n", y, x);
```

- Saídas em tela:

Imprime: 2 e 3 na tela

Imprime: 3 e 3 na tela

Operadores Relacionais

- Comparam duas partes em uma expressão



Sempre retornam Verdadeiro ou Falso

Operador	Exemplo	Descrição
==	<code>x == y</code>	Verifica se o valor de x é igual ao de y
!=	<code>x != y</code>	Verifica se o valor de x é diferente ao de y
>	<code>x > y</code>	Verifica se o valor de x é maior ao de y
>=	<code>x >= y</code>	Verifica se o valor de x é maior ou igual ao de y
<	<code>x < y</code>	Verifica se o valor de x é menor ao de y
<=	<code>x <= y</code>	Verifica se o valor de x é menor ou igual ao de y

Operadores Lógicos

- Conectam duas expressões lógicas de acordo com as tabelas verdade!

Operador	Descrição
&&	E lógico
 	OU lógico
!	Não lógico

E1	E2	E1 && E2	E1 E2
V	V	V	V
V	F	F	V
F	V	F	V
F	F	F	F

- Exemplos: Verificar se x é positivo e par:

(x > 0) && (x % 2 == 0)

Só retorna V se x for positivo E par ao mesmo tempo!

E1	!E1
V	F
F	V

Caracteres de Escape

- São caracteres que modificam o efeito do seu sucessor.
- Em linguagem C é representado pela barra invertida!
- Exemplos importantes:
 - `\n` – *nova linha no terminal*
 - `\t` – *tabulação horizontal*
 - `\\` – *imprime o caractere barra invertida*
 - `\"` – *imprime o caractere aspas duplas*
 - `\'` – *imprime o caractere aspas simples*

Comentários no código!

- Comentários são mecanismos de documentação em seu código-fonte!
- Para uma única linha ou trecho, utiliza-se o símbolo `//` :

```
int x; // Somente esta parte é comentário!
```

- Para blocos, utilizam-se os símbolos `/*` e `*/` :

```
/*
```

Este é um esquema de comentário que pode ocupar várias linhas!!! Lembre-se: os comentários são desconsiderados pelo compilador e não interferem no código-fonte do programa!

```
*/
```

MINIAULA DE ALGORITMOS ESTRUTURAS CONDICIONAIS (IF)

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Condicionais: o que são?

- São estruturas no código que permitem a escolha de caminhos durante a execução
- Os caminhos são definidos de acordo com o resultado da avaliação de uma ou mais expressões lógicas
- Expressões lógicas tem apenas dois resultados possíveis:
 - *VERDADEIRO* ou *FALSO*

Condicionais: IF

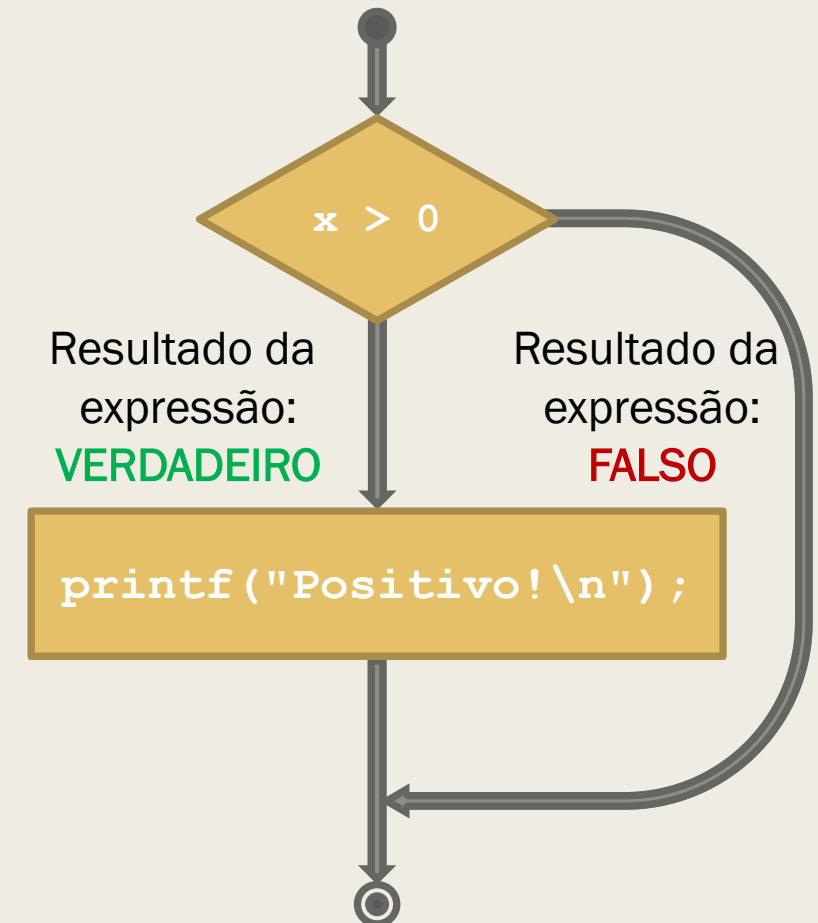
- Observe o código:

```
if (x>0) {  
    printf("Positivo!\n");  
}
```



A instrução *printf* será executada se, e somente se, o valor contido na **variável** *x* for, no momento da verificação, **estritamente maior que 0**.

- A figura abaixo ilustra o comando do código ao lado:



Condicionais: exemplo completo (IF)

```
01. #include <stdio.h>
02. int main () {
03.     int num;
04.     printf("Digite um número: ");
05.     scanf("%d", &num);
06.     if (num == 0) {
07.         printf("O número é zero!\n ");
08.     }
09.     return 0;
10. }
```

Condicionais: IF-ELSE

- A estrutura IF – ELSE permite a execução de comandos em dois blocos separados:
 - *Caso o resultado da expressão seja VERDEIRO, o bloco de instruções ligado ao IF será executado!*
 - *Caso o resultado da expressão seja FALSO, o bloco de instruções ligado ao ELSE será executado*
- Logo, ambos os resultados da expressão terão um bloco de comandos sendo executado!

Condicionais: IF-ELSE

- Observe o código:

```
if (x == 0) {  
    printf("É ZERO!");  
}  
else {  
    printf("Diferente de 0");  
}
```

- Neste caso, se e somente se, o valor de **x** for **igual a zero** o terminal mostrará a mensagem “É ZERO!”.
- Caso contrário, ou seja, se **x** contiver qualquer valor **diferente de zero**, a mensagem “Diferente de 0.” será impressa.



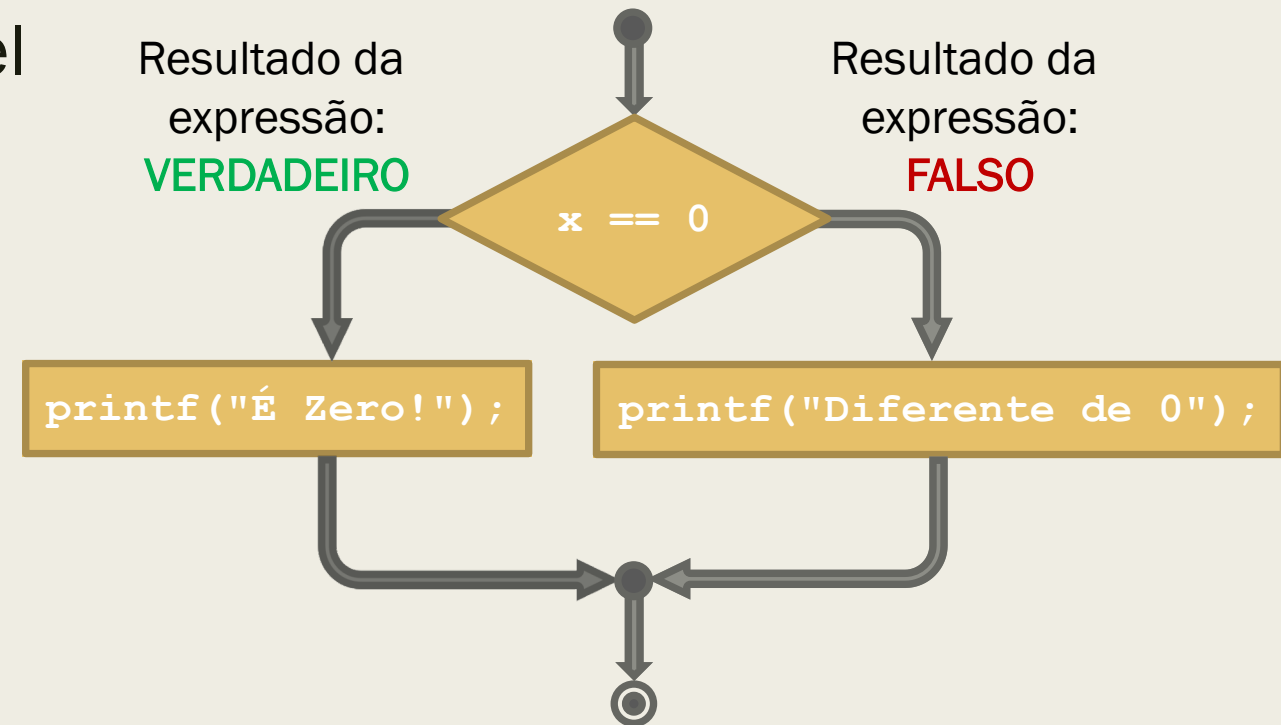
Importante: apenas uma das duas mensagens será impressa, pois um número ou é igual ou diferente de 0, e não pode ser os dois ao mesmo tempo!

Condicionais: exemplo completo (IF-ELSE)

```
01. #include <stdio.h>
02. int main () {
03.     int num;
04.     printf("Digite um número: ");
05.     scanf("%d", &num);
06.     if (num == 0) {
07.         printf("É ZERO!\n ");
08.     }
09.     else{
10.         printf("Diferente de 0!\n");
11.     }
12.     return 0;
13. }
```

Condicionais: IF-ELSE

- Na figura ao lado é possível observar que o código só pode seguir um caminho!
- Ou a expressão assume o valor **VERDADEIRO** na comparação, ou o valor **FALSO**, seguindo apenas um dos dois caminhos!



MINIAULA DE ALGORITMOS AINDA MAIS SOBRE IF-ELSE

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



IF-ELSE aninhados

- Uma estrutura IF-Else simples permite a criação de dois caminhos em seu programa:
 - O bloco de comandos do **IF**, executado quando a expressão lógica é **verdadeira**
 - O bloco de comando do **ELSE**, executado quando a expressão lógica do IF é **falsa**
- O uso do **ELSE** não é obrigatório!
 - Existem IF simples, sem o uso do ELSE. Entretanto:
 - Um ELSE deve sempre estar ligado a um comando IF

IF-ELSE aninhados

- Quando o programa demanda mais de duas possibilidades de escolhas por meio de uma estrutura condicional, podemos utilizar IF-ELSE aninhados:

```
if (num == 0) {  
    printf("É zero!");  
}  
else {  
    if (num < 0) {  
        printf("É Negativo!");  
    }  
    else {  
        printf("É positivo");  
    }  
}
```


Condicionais: exemplo (IF-ELSE aninhados)

```
01. #include <stdio.h>
02. int main () {
03.     int num;
04.     printf("Digite um número: ");
05.     scanf("%d", &num);
06.     if (num == 0) {
07.         printf("É zero!");
08.     }
09.     else {
10.         if (num < 0) {
11.             printf("É Negativo!");
12.         }
13.         else {
14.             printf("É positivo");
15.         }
16.     }
17.     return 0;
18. }
```

Cenário de uso 1:

O usuário digita o número 0

O comando IF verifica o valor de `num`

`num == 0` ? (VERDADEIRO)

Logo o bloco de comandos verde é executado

A instrução `printf("É zero!");` é executada

Como o bloco de instruções do IF foi executado, o bloco de instruções do ELSE é ignorado e o programa vai para a linha 17 e termina sua execução.

Condicionais: exemplo (IF-ELSE aninhados)

```
01. #include <stdio.h>
02. int main () {
03.     int num;
04.     printf("Digite um número: ");
05.     scanf("%d", &num);
06.     if (num == 0) {
07.         printf("É zero!");
08.     }
09.     else {
10.         if (num < 0) {
11.             printf("É Negativo!");
12.         }
13.         else {
14.             printf("É positivo");
15.         }
16.     }
17.     return 0;
18. }
```

Cenário de uso 2:

O usuário digita o número -1

O comando IF verifica o valor de num

num == 0 ? (FALSO)

Com o resultado da expressão FALSO o bloco do ELSE será executado!

Entretanto existe um novo IF para ser verificado

num < 0 ? (VERDADEIRO)

Logo o bloco de comandos amarelo é executado

A instrução printf("É negativo!"); é executada e o programa segue para a linha 17, pois o bloco azul será ignorado!

Condicionais: exemplo (IF-ELSE aninhados)

```
01. #include <stdio.h>
02. int main () {
03.     int num;
04.     printf("Digite um número: ");
05.     scanf("%d", &num);
06.     if (num == 0) {
07.         printf("É zero!");
08.     }
09.     else {
10.         if (num < 0) {
11.             printf("É Negativo!");
12.         }
13.         else {
14.             printf("É positivo");
15.         }
16.     }
17.     return 0;
18. }
```

Cenário de uso 3:

O usuário digita o número 7

O comando IF verifica o valor de **num**

num == 0 ? (FALSO)

Com o resultado da expressão **FALSO** o bloco do **ELSE** será executado!

Entretanto existe um novo **IF** para ser verificado

num < 0 ? (FALSO)

O bloco de comandos do **IF** é ignorado e o bloco do **ELSE** será executado

A instrução `printf("É positivo!");` é executada e o programa segue para a linha 17, terminado sua execução

IF-ELSE aninhados: pratique!

- Uma empresa gostaria de calcular o valor de novo salário de seus funcionários de acordo com a tabela:

Salário	% de aumento
Até R\$ 1.200,00 (menor ou igual)	10
De R\$ 1.200,00 a R\$ 2.500,00 (menor ou igual)	8
Acima de R\$ 2.500,00	5

- É importante entender o problema!
- Uma análise inicial revela 3 faixa de aumento, e portanto três possíveis caminhos de execução em nosso programa!

IF-ELSE aninhados: pratique!

Salários <= R\$ 1.200,00

Salários > R\$ 1.200,00

Salários <= R\$ 2.500,00

Salários > R\$ 2.500,00

```
if (salario <= 1200) {  
    //cálculo do salário  
}  
else if (salario <=2500) {  
    //cálculo do salário  
}  
else {  
    //cálculo do salário  
}
```

- Atenção: ao usarmos uma expressão **else if**, o uso de chaves no bloco do else é opcional!

<< Veja o exemplo ao lado!

MINIAULA DE ALGORITMOS ESTRUTURAS CONDICIONAIS (SWITCH)

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Condicionais: SWITCH – CASE

- Algumas situações possuem opções mutuamente exclusivas, isto é, se uma situação for executada, as demais não serão.
- Por exemplo, verificar se uma variável contém um dos valores estabelecidos em um conjunto:

x é igual a um desses valores {2, 3, 7, 11, 33}?

- Quando for este o caso, um comando condicional como o Switch-case é o mais indicado.

Condicionais: exemplo Switch

```
01. #include <stdio.h>
02. int main () {
03.     int x;
04.     printf("Digite um número: ");
05.     scanf("%d", &x);
06.     switch(x) {
07.         case 2: printf("O número é 2!\n");
08.             break;
09.         case 3: printf("O número é 3!\n");
10.             break;
11.         case 7: printf("O número é 7!\n");
12.             break;
13.         case 11: printf("O número é 11!\n");
14.             break;
15.         case 33: printf("O número é 33!\n");
16.             break;
17.         default: printf("O num não está no conjunto!\n");
18.     }
19.     return 0;
20. }
```


Condicionais: SWITCH – CASE

- No exemplo anterior, a igualdade da variável é verificada individualmente com cada valor nos CASE.
- Se o valor da variável for igual o valor do CASE, o bloco de instruções é executado.
- A estrutura switch-case funciona para comparar variáveis de tipo inteiro (INT, LONG) e do tipo literais (CHAR).
- O comando break é importante para finalizar propositamente a execução do bloco selecionado.
- Caso nenhuma alternativa seja verdadeira, as instruções do caso DEFAULT serão executadas!

```

01 #include <stdio.h>
02 int main () {
03     int x;
04     printf("Digite um número: ");
05     scanf("%d", &x);
06     switch(x) {
07         case 2: printf("O número é 2!\n");
08                 break;
09         case 3: printf("O número é 3!\n");
10                 break;
11         case 7: printf("O número é 7!\n");
12                 break;
13         case 11: printf("O número é 11!\n");
14                  break;
15         case 33: printf("O número é 33!\n");
16                  break;
17         default: printf("O num não está no conjunto!\n");
18     }
19     return 0;
20 }

```

Cenário de uso 1:

O usuário digita o número 7

O comando SWITCH inicia sua execução

A variável x é comparada com os valores

x == 2 ? (**FALSO**)

x == 3 ? (**FALSO**)

x == 7 ? (**VERDADEIRO**)

A instrução printf("O número é 7!\n");
é executada

A instrução break; é executada e
direciona o programa para a **linha 18**

```

01 #include <stdio.h>
02 int main () {
03     int x;
04     printf("Digite um número: ");
05     scanf("%d", &x);
06     switch(x) {
07         case 2: printf("O número é 2!\n");
08                 break;
09         case 3: printf("O número é 3!\n");
10                 break;
11         case 7: printf("O número é 7!\n");
12                 break;
13         case 11: printf("O número é 11!\n");
14                  break;
15         case 33: printf("O número é 33!\n");
16                  break;
17         default: printf("O num não está no
18                    conjunto!\n");
19     }
20     return 0;
21 }

```

Cenário de uso 2:

O usuário digita o número 5

O comando SWITCH inicia sua execução

A variável x é comparada com os valores

x == 2 ? (**FALSO**)

x == 3 ? (**FALSO**)

x == 7 ? (**FALSO**)

x == 11 ? (**FALSO**)

x == 33 ? (**FALSO**)

TODAS as comparações resultaram em
FALSO!

Nesse caso, e somente nesse, a opção
DEFAULT é selecionada e seu printf
executado!

O programa continua então na **linha 18**

Condicionais: exemplo Switch

```
01. #include <stdio.h>
02. int main () {
03.     char ch;
04.     printf("Digite um caractere: ");
05.     scanf("%c", &ch);
06.     switch(ch) {
07.         case 'a': printf("É a letra A!\n");
08.             break;
09.         case 'b': printf("É a letra B!\n");
10.             break;
11.         case 'c': printf("É a letra C!\n");
12.             break;
13.         case '!': printf("É a exclamação!\n");
14.             break;
15.         case '$': printf("É o cifrão!\n");
16.             break;
17.         default: printf("O char não está no conjunto!\n");
18.     }
19.     return 0;
20. }
```

Condicionais: dicas Switch-Case!

- Lembre-se: somente variáveis dos tipos **INT**, **LONG** e **CHAR**!
- As comparações são **automáticas**, por conta da estrutura!
- O comando **SWITCH** contém **apenas o nome da variável**!
- Para cada opção não se esqueça da palavra **CASE**!
- O comando **break** é utilizado para encerrar um bloco!

MINIAULA DE ALGORITMOS ESTRUTURAS DE REPETIÇÃO

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Repetição: o que é?

- Uma estrutura de repetição é utilizada quando um trecho do algoritmo ou até mesmo o algoritmo inteiro precisa ser repetido.
 - *Exemplo: imprima os números de 1 a 1000 na tela!*
 - *O comando é sempre o mesmo: `printf("%d \n", num);`*
 - *O que muda é o valor da variável **num**!*
- O número de repetições pode ser fixo ou estar atrelado a uma condição. Assim, existem diferentes estruturas para cada uma das situações.

Estruturas de Repetição

- As estruturas de repetição devem ser bem definidas!
 - *Um número limitado e preciso de vezes para executar*
Ex.: imprima os número de 1 a 1000 na tela!
 - *Executa enquanto uma condição é verdadeira*
Ex: enquanto o número digitado for diferente de 0 imprima: Não!
- Nos dois casos as estruturas vão executar um número **finito** de vezes!
 - *No primeiro, sabemos de antemão quantas vezes*
 - *No segundo embora finito, o número é indeterminado*

Iteração (s.f. ato de iterar; repetição)

- Chamamos de iteração a cada rodada de execução da estrutura de repetição
- A execução do bloco de instruções depende do critério de parada!
- Em cada iteração:
 - *Verifica-se se o critério de parada foi atingido*
 - *Expressão lógica retorna VERDADEIRO:*
 - Executa o bloco de instruções da repetição
 - *Expressão lógica retorna FALSO:*
 - Para a execução, saindo da repetição

As 3 Estruturas de Repetição em C

■ Para-faça: FOR

```
int i;  
for(i=1; i<=1000; i++){  
    //instruções  
}
```

■ Enquanto: WHILE

```
int i=1;  
while(i<=1000){  
    //instruções  
    i++;  
}
```

■ Faça-enquanto: DO-WHILE

```
int i=1;  
do{  
    //instruções  
    i++;  
} while(i<=1000);
```

Variável de controle

- Em geral, é a variável (ou variáveis) responsável pelo controle da execução e da parada da repetição
- Na execução podem controlar a quantidade de vezes ou algum valor pertinente à repetição
- Na parada, ao atingir um determinado valor causam o término da repetição

Repetição: exemplo completo (FOR)

Exercício: Imprima os números de 1 a 1000 na tela!

```
01. #include <stdio.h>
02. int main () {
03.     int i;
04.     for(i=1; i<=1000; i++) {
05.         printf("%d \n", i);
06.     }
07.     return 0;
08. }
```

Repetição: exemplo completo (WHILE)

Exercício: Imprima os números de 1 a 1000 na tela!

```
01. #include <stdio.h>
02. int main () {
03.     int i;
04.     i=1;
05.     while (i<=1000) {
06.         printf("%d \n", i);
07.         i++;
08.     }
09.     return 0;
10. }
```

Repetição: exemplo completo (DO-WHILE)

Exercício: Imprima os números de 1 a 1000 na tela!

```
01. #include <stdio.h>
02. int main () {
03.     int i;
04.     i=1;
05.     do{
06.         printf("%d \n", i);
07.         i++;
08.     }while (i<=1000);
09.     return 0;
10. }
```

MINIAULA DE ALGORITMOS ESTRUTURA DE REPETIÇÃO FOR

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



A estrutura FOR

- O FOR é geralmente utilizado quando já se sabe de antemão quantas vezes sua repetição será executada.
 - *Não é restrito à apenas esse tipo de solução!*
- O comando agrupa as três partes que ilustram a repetição em uma única instrução.
- A partir de um valor inicial, segue uma contagem em incremento ou decremento determinado pelo usuário, até um valor limite, onde ocorre o término da repetição.

Estrutura geral: FOR

```
for( i=1 ; i <= 1000 ; i++ ) {  
    //bloco de instruções  
}
```

- A primeira parte **atribui um valor inicial à variável de controle i**.
 - *Essa atribuição é feita somente na primeira iteração!*
- A segunda parte corresponde a **uma expressão lógica que, quando retornar falso, determinará o fim da repetição**.
- A terceira parte é responsável por **alterar o valor da variável de controle i (incremento ou decremento)** com o objetivo de, em algum momento, fazer com que a expressão lógica retorne valor falso, e pare a repetição.

Estrutura geral: FOR

```
for( i=1 ; i <= 1000 ; i++ ) {  
    //bloco de instruções  
}
```

- A variável **i** é inicializada na entrada do comando FOR (**i=1**)
- Enquanto a expressão lógica for verdadeira (**i<=1000**), o bloco de instruções será executado.
- A cada iteração, a variável de controle **i** é incrementada (**i++**), e a expressão lógica verificada novamente:
 - *Se for verdadeira, a repetição continua*
 - *Caso seja falsa, a repetição para.*

Repetição FOR: Dicas

- O símbolo de separação das três partes é o “;”
- O valor da variável de controle deve sempre ser inicializado!
 - *Pode ser feito dentro ou fora (antes) do FOR*

```
int i=1;  
  
for(; i<=1000; i++){  
    // instruções  
}
```

É IGUAL A:

```
int i;  
  
for(i=1; i<=1000; i++){  
    // instruções  
}
```

- A expressão lógica deve ser factível!

Teste de mesa: FOR

Exercício: Imprima os números de 1 a 5 na tela!

```
01. #include <stdio.h>
02. int main () {
03.     int i;
04.     for(i=1; i<=5; i++) {
05.         printf("%d \n", i);
06.     }
07.     return 0;
08. }
```

Iteração	Valor de i	i<=5 ?	Tela
1 (Início)	1	V	1
2	2	V	2
3	3	V	3
4	4	V	4
5	5	V	5
6	6	F	

MINIAULA DE ALGORITMOS

ESTRUTURA DE REPETIÇÃO

WHILE

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



A estrutura WHILE

- O WHILE é geralmente utilizado quando não se sabe antecipadamente quantas vezes sua repetição será executada.
 - *Também não é restrito à apenas esse tipo de solução!*
- O comando demanda que a(s) variável(eis) de controle sejam inicializadas e modificadas **manualmente!**
- Assim como no FOR, a partir de um valor inicial, segue uma contagem em incremento ou decremento determinado pelo usuário, até um valor limite, onde ocorre o término da repetição.

Estrutura geral: WHILE

```
i=1;  
while(i <= 1000) {  
    //bloco de instruções  
    i++;  
}
```

- A **variável de controle i** deve ser inicializada antes do começo da execução do WHILE!
- A **expressão lógica** entre **parentese** determinará o **fim da repetição, quando** retornar **falso**.
- A alteração **da variável de controle i (incremento ou decremento)** deve estar presente em algum ponto **no bloco de instruções!**

Estrutura geral: WHILE

```
i=1;  
while(i <= 1000) {  
    //bloco de instruções  
    i++;  
}
```

- A variável **i** é inicializada antes do comando for (**i=1**)
- Enquanto a expressão lógica for verdadeira (**i<=1000**), o bloco de instruções será executado.
- A cada iteração, a variável de controle **i** é incrementada (**i++**) pois o bloco de instruções foi executado, e a expressão lógica é verificada novamente:
 - *Se for verdadeira, a repetição continua*
 - *Caso seja falsa, a repetição para.*

Teste de mesa: WHILE

Exercício: Imprima os números de 1 a 5 na tela!

```
01. #include <stdio.h>
02. int main () {
03.     int i;
04.     i=1;
05.     while (i<=5) {
06.         printf("%d \n", i);
07.         i++;
08.     }
09.     return 0;
10. }
```

Iteração	Valor de i	i<=5 ?	Tela
	1		
1	1	V	1
2	2	V	2
3	3	V	3
4	4	V	4
5	5	V	5
6	6	F	

MINIAULA DE ALGORITMOS ESTRUTURA DE REPETIÇÃO DO-WHILE

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



A estrutura DO-WHILE

- O DO-WHILE é muito similar ao WHILE, com apenas uma exceção: a quantidade de vezes que a repetição executa!



*O bloco de instruções **SEMPRE** será executado uma vez, não importando o retorno da expressão lógica!*

- Assim como no WHILE, o comando demanda que a(s) variável(eis) de controle sejam inicializadas e modificadas explicitamente!

Estrutura geral: DO-WHILE

```
i=1;  
do{  
    //bloco de instruções  
    i++;  
}while(i <= 1000);
```

- A variável **i** é inicializada antes do comando for (**i=1**)
- O bloco de instruções é executado uma vez e já causa um incremento na variável de controle (**i++**)!
- Enquanto a expressão lógica for verdadeira (**i<=1000**), o bloco de instruções será executado.
- Ao final da iteração a expressão lógica é verificada novamente:
 - Se for verdadeira, a repetição continua
 - Caso seja falsa, a repetição para.

Teste de mesa: DO-WHILE

Exercício: Imprima os números de 1 a 5 na tela!

```
01. #include <stdio.h>
02. int main () {
03.     int i;
04.     i=1;
05.     do{
06.         printf("%d \n", i);
07.         i++;
08.     }while (i<=5);
09.     return 0;
10. }
```

Iteração	Tela	Valor de i	i<=5 ?
		1	
1	1	2	V
2	2	3	V
3	3	4	V
4	4	5	V
5	5	6	F

MINIAULA DE ALGORITMOS

VETORES

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Vetores: o que são?

- Variáveis compostas, homogêneas e unidimensionais
- Agregam um conjunto de variáveis:
 - *de mesmo tipo*
 - *com um mesmo nome*
 - *com índices (posições) diferentes*
 - *alocadas sequencialmente na memória*
- Cada índice do vetor é acessado como uma variável comum

Vetores: o que são?

- Exemplo: um vetor de números inteiros com 5 posições

```
int v[5];           // Declaração do vetor
v[0] = 5;           // índice 0 recebe o valor 5
v[1] = 7;           // índice 1 recebe o valor 7
v[2] = 12;          // índice 2 recebe o valor 12
v[3] = -1;          // índice 3 recebe o valor -1
v[4] = 0;           // índice 4 recebe o valor 0
```

v	5	7	12	-1	0
[]	0	1	2	3	4

Vetores: índices

- Permitem o acesso direto a cada posição do vetor
- São representados por valores inteiros e sequenciais



Sempre começam no índice ZERO!

- Representam a posição em memória da variável dentro do vetor
 - *Deslocamento dos endereços (tamanho em bytes) a partir da posição inicial*

Vetores: declaração

- Similar às variáveis simples, apenas adicionando os “[]”
- Podem ser de qualquer tipo:
`int v[10]; // vetor de inteiro com 10 posições`
`float v1[15]; // vetor de float com 15 posições`
`char str[50]; // vetor de char com 50 posições`
- Atenção: os vetores do tipo CHAR são também conhecidos como STRINGS. São um tipo especial e serão estudados em aula específica.

Vetores: manipulação

- Nas declarações, o vetor pode ser inicializado em sua totalidade:

```
int v[5] = {5, 7, 12, -1, 0};
```

índices: 0 1 2 3 4

- No caso acima, o vetor será declarado e inicializado com os números nas respectivas posições. A quantidade de elementos do conjunto deve ser do mesmo tamanho do vetor.

```
int vet[10] = {0};
```

- Nesse caso, todas as posições do vetor serão inicializadas com 0. Esse mecanismo só funciona para o número zero.

Vetores: manipulação

- As posições do vetor recebem valores como variáveis simples:

- *Atribuição direta:* `v[0] = 5;`
- *Comando de entrada:* `scanf ("%d", &v[0]);`



ATENÇÃO: as posições do vetor devem ser manipuladas **INDIVIDUALMENTE**, uma por vez, mesmo que em uma estrutura de repetição!

Vetores: manipulação + repetição

```
1. #include <stdio.h>
2. int main() {
3.     int i, v[5];
4.     for(i=0; i<5; i++) {
5.         printf("Número: ");
6.         scanf("%d", &v[i]);
7.     }
8.     return 0;
9. }
```

Exemplo de execução:

Número: 4 $i == 0 \rightarrow v[0] = 4;$

Número: 13 $i == 1 \rightarrow v[1] = 13;$

Número: -8 $i == 2 \rightarrow v[2] = -8;$

Número: 7 $i == 3 \rightarrow v[3] = 7;$

Número: 25 $i == 4 \rightarrow v[4] = 25;$

v	4	13	-8	7	25
[]	0	1	2	3	4

Vetores: manipulação + repetição

```
1. #include <stdio.h>
2. int main() {
3.     int i, v[5] = {5,4,7,2,1};
4.     for(i=0; i<5; i++) {
5.         printf("V[%d]: %d\n", i, v[i]);
6.     }
7.     return 0;
8. }
```

Exemplo de execução:

V[0]: 5

V[1]: 4

V[2]: 7

V[3]: 2

V[4]: 1

v	5	4	7	2	1
[]	0	1	2	3	4

Vetores: dicas

- Índices sempre começam em ZERO!
 - *Lembre-se do limite da variável contadora na repetição!*
- Manipulação individual das posições!
 - *Todas as posições manipuladas uma por vez!*
 - *Não se esqueça do & no scanf! (`scanf("%d", &v[i]);`)*
- Use como uma variável comum:
 - `if (v[i] > 0) { ... }`
 - `switch (v[i]) { ... }`
 - `soma = soma + v[i];`

MINIAULA DE ALGORITMOS STRINGS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Strings: o que são?

- São um tipo específico de vetor de caracteres
 - *Conhecidos como cadeias de caracteres*
- São declarados com o tipo CHAR
 - *char str[50];*
- Utiliza de um caractere a mais na quantidade para o controle ('\\0')
 - *Esse caractere é inserido automaticamente durante a leitura e utilizado por diversas funções para localizar o fim do preenchimento de uma cadeia.*



*Um conjunto de funções predefinidas está disponível na biblioteca **string.h***

Strings: declaração

- Utilizamos a mesma forma dos vetores:

```
char str[11];
```



No caso acima temos 11 posições na cadeia str, mas devemos ocupar no máximo 10, pois reservamos uma casa para o caractere de controle!

str	A	l	g	o	r	i	t	m	o	s	\0
	0	1	2	3	4	5	6	7	8	9	10
str	P	r	o	g	r	a	m	a	\0		
	0	1	2	3	4	5	6	7	8	9	10

Strings: inicialização (declaração)

- As cadeias podem ser declaradas automaticamente de acordo com o tamanho de uma palavra:
 - `char sistema[] = {'L', 'i', 'n', 'u', 'x'};`
 - `char sistema[] = "Linux";`
- As duas declarações acima têm o mesmo efeito, mas a segunda é mais utilizada por ser mais fácil.
- Note que nesse caso, não existe a necessidade de informar o tamanho da cadeia.
 - *O próprio compilador vai se encarregar de determinar o tamanho necessário para a cadeia.*

sistema
[]

L	i	n	u	x	\0
0	1	2	3	4	5

Strings: inicialização (leitura)

- Podemos inicializar uma cadeia com o scanf:

```
char str[50];
```

```
scanf("%s", str);
```

- *Atenção: nesse uso da função scanf, o comando só vai ler até o primeiro espaço da frase e portanto múltiplas palavras não serão armazenadas em str!*



```
scanf("%[^\n]", str);
```

- *Para resolver o problema, utilizamos uma expressão regular, em que serão lidos caracteres até que seja pressionada a tecla ENTER (\n).*

Strings: inicialização (leitura)

- Quando utilizando o scanf com cadeias, não devemos utilizar o caractere "&" e nem o par de caracteres "[]", apenas o **nome da string**!

```
scanf ("%[^\\n]", str);
```

- *Diferentemente dos vetores numéricos, aqui não vamos preencher as posições uma a uma manualmente*
- A função scanf vai se encarregar de colocar cada caractere no seu índice respectivo e colocar o marcador **\\0** ao final do preenchimento!

Strings: atribuição

- Por se tratar de um vetor, a atribuição deve ser feita posição a posição.
 - *Mas como estamos trabalhando com palavras, existe uma função que faz a cópia da palavra completa de uma vez só para uma cadeia, a STRCPY.*

```
char strA[10];
```

```
strcpy(strA, "Teste123");
```

- Nesse exemplo a literal Teste123 será copiada e armazenada na cadeia strA, sobrescrevendo seu conteúdo anterior, se houver.
 - *O marcador de final (\0) será adicionado automaticamente.*

Strings: cópia

- A função STRCPY também pode ser utilizada para copiar valores de uma cadeia para outra.

```
char strA[10];  
char strB[] = "Teste123";  
strcpy(strA, strB);
```

- Nesse exemplo o conteúdo da string strB será copiado para a strA, sobrescrevendo qualquer caractere nas posições existentes.
 - *Só serão sobrescritos os caracteres necessários para copiar a palavra completa e o marcador final \0.*

Strings: concatenação

- Para juntar duas strings existe o comando **STRCAT**:

```
strcat(str1, str2) ;
```

- A função **STRCAT** concatena (junta) a **str2** na **str1**. Lembre-se: é preciso ter espaço suficiente na **str1**!

```
strncat(str1, str2, n) ;
```

- A função **STRNCAT** concatena apenas os **n** primeiros caracteres da string **str2** na **str1**.
- Ambas as funções estão presentes na biblioteca **string.h**!

Strings: comparação

- A função que compara cadeias retorna um número inteiro. É a função **STRCMP**.

```
int resultado = strcmp(str1, str2);
```

- A variável **resultado** vai conter:
 - Zero se as duas cadeias forem iguais;
 - Um número negativo se a *str1* for alfabeticamente menor que *str2*;
 - Um número positivo se a *str1* for alfabeticamente maior que *str2*;

Strings: comparação

- Também podem ser comparados apenas os primeiros caracteres de duas cadeias usando a função **STRNCMP**:

```
int resultado = strncmp(str1, str2, n);
```

- Assim como na função anterior, quando os *n* primeiros caracteres forem iguais, o retorno será 0.
- Caso contrário um número positivo ou negativo será retornado.

Strings: tamanho

- Para descobrir quantos caracteres estão ocupados em uma cadeia, usamos a função STRLEN.

```
char str[11];
```

```
strcpy(str, "Programa");
```

```
int tamanho = strlen(str);
```

- Qual o valor armazenado em tamanho?
 - A resposta é 8 (apenas os caracteres ocupados)!

str	P	r	o	g	r	a	m	a	\0		
	0	1	2	3	4	5	6	7	8	9	10

Strings: funções adicionais

- **STRREV** – inverte todos os caracteres da cadeia
 - `strrev(str);`
- **TOLOWER** – converte um caractere da cadeia para minúsculo. Utilize em um laço de repetição para converter a cadeia completa!

```
for(i=0; i<strlen(str); i++){  
    str[i] = tolower(str[i]);  
}
```
- **TOUPPER** – converte um caractere da cadeia para maiúsculo. Utilize em um laço de repetição para converter a cadeia completa!

```
for(i=0; i<strlen(str); i++){  
    str[i] = toupper(str[i]);  
}
```

MINIAULA DE ALGORITMOS MATRIZES

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Matrizes: o que são?

- Variáveis compostas, homogêneas e multidimensionais
 - *Trabalham com DUAS ou mais dimensões!*
- Agregam um conjunto de variáveis:
 - *de mesmo tipo*
 - *com um mesmo nome*
 - *com índices (posições) diferentes*
 - *alocadas na memória (vetor de vetores)*
- Cada índice da matriz é acessado como uma variável comum

Matrizes: o que são?

- Exemplo: uma matriz de números inteiros com
2 linhas e 3 colunas

```
int m[2][3];    // Declaração da matriz
m[0][0] = 1;    // matriz L0,C0 recebe o valor 1
m[0][1] = 3;    // matriz L0,C1 recebe o valor 3
m[0][2] = 5;    // matriz L0,C2 recebe o valor 5
m[1][0] = 1;    // matriz L1,C0 recebe o valor 2
m[1][1] = 3;    // matriz L1,C1 recebe o valor 4
m[1][2] = 5;    // matriz L1,C2 recebe o valor 5
```

Matrizes: ilustrando o exemplo anterior

- Matriz m com 2 linhas e 3 colunas

	0	1	2
0	1	3	5
1	2	4	6

- Referenciar os elementos na forma:

$m[\text{ÍNDICE DA LINHA}][\text{ÍNDICE DA COLUNA}]$:

$m[0][0]$, $m[0][1]$, $m[0][2]$, ...

Matrizes: índices

- Assim como nos vetores, permitem o acesso direto a cada posição do vetor
- São representados por valores inteiros e sequenciais, e sempre vem no mínimo em pares (linha e coluna)!



Sempre começam no índice ZERO!

- Linhas e colunas manipuladas individualmente!

Matrizes: declaração

- Similar às variáveis simples, apenas adicionando os “[]”, um para cada dimensão!
- Podem ser de qualquer tipo:

```
int m[2][3]; // matriz de inteiros com 2 linhas  
e 3 colunas
```

```
float m2[5][7]; // matriz de float com 5 linhas  
e 7 colunas
```

```
char velha[3][3]; //matriz de char com 3 linhas  
e 3 colunas. Poderia ser utilizada para  
controlar o jogo da velha ('X' ou 'O')
```

Matrizes: manipulação

- Nas declarações, a matriz pode ser inicializada em sua totalidade:

```
linhas:      0      1
int m[2][3] = { {1,3,5}, {2,4,6} };
colunas:    0  1  2      0  1  2
```

- No caso acima, a matriz será declarado e inicializada com os números nas respectivas posições. Devemos ter uma **quantidade de conjuntos igual ao número de linhas**. Cada conjunto deve ter a **quantidade** de elementos igual ao **número de colunas da matriz**.

```
int mat[128][256] = {0};
```

- Nesse caso, todas as posições da matriz serão inicializadas com 0. Esse mecanismo só funciona para o número zero.

Matrizes: manipulação

- As posições da matriz recebem valores como variáveis simples:
 - *Atribuição direta:* `m[0][2] = 5;`
 - *Comando de entrada:* `scanf ("%d", &m[0][2]);`



ATENÇÃO: *as posições da matriz devem ser manipuladas INDIVIDUALMENTE, uma por vez, mesmo que em uma estrutura de repetição!*

Matrizes: manipulação + repetição

```
01. #include <stdio.h>
02. int main() {
03.     int i, j;
04.     int m[2][3];
05. → for(i=0; i<2; i++) {
06.         for(j=0; j<3; j++) {
07.             printf("Número: ");
08.             scanf("%d", &m[i][j]);
09.         }
10.     }
11.     return 0;
12. }
```

i	j	m[i][j]
0	0	Número: 1
0	1	Número: 3
0	2	Número: 5
1	0	Número: 2
1	1	Número: 4
1	2	Número: 6

	0	1	2
0	1	3	5
1	2	4	6

Matrizes: inversão dos FOR!

```
01. #include <stdio.h>
02. int main() {
03.     int i, j;
04.     int m[2][3];
06. → for(j=0; j<3; j++) {
05.         for(i=0; i<2; i++) {
07.             printf("Número: ");
08.             scanf("%d", &m[i][j]);
09.         }
10.     }
11.     return 0;
12. }
```

i	j	m[i][j]
0	0	Número: 1
1	0	Número: 3
0	1	Número: 5
1	1	Número: 2
0	2	Número: 4
1	2	Número: 6

	0	1	2
0	1	5	4
1	3	2	6

Matrizes: manipulação + repetição

```
01. #include <stdio.h>
02. int main() {
03.     int i, j, m[2][3]={{1,3,5},{2,4,6}};
04.     for(i=0; i<2; i++){
05.         for(j=0; j<3; j++){
06.             printf("M[%d][%d]: %d",
07.                 i, j, m[i][j]);
08.         }
09.     return 0;
10. }
```

i	j	Tela
0	0	M[0][0]: 1
0	1	M[0][1]: 3
0	2	M[0][2]: 5
1	0	M[1][0]: 2
1	1	M[1][1]: 4
1	2	M[1][2]: 6

	0	1	2
0	1	3	5
1	2	4	6

Matrizes: impressão em formato

```
01. #include <stdio.h>
02. int main() {
03.     int i, j;
04.     int m[2][3]={{1,3,5},{2,4,6}};
05.     for(i=0; i<2; i++) {
06.         for(j=0; j<3; j++) {
07.             printf("%d  ", m[i][j]);
08.         }
09.         printf("\n");
10.     }
11.     return 0;
12. }
```

Saída em Tela:

```
1   3   5
2   4   6
```

	0	1	2
0	1	3	5
1	2	4	6

Matrizes: dicas

- Índices sempre começam em ZERO!
 - *Lembre-se do limite das variáveis contadora na repetição!*
 - *Associação facilitada: $i \rightarrow$ linhas e $j \rightarrow$ colunas*
 - *Jamais inverta os índices i e j na instrução da matriz!*
- Manipulação individual das posições!
 - *Todas as posições manipuladas uma por vez!*
 - *Não se esqueça do $\&$ no scanf! (`scanf("%d", &m[i][j]);`)*
- Use como uma variável comum:
 - `if (m[i][j] > 0) { ... }`
 - `switch(m[i][j]) { ... }`
 - `soma = soma + m[i][j];`

MINIAULA DE ALGORITMOS SUBRPOGRAMAÇÃO

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Subprogramação: o que é?

- Mecanismo de organizar seu código!
- Maneira de lidar com o aumento da complexidade
- Realizado através da criação de funções
- Tarefas repetidas ao longo do código são organizadas e centralizadas em um ponto

Funções: o que são?

- As funções são trechos de código com um objetivo específico
- Podem ser chamadas ao longo do programa, conforme a necessidade
- Como desenvolver:
 - *Procurar por trechos do algoritmo com ações específicas*
 - *Isolar e atribuir um nome para esse trecho*
 - Definição das informações recebidas/devolvidas
 - *Utilizar esse trecho de código através da função gerada*

Funções: exemplificando

- Observe o algoritmo ao lado
- Uma mesma solução (X) desenvolvida foi repetida em dois pontos do algoritmo...
- E se a solução estivesse presente em muitos pontos do algoritmo?
- Como fazer e controlar modificações uniformes?

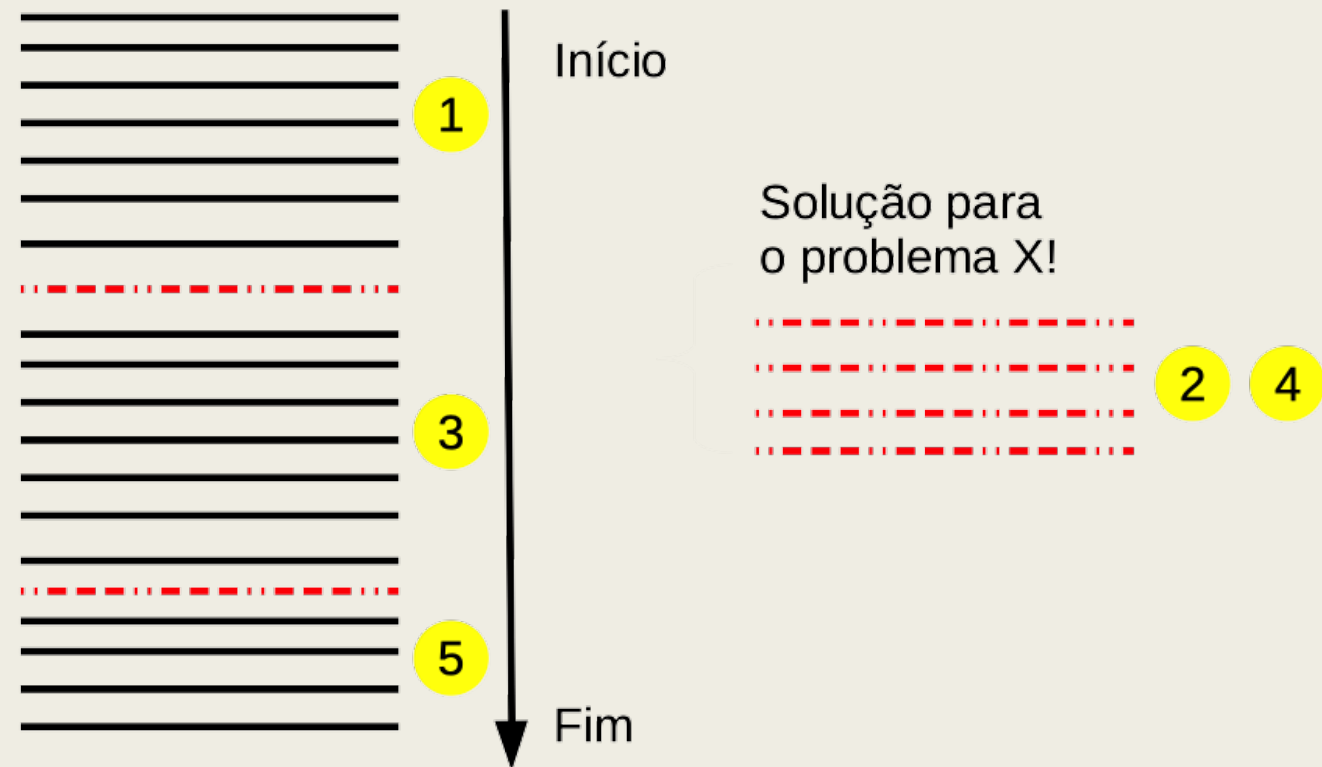
Algoritmo sem subprogramação (funções)



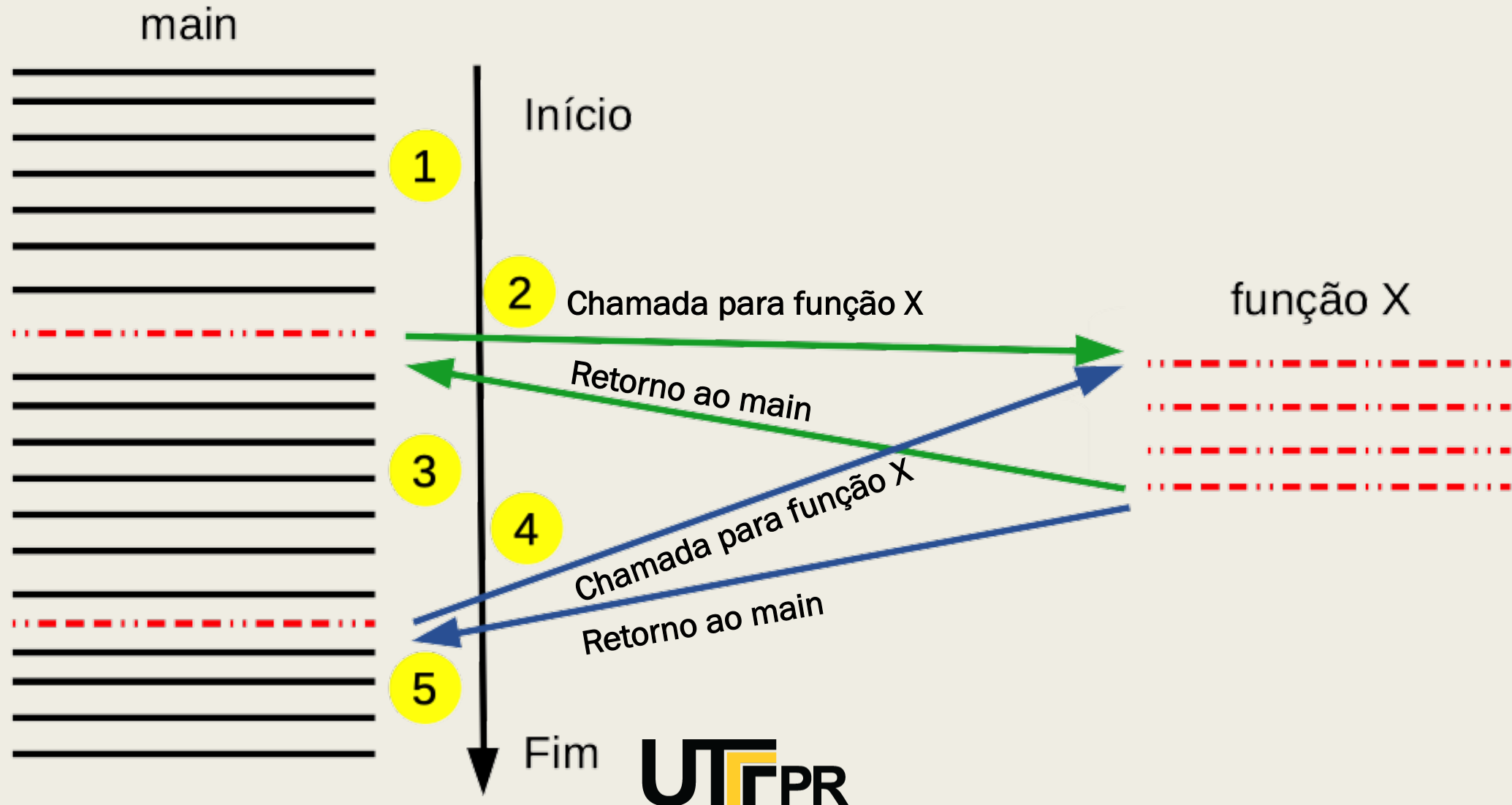
Funções: exemplificando

- Identificamos que a solução de X pode ser uma função!
- Isolamos e atribuímos um nome para esse trecho de código
- Substituímos esse código pela chamada da nova função

Algoritmo com subprogramação (funções)



Funções: exemplificando



Escopo de variáveis

- É o conceito que define a visibilidade e contexto das variáveis!
- **Variáveis locais:** são visíveis dentro de uma função onde foram declaradas:
 - *Só conseguem ser utilizadas dentro dessa função*
- **Variáveis globais:** são declaradas fora das funções e visíveis por qualquer função do seu programa:
 - *Podem ser utilizadas em qualquer função dentro de seu programa*

Passagem de parâmetros e Retorno

- Para permitir a comunicação entre funções existem esses dois conceitos
- Passagem de parâmetros: permite que a função receba valores enviados pela instrução que está chamando a função quando esta for iniciar sua execução
- Retorno: permite a devolução de um valor calculado pela função para a instrução que a chamou

Passagem de parâmetros e Retorno

- Exemplo: Um programa possui uma função que soma dois números. Nesse caso, quando a função for utilizada o programa deve enviar dois números e receber o resultado da soma.
- **Passagem de parâmetros**, no exemplo, é o envio dos para que a função faça a soma deles
- **Retorno**, no exemplo, é a devolução do resultado da soma dos números para o programa principal

Funções, parâmetros e retorno

- O uso de parâmetros e retorno é opcional!
- Uma função pode ter diversos parâmetros!
- A princípio, uma função pode devolver um único valor!
- O desenvolvedor pode criar quantas funções forem necessárias!
- Estudaremos quatro tipos de funções:
 1. *Sem passagem de parâmetros e sem retorno;*
 2. *Com passagem de parâmetros e sem retorno;*
 3. *Sem passagem de parâmetros e com retorno;*
 4. *COM passagem de parâmetros e COM retorno!*

MINIAULA DE ALGORITMOS FUNÇÕES

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



1. Sem parâmetros e sem retorno

- São as funções mais simples, pois:
 - *Não recebem valores por parâmetros*
 - *Não devolvem valor como retorno*
 - *Apenas executam um trecho de código*
- Essas funções são do tipo VOID:
 - *Desobriga o retorno de valores*

Exemplo: soma de dois números

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

Declaração de bibliotecas

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

Função soma: solicita dois números, recebe os dados digitados pelo usuário, armazena em variáveis locais, faz a soma e apresenta o resultado em tela

```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

Função principal (main), inicia a execução do programa e faz a chamada das funções para que a soma seja realizada

Exemplo!

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

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

A função **main()** inicia sua execução quando o programa tem início...

Exemplo!

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

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

A função **soma()** é chamada! Nesse ponto a função **main()** suspende sua execução e começa a execução da função **soma()** ...

Exemplo!

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

```
void soma() {
```

← Começa a execução da função **soma()** ...

```
    int n1, n2, s;
```

```
    printf("Primeiro número:\n");
```

```
    scanf("%d", &n1);
```

```
    printf("Segundo número:\n");
```

```
    scanf("%d", &n2);
```

```
    s = n1 + n2;
```

```
    printf("A soma é: %d\n", s);
```

```
}
```

```
void main() {
```

```
    soma() ;
```

```
    soma() ;
```

```
    soma() ;
```


```
}
```

Exemplo!

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

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

A função segue a
execução, linha a
linha, até o fim



```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

Exemplo!

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

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

← A execução da função **soma()** termina e controle volta para a função **main()**

```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

Exemplo!

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

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

Novamente a função **soma()** é chamada! E procede da mesma forma: a função **main()** suspende sua execução e começa a execução da função **soma()** ...

Exemplo!

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

```
void soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    soma();  
    soma();  
    soma();  
}
```

- E assim o programa segue sua execução até o fim
- Sempre que uma função é chamada, a função em execução é suspensa para que a função que foi chamada possa ser executada
- Ao terminar a execução, a função que foi chamada é terminada (seus dados são destruídos) e o controle retorna para quem a chamou

2. Com parâmetros e sem retorno

- São as funções que:
 - *Recebem valores para processar*
 - *Não devolvem valor como retorno*
- Essas funções ainda são do tipo VOID:
 - *Desobriga o retorno de valores*

Exemplo: soma de dois números

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);  
}
```

Função soma: RECEBE dois números por parâmetro, armazena em variáveis locais (parâmetros), faz a soma e apresenta o resultado na tela

Função principal (main), inicia a execução do programa e solicita dois números para o usuário, faz a chamada das funções passando os números via parâmetro para que a soma seja realizada dentro da função

Declaração de parâmetros

- Exemplo: `void soma(int n1, int n2) { ... }`
 - *Os parâmetros ficam entre os parênteses*
 - *Quantos forem necessários*
 - *Separados por vírgula*
 - *Cada qual com seu tipo de dados: char, int, long, float, double, dentre outros...*
- Quando declarados, obrigam a chamada da função a passar uma quantidade de parâmetros iguais à definida, na mesma ordem, e com os mesmo tipos (ou equivalentes)!

Exemplo!

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {
```

← A função **main()** inicia sua execução quando o programa tem início...

```
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);
```

```
}
```

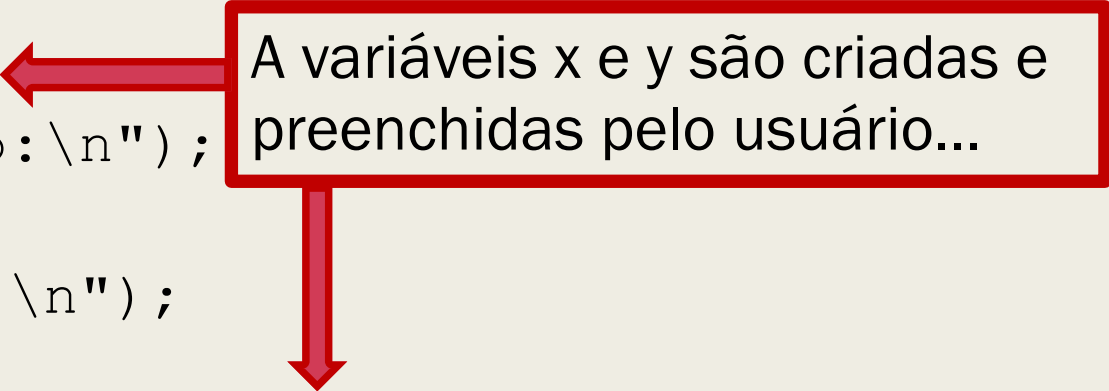
Exemplo!

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);  
}
```

A variáveis x e y são criadas e preenchidas pelo usuário...



Exemplo!

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);  
}
```

A função **soma()** é chamada! Nessa chamada devem, obrigatoriamente, ser passados os parâmetros na ordem esperada, em número igual e em tipos compatíveis: dois números inteiros!

A função **main()** suspende sua execução e começa a execução da função **soma()** ...

Nesse momento, os valores de x e y, SÃO COPIADOS, para n1 e n2 (da função soma), respectivamente...

Exemplo!

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

```
void main() {  
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);  
}
```

Quando **soma()** inicia, **n1** e **n2**, recebem automaticamente, o valores que foram enviados pela chamada, ou seja, os valores de **x** e **y**, respectivamente.


As variáveis **n1** e **n2** são definidas nos parâmetros e não precisam ser definidas em outro lugar. São consideradas variáveis locais normais, e que sempre são inicializadas durante a chamada da função!

Exemplo!

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

A função segue a
execução, linha a
linha, até o fim



```
void main() {  
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);  
}
```

Exemplo!

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

```
void soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    printf("A soma é: %d\n", s);  
}
```

← A execução da função **soma()** termina e controle volta para a função **main()**

```
void main() {  
    int x, y;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    soma(x, y);  
    soma(x, 10);  
    soma(10, 15);  
}
```

Exemplo: soma de dois números

- Os parâmetros da chamada podem variar conforme a necessidade! Observe:
- Duas variáveis inteiras:

soma (x, y) ;

- Uma variável inteira e um número inteiro:

soma (x, 10) ;

- Dois números inteiros:

soma (10, 15) ;



O importante é a ordem, a quantidade e o tipo!

3. Sem parâmetros e com retorno

- São as funções que:
 - *Não recebem valores por parâmetros*
 - *Retornam obrigatoriamente um valor*
- Essas funções utilizam tipos diferentes de VOID:
 - *char, int, long, float, double, dentre outros....*
- Devem usar a instrução **return** ao final da função!

Exemplo: soma de dois números

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

Declaração de bibliotecas

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

Função soma: solicita dois números, recebe e armazena em variáveis locais, faz a soma e retorna o valor para a função chamadora! Observe que o tipo da função é **int**!

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

Função principal (main), inicia a execução do programa e faz a chamada da função soma, aguardando o retorno para que seja armazenado na variável res. Em seguida o valor é impresso em tela...

Exemplo!

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

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

A função **main()** inicia sua execução quando o programa tem início...

Exemplo!

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

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

A função **soma()** é chamada! A função **main()** suspende sua execução e fica aguardando o resultado da função **soma()** ...

Exemplo!

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

```
int soma () {
```



Começa a execução da função **soma ()** ...

```
    int n1, n2, s;
```

```
    printf("Primeiro número:\n");
```

```
    scanf("%d", &n1);
```

```
    printf("Segundo número:\n");
```

```
    scanf("%d", &n2);
```

```
    s = n1 + n2;
```

```
    return s;
```

```
}
```

```
void main() {
```

```
    int res;
```

```
    res = soma ();
```

```
    printf("A soma é: %d\n", res);
```


```
}
```

Exemplo!

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

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

A função segue a
execução, linha a
linha, até o fim



```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

Exemplo!

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

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

Ao final a instrução **return** é executada. É a última instrução da função antes de seu término e exclusão. Nesse momento, o valor contido na variável da instrução **return** (nesse caso, **s**) é COPIADO para a variável receptora na função chamadora. A função **soma()** termina e o controle volta para a função **main()** ...

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

Exemplo!

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

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

Quando a função **soma()** termina sua execução o valor retornado é armazenado na variável **res**, que estava esperando o retorno!

Exemplo!

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

```
int soma() {  
    int n1, n2, s;  
    printf("Primeiro número:\n");  
    scanf("%d", &n1);  
    printf("Segundo número:\n");  
    scanf("%d", &n2);  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

A função **main()** então termina sua execução com a impressão do valor da soma em tela!

Pontos importantes!

- Nenhuma instrução deve ser colocada após o **return**!
- A variável que vai receber o retorno da função deve ser do mesmo tipo da função declarada!
- A variável de retorno também deve ser do mesmo tipo da função declarada!

```
int soma() {  
    int ... s;  
    ...  
    return s;  
}
```

```
void main() {  
    int res;  
    res = soma();  
    printf("A soma é: %d\n", res);  
}
```

4. Com parâmetros e com retorno

- São funções mais elaboradas que:
 - *Recebem valores por parâmetros*
 - *Retornam obrigatoriamente um valor*
- Essas funções utilizam tipos diferentes de VOID:
 - *char, int, long, float, double, dentre outros....*
- Devem usar a instrução **return** ao final da função!

Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

Função soma: RECEBE dois números, armazena em variáveis locais (parâmetros), faz a soma e retorna o resultado para a função chamadora!

Função principal (main), inicia a execução do programa e solicita dois números para o usuário, faz a chamada das funções passando os números via parâmetro para que a soma seja realizada. Recebe o retorno da função na variável res, e ao final imprime em tela o valor da soma!

Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

A função **main()** inicia sua execução quando o programa tem início...

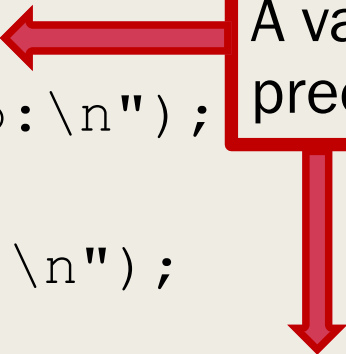
Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

A variáveis são declaradas, e x e y preenchidas pelo usuário...



Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

A função **soma()** é chamada! A função **main()** suspende sua execução e fica aguardando o retorno da função **soma()**. Nesse momento, os valores de **x** e **y**, SÃO COPIADOS, para **n1** e **n2** (da função **soma**), respectivamente...

Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

Quando **soma()** inicia, **n1** e **n2**, recebem automaticamente, o valores que foram enviados pela chamada, ou seja, os valores de **x** e **y**.

As variáveis **n1** e **n2** são definidas nos parâmetros e sempre são inicializadas durante a chamada da função!


```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

A função segue a execução, linha a linha, até o fim



```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```


Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

Ao final a instrução **return** é executada. Nesse momento, o valor contido na variável da `s` é COPIADO para a variável `res` na função chamadora. A função **soma()** termina e o controle volta para a função **main()** ...

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

Quando a função **soma ()** termina sua execução o valor retornado é armazenado na variável **res**, que estava esperando o retorno!

Exemplo: soma de dois números

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

```
int soma(int n1, int n2) {  
    int s;  
    s = n1 + n2;  
    return s;  
}
```

```
void main() {  
    int x, y, res;  
    printf("Primeiro número:\n");  
    scanf("%d", &x);  
    printf("Segundo número:\n");  
    scanf("%d", &y);  
    res = soma(x, y);  
    printf("A soma é: %d\n", res);  
}
```

A função **main()** então termina sua execução com a impressão do valor da soma em tela!

Atenção!

- Sempre conferir:
 - *Ordem, quantidade e tipos dos parâmetros*
- Na declaração da função e na chamada da função!
- Sempre declarar uma função antes de seu uso, isto é, em geral, antes da função main
 - *Exceção: uso de protótipos de função*

Protótipos de função

- Em qualquer programa, podemos escrever funções antes ou depois da função main.
- Se optarmos por escrevê-las antes, nenhum cuidado especial será necessário.
- Se optarmos por escrevê-las abaixo da função main, devemos fazer uso dos protótipos de função.
 - *É uma declaração do cabeçalho da função seguida de ponto e vírgula (;)*



O uso de protótipos de função pode auxiliar quando o arquivo contém muitas funções interdependentes!

Exemplo de protótipo!

```
#include <stdio.h>

int soma(int n1, int n2); ←

void main() {
    int x, y, res;
    printf("Primeiro número:\n");
    scanf("%d", &x);
    printf("Segundo número:\n");
    scanf("%d", &y);
    res = soma(x, y);
    printf("A soma é: %d\n", res);
}

int soma(int n1, int n2) { ←
    int s;
    s = n1 + n2;
    return s;
}
```

O uso do protótipo desobriga a declaração da função antes de ser utilizada pelo main!

Nesse caso, a função pode ser desenvolvida em qualquer ponto do seu arquivo de código fonte!

MINIAULA DE ALGORITMOS TIPOS DE PASSAGEM DE PARÂMETROS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Passagem de parâmetros: 2 maneiras!

- Passagem de parâmetros por valor: a função trabalhará com cópias dos valores passados no momento de sua chamada.
 - *Foi este tipo de passagem que utilizamos até o momento. Neste esquema quando a função termina suas variáveis locais são perdidas.*
- Passagem de parâmetros por referência: nesta modalidade os parâmetros passados para uma função correspondem a endereços de memória ocupados pelas variáveis.
 - *Toda vez que for necessário acessar o valor de uma variável, isso será feito por meio de referência, ou seja, apontando ao seu endereço.*
 - *Ainda, toda vez que a função modifica uma variável, estamos modificando o valor no endereço da variável original, ou seja, o valor original é modificado!*

Passagem de parâmetros: tipos!

- Por valor:

```
res = soma (x, y) ;
```

- Por referência:

```
res = soma (&x, &y) ;
```

- A diferença é o símbolo **&** que indica o endereço da variável, ao invés de seu conteúdo!
- Mas existem diferenças na declaração dos parâmetros na função também!

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Exemplo: soma dobro!

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

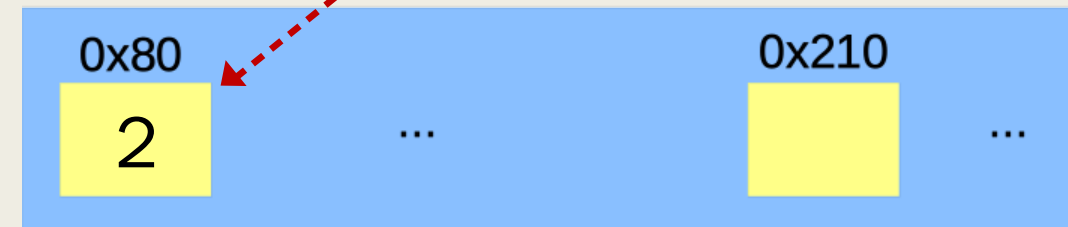
```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
	2	

Memória Principal (com endereços físicos)



O usuário digita
o valor 2!

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
	2	3

Memória Principal (com endereços físicos)

0x80		0x210
2	...	3

O usuário digita
o valor 3!

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Variáveis da soma_dobro()

soma	*n1	*n2
	0x80	0x210

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
	2	3

Memória Principal (com endereços físicos)

0x80		0x210
2	...	3

A função soma_dobro é chamada!
Ao invés de valores, são enviados os
endereços das variáveis!

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Na função soma_dobro quando **n1** é manipulado, estamos manipulando o conteúdo de x, através do endereço dele! Logo, estamos dobrando o valor de x!

Variáveis da soma_dobro()

soma	*n1	*n2
	0x80	0x210

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
	4	3

Memória Principal (com endereços físicos)

0x80	...	0x210	...
4		3	

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

O mesmo vale para n2,
que altera o valor do
endereço da variável y!

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Variáveis da soma_dobro()

soma	*n1	*n2
	0x80	0x210

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
	4	6

Memória Principal (com endereços físicos)

0x80	...	0x210	...
4		6	

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

soma recebe a soma dos conteúdos de n1 e n2, que apontam, respectivamente, para x e y!

Variáveis da soma_dobro()

soma	*n1	*n2
10	0x80	0x210

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
	4	6

Memória Principal (com endereços físicos)

0x80		0x210
4	...	6

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Na instrução return, o valor de soma é copiado para res, antes que seja destruído!

Variáveis da soma_dobro()

soma	*n1	*n2
10	0x80	0x210

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
10	4	6

Memória Principal (com endereços físicos)

0x80	...	0x210	...
4		6	

Exemplo: soma dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
10	4	6

Memória Principal (com endereços físicos)

0x80		0x210
4	...	6

A função termina e o controle volta para o main(), destruindo o conteúdo de soma_dobro!

Exemplo: soma dobro!

A função main() termina com a impressão em tela dos valores, incluindo aqueles que foram modificados pela função soma_dobro!

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

```
int soma_dobro(int *n1, int *n2) {  
    int soma;  
    *n1 = 2 * (*n1);  
    *n2 = 2 * (*n2);  
    soma = *n1 + *n2;  
    return soma;  
}
```

```
void main () {  
    int x, y, res;  
    printf("Digite o primeiro valor: \n");  
    scanf("%d", &x);  
    printf("Digite o segundo valor: \n");  
    scanf("%d", &y);  
    res = soma_dobro(&x, &y);  
    printf("A soma entre %d e %d é %d\n", x, y, res);  
}
```

Variáveis do main() (nome/endereço físico)

res (0x300)	x (0x80)	y (0x210)
10	4	6

Memória Principal (com endereços físicos)

0x80	...	0x210	...
4		6	

Lembrete!

- Na passagem por referência:
 - Na declaração, utiliza-se o *** nos parâmetros

```
int soma_dobro(int *n1, int *n2) { ... }
```

- Na chamada, utiliza-se o *&* nos parâmetros

```
res = soma_dobro(&x, &y);
```

MINIAULA DE ALGORITMOS VETORES E MATRIZES VIA PARÂMETROS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

ipolato@utfpr.edu.br



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



Vetores via parâmetro

- Como sabemos, vetores são variáveis compostas
 - *Agregado unidimensional homogêneo*
- Não podem ser passados por valor
 - *Somente por REFERÊNCIA!*
- Em geral, junto com ponteiro para o vetor, passamos a quantidade de elementos do vetor
- Caso, isso não ocorra, podemos calcular o tamanho:
int tam = sizeof(vetor)/sizeof(vetor[0]);
A quantidade de elementos é o tamanho do vetor em bytes/tamanho de um elemento do vetor!

Exemplo: média de um vetor

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

```
//As duas formas listadas são válidas! Use apenas uma!
```

```
//float mediaVetor(int *vet, int tam) {  
float mediaVetor(int vet[], int tam) {  
    float m=0;  
    for (int i=0; i<tam; i++)  
        m += vet[i];  
    m = m / tam;  
    return m;  
}
```

```
int main () {  
    int v[5] = {1, 2, 3, 4, 5};  
    float media = mediaVetor(v, 5);  
    //media = mediaVetor(v, sizeof(v)/sizeof(v[0]));  
    printf("Média: %.1f\n", media);  
    return 0;  
}
```

Matrizes via parâmetro

- Funciona de maneira similar aos vetores
 - Entretanto, para matrizes, apenas a primeira dimensão pode ser enviada vazia, comente com os colchetes, como nos vetores :
 - *As outras dimensões (2, 3, ...) devem ser explicitadas!*
- ```
float mediaMatriz(int mat[][3], int lin, int col)
```
- Também devemos enviar as dimensões (linhas e colunas)!



# Exemplo: média de uma matriz

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

```
//As duas formas listadas são válidas! Use apenas uma!
```

```
//float mediaMatriz(int (*mat)[3], int lin, int col) {
float mediaMatriz(int mat[][3], int lin, int col) {
 float m=0;
 for(int i=0; i<lin; i++)
 for(int j=0; j<col; j++)
 m += mat[i][j];
 m = m / (lin*col);
 return m;
}
```

```
int main () {
 int m[2][3] = {{1,2,3},{4,5,6}};
 float media = mediaMatriz(m, 2, 3);
 printf("Média: %.1f\n", media);
 return 0;
}
```

# MINIAULA DE ALGORITMOS

# REGISTROS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

[ipolato@utfpr.edu.br](mailto:ipolato@utfpr.edu.br)



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



# Registros: o que são?

- São estruturas compostas heterogêneas que agregam informações de diferentes tipos em um lugar só
- Cada informação em um registro é chamada de campo
- Cada campo tem seu tipo:
  - *Simples: char, int, long, float, double*
  - *Complexo: strings, vetores, matrizes*
- Declarados através da instrução STRUCT
  - *Define um tipo de dados personalizado, a partir do qual são criadas variáveis, manipuladas pelo programa*

# Registros: exemplo

```
struct CONTA{
 int codigo;
 char nome[51];
 long telefone;
 float saldo;
};

struct CONTA cliente;
```

- O código ao lado declara uma estrutura chamada CONTA.
- CONTA é um tipo de dados complexo, que reúne diversos campos: codigo, nome, saldo e telefone.
- A seguir, definimos a **variável cliente**, que é do tipo CONTA!
  - *Nesse caso, temos espaço para armazenar um cliente*

# Registros: alternativamente!

```
struct CONTA{
 int codigo;
 char nome[51];
 long telefone;
 float saldo;
};
```

```
struct CONTA cliente;
```

```
struct CONTA{
 int codigo;
 char nome[51];
 long telefone;
 float saldo;
}cliente;
```

# Registros: exemplo

- Não podemos acessar todos os campos de uma vez
- Devemos acessar individualmente cada campo, como se fosse uma variável comum
- Adicionamos o prefixo <nomeDaVariável> seguida de um ‘.’

```
cliente.codigo = 123;
```

```
strcpy(cliente.nome, "Peter Griffin");
```

```
cliente.telefone = 5551234;
```

```
cliente.saldo = 299.99;
```

# Registros: exemplo

- Podemos usar a função `scanf()`:

...

```
printf("Código: ");
scanf("%d", &cliente.codigo);
printf("Nome: ");
scanf("%[^\n]", cliente.nome);
printf("Telefone: ");
scanf("%ld", &cliente.telefone);
printf("Saldo: ");
scanf("%f", &cliente.saldo);
```

...

- Toda a entrada de dados é feita usando `scanf()`, solicitando as informações ao usuário
- Os campos são tratados como variáveis regulares
  - *Não se esqueça do **&** onde for necessário!*

# Registros: exemplo

- Exibindo as informações armazenadas:

...

```
printf("Código: %d", cliente.codigo);
printf("Nome: %s", cliente.nome);
printf("Telefone: %ld", cliente.telefone);
printf("Saldo: %.2f", cliente.saldo);
```

...



# Registros + Vetores

```
struct CONTA{
 int codigo;
 char nome[51];
 char telefone[15];
 float saldo;
};
```

```
struct CONTA clientes[5];
```

- Agora criamos um vetor clientes com 10 posições
  - *Cada posição é uma estrutura completa, com todos os campos*
- Cada posição no vetor tem que ser manipulada individualmente também!

# Registros + Vetores: visualizando

```
struct CONTA{
 int codigo;
 char nome[51];
 char telefone[15];
 float saldo;
}clientes[5];
```

- Cada posição do vetor é uma variável do tipo CONTA, e possui todos os campos declarados no tipo.
- Devem ser acessados da mesma maneira, mas considerando seu **índice no vetor**.

|     |                                     |                                     |                                     |                                     |                                     |
|-----|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
|     | codigo<br>nome<br>telefone<br>saldo | codigo<br>nome<br>telefone<br>saldo | codigo<br>nome<br>telefone<br>saldo | codigo<br>nome<br>telefone<br>saldo | codigo<br>nome<br>telefone<br>saldo |
| [ ] | 0                                   | 1                                   | 2                                   | 3                                   | 4                                   |

# Registros + Vetores: manipulação

- Acessar posições do **vetor** e seus campos individualmente!

...

```
clientes[0].codigo = 123;
strcpy(clientes[0].nome, "Peter Griffin");
clientes[0].telefone = 5551234;
clientes[0].saldo = 299.99;
clientes[1].codigo = 456;
strcpy(clientes[1].nome, "Stewie");
clientes[1].telefone = 5556666;
clientes[1].saldo = 0.99;
```

...

# Registros + Vetores: manipulação

```
...
for(int i=0; i<5; i++){
 printf("Código: ");
 scanf("%d", &clientes[i].codigo);
 printf("Nome: ");
 scanf("%[^\\n]", clientes[i].nome);
 printf("Telefone: ");
 scanf("%ld", &clientes[i].telefone);
 printf("Saldo: ");
 scanf("%f", &clientes[i].saldo);
}
...
```

- Assim como nos vetores de tipos simples, podemos utilizar estruturas de repetição para facilitar a manipulação do vetor completo de uma vez só!

# Registros + Vetores: manipulação

- Exibindo as informações armazenadas:

...

```
printf("Código: %d", clientes[0].codigo);
printf("Nome: %s", clientes[0].nome);
printf("Telefone: %ld", clientes[0].telefone);
printf("Saldo: %.2f", clientes[0].saldo);
```

...

# Registros: lembretes!

- Nunca manipule o tipo, e sim a variável criada a partir dele!

```
struct CONTA clientes[5];
```

- Cada campo deve ter seu tipo individual!
- Lembre-se que o índice pertence ao vetor!

*Certo:*

✓ **clientes**[0].codigo = 123;

*Errado:*

✗ **clientes**.codigo[0] = 123;

# MINIAULA DE ALGORITMOS

# ARQUIVOS

Prof. Ivanilton Polato

Departamento Acadêmico de Computação (DACOM-CM)

[ipolato@utfpr.edu.br](mailto:ipolato@utfpr.edu.br)



[4.0 Internacional](https://creativecommons.org/licenses/by/4.0/)



# Arquivos: o que são?

- Conjuntos de dados armazenados em memória secundária
  - *Disco rígido (HD, SSD, CD, DVD)*
- Necessidade de persistir os dados da memória principal
  - *Podem ser acessados em diferentes execuções de um programa*
  - *Podem ser acessados por diversos programas*
- Evolução: SGBDs



# Arquivos: declaração

- Criamos um ponteiro para o arquivo:  
`FILE *arq;`
- `arq` é uma variável (ponteiro) que armazena o endereço inicial de memória ocupado por um arquivo
  - *se o arquivo não puder ser aberto `arq` recebe `NULL` e a operação falha!*
- Erros: arquivo inexistente, permissões, espaço em disco

# Arquivos: abertura

- Usamos a função `fopen`:

```
arq = fopen(nome_arquivo, modo_de_abertura);
```

- O nome do arquivo é o identificador do arquivo que se deseja abrir, podendo incluir o caminho para o arquivo

```
arq = fopen("teste.txt", "r");
```

```
arq = fopen("/pasta1/pasta2/teste.txt", "r");
```

- O modo de abertura determina como o arquivo poderá ser manipulado: leitura (`r`), escrita (`w` ou `a`), o tipo do arquivo (texto ou binário) e atualizações permitidas (`+`)

# Arquivos: modo de abertura

| Sigla                                                 | Modo                                                                                           | Efeito                                                                                                                              |
|-------------------------------------------------------|------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <b>r</b>                                              | READ: Abre o arquivo em formato texto no modo somente leitura                                  | Se o arquivo não existir, ocorrerá um erro!                                                                                         |
| <b>w</b>                                              | WRITE: Abre o arquivo em formato texto no modo escrita                                         | Se o arquivo não existir o arquivo será criado;<br>Se existir o arquivo será sobrescrito e o conteúdo original apagado!             |
| <b>a</b>                                              | APPEND: Abre o arquivo em formato texto no modo anexo                                          | Se o arquivo não existir o arquivo será criado<br>Se existir o arquivo será aberto e o conteúdo original preservado!                |
| <b>rb</b><br><b>wb</b><br><b>ab</b>                   | BINARY: Modos similares, mas em formato binário                                                | Utilizado para arquivos binários, em geral, gravar estruturas complexas (registros) através de um streaming de bytes.               |
| <b>r+ / rb+</b><br><b>w+ / wb+</b><br><b>a+ / ab+</b> | <b>+</b> : Modos similares, mas permite atualização de dados no arquivo (gravação sobrescrita) | O modificador <b>+</b> pode ser utilizado em situações onde são necessárias correções em dados gravados anteriormente nos arquivos. |

# Exemplo

```
#include <stdio.h>

int main () {
 FILE *arq;
 arq = fopen("arquivoTeste.txt", "r") ;

 if(arq == NULL)
 printf("Erro: arquivo não pode ser aberto!");
 else{
 printf("Arquivo aberto com sucesso!\n");
 fclose(arq) ;
 }
}
```

# Arquivo: fechamento

- É extremamente recomendável fechar um arquivo depois de utilizá-lo.
- Caso o arquivo não seja fechado corretamente podem ocorrer erros
  - *Perda de dados!*
  - *Perda do arquivo!*
- A função `fclose()` fecha um arquivo  
**`fclose(arq) ;`**

# Arquivos: gravando dados

- Para gravar um caractere no arquivo:
  - Função `fputc(char ch, FILE *arq);`  
`char c = '@' ;`  
`fputc(c, arq) ;`
- Para gravar uma string no arquivo:
  - Função `fputs(char *cadeia, FILE *arq);`  
`char str[]="Algoritmos" ;`  
`fputc(str, arq) ;`

# Arquivos: lendo dados

- Para ler um **caractere** no arquivo:

- *Função `fgetc(FILE *arq);`*  
**`char c = fgetc(arq);`**

- Para ler uma **string** no arquivo:

- *Função `fgets(char *cadeia, int tamanho, FILE *arq);`*  
**`char str[100];`**  
**`fgets(str, 100, arq);`**
- *Essa função lê uma string do arquivo até encontrar uma quebra de linha (`\n`) ou até o tamanho máximo definido*
- *Caso não seja possível, a função retorna NULL*

# Exemplo de gravação (caracteres)

```
#include <stdio.h>
int main () {
 FILE *arq;
 char caracas = '!';
 arq = fopen("dados.txt", "a");
 if (arq == NULL)
 printf("Erro na abertura!\n");
 else {
 while (caracas != '0') {
 printf("Digite um caractere, ou 0 para sair:\n");
 scanf(" %c", &caracas);
 fputc(caracas, arq);
 if (ferror(arq))
 printf("Erro ao tentar escrever no arquivo\n");
 }
 }
 fclose(arq);
}
```



# Arquivos: `ferror()`

- A função `ferror()` detecta se ocorreu algum erro durante uma operação com arquivos.
  - *A sintaxe correta é: `ferror(FILE *arq);`*
- A função `ferror()` retorna um número inteiro e deve ser chamada logo depois que qualquer outra função for invocada.
  - *Se o número retornado for diferente de zero, isto significa erro durante a última operação.*
  - *Se for zero não ocorreu erro.*

# Exemplo de leitura (caracteres)

```
#include <stdio.h>
int main () {
 FILE *arq;
 char caracas = '!';
 arq = fopen("dados.txt", "a");
 if (arq == NULL)
 printf("Erro na abertura!\n");
 else {
 while ((caracas = fgetc(arq)) != EOF) {
 if (ferror(arq))
 printf("Erro na leitura do caractere\n");
 else
 printf("Caractere: %c\n", caracas);
 }
 }
 fclose(arq);
}
```

# Arquivos: EOF

- É o caractere (marcador) que indica o fim de arquivo
- Usamos para verificar, em uma repetição, se chegamos ao final do arquivo e interromper a leitura no momento correto
- No exemplo, comparamos o resultado da operação `fgetc()`, que lê um caractere do arquivo e armazena em uma variável, com o EOF.
  - *Caso a operação de leitura retorne o EOF, a execução termina, e não são impressos caracteres indevidos!*

# Arquivos: função `fprintf()`

- A função `fprintf()` envia texto formatado, assim como o `printf()`, para um arquivo.
- Funciona da mesma forma, mas agora temos que adicionar o ponteiro para o arquivo. Veja o exemplo.

```
#include <stdio.h>
int main() {
 FILE * arq;
 arq = fopen ("teste.txt", "a");
 fprintf(arq, "%s %d %c", "Estamos em", 2020, '!');
 fclose(arq);
}
```

# Arquivos: função **fscanf()**

- A função **fscanf()** funciona de maneira similar ao **scanf()**, mas lendo dados de um arquivo e armazenando nas respectivas variáveis!
- Devem ser respeitadas as mesmas regras para o uso do **&** nas variáveis
- Podem ser feitas múltiplas leituras de uma vez só, se conhecermos a maneira como foram armazenados no arquivo!

# Arquivos: função `fscanf()`

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

```
int main() {
```

```
 char str[11];
```

```
 int ano;
```

```
 char caracas;
```

```
 FILE *arq;
```

```
 arq = fopen ("teste.txt", "r+");
```

```
 fscanf(arq, "%s %d %c", str, &ano, &caracas);
```

```
 printf("Dados lidos do arquivo!\n");
```

```
 printf("Frase: %s %d%c", str, ano, caracas);
```

```
 fclose(arq);
```

```
}
```

# Arquivos binários: gravando bytes!

- Arquivos armazenam uma sequência de caracteres ou de bytes.
- Em alguns programas é mais útil e prático ler parte do conteúdo de um arquivo e gravar diretamente em uma variável simples, como int ou float, ou ainda em uma variável de um tipo struct.
  - *A função **fscanf()** permite fazer isso com tipos simples e strings, mas não com structs!*
- Quando isso for necessário, o programa deverá abrir com arquivos binários.
- Toda vez que uma operação de leitura ou de escrita for realizada, deverá ser informado o número de bytes que serão lidos ou gravados.
  - *Para isso, a função **sizeof()** será utilizada intensamente, uma vez que ela permite descobrir quantos bytes uma variável ou struct ocupa.*

# Arquivos: função `fwrite()`

- A função `fwrite()` pode gravar qualquer tipo de dados, e não apenas caracteres ou strings! A forma geral é:

`fwrite(void *dados, size_t qtBytes, size_t numItens, FILE *arq);`

- *dados* representa a variável com o conteúdo a ser gravado no arquivo
  - *qtBytes* é o tamanho em bytes que será escrito no arquivo
  - *numItens* é o número de itens de tamanho `qtBytes` escritos no arquivo
  - *arq* é a referência para o arquivo onde as informações serão escritas
- Quando a função `fwrite()` é bem sucedida, gera como retorno um valor igual ao número de gravações realizadas, igual ao parâmetro *numItens*!
    - Se ocorrer algum erro, o valor retornado será menor que *numItems*.



# Exemplo de uso do fwrite()

```
#include <stdio.h>

struct DADOS{
 int codigo;
 char nome[20];
}cliente;

int main (){
 FILE *arq;
 arq = fopen("dados.dat", "ab+");
 if (arq == NULL) printf("Erro na abertura do arquivo.\n");
 else{
 printf("Digite o código do cliente: ");
 scanf(" %d", &cliente.codigo);
 printf("Digite o nome: ");
 scanf("%[^\\n]", cliente.nome);
 fwrite(&cliente, sizeof(cliente), 1, arq);
 if (ferror(arq))
 printf("Erro na gravação de dados.\n");
 else
 printf("Dados gravados com sucesso!\n");
 fclose(arq);
 }
}
```

# Arquivos: função `sizeof()`

- A função `sizeof()` calcula o tamanho em bytes de um tipo de dados
- Seu uso é importante quando gravando bytes em arquivo, pois as funções `fwrite()` e `fread()` precisam saber quantos bytes serão gravados no arquivo!
- Podemos aplicar aos tipos simples também!

```
int x;
```

```
printf("Tamanho do int: %d bytes!", sizeof(x));
```

– Nesse caso, teríamos: *Tamanho do int: 4 bytes!*

# Arquivos: função `fread()`

- A função `fread()` funciona como a `fwrite()`, mas lê qualquer tipo de dados do arquivo! A forma geral é:

`fread(void *dados, size_t qtBytes, size_t numItems, FILE *arq);`

- *dados* representa a variável que vai receber o conteúdo do arquivo
  - *qtBytes* é o tamanho em bytes que será lido do arquivo
  - *numItems* é o número de itens de tamanho *qtBytes* lidos do arquivo
  - *arq* é a referência para o arquivo de onde as informações serão lidas
- Quando a função `fread()` é bem sucedida, gera como retorno um valor igual ao número de leituras realizadas, igual ao parâmetro *numItems*!
    - Se ocorrer algum erro, o valor retornado será menor que *numItems*.

# Exemplo de uso do fread()

```
#include <stdio.h>

struct DADOS{
 int codigo;
 char nome[20];
}cliente;

int main (){
 FILE *arq;
 arq = fopen("dados.dat", "ab+");
 if (arq == NULL) printf("Erro na abertura do arquivo.\n");
 else{
 while((fread(&cliente, sizeof(cliente), 1, arq)) != EOF){
 if(ferror(arq))
 printf("Erro na leitura de dados do arquivo.\n");
 else{
 printf("Código do cliente: %d\n", cliente.codigo);
 printf("Nome do cliente: %s\n", cliente.nome);
 }
 }
 fclose(arq);
 }
}
```

# Funções adicionais: **rewind()**

- Cursor é um ponteiro que indica a partir de que posição, dentro do arquivo, uma operação será executada.
- Por exemplo, quando um arquivo acaba de ser aberto, seu cursor está apontando para a posição zero.
- Caso seja feita uma leitura com o comando `fread()`, o cursor se movimentará quantos bytes forem lidos.
- A função **rewind()** reposiciona o cursor de volta ao início do arquivo. Sua sintaxe é:

```
rewind(FILE *arq) ;
```

# Funções adicionais: **fseek()**

- A função **fseek()** é utilizada para mudar a posição do cursor sem que haja necessidade de leituras ou escritas no arquivo. Sua sintaxe é:

**fseek(FILE \*arq, long qtBytes, int pos);**

- *arq* – representa o arquivo
- *qtd\_bytes* – representa a quantidade de bytes que o cursor será movimentado a partir de *pos*
- *pos* – representa o ponto a partir do qual a movimentação será executada

# Funções adicionais: **fseek()**

- O parâmetro pos da função fseek() pode assumir três valores:
  - *SEEK\_SET* – movimenta qtBytes a partir da posição inicial do arquivo
  - *SEEK\_CUR* – movimenta qtBytes (positivos para frente, negativos para retroceder) a partir do ponto atual do cursor;
  - *SEEK\_END* – movimenta qtBytes (negativos, para retorceder) a partir da posição final do arquivo.
- Para o SEEK\_SET e SEEK\_END, se utilizarmos 0 bytes, o ponteiro vai ficar no início e no fim do arquivo, respectivamente!