



# Algoritmos e Estruturas de Dados II

## ED2

### Módulo 1 - Modularização em C

Profa. Msc. **Cilene Aparecida Mainente**

# **1 - Introdução à Modularização**

# Introdução

Em C existe um subprograma principal chamado **main** que determina o início da execução do programa.

```
#include "stdio.h"

int main () {

    int x = 10;

    printf ("O valor de X = %d", x );

    return 0;

}
```

# Introdução

A Modularização consiste em implementar códigos por meio de funções ou procedimentos, que realizam tarefas específicas.

O código passa a ter, além da função main em C, outras funções necessárias para o programa atingir seu objetivo.

Uma função (ou procedimento) é um trecho de código ao qual damos um nome para uso posterior (chamada).

Uma função (ou procedimento) pode receber entradas (parâmetros).

Uma função pode retornar um (e somente um) valor. Procedimentos não retornam valor.

## **2 - Funções das Bibliotecas C - Exemplos**

# Funções das Bibliotecas C

Em C há um vasto conjunto de funções organizadas em bibliotecas

O programa abaixo mostra o uso das seguintes “funções” printf e scanf presentes na biblioteca stdio.h

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int x, y;
    printf("Digite o primeiro numero: \n");
    scanf("%d", &x);
    printf("Digite o segundo numero: \n");
    scanf("%d", &y);
    printf("A soma eh: %d \n", soma(x, y));
    system("pause");
}
```

# Funções das Bibliotecas C

Em C há um vasto conjunto de funções organizadas em bibliotecas

- O programa abaixo mostra o uso das seguintes “funções” strcpy (Biblioteca string) e system (Biblioteca stdlib).

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

int main () {
    char command[50];

    strcpy( command, "dir" );
    system(command);

    return(0);
}
```

- O programa lista na tela o nome e tamanho dos arquivos presentes na pasta atual.

## **3 - Criando suas próprias funções em C**



# Funções de Usuário

**Assim como a maioria das linguagens de programação, a linguagem C permite escrever nossas próprias funções**

**Ao escrevê-las, pode-se “QUEBRAR” um programa complexo em vários subprogramas.**


**Essa ação recebe o nome de Modularização.**

# Exemplo de Função de Usuário em C

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

int main() {
    int x, y;
    printf("Digite o primeiro numero: \n");
    scanf("%d", &x);
    printf("Digite o segundo numero: \n");
    scanf("%d", &y);
    printf("A soma eh: %d \n", soma(x, y));
    system("pause");
}

int soma(int a, int b) {
    int resultado;
    resultado = a + b;
    return resultado;
}
```




## COMPONENTES DESTACADOS

1. Protótipo da Função
2. Chamada
3. Parâmetros
4. Assinatura da Função
5. Entrada de valores
6. Retorno da Função

# Forma Geral de um Função C

```
tipo_de_retorno_da_função nome_da_função (lista de parâmetros) {  
  
    variáveis locais;  
    ...  
    instruções;  
    [ return valor ;]  
}
```



## Lista de Parâmetros

São variáveis que se comunicam com outras funções. Ex. int v1, int v2, float v3;

## Variáveis Locais

São variáveis utilizadas apenas dentro da função, geralmente para auxiliar na tarefa e que não se comunicam com outras funções. Exemplo: int aux1, int aux2, num;

## Return

Toda função pode retornar no máximo **UM VALOR** (de qualquer tipo) através do comando return. É este valor retornado que determina o tipo da função. Se o valor retornado for um FLOAT a função será do tipo float, e assim por diante.

# Exemplo de código sem uso de função de usuário - Java

```
import java.io.*;
public class ex03 {
    public static void main(String[] args) {
        float nota1, nota2, media;
        BufferedReader teclado;
        teclado = new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.print("Digite a primeira nota entre 0 e 10 : ");
            nota1 = Float.parseFloat (teclado.readLine());
            while (nota1 < 0 || nota1 > 10){
                System.out.print("Nota Invalida. Redigite nota entre 0 e 10: ");
                nota1 = Float.parseFloat (teclado.readLine());
            }
            /* ----- */
            System.out.print("Digite a segunda nota entre 0 e 10 : ");
            nota2 = Float.parseFloat (teclado.readLine());
            while (nota1 < 0 || nota1 > 10){
                System.out.print("Nota Invalida. Redigite nota entre 0 e 10: ");
                nota1 = Float.parseFloat (teclado.readLine());
            }
            media = (nota1 + nota2)/2;
            System.out.println("Media do aluno= " + media);
        }
        catch (Exception e){
            System.out.println("Erro digitacao ! ");
        }
    }
}
```

**NOTE QUE OS 2 TRECHOS  
MARCADOS  
SÃO REPETITIVOS:**

fazem a mesma coisa (para  
nota1 e nota2)

# Função de usuário: propiciando o reuso

O trecho “repetitivousado na LEITURA da variável foi transformado na função `Leitura_Float`

```
static float Leitura_Float ( ) {  
    BufferedReader teclado;  
    teclado = new BufferedReader(new InputStreamReader(System.in));  
    float nota = 0;  
    System.out.print("Digite uma nota entre 0 e 10 ");  
    nota = Float.parseFloat (teclado.readLine());  
    while (nota < 0 || nota > 10){  
        System.out.print("Nota Invalida. Redigite nota entre 0 e 10: ");  
        nota = Float.parseFloat (teclado.readLine());  
    }  
    return nota;  
}
```

# Função de usuário: propiciando o reuso

- Note que o programa abaixo chama a função **Leitura\_Float** duas vezes: uma para cada nota.
- A função **Leitura\_Float** pode ser colocada numa **biblioteca** e reusada em qualquer programa

```
public class ex04 {  
  
    public static void main(String[] args) {  
        float nota1, nota2, media;  
  
        nota1 = Leitura_Float();  
  
        nota2 = Leitura_Float();  
  
        if (nota1 >= 0 && nota2 >= 0) {  
            media = (nota1 + nota2)/2;  
            System.out.println("Media do aluno= " + media);  
        }  
  
    }  
}
```

## **4 – Vantagens da Modularização**

# Modularização - Vantagens

1 - Dividir um problema em subproblemas



# Modularização - Vantagens

2 - Dividir o desenvolvimento entre vários programadores

# Modularização - Vantagens

3- Módulos menores facilitam a depuração.

# Modularização - Vantagens

4- Reutilização de trechos de programas.

# Exercícios

# Exercício 1

Inclua setas para indicar a ordem de retorno dos métodos, a partir da execução do seguinte código C:

```
1  int funcao_c (int c);
2  int funcao_b (int i);
3  int funcao_a (int n);

4  int main () {
5      int result;
6      result = funcao_a(10);
7      printf ("Resultado = %d" , result);
8  }

9  int funcao_c (int c) {
10     return (c/2);
11 }

12 int funcao_b (int i) {
13     int k;
14     k = funcao_c (i-3) * 3 ;
15     return (k);
16 }

17 int funcao_a (int n) {
18     int x;
19     x = funcao_b (n-1) + 6;
20     return (x);
21 }
```

## Exercício 2

Considere o método **abs** descrito abaixo. Escreva instruções para:

- a. apresentar na tela o valor absoluto de -20: \_\_\_\_\_
- b. guardar na variável **aux** o valor absoluto de -20: \_\_\_\_\_

```
float abs (float x) {  
    if (x < 0) x = (-1) * x;  
    return x;  
}
```

# Exercício 2 - Resolução

---

Considere o método **abs** descrito abaixo. Escreva instruções para:

a. apresentar na tela o valor absoluto de -20:

**printf( "%d", abs (-20) );**

b. guardar na variável **aux** o valor absoluto de -20:

**float aux = abs (-20);**

```
float abs (float x) {  
    if (x < 0) x = (-1)* x;  
    return x;  
  
}
```

## Exercício 2 - Resolução Completa Para Teste

---

```
#include "stdio.h"
```

```
float abs (float x) {  
    if (x < 0) x = (-1)* x;  
    return x;  
}
```

```
int main () {  
    float x = 10;  
  
    printf ("O valor de X = %f", abs(x) );  
  
    return 0;  
}
```



# Exercício 3

Informe quais as saídas geradas durante a execução das seguintes aplicações:

```
void muda (int n);  
void muda2 ( );
```

```
int main () {  
    int n=5;  
    printf (" n = %d antes da chamada de muda", n);  
  
    muda (n);  
    printf (" n = %d depois da chamada de muda", n);  
    n=5;  
    printf (" n = %d antes da chamada de muda2", n);  
    muda2();  
    printf (" n = %d depois da chamada de muda2", n);  
}
```

```
void muda (int n) {  
    n = 10;  
    printf (" n = %d dentro da chamada de muda", n);  
}
```

```
void muda2 ( ) {  
    int n=10;  
    printf (" n = %d dentro da chamada de muda2", n);  
}
```

1

3

4

6

2

5

## Exercício 3 - Resolução

---

<b>main</b>
<b>n = 5</b>
Chama <b>muda (5)</b>
<b>Imprime 5</b>
<b>n = 5</b>
Chama <b>muda2 ( )</b>
<b>Imprime 5</b>

<b>muda (5)</b>
<b>n = 5</b>
<b>n = 10</b>
<b>Imprime 10</b>
Retorna para <b>main</b>

<b>muda2 ( )</b>
<b>n = 10</b>
<b>Imprime 10</b>
Retorna para <b>main</b>

## Exercício 4

Escrever uma função chamada **verificaIntervalo** que verifica se um valor inteiro **x** encontra-se no intervalo de limite inferior **min** e limite superior **max**.

```
_____ ( _____ ) {
```

```
}
```

## Exercício 4

Escrever uma função chamada **verificaIntervalo** que verifica se um valor inteiro **x** encontra-se no intervalo de limite inferior **min** e limite superior **max**.

## Exercício 4 - Resolução

---

Escrever uma função (e o método *main* que a invoque) que tenha como parâmetros 3 valores inteiros **a**, **b** e **c** e retorna a **posição do maior** elemento

```
int verificaIntervalo (int min, int max, int valor){  
    int retorno = 0;  
    if (valor >= min and valor <= max) retorno = 1;  
    return (retorno);  
}
```

## Exercício 5

Escrever uma função (e o método *main* que a invoque) que tenha como parâmetros 3 valores inteiros **a**, **b** e **c** e retorna a **posição do maior** elemento.

Exemplo:

Se a = 7, b = 1 e c = 5,

o método deve retornar:

1 como a posição do maior.

## Exercício 5 - Resolução Lucas

---

```
#include <stdio.h>
#include <locale.h>
int main (){
    setlocale(LC_ALL, "Portuguese");
    int a, b, c;
    printf("Inserir o primeiro valor: "); scanf("%d", &a);
    printf("Inserir o segundo valor: "); scanf("%d", &b);
    printf("Inserir o terceiro valor: "); scanf("%d", &c);
    printf("\nPosição do maior elemento: %dª posição", verificaPosicao(a, b,
c));    return 0;
}
int verificaPosicao (int a, int b, int c){
    int maior;
    if(a > b && a > c) maior = 1;
    if(b > a && b > c) maior = 2;
    if(c > a && c > b) maior = 3;
    return maior;
}
```

## Exercício 5 - Resolução

---

Escrever uma função (e o método *main* que a invoque) que tenha como parâmetros 3 valores inteiros **a**, **b** e **c** e retorna a **posição do maior** elemento.

```
int verificaPosicao (int a, int b, int c){  
    int posicaoDoMaiorElemento = 1;  
    if(b > a && b > c) posicaoDoMaiorElemento = 2;  
    if(c > a && c > b) posicaoDoMaiorElemento = 3;  
    posicaoDoMaiorElemento maior;  
}  
  
int main () {  
    printf ("Na sequencia 4-5-9 o maior elemento está na posição %d",  
           verificaPosicao (4,5,9) );  
    printf ("Na sequencia 5-2-1 o maior elemento está na posição %d",  
           verificaPosicao (5,2,1) );  
}
```



## Exercício 6

Explique “o que” faz o método abaixo.

- Seja conciso e evite vícios de linguagem
- Não parafraseie o código.

```
int xxx (int *a, int tamanhoA, int *b, int tamanhoB) {  
  
    if(tamanhoA == 0 && tamanhoB == 0) return 0;  
  
    int aux [tamanhoA + tamanhoB];  
  
    for(int i=0; i < tamanhoA; i++) aux[i] = a[i];  
    for(int i=0; i < tamanhoB; i++) aux[tamanhoA + i] = b[i];  
    return 1;  
}
```

## Exercício 6 - Resolução

---

Vetor **a**



Vetor **b**



Vetor **aux**



## **4 - Passagem de Parâmetros**

# Tipos de Passagem de Parâmetro

Existem duas formas de passagem de parâmetros entre funções:

A. Passagem por Valor

B. Passagem por Referência (Endereço)

## A) Passagem de Parâmetros por Valor

- É a passagem normal do valor dos argumentos para a função.
- Os valores dos argumentos passados não são modificados, pois é fornecida uma **cópia** dos valores para a função.

# Exemplo de Passagem de Parâmetros por Valor

A alteração é feita em uma cópia de n1 e n2 (a,b)

Quando forem testar, renomeie os nomes dos parâmetros de a,b para n1,n2 e veja os resultados !!!

Continua imprimindo 1 para n1 e 1 para n2

```
/* Função com chamada por valor */

#include <stdio.h>
#include <stdlib.h>
int valor(int a, int b)
{
    a = a + 1; /* primeiro argumento foi modificado */
    b = b + 2; /* segundo argumento foi modificado */
    printf("Valores das variaveis dentro da funcao: \n");
    printf("Valor 1 = %d \n", a);
    printf("Valor 2 = %d \n", b);
}

main()
{
    int n1 = 1, n2 = 1 total;
    printf("Valores iniciais de n1 e n2: %d e %d \n", n1, n2);
    printf("Chamando a funcao ... \n");
    valor(n1, n2);
    printf("Valores depois de chamada a funcao: %d e %d \n", n1, n2);
}
```

## Observações sobre o exemplo

As alterações de valor de  $n1$  e  $n2$  na função não são refletidas no main, independente do nome que essas variáveis tenham na função ( $a, b, n1, n2$ ).

A razão pela qual o main não enxerga as alterações da função é:

As variáveis FORAM PASSADOS POR VALOR !!!  
(ou seja, uma cópia, uma "xerox")

## Passagem de Parâmetros por Valor - Outro Exemplo

```
...  
int main ( ) {  
  
    int a = 5, resultado;  
    resultado = funcao_Exemplo (a);  
    printf ("a = %d", a);  
}  
  
int funcao_Exemplo ( int a) {  
    a = a * 2;  
    return a ;  
}
```

A passagem de parâmetro está enviando o VALOR da variável a (UMA CÓPIA)!!

Imprime 5



## B) Passagem de Parâmetros por Referência (ENDEREÇO)

Na **Chamada por Referência** são passados os **endereços de memória** dos argumentos.

- Portanto, os valores podem ser modificados.
- Se forem modificados na função destino, a função que a chamou vai visualizar as alterações.

## Exemplo com Scanf

....

```
int a = 10;  
scanf ("%d" , &a);
```

// Se o usuário digitou 5, o novo valor de a será 5  
// e não mais 10. Ou seja, a função scanf ALTEROU  
// o valor passado como parâmetro.

// Note que passamos o ENDEREÇO de a

# Exemplo de Passagem por Referência

```
#include <stdio.h>
#include <stdlib.h>
int valor(int * a, int * b)
{
    *a = *a + 1; /* primeiro argumento foi modificado */
    *b = *b + 2; /* segundo argumento foi modificado */
    printf("Valores das variaveis dentro da funcao: \n");
    printf("Valor 1 = %d \n", *a);
    printf("Valor 2 = %d \n", *b);
}

main()
{
    int n1 = 1, n2 = 1, total;
    printf("Valores iniciais de n1 e n2: %d e %d \n", n1, n2);
    printf("Chamando a funcao ... \n");
    valor(&n1, &n2); // passado o endereco da variavel
    printf("Valores depois de chamada a funcao: %d e %d \n", n1, n2);
}
```

O \* indica que vai receber um endereço

Na chamada estamos passando o endereço - &

## Passagem de Parâmetros por Referência - Exemplo

```
...  
int main ( ) {  
  
    int a = 5, resultado;  
    resultado=funcao_Exemplo (&a);  
    printf ("a = %d", a);  
}  
  
int funcao_Exemplo ( int *a) {  
    a = a * 2;  
    return a ;  
}
```

A passagem de parâmetro está enviando  
o ENDEREÇO da variável a !!

Imprime 10

Ainda que se escreva (\*b) o resultado será  
o mesmo.

## Outro exemplo de passagem por referência

- Para que os valores enviados para a função **swap** possam retornar ao ponto de chamada – já trocados – é preciso passá-los por referência, e não por valor.
- Na função `main( )` utilizamos a chamada **swap (&i, &j)**, em que “&” representa o endereço da variável, e não seu valor.

```
// Função de troca de valores  
void swap (int * x, int * y)  
{  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

## Importante !

- **As Strings e vetores são sempre chamadas por referência.**  
Quando a linguagem C passa um vetor ou uma string para uma função, o que se passa, na verdade, é o endereço de memória inicial do mesmo.
- **As variáveis declaradas dentro de uma função são locais:**  
Visíveis apenas naquele escopo (dentro daquela função)  
e destruídas após o término de execução da mesma.
- Caso deseje **manter o valor da variável entre as chamadas a uma função**, ela deve ser declarada **static**.

## Exemplo de Passagem de Vetor como Parâmetro

```
float media (int n, float *vnotas) {  
    int i;  
    float m, soma = 0;  
    for (i = 0; i < n; i++) soma += vnotas[i];  
    m = soma / n;  
    return m;  
}
```

```
int main () {  
    float vnotas[10];  
    float media_notas;  
    int i;  
  
    /* leitura das notas */  
    for (i = 0; i < 10; i++) {  
        printf ("Digite os valores das notas: ");  
        scanf ("%f", &vnotas[i]);  
    }  
  
    //chamada da função  
    media_notas = media(10, vnotas);  
  
    printf ( "\nMedia = %.1f \n", media_notas );  
  
    system("pause");  
    return 0;  
}
```

## **5 - Argumentos da Linha de Comando**



# Argumentos da Linha de Comando

- A função `main()` tem, em sua definição completa a possibilidade de passar argumentos para a linha de comando do sistema operacional.
- Por ser a primeira função a ser chamada pelo programa, seus argumentos são passados junto com o comando que executa o programa.

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

int main(int argc, char *argv[]) // aqui a definição completa de main()
{
    printf("Numero de argumentos de main(): %d \n", argc);
    printf("Valores dos argumentos de main(): %s e %s \n", argv[0], argv[1]);
    return 0;
}
```

- **argc**: o número de argumentos passado para o programa
- **argv**: vetor de strings com os valores de tais argumentos.  
Armazena o nome do programa e os argumentos em forma de cadeia de caracteres.  
**argv[0]** é o nome do programa,  
**argv[1]**, o primeiro argumento passado ao programa, e assim por diante.

## **6 - Recursividade**

# Introdução

```
int fatorial (int n) {  
    int fat;  
    if (n==0) fat = 1;  
    else     fat = n * fatorial (n-1);  
    return fat;  
}
```

- Fatorial de 4 =  $4 * 3 * 2 * 1 = 24$
- Qual o tipo de retorno da função acima ?
- Qual a principal característica da implementação da função fatorial ?

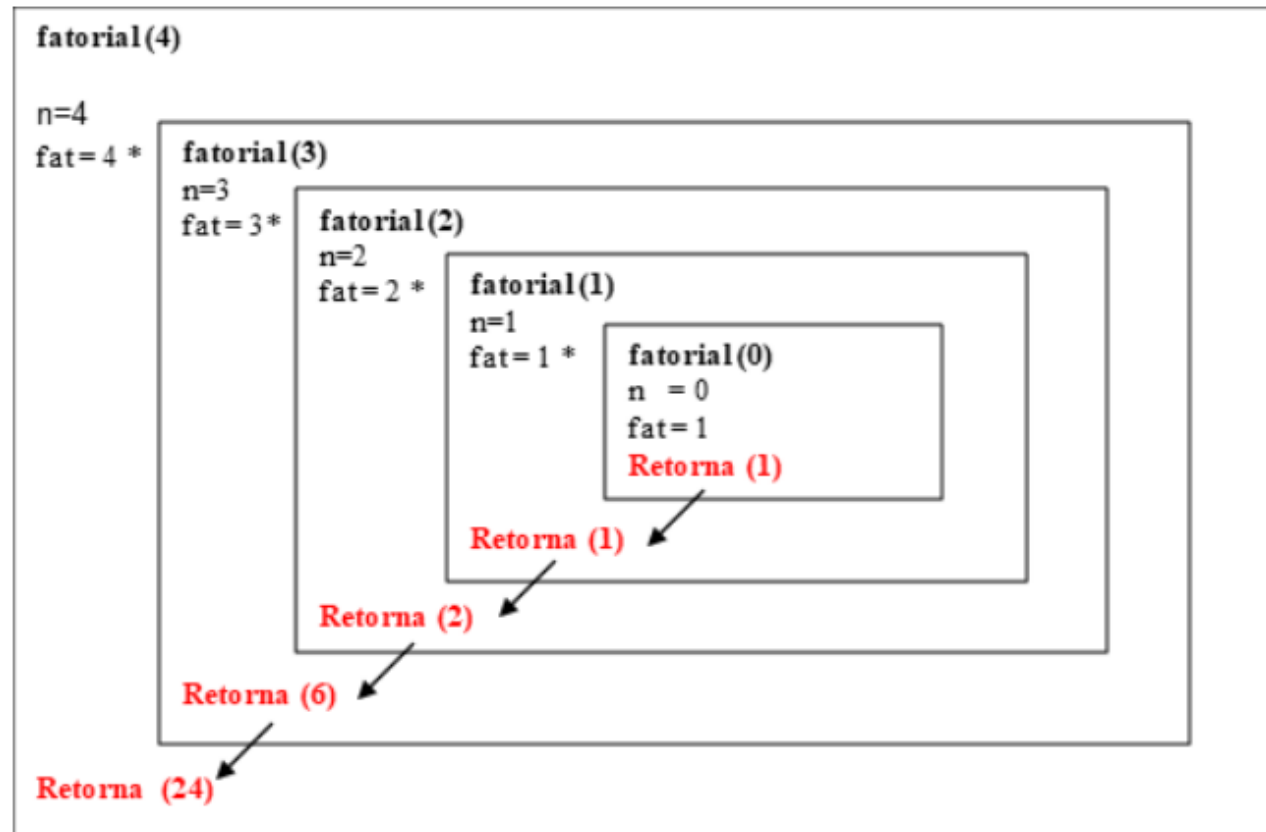
# 01 - Teste de Mesa da Recursividade

```
int fatorial (int n) {  
    int fat;  
    if (n==0) fat = 1;  
    else      fat = n * fatorial (n-1);  
    return fat;  
}
```

Efetue o teste de mesa da chamada da função fatorial, considerando como parâmetro o valor 4.

**fatorial(4);**

# Resposta




```
int fatorial (int n) {
    int fat;
    if (n==0) fat = 1;
    else fat = n * fatorial (n-1);
    return fat;
}
```

## 02 - Teste de Mesa da Recursividade


Quais serão as saídas geradas quando a função for chamada passando o valor 2 ?

**magico** (2);

```
void magico (int n)
```

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

```
        if (n < 5) magico (n + 1);
```

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

```
}
```

# Resposta

```
void magico (int n)

    printf ("%d", n);

    if (n < 5)  magico (n + 1);

    printf ("%d", n);

}
```

## Saídas Geradas

2  
3  
4  
5  
5  
4  
3  
2

magico (2)

n=2

Imprime 2

Chama

magico (3)

n=3

Imprime 3

Chama

magico (4)

n=4

Imprime 4

Chama

magico (5)

n=5

Imprime 5

Imprime 5

Imprime 4

Imprime 3

Imprime 2

## 03 - Função Recursiva versus Iterativa

Reescreva a função recursiva fatorial abaixo, de forma iterativa, ou seja, substituindo a recursividade (chamada a ela mesma) pelo uso de comandos de repetição (iteração).

```
int fatorial (int n) {  
    int fat;  
    if (n==0) fat = 1;  
    else      fat = n * fatorial (n-1);  
    return fat;  
}
```



# Resposta

```
int fatorial (int n) {  
    int fat;  
    if (n==0) fat = 1;  
    else      fat = n * fatorial (n-1);  
    return fat;  
}
```

```
int fatorial (int n){  
    int fat = 1, i;  
  
    for(i=1; i<=n; i++){  
        fat = fat * i;  
    }  
    return fat;  
}
```

```
int fatorial (int n){  
    int fat = 1;  
  
    for( ;n > 1; n--){  
        fat = fat * n;  
    }  
    return fat;  
}
```

```
int fatorial(int n){  
    int fat=n-1;  
    while(fat>0) {  
        n*=fat;  
        fat--;  
    }  
    return fat;  
}
```