Introdução à Programação
AULA 10 – Funções e Procedimentos

Profª. Glaucia M. M. Campos
glauciamelissa@uern.br

Universidade do Estado do Rio Grande do Norte - CaN

Hierarquia entre Funções

- Sempre é possível que um programa principal chame uma função que por sua vez chame outra função ... e assim sucessivamente.
- Quando isto acontece dizemos que a função chamadora tem hierarquia maior (ou superior) a função chamada. Ou que a função chamadora está em um nível hierárquico superior a função chamada.
- Quando isto ocorre, devemos ter o cuidado de definir (ou incluir) as funções em ordem crescente de hierarquia, isto é, uma função chamada é escrita antes de uma função chamadora. Isto se deve ao fato de que o compilador deve conhecer uma função antes de que chamada seja compilada.

Hierarquia entre Funções – Exemplo

```c
#include <stdio.h>

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

int multsum(int a, int b){
    int m=0, i;

    for(i=1;i<=b;i++){
        m = soma(m,a);
    }
    return m;
}

int main(){

    int n1,n2,res;

    printf("Digite um número: \n");
    scanf("%d", &n1);
    printf("Digite outro número: \n");
    scanf("%d", &n2);

    res = multsum(n1,n2);
    printf("O produto entre os números é %d", res);

}
```

Regras de Escopo para Variáveis

- A regra de escopo define o âmbito de validade de variáveis, ou seja, define onde as variáveis e funções são reconhecidas.
- Em C, uma variável só pode ser usada após ser declarada. Isto por que o processador deve reservar um local da memória para armazenar os valores atribuídos à variável.
- Porém o local, do programa, onde uma variável é declarada define ainda seu escopo de validade. Uma variável pode ser local, global ou formal de acordo com o local de declaração.

Variáveis Locais

- Uma variável é dita local, se for declarada dentro do bloco de uma função .
- Uma variável local tem validade apenas dentro do bloco onde é declarada, isto significa que podem ser apenas acessadas e modificadas dentro de um bloco.
- O espaço de memória alocado para esta variável é criado quando a execução do bloco é iniciada e destruído quando encerrado, assim variáveis de mesmo nome mas declaradas em blocos distintos, são para todos os efeitos, variáveis distintas.

Variáveis Locais - Exemplos

```c
int main(){

    int n1,n2,res;

    printf("Digite um número: \n");
    scanf("%d", &n1);
    printf("Digite outro número: \n");
    scanf("%d", &n2);

    res = multsum(n1,n2);
    printf("O produto entre os números é %d", res);

}
```

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

int multsum(int a, int b){
    int m=0, i;

    for(i=1;i<=b;i++){
        m = soma(m,a);
    }
    return m;
}
```

Variáveis Formais

- É uma variável local declarada na lista de parâmetros de uma função . Deste modo, tem validade apenas dentro da função onde é declarada, porém serve de suporte para os valores passados pelas funções.
- As variáveis formais na declaração da função e na chamada da função podem ter nomes distintos. A única exigência é de que sejam do mesmo tipo .
- Por serem variáveis locais, os valores que uma função passa para outra não são alterados pela função chamada. (passagem por valor).

Variáveis Formais - Exemplo

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

int multsu(int a, int b){
    int m=0, i;

    for(i=1;i<=b;i++){
        m = soma(m,a);
    }
    return m;
}
```

Variáveis Globais

- Uma variável é dita global, se for declarada fora do bloco de uma função . Uma variável global tem validade no escopo de todas as funções, isto é, pode ser acessadas e modificada por qualquer função .
- O espaço de memória alocado para esta variável é criado no momento de sua declaração e destruído apenas quando o programa é encerrado.

Variáveis Globais - Exemplo

```c
#include <stdio.h>

int vg=0;

int somavg(int a, int b){
    int s;
    s = a + b;
    return s;
}

int multsum(int a, int b){
    int m=0, i;

    for(i=1;i<=b;i++){
        m = somavg(m,a);
    }
    vg = vg - 1;
    return m;
}

int main(){

    int n1,n2,res;

    printf("Digite um número: \n");
    scanf("%d", &n1);
    printf("Digite outro número: \n");
    scanf("%d", &n2);
    vg = n1+n2;

    res = multsum(n1,n2);
    printf("O produto entre os números é %d\n", res);
    printf("O valor de vg é %d\n", vg);
}
```

Recursividade

- A recursividade é uma das mais importantes vantagens do uso de funções em C.
- Recursão é o processo pelo qual uma função chama a si mesma repetidamente um número finito de vezes. Este recurso é muito útil em alguns tipos de algoritmos chamados de algoritmos recursivos.
- Vejamos um exemplo clássico: cálculo do fatorial de um número. A definição de fatorial é:
  - n! = n x (n-1) x (n-2) x ... x 3 x 2 x 1 , n é um inteiro positivo
  - 0! = 1
- Uma propriedade (facilmente verificável) dos fatoriais é que: n! = n . (n-1)!

Recursividade

- Esta propriedade é chamada de propriedade recursiva: o fatorial de um numero pode ser calculado através do fatorial de seu antecessor .
- Podemos utilizar esta propriedade para escrevermos uma rotina recursiva para o calculo de fatoriais. Para criarmos uma rotina recursiva, em C, basta criar uma chamada a própria função dentro dela mesma.
- Uma função recursiva cria a cada chamada um novo conjunto de variáveis locais. Não existe ganho de velocidade ou espaço de memória significativo com o uso de funções recursivas.

Recursividade

- Teoricamente uma algoritmo recursivo pode ser escrito de forma iterativa e vice-versa.
- A principal vantagem destes algoritmos é que algumas classes de algoritmos [de inteligência artificial, simulação numérica, busca e ordenação em arvore binária, etc.] são mais facilmente implementadas com o uso de rotinas recursivas.

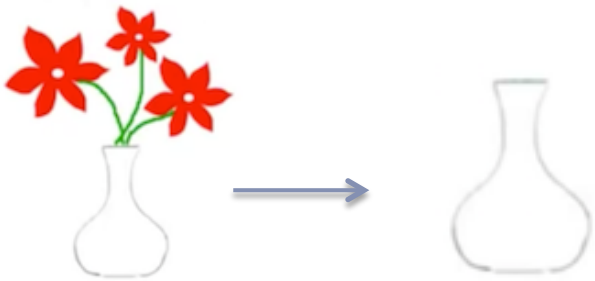
Recursividade

```c
/*
 * Rercusividade
 *
 *  Created on: 24 de set de 2017
 *      Author: macbook
 */

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

void imprime(int n){
    int i;
    for (i=1;i<=n;i++){
        printf("Linha %d \n", i);
    }
}

int main(){
    imprime(5);
    printf("Fim do programa! \n");
    return 0;
}
```

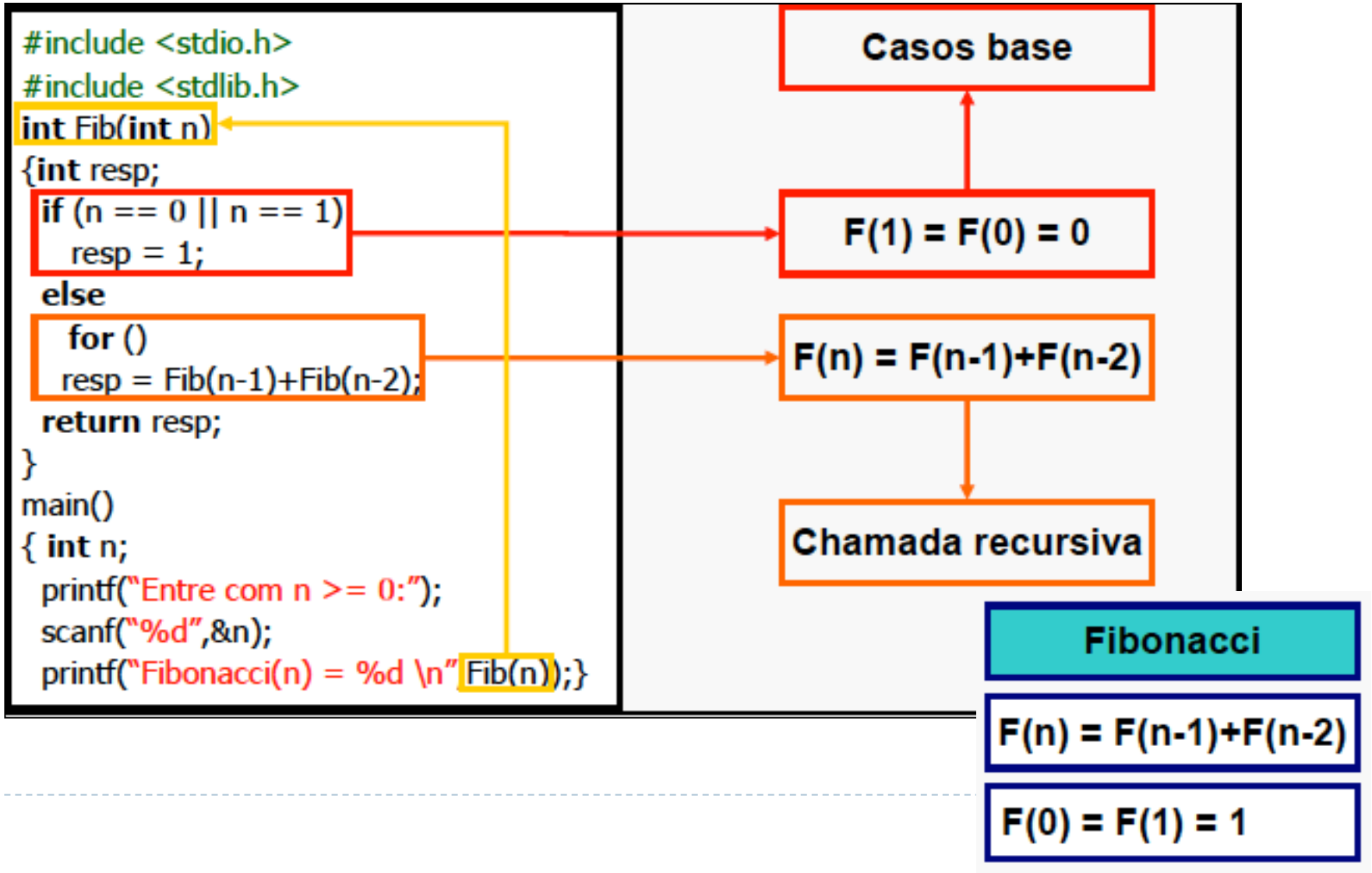
Recursividade

```c
/*
 * Recursividade
 *
 *  Created on: 24 de set de 2017
 *      Author: macbook
 */


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

int main(){
    //Como esvaziar um jarro com três flores

    return 0;
}
```

Recursividade

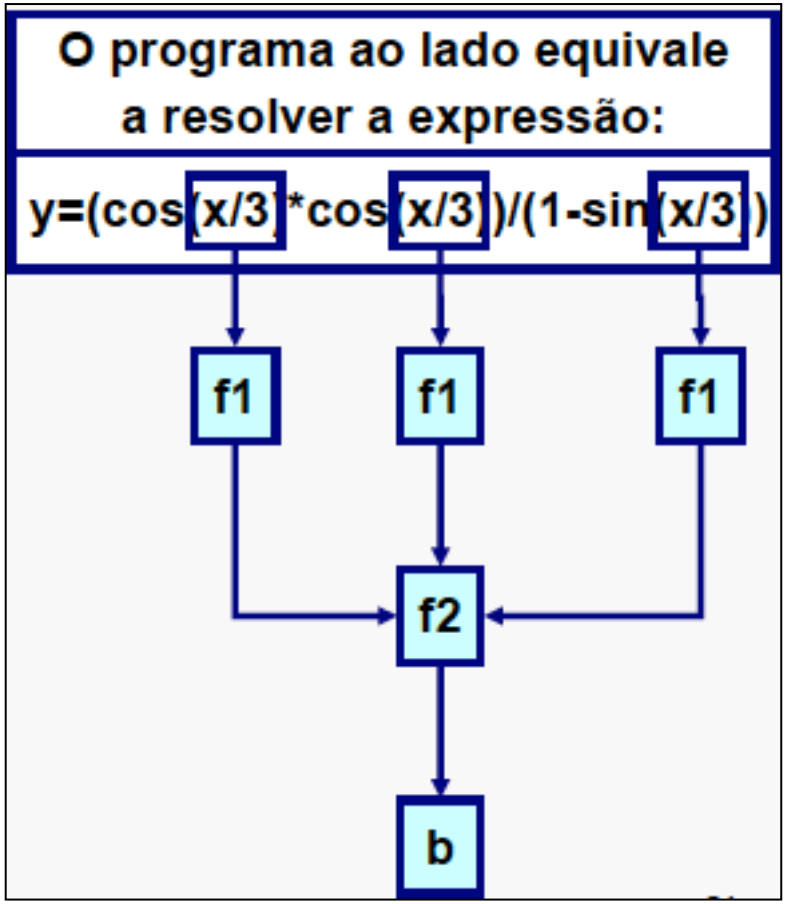
```c
/*
 * Recursividade
 *
 *  Created on: 24 de set de 2017
 *      Author: macbook
 */


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

int main(){
    //Como esvaziar um jarro com quatro flores



    return 0;
}
```

Recursividade – Exemplo 1



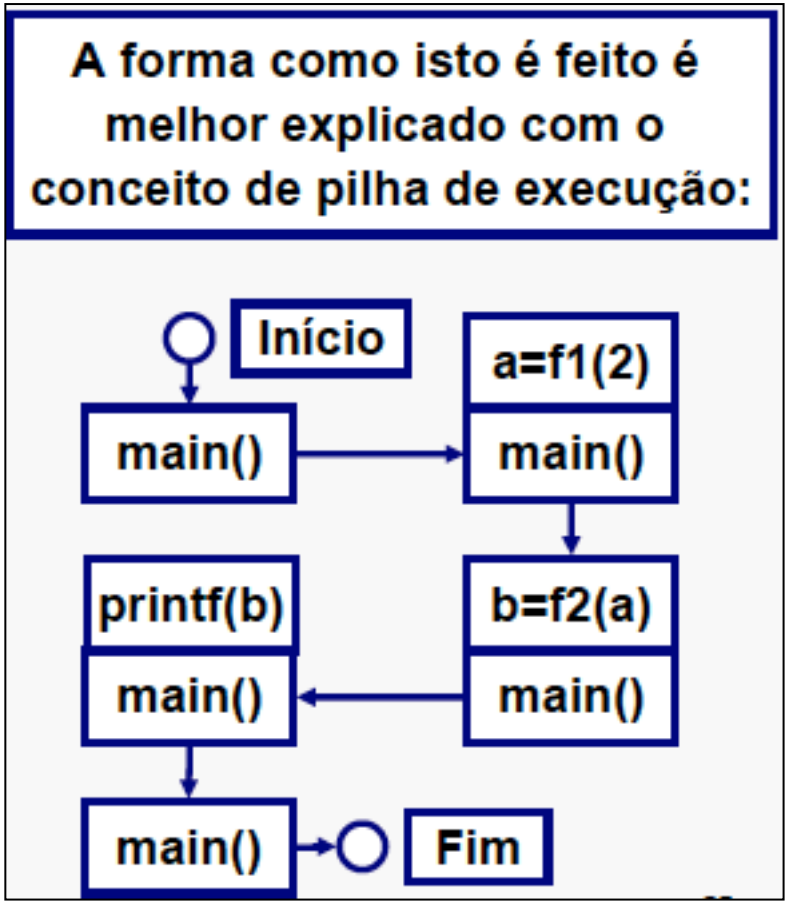
Recursividade – Exemplo 2

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int f1(int x)
{ return (x/3);}

int f2(int x)
{ return ((cos(x)*cos(x))/(1-sin(x)));}

main()
{int a, b;
  a = f1(2);
  b = f2(a);
  printf("b = f2(f1(2)) = %d \n", b);
}
```

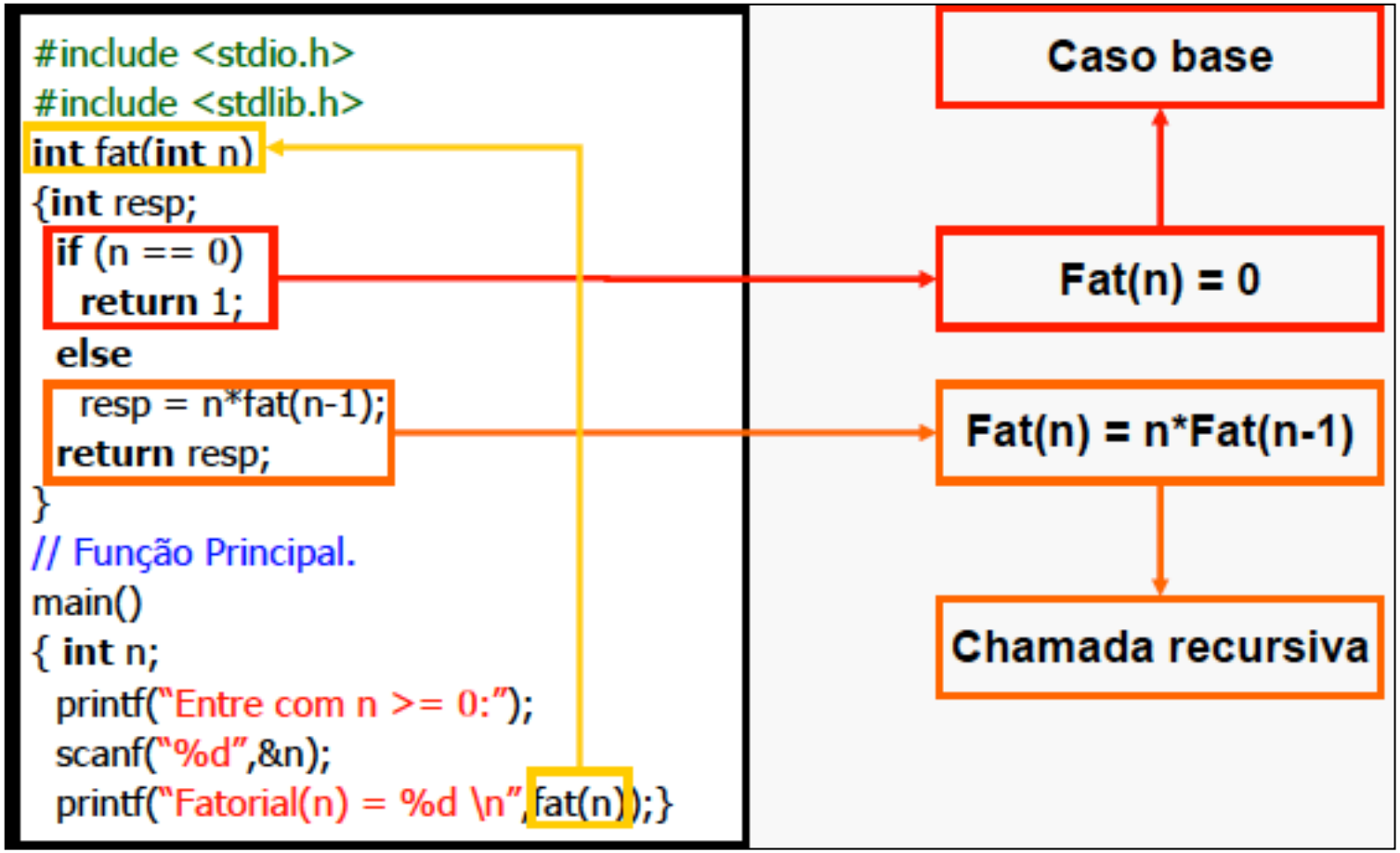


Recursividade – Exemplo 2

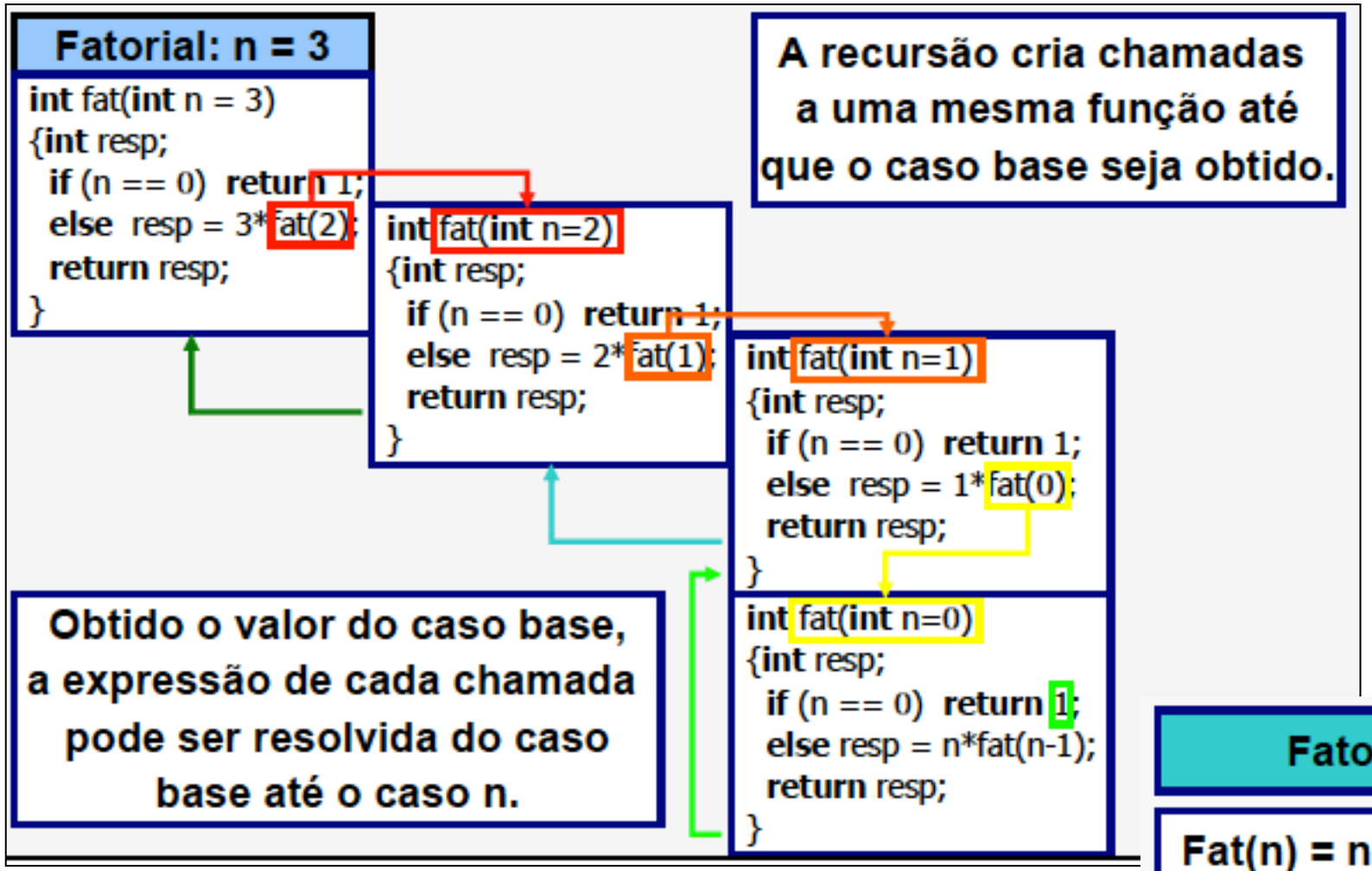
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int f1(int x)
{ return (x/3);}

int f2(int x)
{ return ((cos(x)*cos(x))/(1-sin(x)));}

main()
{int a, b;
  a = f1(2);
  b = f2(a);
  printf("b = f2(f1(2)) = %d \n", b);
}
```



Recursividade – Exemplo 3



Recursividade – Exemplo 3



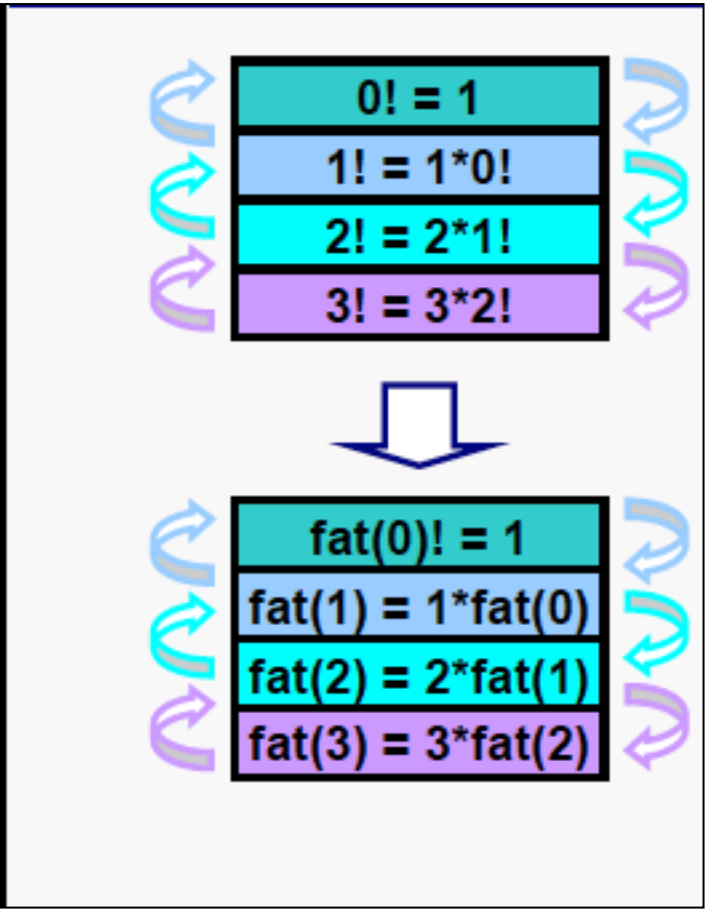
Fatorial

Fat(n) = n*Fat(n-1)

Fat(0) = 1

Recursividade – Exemplo 3

```
#include <stdio.h>
#include <stdlib.h>
int fat(int n)
{int resp;
 if (n == 0)
   return 1;
 else
   resp = n*fat(n-1);
 return resp;
}
// Função Principal.
main()
{ int n;
  printf("Entre com n >= 0:");
  scanf("%d",&n);
  printf("Fatorial(n) = %d \n",fat(n));}
```



Passagem de parâmetros

- O mecanismo de informar os valores a serem processados pela função chama-se passagem de parâmetros
- A Linguagem C define duas categorias de passagem de parâmetros: passagem por valor e passagem por endereço (ou passagem por referência).

Passagem por valor

- Considere o exemplo abaixo:

```
void alterar(int x, int y, int z)
{
  printf("Valores recebidos ... %d, %d e %d\n",x,y,z);
  x++;
  y++;
  z++;
  printf("Valores alterados ... %d, %d e %d\n",x,y,z);
}

void main()
{
  int a = 1, b = 2, c = 3;

  alterar(a,b,c);
  printf("Valores finais ..... %d, %d e %d\n",a,b,c);
}
```

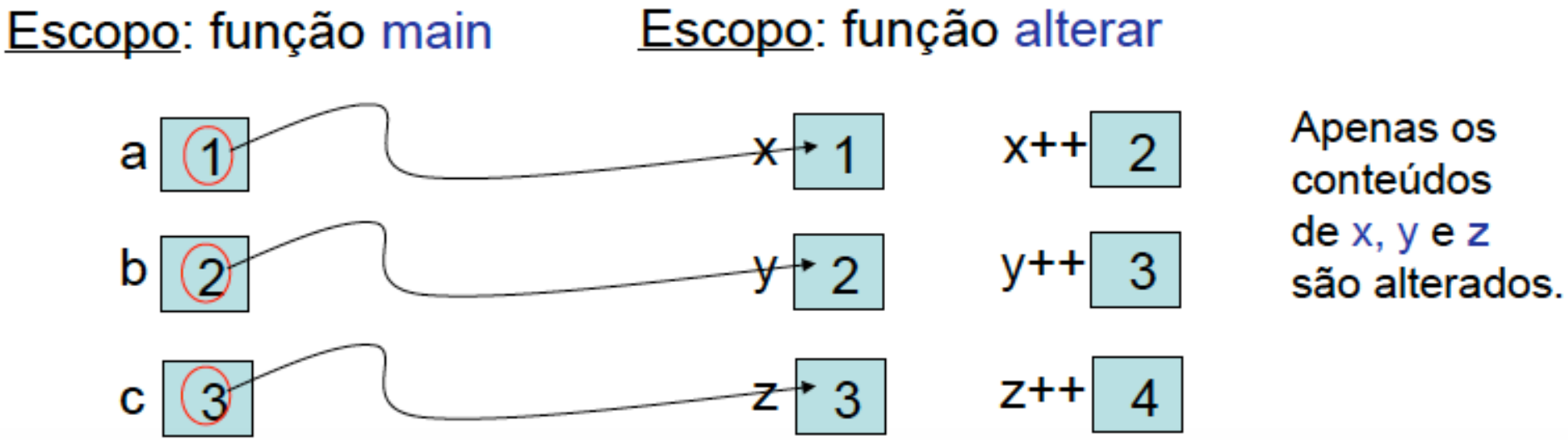
- O que este programa irá exibir?

```
Valores recebidos  ... 1, 2 e 3
Valores alterados ...  2, 3 e 4
Valores finais .....   1, 2 e 3
```

Passagem por valor

- Observe que os valores das variáveis a, b e c não foram modificados na função alterar. Por quê?
- O tipo de passagem de parâmetros utilizado é por valor.

Ou seja, são feitas apenas cópias dos valores das variáveis a, b, e c nas variáveis x, y e z.



Passagem por referência

- Mas, e se quisermos que a função modifique os valores das variáveis a, b e c passadas a ela como parâmetros?
- Neste caso, em vez de passar para a função os valores destas variáveis, é preciso passar os seus endereços.

Passagem por referência

- Ou seja:

Endereço	Conteúdo	Variável
F000	1	a
F010	2	b
F020	3	c

- Sabemos, portanto, que:

  &a = F000 (endereço de a);

  &b = F010 (endereço de b);

  &c = F020 (endereço de c);

  a = 1, b = 2, c = 3 (valores das variáveis).

Passagem por referência

- Considere uma variável declarada como:

```
int *x;
```

- x é um ponteiro para int, ou seja, x é uma variável que armazena o endereço de uma variável do tipo int.

- Considere agora que:
- Neste caso, x armazena o valor F000.

```
x = &a;
```

Define-se *x, como sendo o valor contido na posição de memória apontada por x. Ou seja, *x vale 1.

Passagem por referência

- Considere o exemplo anterior:

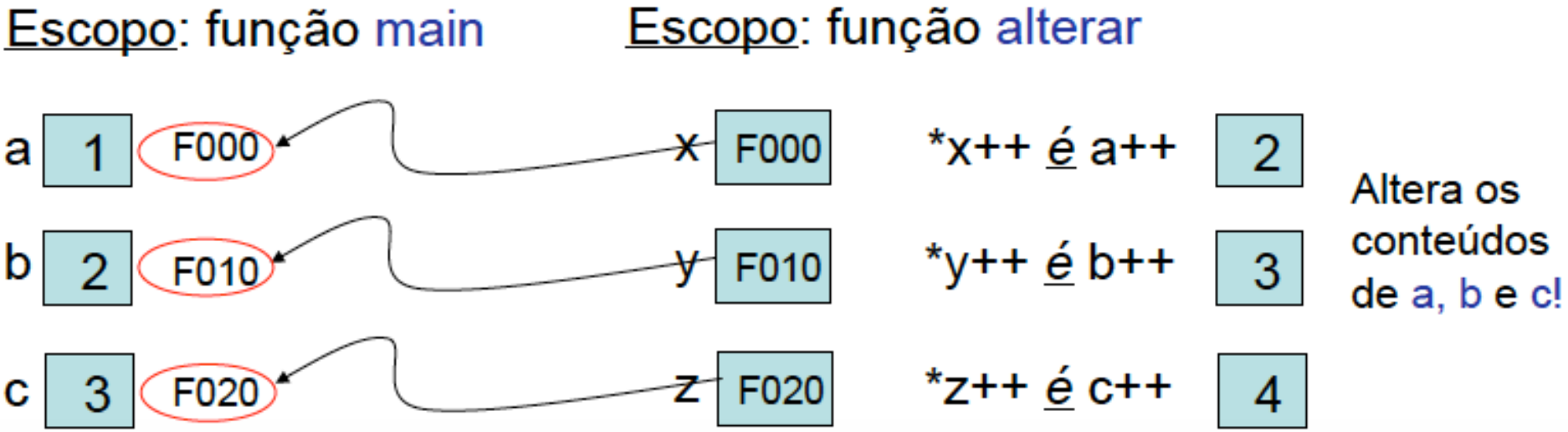
```
void alterar(int *x, int *y, int *z)
{
  printf("Valores recebidos ... %d, %d e %d\n",*x,*y,*z);
  *x++;
  *y++;
  *z++;
  printf("Valores alterados ... %d, %d e %d\n",*x,*y,*z);
}

void main()
{
  int a = 1, b = 2, c = 3;

  alterar(&a,&b,&c);
  printf("Valores finais ..... %d, %d e %d\n",a,b,c);
}
```

- O que este programa vai exibir?

```
Valores recebidos  ... 1, 2 e 3
Valores alterados ...  2, 3 e 4
Valores finais .....  2, 3 e 4
```

Passagem por referência

- Observe agora que os valores das variáveis a, b e c foram modificados na função alterar. Por quê?
- O tipo de passagem de parâmetros utilizado é por referência. Ou seja, são passados os endereços das variáveis a, b, e c para os ponteiros x, y e z



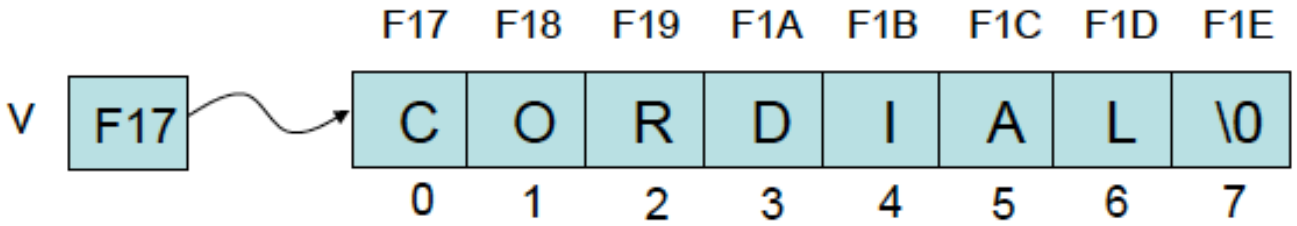
Vetores como parâmetros

- No caso de uma função ter como parâmetro um vetor, temos um caso particular de grande importância.

Por quê? Porque o nome de um vetor nada mais é que um ponteiro para sua primeira posição.

- Exemplo:

```
char v[8];
```



Vetores como parâmetros

- Mas, então como é possível usarmos a notação:

```
nome_do_vetor[índice]
```

- Isto pode ser facilmente explicado, desde que se entenda que a notação acima é absolutamente equivalente a:

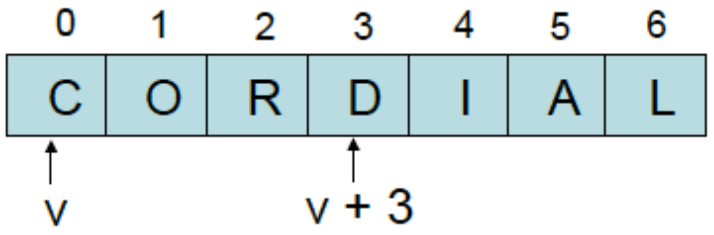
```
*(nome_do_vetor + índice)
```

- Mas, o que significa somar (ou subtrair) um valor a um ponteiro?

Vetores como parâmetros

- Quando incrementamos um ponteiro, ele passa a apontar para o próximo valor do mesmo tipo.

- Exemplo: ao incrementar um ponteiro para char, ele anda 1 byte na memória e ao incrementar um ponteiro para double ele anda 8 bytes.




```
 v ≡ &v[0]
*v ≡ v[0] = 'C'
*(v + 3) ≡ v[3] = 'D'
```

São equivalentes!!!

Vetores como parâmetros

- Logo, como o nome de um vetor é também um ponteiro, a passagem de parâmetro para vetores é sempre por referência.

Assim, qualquer modificação ocorrida no vetor dentro da função será, na realidade, feita sobre o parâmetro usado na chamada.

```
void ordenar_por_selecao(int x[], int n)
{
  int menor,pos;
  int i,k = 0;
     ...

// Classificar vetor
ordenar_por_selecao(a,n);
     ...
```

Alterações no vetor **x** dentro da função ordenar_por_selecao serão também realizadas no vetor **a**.

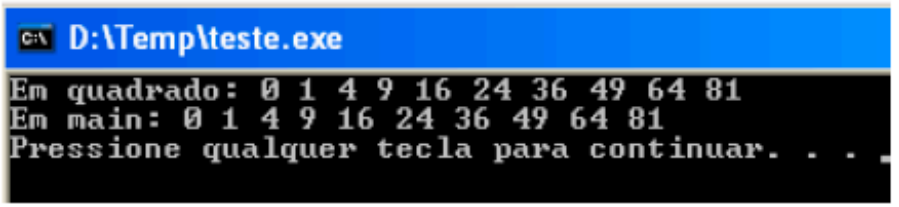
Na definição da função, podemos substituir **int x[ ]** por **int x[TAM_MAX]** ou ainda **int *x**.

Vetores como parâmetros

- O que devemos fazer se desejarmos que os elementos de um vetor, passado como parâmetro para uma função, não sejam alterados?
- Resposta: dentro da função é preciso atribuir os elementos do vetor a uma variável local.
- Considere o exemplo mostrado a seguir:

Vetores como parâmetros

```c
#define TAM_MAX 10

void quadrado(int v[], int n)
{
  int i;
  for (i = 0; i < n; i++)
    v[i] = pow(v[i],2);
  printf("Em quadrado: ");
  for (i = 0; i < n; i++)
    printf("%d ",v[i]);
  printf("\n");
}

void main()
{
  int i;
  int a[TAM_MAX] = { 0,1,2,3,4,5,6,7,8,9 };
  quadrado(a,10);
  printf("Em main: ");
  for (i = 0; i < 10; i++)
    printf("%d ",a[i]);
  printf("\n");
  system("pause");
}
```



Sim! A passagem é por referência.

Os elementos do vetor a terão seus valores modificados pela função quadrado?

Vetores como parâmetros

```c
#define TAM_MAX 10

void quadrado(int v[], int n)
{
  int i, x[TAM_MAX];
  for (i = 0; i < n; i++)
    x[i] = v[i];
  for (i = 0; i < n; i++)
    x[i] = pow(x[i],2);
  printf("Em quadrado: ");
  for (i = 0; i < n; i++)
    printf("%d ",x[i]);
  printf("\n");
}

void main()
{
  int i;
  int a[TAM_MAX] = { 0,1,2,3,4,5,6,7,8,9 };
  quadrado(a,10);
  printf("Em main: ");
  for (i = 0; i < 10; i++)
    printf("%d ",a[i]);
  printf("\n");
  system("pause");
}
```

Atenção! Observe que a cópia dos elementos foi feita um a um. O que aconteceria se fizéssemos: x = v?

Não! Foi feita uma cópia de v em x.

E agora, os elementos do vetor a terão seus valores modificados pela função quadrado?

Exemplo 1 :: *Vetor como parâmetro*

```
#include <stdio.h>
void imprimeVetor(int m[3]){
    int j;
    for(j=0; j<3;j++)
        printf("%d \n", m[j]);
}
int main() {
    int vett[3]= {1, 2, 3};
    imprimeVetor(vet);
    return 0;
}
```

Exemplo 2 :: *Vetor como parâmetro*

```
#include <stdio.h>
void imprimeVetor(int m[], int j){
        int i;
        for(i=0; i<j;i++)
        printf("%d \n", m[j]);
}
int main() {
    int vet[3]= {1, 2, 3};
    imprimeVetor(vet, 3);
    return 0;
}
```

Exemplo 3 :: *Matriz como parâmetro*

```
#include <stdio.h>
void imprimeMatriz(int m[][2], int n){
    int i,j;
    for (i=0; i<n; i++)
        for(j=0; j<2;j++)
            printf("%d \n", m[i][j]);
}
int main(){
    int mat[3][2] = {{1,2}, {3,4}, {5,6}};
    imprimeMatriz(mat,3);
    return 0;
}
```

Exemplo 4 :: *Matriz como parâmetro*

```
#include <stdio.h>
void imprimeMatriz(int m[3][2]){
    int i,j;
    for (i=0; i<n; i++)
        for(j=0; j<2;j++)
            printf("%d \n", m[i][j]);
}
int main(){
    int mat[3][2] = {{1,2}, {3,4}, {5,6}};
    imprimeMatriz(mat);
    return 0;
}
```

Exemplo 5 :: *Matriz como parâmetro*

```
#include <stdio.h>
void imprimeMatriz(int m[3][2]){
    int i,j;
    for (i=0; i<n; i++)
        for(j=0; j<2;j++)
            printf("%d \n", m[i][j]);
}
int main(){
    int mat[3][2] = {{1,2}, {3,4}, {5,6}};
    imprimeMatriz(mat[3][2]);
    return 0;
}
```

Não funciona!!!!!

Exemplo 6 :: *Matriz como parâmetro*

```
#include <stdio.h>
void imprimeMatriz(int m[3][2]){
    int i,j;
    for (i=0; i<n; i++)
        for(j=0; j<2;j++)
            printf("%d \n", m[i][j]);
}
int main(){
    int mat[3][2] = {{1,2}, {3,4}, {5,6}};
    imprimeMatriz(mat[][]);
    return 0;
}
```

Não funciona!!!!!

Exemplo 7 :: *Struct como parâmetro*

```
#include <stdio.h>
struct Ponto{
    int x, y;
};
void ImprimeValor(int n){
    printf("%d", n);
}
int main() {
    struct Ponto p1 = {10,20};
    ImprimeValor(p1.x);
    ImprimeValor(p1.y);
    return 0;
}
```

Passando apenas um campo da struct

Exemplo 8 :: *Struct como parâmetro*

```
#include <stdio.h>
struct Ponto{
    int x, y;
};
void ImprimeValor(struct Ponto p){
    printf("%d", p.x);
    printf("%d", p.y);
}
int main() {
    struct Ponto p1 = {10,20};
    ImprimeValor(p1);
    return 0;
}
```

Passando toda a struct como parâmetro

Exercícios

1. Crie uma função que recebe como parâmetro um número inteiro e devolve o seu dobro.
2. Faça uma função que receba a data atual (dia, mês e ano em inteiro) e exiba-a na tela no formato textual por extenso. Exemplo: Data: 01/01/2000, Imprimir: 1 de janeiro de 2000.
3. Faça uma função para verificar se um número é positivo ou negativo. Sendo que o valor de retorno será 1 se positivo, -1 se negativo e 0 se for igual a 0.
4. Faça uma função para verificar se um número é um quadrado perfeito. Um quadrado perfeito é um número inteiro não negativo que pode ser expresso como o quadrado de outro número inteiro. Ex: 1, 4, 9...

Exercícios

5. Faça uma função e um programa de teste para o cálculo do volume de uma esfera. Sendo que o raio é passado por parâmetro. $V = 4/3 * \pi * R3$.
6. Faça uma função que receba 3 números inteiros como parâmetro, representando horas, minutos e segundos, e os converta em segundos.
7. Faça uma função que receba uma temperatura em graus Celsius e retorne-a convertida em graus Fahrenheit. A fórmula de conversão é: $F = C * (9.0/5.0) + 32.0$, sendo F a temperatura em Fahrenheit e C a temperatura em Celsius.
8. Faça uma função que receba duas strings e retorne a intercalação letra a letra da primeira com a segunda string. A string intercalada deve ser retornada na primeira string.

Exercícios

9. Considerando a estrutura struct Vetor{ float x; float y; float z;}; para representar um vetor no R3, implemente uma função que calcule a soma de dois vetores. Essa função deve obedecer ao protótipo: void soma (struct Vetor* v1, struct Vetor* v2, struct Vetor* res); onde os parâmetros v1 e v2 são ponteiros para os vetores a serem somados, e o parâmetro res é um ponteiro para uma estrutura vetor onde o resultado da operação deve ser armazenado.
10. Faça uma rotina que receba como parâmetro um vetor de caracteres e seu tamanho. A função deverá ler uma string do teclado, caractere por caractere usando a função getchar() ate que o usuário digite enter ou o tamanho máximo do vetor seja alcançado.

Exercícios

11. Considerando a estrutura: struct Ponto{ int x; int y; }; para representar um ponto em uma grade 2D, implemente uma função que indique se um ponto está localizado dentro ou fora de um retângulo. O retângulo é definido por seus vértices inferior esquerdo v1 e superior direito v2. A função deve retornar 1 caso o ponto esteja localizado dentro do retângulo e 0 caso contrário. Essa função deve obedecer ao protótipo: int dentroRet (struct Ponto* v1, struct Ponto* v2, struct Ponto* p);

Exercícios

12. Faça uma função que receba, por parâmetro, duas matrizes quadradas de ordem N (A e B), e retorna uma matriz C, também por parâmetro, que seja o produto matricial de A e B.
13. Faça uma função que receba uma matriz de 3 x 3 elementos. Calcule e retorne a soma dos elementos que estão abaixo da diagonal principal.
14. Faça uma função que receba um vetor de reais e retorne a média dele.
15. Faça uma função que receba um vetor de reais e modifique cada um dos elementos multiplicando-os por 3.

Exercícios :: Recursividade

- 1. Faça uma função recursiva que receba um número inteiro positivo N e imprima todos os números naturais de 0 até N em ordem crescente.
- 2. Faça uma função recursiva que receba um número inteiro positivo N e imprima todos os números naturais de 0 até N em ordem decrescente.
- 3. Faça uma função recursiva que receba um número inteiro positivo par N e imprima todos os números pares de 0 até N em ordem crescente.
- 4. Faça uma função recursiva que receba um número inteiro positivo par N e imprima todos os números pares de 0 até N em ordem decrescente

Exercícios:: Recursividade

- 12. Crie um programa que contenha uma função recursiva para encontrar o menor elemento em um vetor.
- 13. Faça uma função recursiva que receba um número inteiro positivo N e retorne o superfatorial desse numero. O superfatorial de um número N é definido pelo produto dos N primeiros fatoriais de N. Assim, o superfatorial de 4 é: $sf(4) = 1! * 2! * 3! * 4! = 288$
- 14. Crie um programa que receba um vetor de números reais com 100 elementos. Escreva uma função recursiva que inverta a ordem dos elementos presentes no vetor.
- 15. Faça uma função recursiva que permita somar os elementos de um vetor de inteiros.

Referências

- Medina, Marco; Fertig, Cristina. Algoritmos e Programação: teoria e prática. São Paulo: Novatec Editora, 2006.
- Lopes, Anita; Garcia, Guto. Introdução à Programação: 500 algoritmos resolvidos. Rio de Janeiro: Editora Campus, 2002.
- Mizrani, Victorine Viviane. Treinamento em Linguagem C, Módulo 1. Editora Makron Books.
- Transparências modificadas do professor Dr. Flavio Luiz Cardeal Pádua, do Centro Federal de Educação Tecnológica de Minas Gerais
- Transparências modificadas do professor Robson Fidalgo, da UFRPE.