

Algoritmos e Estruturas de Dados I

1º Período Engenharia da Computação

Prof. Edwaldo Soares Rodrigues

Email: edwaldo.rodrigues@uemg.br

Material adaptado do prof. André Backes

Funções

Função

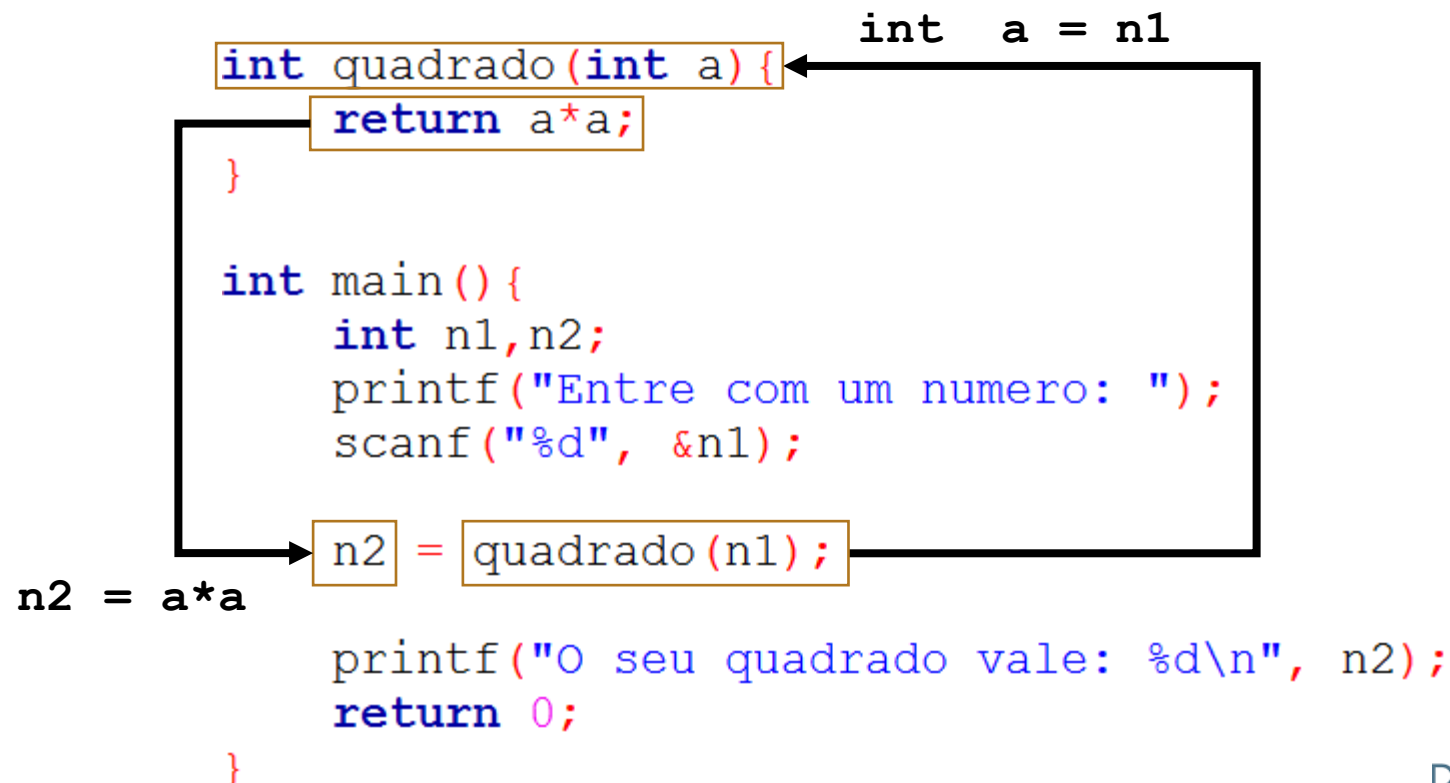
- Funções são blocos de código que podem ser nomeados e chamados de dentro de um programa.
 - **printf()**: função que escreve na tela;
 - **scanf()**: função que lê o teclado;

Função

- Facilitam a estruturação e reutilização do código;
 - Estruturação: programas grandes e complexos são construídos bloco a bloco;
 - Reutilização: o uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros;

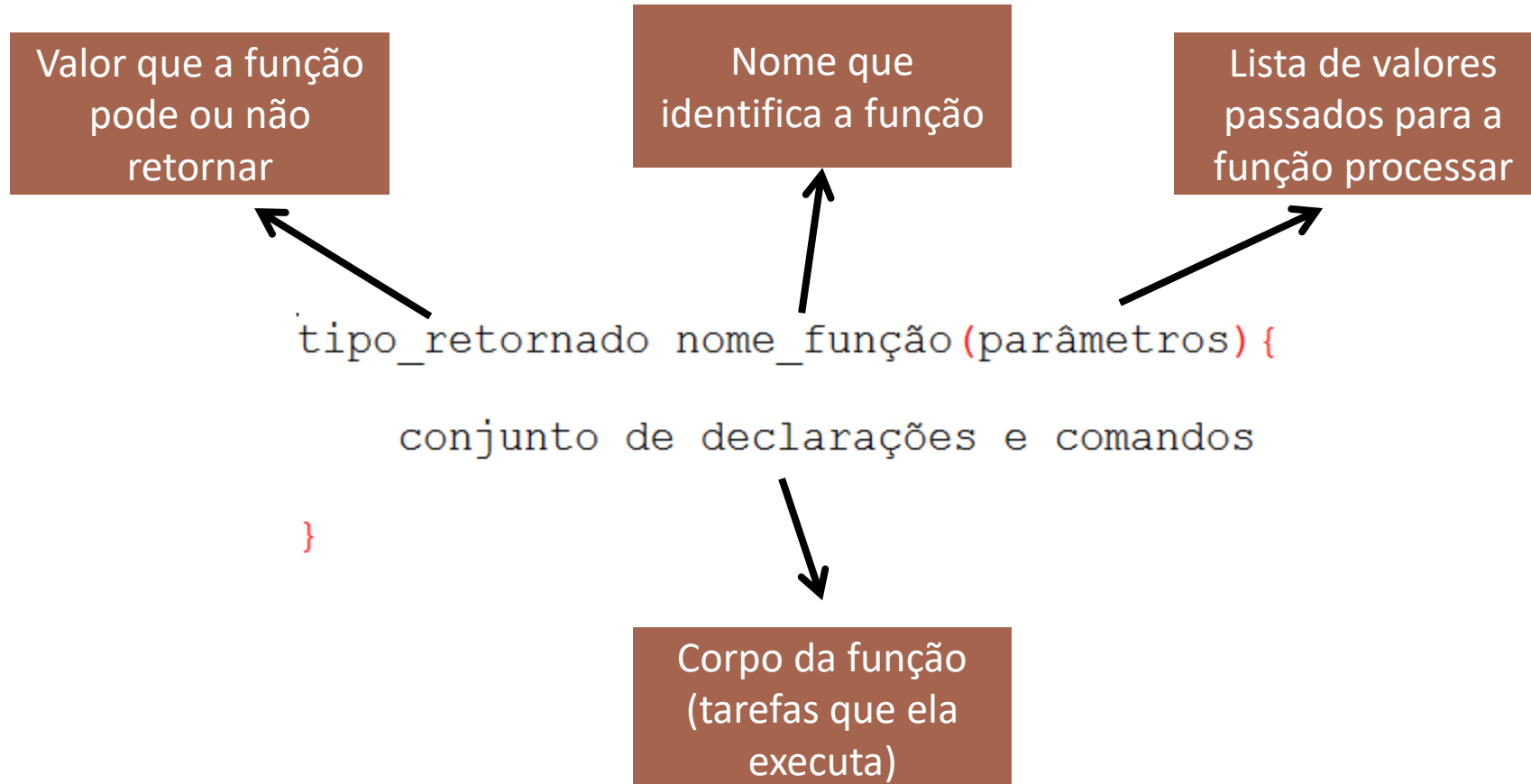
Função – Ordem de Execução

- Ao chamar uma função, o programa que a chamou é pausado até que a função termine a sua execução



Função - Estrutura

- Forma geral de uma função:



Função - Corpo

- O corpo da função é a sua alma;
 - É formado pelos comandos que a função deve executar;
 - Ele processa os parâmetros (se houver), realiza outras tarefas e gera saídas (se necessário);
 - Similar a cláusula **main()**;

```
int main() {  
  
    //conjunto de declarações e comandos  
  
    return 0;  
}
```

Função - Corpo

- De modo geral, evita-se fazer operações de leitura e escrita dentro de uma função;
 - Uma função é construída com o intuito de realizar uma tarefa específica e bem-definida;
 - As operações de entrada e saída de dados (funções **scanf()** e **printf()**) devem ser feitas em quem chamou a função (por exemplo, na **main()**);
 - Isso assegura que a função construída possa ser utilizada nas mais diversas aplicações, garantindo a sua generalidade;

Função - Parâmetros

- A declaração de parâmetros é uma lista de variáveis juntamente com seus tipos:
 - *tipo1 nome1, tipo2 nome2, ... , tipoN nomeN;*
- Pode-se definir quantos parâmetros achar necessários;



```
//Declaração CORRETA de parâmetros
int soma(int x, int y) {
    return x + y;
}


//Declaração ERRADA de parâmetros
int soma(int x, y) {
    return x + y;
}
```

Função - Parâmetros

- É por meio dos parâmetros que uma função recebe informação do programa principal (isto é, de quem a chamou);
- Não é preciso fazer a leitura das variáveis dos parâmetros dentro da função

```
int x = 2;  
int y = 3;  
  
int soma(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int z = soma(2, 3);  
  
    return 0;  
}
```

```
int soma(int x, int y) {  
  
    scanf("%d", &x);  
    scanf("%d", &y);  
  
    return x + y;  
}
```



Função - Parâmetros

- Podemos criar uma função que não recebe nenhum parâmetro de entrada;
- Isso pode ser feito de duas formas:
 - Podemos deixar a lista de parâmetros vazia;
 - Podemos colocar **void** entre os parênteses;

```
void imprime() {  
    printf("Teste\n");  
}
```

```
void imprime(void) {  
    printf("Teste\n");  
}
```

Função - Retorno

- Uma função pode ou não retornar um valor;
 - Se ela retornar um valor, alguém deverá receber este valor;
 - Uma função que retorna nada é definida colocando-se o tipo **void** como valor retornado;
- Podemos retornar qualquer valor válido em C;
 - tipos pré-definidos: int, char, float e double;
 - tipos definidos pelo usuário: struct;

Comando return

- O valor retornado pela função é dado pelo comando **return**;
- Forma geral:
 - **return** *valor ou expressão*;
 - **return**;
 - Usada para terminar uma função que não retorna valor;
- É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função;

Comando return

Função com retorno de valor

```
int soma(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int z = soma(2, 3);  
  
    return 0;  
}
```

Função sem retorno de valor

```
void imprime() {  
    printf("Teste\n");  
}
```

Comando return

- Uma função pode ter mais de uma declaração **return**;
- Quando o comando **return** é executado, a função termina imediatamente;
- Todos os comandos restantes são **ignorados**;

```
int maior(int x, int y) {  
    if(x > y)  
        return x;  
    else  
        return y;  
    printf("Esse texto nao sera impresso\n");  
}
```

Declaração de Funções

- Funções devem declaradas antes de serem utilizadas, ou seja, antes da cláusula **main**.
 - Uma função criada pelo programador pode utilizar qualquer outra função, inclusive as que foram criadas

```
int quadrado(int a) {  
    return a*a;  
}
```

```
int main() {  
    int n1, n2;  
    printf("Entre com um numero: ");  
    scanf("%d", &n1);  
  
    n2 = quadrado(n1);  
  
    printf("O seu quadrado vale: %d\n", n2);  
    return 0;  
}
```


Declaração de Funções

- Podemos definir apenas o protótipo da função antes da cláusula **main**;
- O protótipo apenas indica a existência da função;
- Desse modo ela pode ser declarada após a cláusula `main()`;

```
tipo_retornado nome_função (parâmetros);
```

Declaração de Funções

- Exemplo de protótipo:

```
int quadrado(int a);
```

```
int main() {  
    int n1, n2;  
    printf("Entre com um numero: ");  
    scanf("%d", &n1);  
  
    n2 = quadrado(n1);  
  
    printf("O seu quadrado vale: %d\n", n2);  
    return 0;  
}  
  
int quadrado(int a) {  
    return a*a;  
}
```

Escopo

- Funções também estão sujeitas ao escopo das variáveis;
- O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa;
 - Variáveis Locais;
 - Variáveis Globais;
 - Parâmetros formais;

Escopo

- Variáveis locais são aquelas que só têm validade dentro do bloco no qual são declaradas;
- Um bloco começa quando abrimos uma chave e termina quando fechamos a chave;
- Ex.: variáveis declaradas dentro da função;

```
int fatorial (int n){  
    if (n == 0)  
        return 1;  
    else{  
        int i;  
        int f = 1;  
        for(i = 1; i <= n; i++)  
            f = f * i;  
        return f;  
    }  
}
```

Escopo

- Parâmetros formais são declarados como sendo as entradas de uma função;
- O parâmetro formal é uma variável local da função.
- Ex.:
 - x é um parâmetro formal

```
float quadrado(float x);
```

Escopo

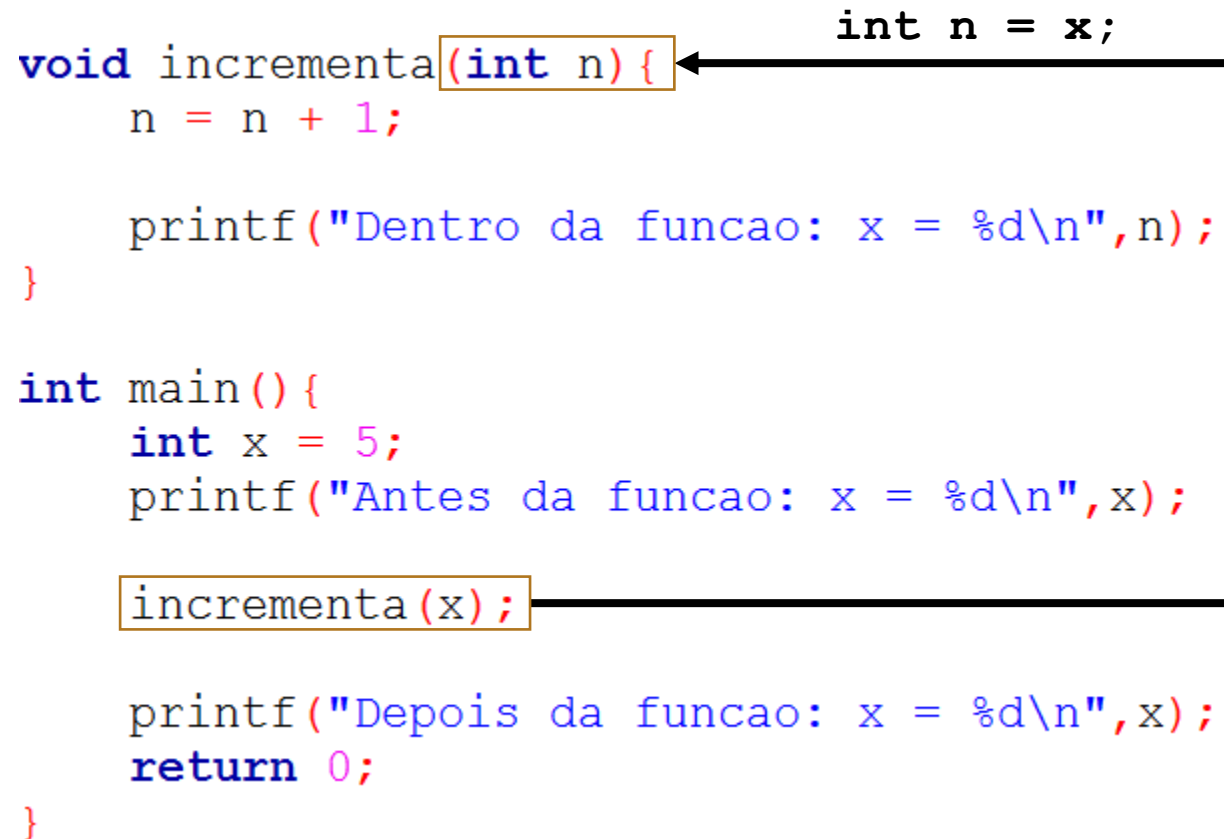
- Variáveis globais são declaradas fora de todas as funções do programa;
- Elas são conhecidas e podem ser alteradas por todas as funções do programa;
 - Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local;
- ***Evite variáveis globais!***

Passagem de Parâmetros

- Na linguagem C, uma possibilidade de passagem de parâmetros é por **valor**, ou seja, uma cópia do valor do parâmetro é feita e passada para a função;
- Mesmo que esse valor mude dentro da função, nada acontece com o valor de fora da função;

Passagem por valor

```
void incrementa(int n) {  
    n = n + 1;  
  
    printf("Dentro da funcao: x = %d\n", n);  
}  
  
int main() {  
    int x = 5;  
    printf("Antes da funcao: x = %d\n", x);  
  
    incrementa(x);  
  
    printf("Depois da funcao: x = %d\n", x);  
    return 0;  
}
```



Saída:

```
Antes da funcao: x = 5  
Dentro da funcao: x = 6  
Depois da funcao: x = 5
```


Passagem por referência

- Quando se quer que o valor da variável mude dentro da função, usa-se passagem de parâmetros por ***referência***;
- Neste tipo de chamada, não se passa para a função o valor da variável, mas a sua ***referência*** (seu endereço na memória);

Passagem por referência

- Utilizando o endereço da variável, qualquer alteração que a variável sofra dentro da função será refletida fora da função;
- Ex: função **scanf()**;

Passagem por referência

- Ex: função **scanf()**:

- Sempre que desejamos ler algo do teclado, passamos para a função **scanf()** o nome da variável onde o dado será armazenado;
- Essa variável tem seu valor modificado dentro da função **scanf()**, e seu valor pode ser acessado no programa principal;

```
int main() {  
    int x = 5;  
    printf("Antes do scanf: x = %d\n", x);  
    printf("Digite um numero: ");  
    scanf("%d", &x);  
    printf("Depois do scanf: x = %d\n", x);  
  
    return 0;  
}
```

Passagem por referência

- Para passar um parâmetro por referência, coloca-se um asterisco “*” na frente do nome do parâmetro na declaração da função:

```
//passagem de parâmetro por valor  
void incrementa(int n);  
  
//passagem de parâmetro por referência  
void incrementa(int *n);
```

- Ao se chamar a função, é necessário agora utilizar o operador “&”, igual como é feito com a função **scanf()**:

```
//passagem de parâmetro por valor  
int x = 10;  
incrementa(x);  
  
//passagem de parâmetro por referência  
int x = 10;  
incrementa(&x);
```

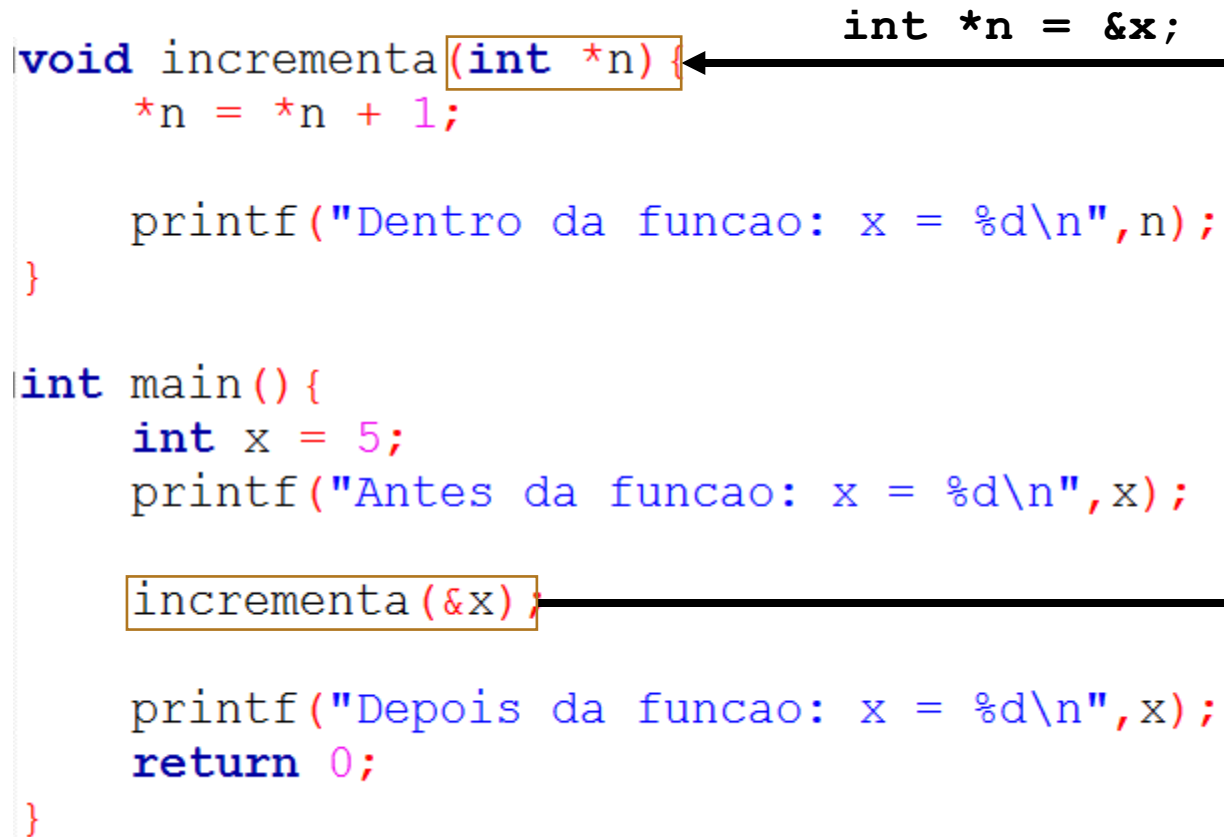
Passagem por referência

- No corpo da função, é necessário usar colocar um asterisco “*” sempre que se desejar acessar o conteúdo do parâmetro passado por referência.

```
//passagem de parâmetro por valor
void incrementa(int n) {
    n = n + 1;
}
//passagem de parâmetro por referência
void incrementa(int *n) {
    *n = *n + 1;
}
```

Passagem por referência

```
void incrementa(int *n) {  
    *n = *n + 1;  
  
    printf("Dentro da funcao: x = %d\n", n);  
}  
  
int main() {  
    int x = 5;  
    printf("Antes da funcao: x = %d\n", x);  
  
    incrementa(&x);  
  
    printf("Depois da funcao: x = %d\n", x);  
    return 0;  
}
```



Saída:

Antes da funcao: x = 5

Dentro da funcao: x = 6

Depois da funcao: x = 6

Passagem por referência

- A maneira mais fácil de identificar quando se deve passar um parâmetro por referência ou por valor é quando a função deve “retornar” mais de um valor:
 - Em C uma função só pode retornar um único valor;
 - Desta forma, quando uma função precisa “retornar” mais de um valor os parâmetros da mesma deverão ser passados por referência, já que se os valores for alterado no escopo da função, automaticamente também serão alterados no escopo da função chamadora;

Exercício

- Crie uma função que troque o valor de dois números inteiros passados por referência.

Exercício

- Crie uma função que troque o valor de dois números inteiros passados por referência.

```
void Troca (int*a, int*b) {  
    int temp;  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Arrays como parâmetros

- Para utilizar arrays como parâmetros de funções alguns cuidados simples são necessários;
 - Arrays são sempre passados por referência para uma função;
 - A passagem de arrays ***por referência*** evita a cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função, o que afetaria o desempenho do programa;

Arrays como parâmetros

- É necessário declarar um segundo parâmetro (em geral uma variável inteira) para passar para a função o tamanho do array separadamente.
- Quando passamos um array por parâmetro, independente do seu tipo, o que é de fato passado é o endereço do primeiro elemento do array;

Arrays como parâmetros

- Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime(int *m, int n);  
void imprime(int m[], int n);  
void imprime(int m[5], int n);
```

Arrays como parâmetros

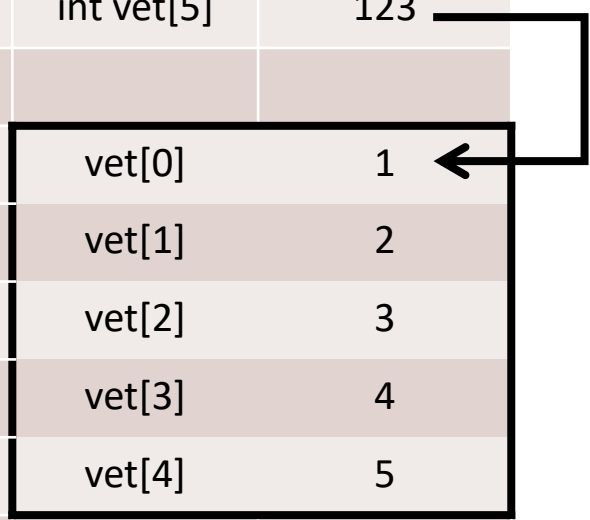
- Exemplo:
 - Função que imprime um array:

```
void imprime(int *m, int n){
    int i;
    for (i=0; i< n;i++)
        printf ("%d \n", m[i]);
}

int main (){
    int vet[5] = {1,2,3,4,5};
    imprime(vet,5);

    return 0;
}
```

| Memória | | |
|---------|------------|----------|
| posição | variável | conteúdo |
| 119 | | |
| 120 | | |
| 121 | int vet[5] | 123 |
| 122 | | |
| 123 | vet[0] | 1 |
| 124 | vet[1] | 2 |
| 125 | vet[2] | 3 |
| 126 | vet[3] | 4 |
| 127 | vet[4] | 5 |
| 128 | | |



Arrays como parâmetros

- Vimos que para arrays, não é necessário especificar o número de elementos para a função;

```
void imprime (int*m, int n);  
void imprime (int m[], int n);
```

- No entanto, para arrays com mais de uma dimensão, é necessário especificar o tamanho de todas as dimensões, exceto a primeira;

```
void imprime (int m[][5], int n);
```

Arrays como parâmetros

- Na passagem de um array para uma função, o compilador precisa saber o tamanho de cada elemento, não o número de elementos;
- Uma matriz pode ser interpretada como um array de arrays:
 - **int m[4][5]**: array de 4 elementos onde cada elemento é um array de 5 posições inteiras;

Arrays como parâmetros

- Logo, o compilador precisa saber o tamanho de cada elemento do array;

```
int m[4][5]
```

```
void imprime (int m[][5], int n);
```

- Na notação acima, informamos ao compilador que estamos passando um array, onde cada elemento dele é outro array de 5 posições inteiras;

Arrays como parâmetros

- Isso é necessário para que o programa saiba que o array possui mais de uma dimensão e mantenha a notação de um conjunto de colchetes por dimensão;
- As notações abaixo funcionam para arrays com mais de uma dimensão. Mas o array é tratado como se tivesse apenas uma dimensão dentro da função;

```
void imprime (int*m, int n);  
void imprime (int m[], int n);
```

Struct como parâmetro

- Podemos passar uma struct por parâmetro ou por referência;
- Temos duas possibilidades:
 - Passar por parâmetro toda a struct;
 - Passar por parâmetro apenas um campo específico da struct;

Struct como parâmetro

- Passar por parâmetro apenas um campo específico da struct:
 - Valem as mesmas regras vistas até o momento;
 - Cada campo da struct é como uma variável independente. Ela pode, portanto, ser passada individualmente por **valor** ou por **referência**;

Struct como parâmetro

- Passar por parâmetro toda a struct:
- Passagem por valor:
 - Valem as mesmas regras vistas até o momento;
 - A struct é tratada como uma variável qualquer e seu valor é copiado para dentro da função;
- Passagem por referência:
 - Valem as regras de uso do asterisco “*” e operador de endereço “&;
 - Devemos acessar o conteúdo da struct para somente depois acessar os seus campos e modificá-los;
 - Uma alternativa é usar o **operador seta** “->;

Struct como parâmetro

Usando “*”

```
struct ponto {  
    int x, y;  
};  
  
void atribui(struct ponto *p) {  
    (*p).x = 10;  
    (*p).y = 20;  
}  
  
struct ponto p1;  
  
atribui(&p1);
```

Usando “->”

```
struct ponto {  
    int x, y;  
};  
  
void atribui(struct ponto *p) {  
    p->x = 10;  
    p->y = 20;  
}  
  
struct ponto p1;  
  
atribui(&p1);
```

Recursão

- Na linguagem C, uma função pode chamar outra função;
 - A função `main()` pode chamar qualquer função, seja ela da biblioteca da linguagem (como a função `printf()`) ou definida pelo programador (função `imprime()`);
- Uma função também pode chamar a si própria;
 - A qual chamamos de ***função recursiva***;

Recursão

- A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo;
- Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número:
 - $3! = 3 * 2!$
 - $4! = 4 * 3!$
 - $n! = n * (n - 1)!$

Recursão

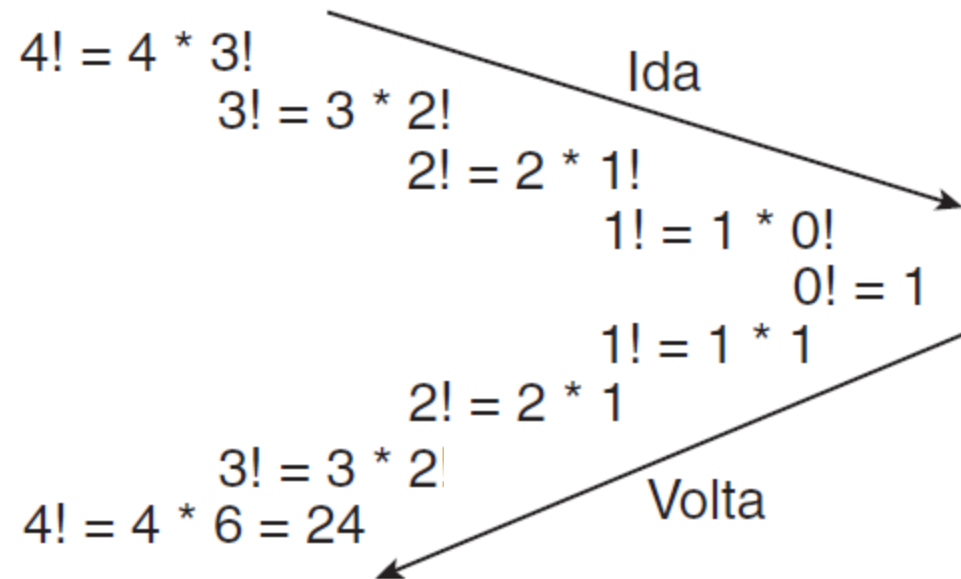
$$0! = 1$$

$$1! = 1 * 0!$$

$$2! = 2 * 1!$$

$$3! = 3 * 2!$$

$$4! = 4 * 3!$$



$$n! = n * (n - 1)! : \text{fórmula geral}$$

$$0! = 1 : \text{caso-base}$$

Recursão

Com Recursão

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

Sem Recursão

```
int fatorial (int n){  
    if (n == 0)  
        return 1;  
    else{  
        int i;  
        int f = 1;  
        for(i = 1; i <= n; i++)  
            f = f * i;  
        return f;  
    }  
}
```

Recursão

- Em geral, formulações recursivas de algoritmos são frequentemente consideradas "mais enxutas" ou "mais elegantes" do que formulações iterativas;
- Porém, algoritmos recursivos tendem a necessitar de mais espaço do que algoritmos iterativos;

Recursão

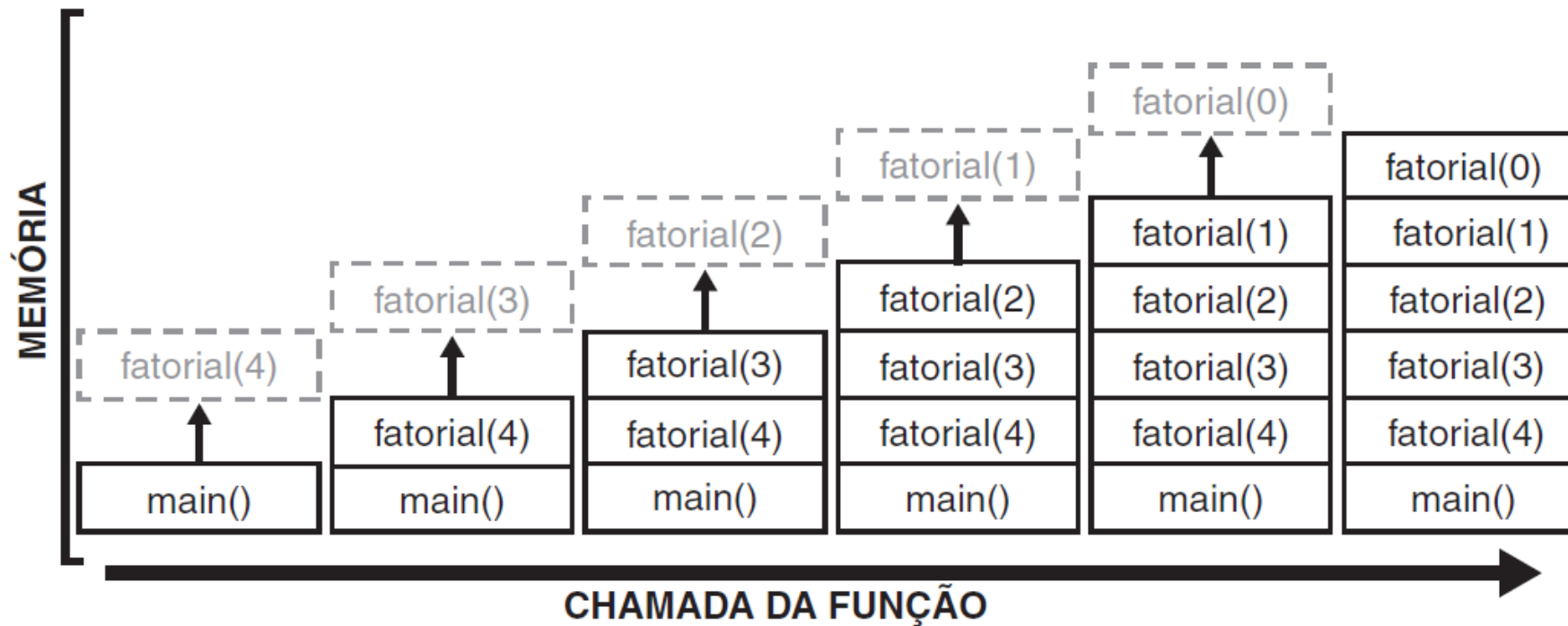
- Todo cuidado é pouco ao se fazer funções recursivas;
 - Critério de parada (caso base): determina quando a função deverá parar de chamar a si mesma;
 - O parâmetro da chamada recursiva deve ser sempre modificado, de forma que a recursão chegue a um término;

```
int fatorial (int n) {  
    if (n == 0) //critério de parada  
        return 1;  
    else /*parâmetro de fatorial sempre muda*/  
        return n*fatorial(n-1);  
}
```

Recursão

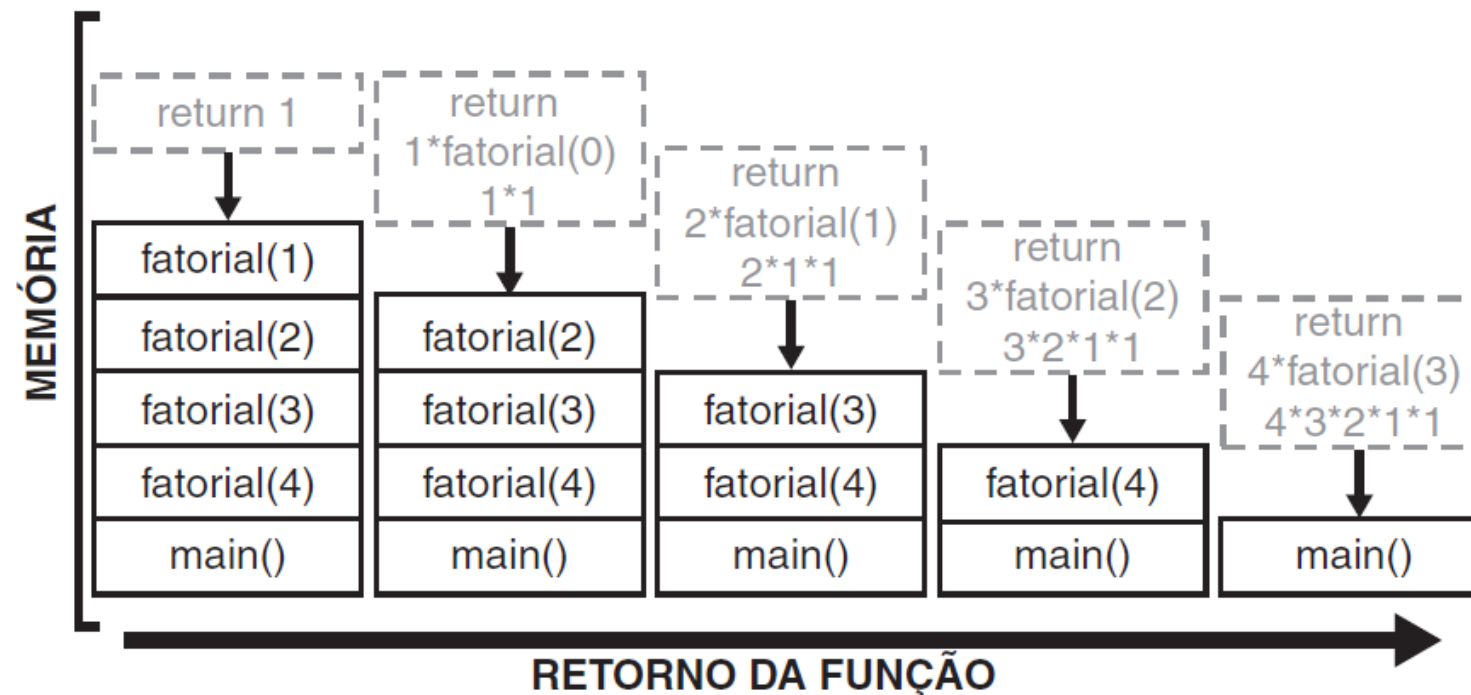
- O que acontece na chamada da função fatorial com um valor como n = 4?

```
int x = fatorial(4);
```



Recursão

- Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão;



Fibonacci

- Essa sequência é um clássico da recursão
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula a seguir;

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

- Sua solução recursiva é muito elegante ...

Recursão

Sem Recursão

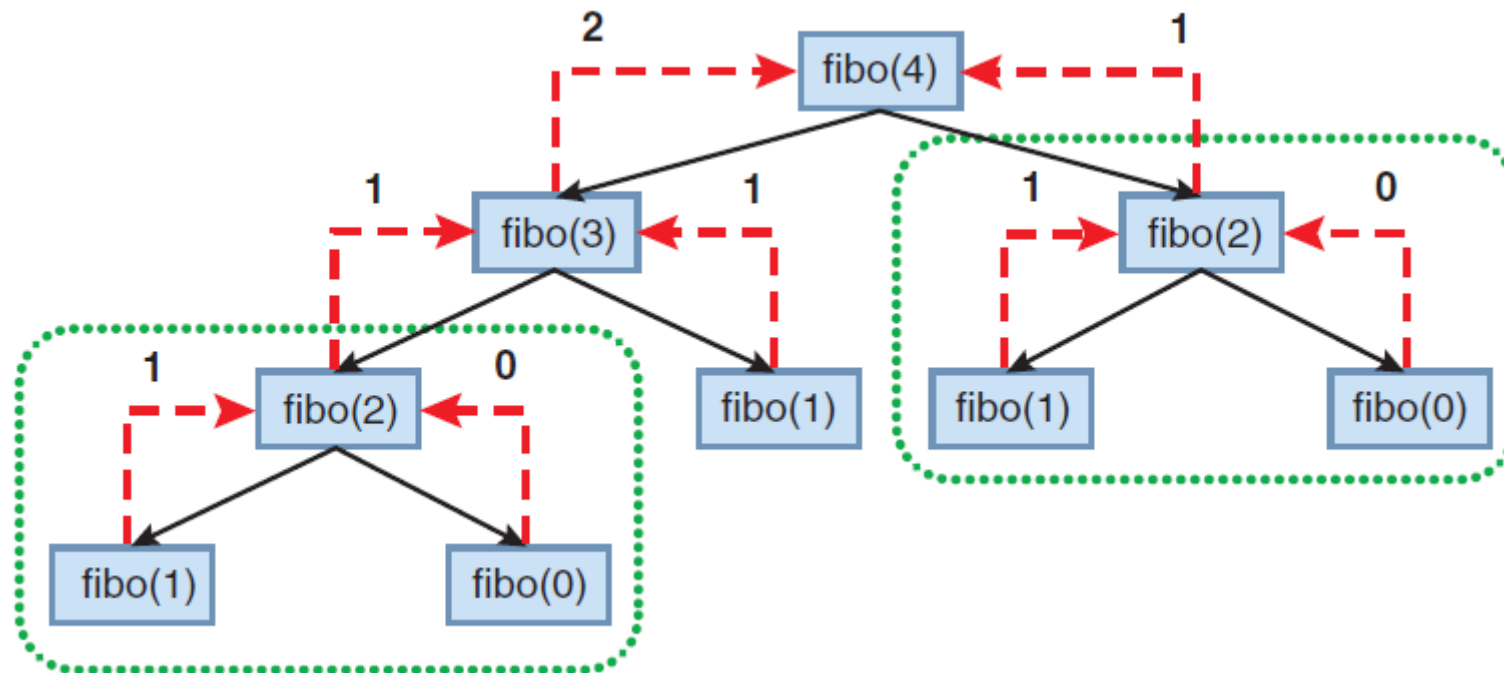
```
int fibo(int n){  
    int i, t, c, a = 0, b = 1;  
    for(i = 0; i < n; i++){  
        c = a + b;  
        a = b;  
        b = c;  
    }  
    return a;  
}
```

Com Recursão

```
int fiboR(int n){  
    if (n == 0 || n == 1)  
        return n;  
    else  
        return fiboR(n-1) + fiboR(n-2);  
}
```

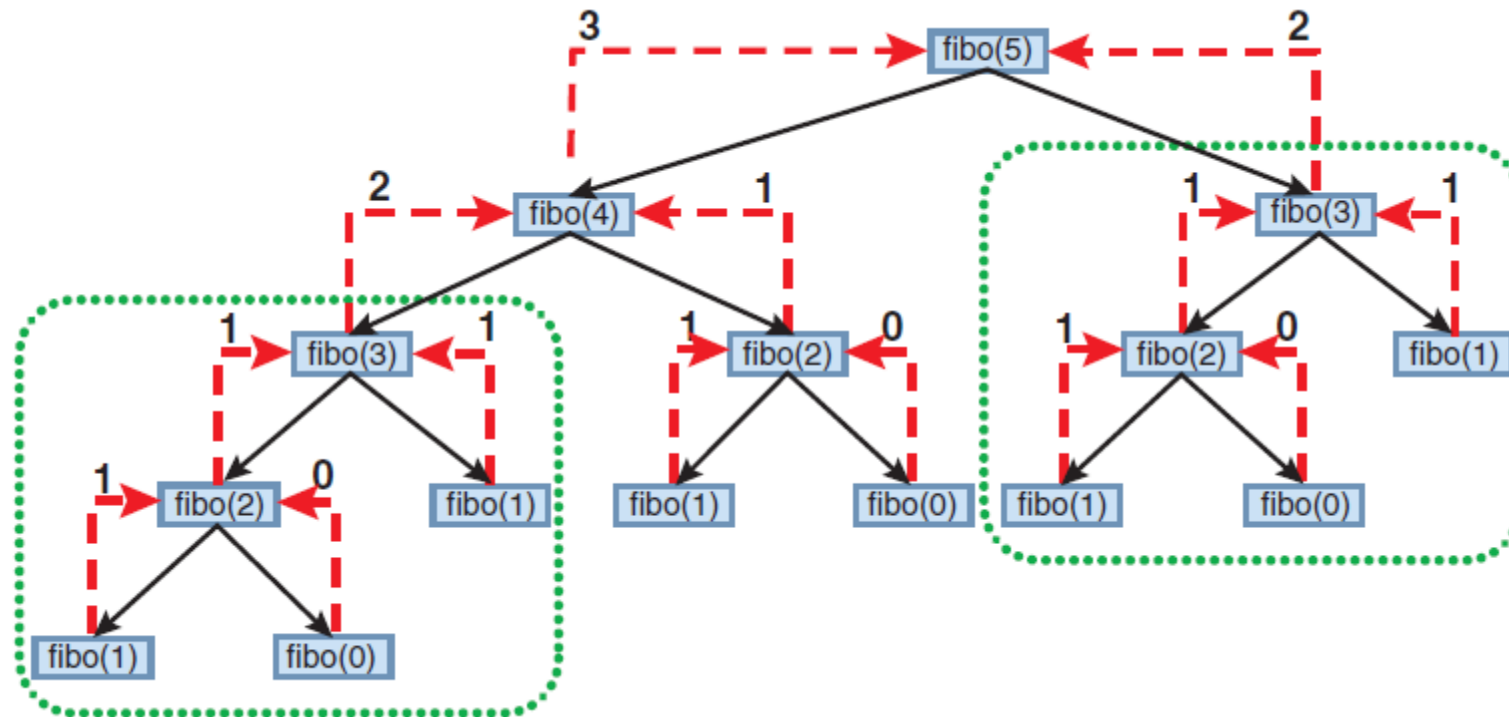
Fibonacci

- ... mas como se verifica na imagem, elegância não significa eficiência



Fibonacci

- Aumentando para **fib(5)**



Exercícios

- Crie uma função para verificar e retornar qual o maior valor entre 3 valores passados como parâmetro. A passagem de parâmetro nesse exercício será feita por valor;
- Crie uma função que receba 5 valores como parâmetro, e que “retorne” qual o menor e qual o maior valor dos 5. Nesta função ao menos dois parâmetros deverão ser passados por referência;
- Crie uma função que receba um array de tamanho 5. Os elementos deste array deverão ser lidos pelo teclado na função main. A função criada deverá trocar de posição o menor e o maior valor deste array. Após a saída da função, já na função main imprima os elementos do array;
- Crie uma função recursiva, para calcular e retornar o fatorial de um valor passado por parâmetro por valor para a função;

Algoritmos e Estruturas de Dados I

- Bibliografia:

- Básica:

- CORMEN, Thomas; RIVEST, Ronald, STEIN, Clifford, LEISERSON, Charles. Algoritmos. Rio de Janeiro: Elsevier, 2002.
 - DEITEL, Paul; DEITEL, Harvey. C++ como programar. 5. ed. São Paulo: Pearson, 2006.
 - MELO, Ana Cristina Vieira de; SILVA, Flávio Soares Corrêa da. Princípios de linguagens de programação. São Paulo: Edgard Blücher, 2003.

- Complementar:

- ASCENCIO, A. F. G. & CAMPOS, E. A. V. Fundamentos da programação de computadores. 2. ed. São Paulo: Pearson Education, 2007.
 - MEDINA, Marcelo, FERTIG, Cristina. Algoritmos e programação: teoria e prática. Novatec. 2005.
 - MIZRAHI, V. V.. Treinamento em linguagem C: módulo 1. São Paulo: Makron Books, 2008.
 - PUGA, S. & RISSETTI, G. Lógica de programação e estruturas de dados com aplicações em java. São Paulo: Prentice Hall, 2004.
 - ZIVIANI, Nívio. Projeto de Algoritmos com Implementação em Pascal e C. Cengage Learning. 2010.

Algoritmos e Estruturas de Dados I

