

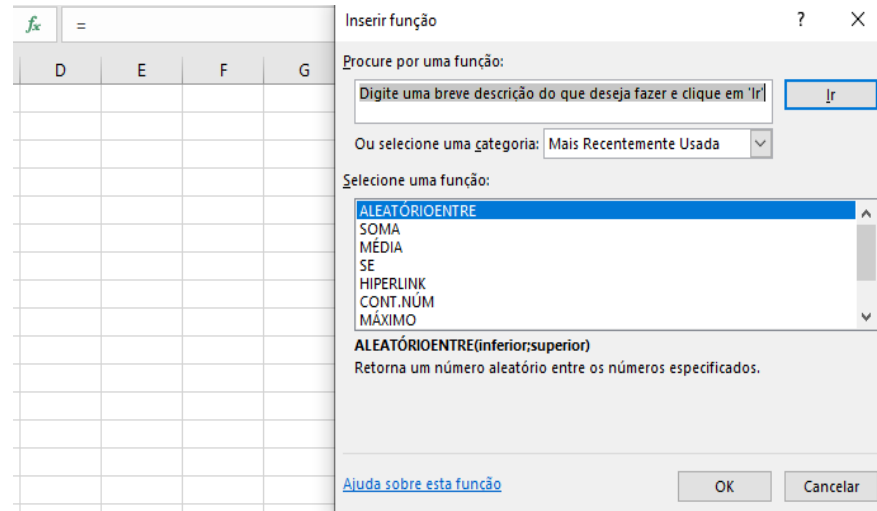
TAD0102

Algoritmos e programação

Prof. Dr. Josenalde Barbosa de Oliveira

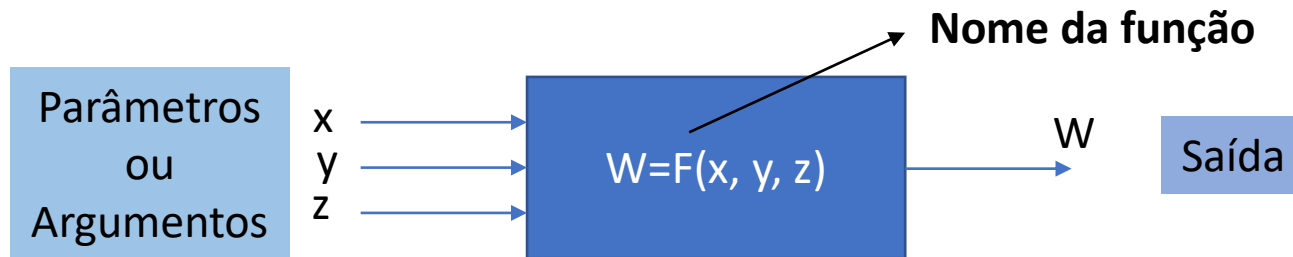
josenalde@eaj.ufrn.br

Aulas: 24T345, 6 CRDS – 90h



Modularização de Código - funções

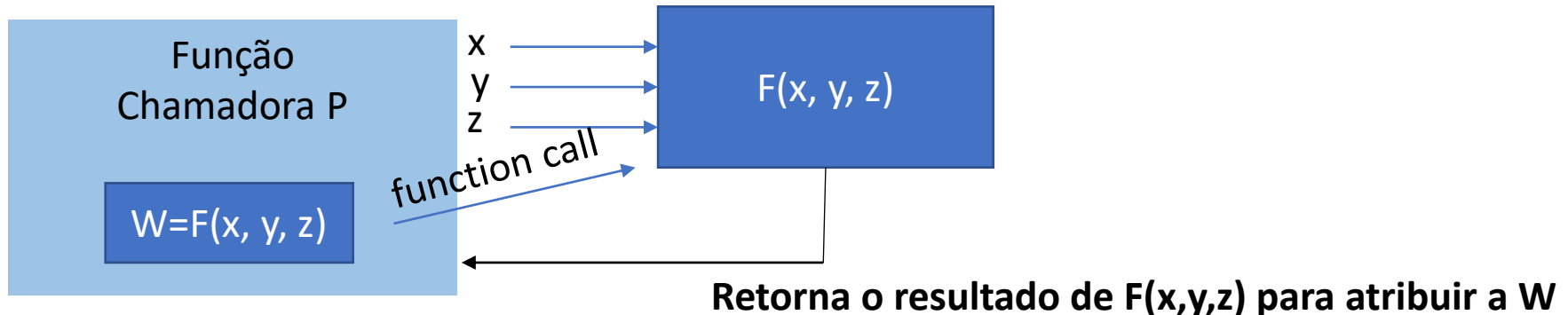
- Ao programar, utilizamos funções de sistema ou de terceiros, naturalmente
- Reutilização de código (não reinventar a roda)
- Organização de código (mais limpo, inteligível)



- Em C/C++ especificamos o tipo de cada parâmetro e o tipo de retorno da função, ou seja, o tipo de W no exemplo acima (int, float, double, char, unsigned, long, bool, string, **void**,...)
- A própria função **main()** é por padrão do tipo int, retorna o número 0 para indicar ao shell-SO que a execução foi bem sucedida e pode ou não receber parâmetros (argc, argv)

Modularização de Código - funções

- Uma função é definida (protótipo, assinatura), implementada (corpo), e é CHAMADA (function call) por outra função (ou código principal)
- Seu valor de retorno pode ser atribuído a uma variável ou utilizado onde se aguarda um valor explícito
- Contudo, uma função pode ser do tipo void (também chamado procedimento), que executa algo (faz algo), mas não retorna valores à função que chamou-a para serem atribuídos



Modularização de Código - funções

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << "Func cout\n";
    std::cout << sqrt(4) << std::endl;
}
```

```
#include <iostream>
#include <cmath>

int somaNum(int n1, int n2) { //definição
    return n1 + n2; // corpo
}

int main() {
    std::cout << "Func cout\n"; // cout em iostream
    std::cout << sqrt(4) << std::endl; // sqrt em cmath
    std::cout << somaNum(5,6) << std::endl;
    // ou
    int x = somaNum(10,20);
    std::cout << x << std::endl;
}
```

Modularização de Código - funções

```
#include <iostream>
#include <string>
using namespace std;

void imprimeNome(std::string s, unsigned char n) {
    unsigned char i = 0;
    while (i < n) {
        std::cout << s << std::endl;
        i++;
    }
}

int main() {
    string a;
    getline(cin, a); // função interna para ler string
    imprimeNome(a, 5); // passa 02 parâmetros
    // função executa algo (ação), mas não retorna valor ao main()
}
```

Modularização de Código - funções

```
#include <iostream>
#include <vector>

using namespace std;

void imprimeVetor(int v[], unsigned short t) { // ou *v
    for (int i = 0; i < t; i++) {
        cout << v[i] << endl;
    }
}

void imprimeVetorSTL(vector<int> v) { // não recebe o tamanho
    for (int i = 0; i < v.size(); i++) { // pois é facilmente calculado aqui
        cout << v.at(i) << endl;
    }
}

int main() {
    int v[5] = {4, 5, 10, 4, 2};
    imprimeVetor(v, 5);
    cout << endl;
    vector<int> z{4,5,10,4,2};
    imprimeVetorSTL(z);
}
```

Modularização de Código - funções

```
#include <iostream>

using namespace std;

void imprimeMatriz(int A[][3], unsigned short m) { // ou *v
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < 3; j++) {
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int M[2][3] = {{4, 5, 1}, {10, 4, 4}};
    imprimeMatriz(M, 2);
}
```

As implementações das funções não precisam estar acima da função principal. Podem estar abaixo ou mesmo noutros arquivos (.h e .cpp equivalentes). Surge o conceito de assinatura ou protótipo da função.

Modularização de Código - funções

```
#include <iostream>
#include <vector>

using namespace std;

void imprimeVetor(int v[], unsigned short t); // ou
//void imprimeVetor(int [], unsigned short); // ou int *
// na assinatura, não é obrigatório o nome dos parâmetros

int main() {
    int v[5] = {4, 5, 10, 4, 2};
    imprimeVetor(v, 5);
}

// implementação abaixo da função principal
void imprimeVetor(int v[], unsigned short t) { // ou *v
    for (int i = 0; i < t; i++) {
        cout << v[i] << endl;
    }
}
```


Modularização de Código – arquivos de cabeçalho

main.cpp

```
#include <impressao.h>
#include <iostream>
using namespace std;
int main() {
    int v[5] = {4, 5, 10, 4, 2};
    imprimeVetor(v, 5);
    cout << NUM << endl;
}
```

impressao.h

```
#define NUM 100
#define NMAX 10000

void imprimeVetor(int v[], unsigned short t);
```

impressao.cpp

```
#include <iostream>
using namespace std;
#include "impressao.h"
void imprimeVetor(int v[], unsigned short t) { // ou *v
    for (int i = 0; i < t; i++) {
        cout << v[i] << endl;
    }
}
```

Modularização de Código – passagem de parâmetros

```
#include <iostream>
using namespace std;

void trocaValor(int n1, int n2) {
    int aux; // n1 recebe cópia de x e n2 de y
    aux = n2; // variáveis aux, n1 e n2 são locais
    n2 = n1; // o que acontece aqui, fica aqui
    n1 = aux;
    cout << "Dentro da função trocaValor: " << endl;
    cout << "n1 = " << n1 << ", n2 = " << n2 << endl;
}

void trocaRef(int *n1, int *n2) { // recebe endereços
    int aux;
    aux = *n2; // manipula diretamente a variável original
    *n2 = *n1;
    *n1 = aux;
    cout << "Dentro da função trocaRef: " << endl;
    cout << "*n1 = " << *n1 << ", *n2 = " << *n2 << endl;
}

int main() {
    int x = 5, y = 10;
    cout << "Antes troca por valor: " << endl;
    cout << "x = " << x << ", y = " << y << endl;
    trocaValor(x, y);
    cout << "Depois troca por valor: " << endl;
    cout << "x = " << x << ", y = " << y << endl;
    trocaRef(&x, &y); // passa os endereços de x e y
    cout << "Depois troca por Referência: " << endl;
    cout << "x = " << x << ", y = " << y << endl;
}
```

Com exceção dos arrays (uni e multidimensionais), os parâmetros passados até agora foram POR VALOR, ou seja, uma cópia dos valores das variáveis é passada para as funções e não as próprias variáveis das funções chamadoras. Pode-se também manipular diretamente as variáveis originais passadas, com reflexo fora da função, ou seja, na função chamadora.

Arrays são sempre passados por REFERÊNCIA, pois o nome do vetor, matriz, etc. é o endereço do primeiro elemento (BASE)

Programação dinâmica

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

```
void imprimeVetor(int a[], int s) {  
    for (int i = 0; i < s; i++) {  
        cout << a[i] << " ";  
    }  
}
```

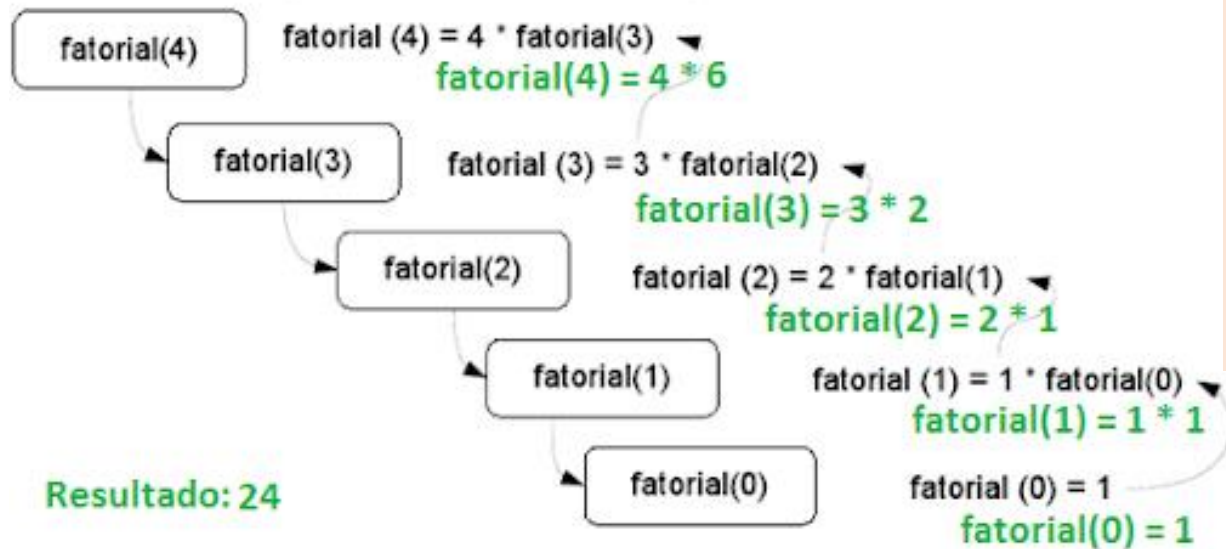
```
void fibo(int t) { // imprime até o t. termo da série de Fibonacci  
    int v[t];  
    v[0] = 0;  
    v[1] = 1;  
    if (t > 2) { // assume t1=0, t2=1 FIXOS  
        for (int k = 2; k < t; k++) {  
            v[k] = v[k-1] + v[k-2]; // atual depende dos anteriores  
        }  
    }  
    imprimeVetor(v, t);  
}
```

```
int main() {  
    int x = 5, y = 10;  
    cout << fatorial(x) << endl;  
    fibo(x);  
}
```

Recursividade

- Chamadas sucessivas a mesma função (auto chamamento)
- Importante definir critério de parada, senão loop infinito
- Emula um laço de repetição onde a variável de controle avança ou recua até não atender determinado critério
- Elegante, porém eficiência deve ser avaliada

*Lembrem-se que a leitura é de baixo para cima!



Exemplo: FATORIAL

$$5! = 5 * 4!$$

$$= 5 * 4 * 3!$$

$$= 5 * 4 * 3 * 2!$$

$$= 5 * 4 * 3 * 2 * 1!$$

Recursividade

```
#include <iostream>
using namespace std;
```

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

```
int fatorialR(int n) {
    if (n == 1) return 1;
    return (n*fatorialR(n-1));
}
```

```
int main() {
    int x = 5, y = 10;
    cout << fatorial(x) << endl;
    cout << fatorialR(x) << endl;
}
```

fatorial(5)

5*fatorialR(4)

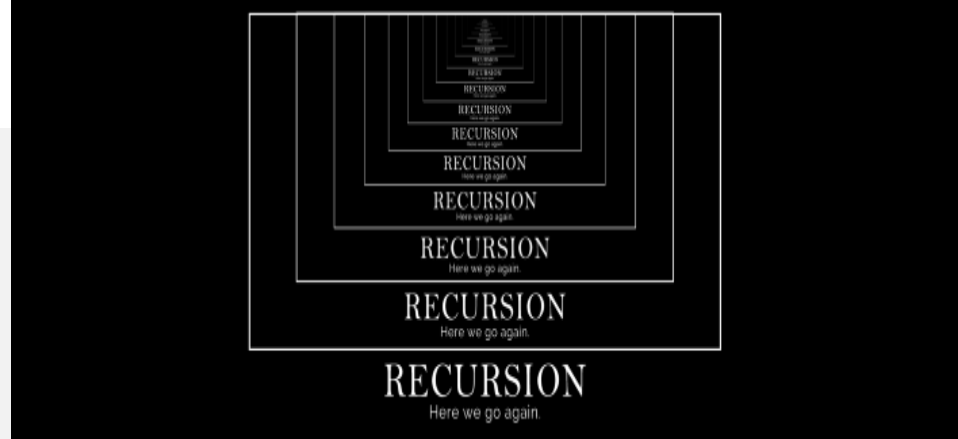
4*fatorialR(3)

3*fatorialR(2)

2*fatorialR(1)

1

Critério de PARADA

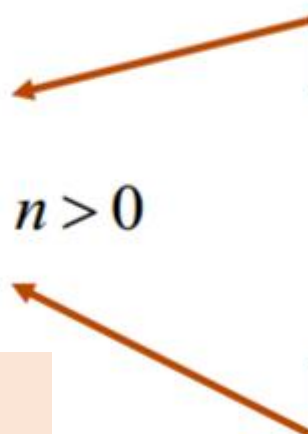


Recursividade

Exercício: forneça a definição recursiva para a operação de potenciação

$$x^n = \begin{cases} 1, & \text{se } n = 0 \\ x \times x^{(n-1)}, & \text{se } n > 0 \end{cases}$$

Caso BASE



```
int pot(int n, int i) {  
    if (i == 0) return 1;  
    return n*pot(n, i-1);  
}  
  
int main() {  
    cout << pot(2, 10) << endl;  
}
```

Passo
Recursivo