

Funções em C++

Alexandro Adário



INSTITUTO FEDERAL
Rio Grande do Sul
Campus Erechim

Subrotina

Bloco de código reutilizável que executa uma tarefa específica, podendo ou não retornar um valor.

Subrotina

Bloco de código reutiliza uma tarefa específica para retornar um valor.

Todo e qualquer código pode ser dividido em blocos, seções e estruturas menores, que facilitam organização e podem ser reutilizados.

Subrotina

Bloco de código **reutilizável** que executa

uma tarefa específica, podendo ou não retornar um valor.

Reuso é uma estratégia que possibilita utilizar novamente soluções de código, evitando duplicidade, redundância ou retrabalho.

Subrotina

Bloco de código reutilizável para executar uma **tarefa específica**, responsável por retornar um valor.

A subrotina deve ser de simples compreensão, evitar duplicidade ou acúmulo de responsabilidades que podem introduzir erros ou dificultar a manutenção

Subrotina

Bloco de código re
uma tarefa específica
retornar um valor.

Normalmente, após a
execução da tarefa,
devolvem ao seu
“chamador” um
resultado da execução,
que pode ser apenas
um “bem sucedido” ou
uma estrutura mais
complexa

Sinônimos: Subrotina

Conforme a linguagem, o tipo e o paradigma, as subrotinas podem ser chamadas de **funções**, **procedimentos** ou **métodos**.

Procedimentos

São as subrotinas que **não devolvem valor** ao “chamador”, apenas realizam uma tarefa específica e retornam.

Funções

É o nome mais genérico dado às subrotinas, mas normalmente são aquelas que **devolvem algum valor** ao “chamador”, após realizarem sua tarefa.

Métodos

Nas **linguagens orientadas a objetos**, as funções que pertencem a uma classe são chamadas de métodos. Podem ou não devolver valor, conforme o caso.

Por Que Modularizar?

Eliminar código repetitivo.

Simplificar a compreensão e visualização do código.

Estruturar e organizar programas.

Facilitar a manutenção do programa.

Estrutura de uma Função

nome da função

tipo de retorno

parâmetros

Cabeçalho

Corpo

```
int nomeFuncao (void) {  
    // comandos  
    return 0;  
}
```

comando de retorno

The diagram illustrates the structure of a function in C. It shows a code snippet with labels and arrows pointing to specific parts: 'nome da função' points to 'nomeFuncao', 'tipo de retorno' points to 'int', 'parâmetros' points to '(void)', 'Cabeçalho' points to the opening curly brace '{', 'Corpo' points to the closing curly brace '}', and 'comando de retorno' points to 'return 0;'. The code snippet is: `int nomeFuncao (void) {
 // comandos
 return 0;
}`

Retorno

Para um método devolver um resultado/valor ao “chamador”, precisa ter um tipo de retorno.

Caso não haja retorno, a função deve ser declarada como void.

Parâmetros

São informações/valores que passamos para a função realizar uma tarefa.

Se a função é uma “receita”, os parâmetros são os ingredientes necessários para o “prato” final.

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

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

int main() {
    int resultado = soma(5, 7);
    cout << "Resultado: " << resultado << endl;
    return 0;
}
```

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


Como a Função Executa?

Como são passados os parâmetros?

Como devolve pro “chamador”?

E se outra função é chamada?

Memória Principal

Área de Código do Programa

Programa/Função Principal

Variáveis da
Função Principal

código

Função 1

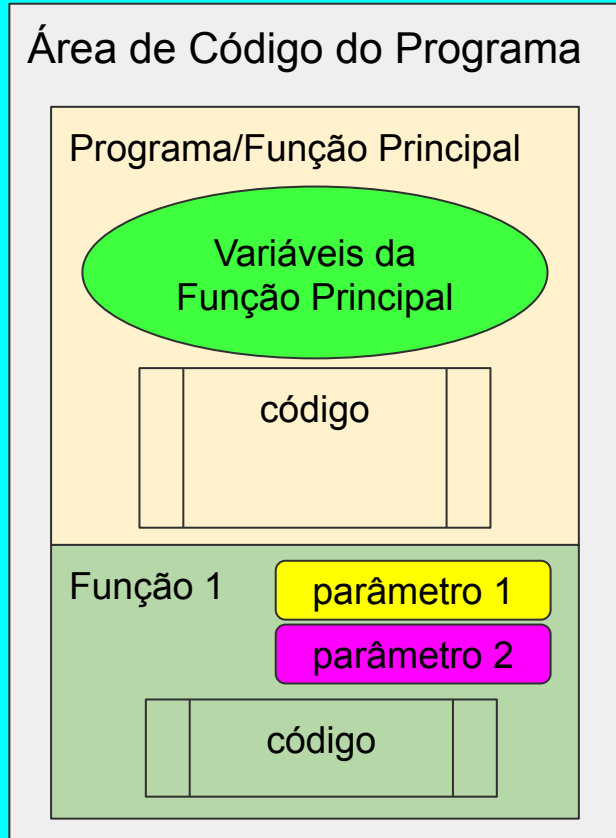
parâmetro 1

parâmetro 2

código

Pilha de Execução do Programa

Memória Principal

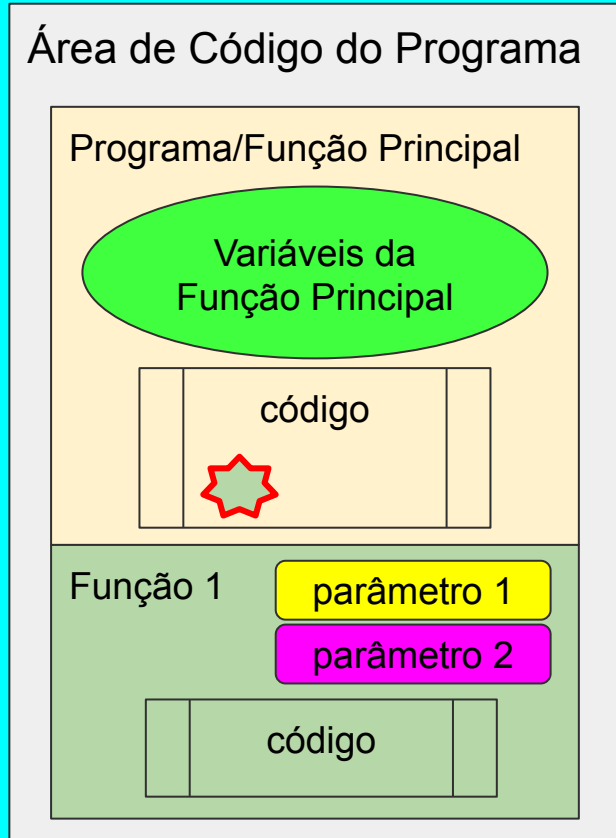


Pilha de Execução do Programa

Criação do espaço na pilha de execução

Espaço de Execução da Função Principal

Memória Principal

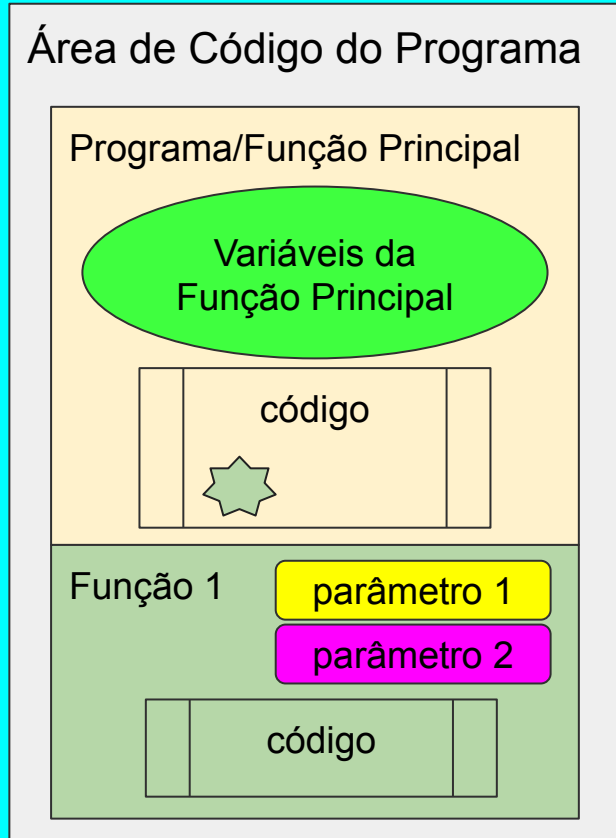


Pilha de Execução do Programa

Ocorre a chamada de execução da função

Espaço de Execução da Função Principal

Memória Principal



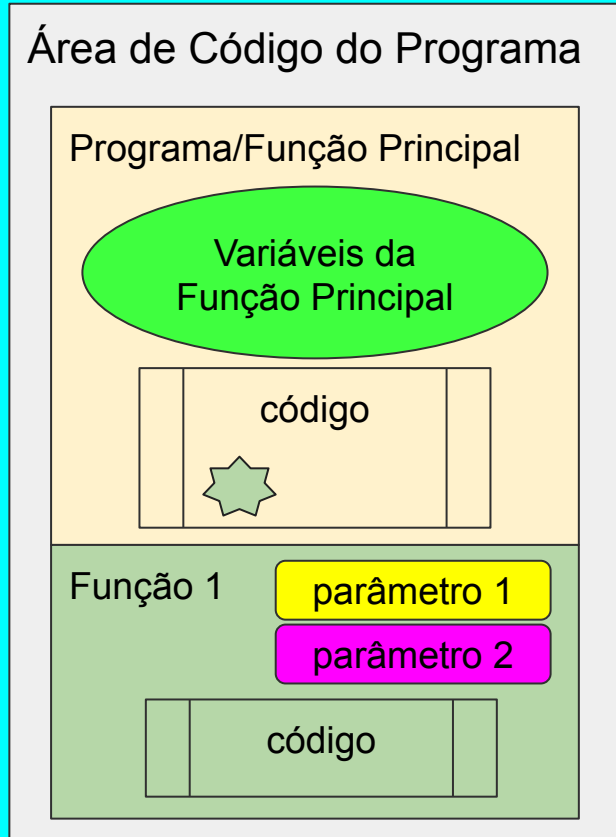
Pilha de Execução do Programa

Criação do espaço da função na pilha de execução

Espaço de Execução da Função 1

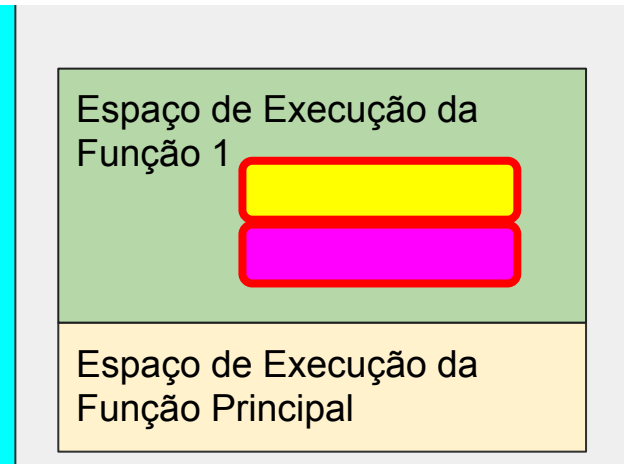
Espaço de Execução da Função Principal

Memória Principal

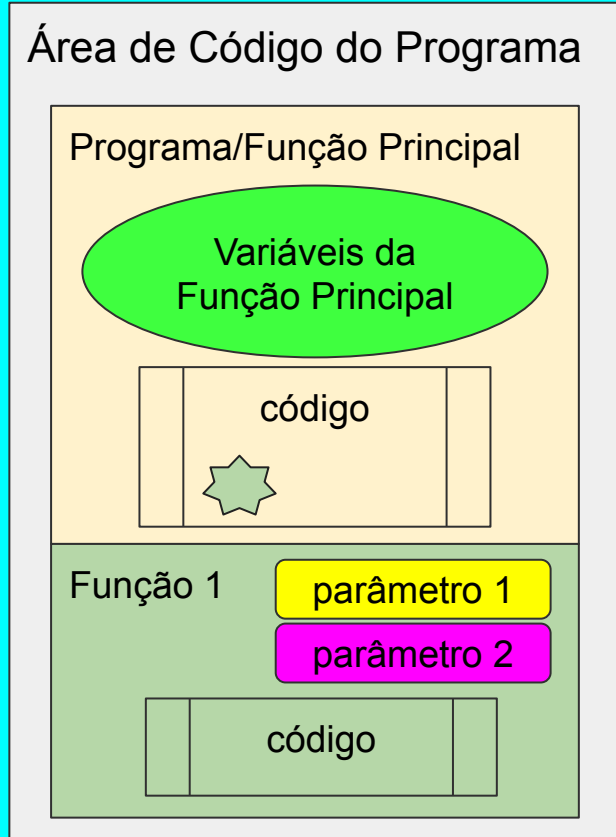


Pilha de Execução do Programa

A área de dados referente aos parâmetros já está definida, ainda sem valor

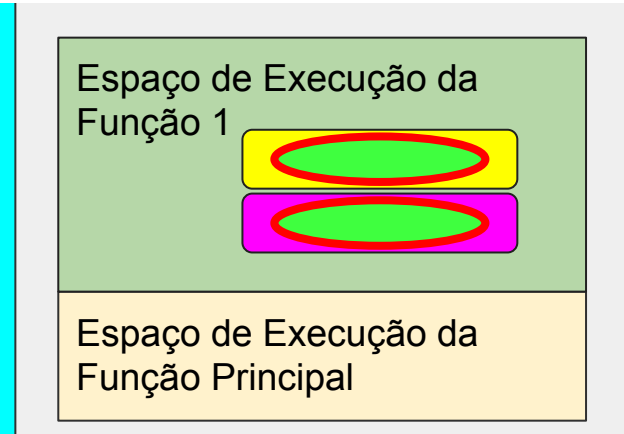


Memória Principal



Pilha de Execução do Programa

Os valores a serem usados como parâmetros são copiados da área principal para o espaço de execução



Passagem por Valor

O valor da variável original é COPIADO para o parâmetro.

Desta forma, a variável nunca é afetada pela execução da função. Toda a tarefa é realizada sobre o parâmetro.


```
#include <iostream>
using namespace std;
void incrementa(int num) {
    num++;
}

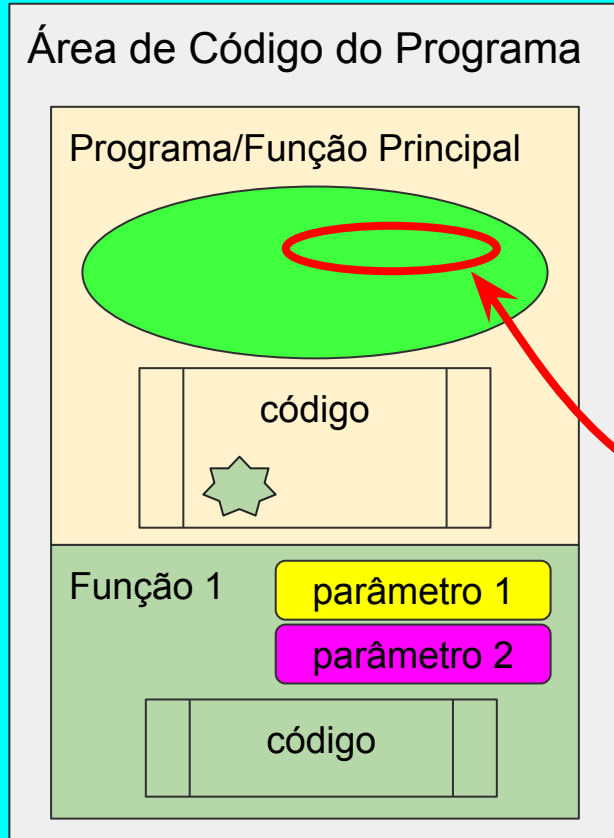
int main() {
    int valor = 10;
    incrementa(valor);
    cout << "Novo Valor: " << valor << endl;
    return 0;
}
```

Passagem por Referência

O ENDEREÇO da variável original é associado ao parâmetro.

Desta forma, toda a tarefa realizada sobre o parâmetro afeta o conteúdo da variável.

Memória Principal




Pilha de Execução do Programa

O endereço da variável fica associado ao parâmetro que é apenas um endereço no espaço de execução.

Espaço de Execução da Função 1

Espaço de Execução da Função Principal

```
#include <iostream>
using namespace std;
void incrementa(int& num) {
    num++;
}
```



```
int main() {
    int valor = 10;
    incrementa(valor);
    cout << "Novo Valor: " << valor << endl;
    return 0;
}
```

Operador &

Quando utilizado, indica que será usado o endereço da variável e não o seu valor.

O endereço é ONDE ela está localizada na memória do computador durante a execução.

```
void troca(int& a, int& b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

Exemplo bastante útil!

Como a e b estão informados por referência, as variáveis que forem passadas como parâmetro terão seus valores trocados de lugar, independente de onde estejam no código.

Recursividade

Quando uma função é definida de maneira que pode chamar a si mesma.

Ou seja, para resolver o problema, ela chama a si mesma, resolvendo um caso ligeiramente menor.

Exemplo: Fatorial (1)

A função fatorial é definida como o produto de todos os antecessores de um número natural, incluindo ele.

Definição prévia: $0! = 1$

Exemplo: Fatorial ⁽²⁾

$0! = 1$ caso base ou trivial

$$1! = 1$$

$$2! = 2 \times 1$$

$$3! = 3 \times 2 \times 1$$

$$4! = 4 \times 3 \times 2 \times 1$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Exemplo: Fatorial ⁽³⁾

$$0! = 1$$

$$1! = 1 \times 0! = 1 \times 1$$

$$2! = 2 \times 1! = 2 \times (1 \times 0!)$$

$$3! = 3 \times 2! = 3 \times (2 \times 1!) = 3 \times (2 \times (1 \times 0!))$$

$$4! = 4 \times 3! = 4 \times (3 \times 2!) = 4 \times (3 \times (2 \times 1!))...$$

$$\begin{aligned} 5! &= 5 \times 4! = 5 \times (4 \times 3!) = 5 \times (4 \times (3 \times 2!)) \\ &= 5 \times (4 \times (3 \times (2 \times 1!)))... \end{aligned}$$

Exemplo: Fatorial ⁽⁴⁾

Generalização do Fatorial:

Se $N = 0$, resultado é 1 caso base ou trivial

Se $N > 0$, resultado é $N \times (N-1)!$ caso geral

Casos Base/Trivial e Geral

Caso base (ou trivial): Condição que encerra a recursão, evitando que seja infinita. Não faz a chamada recursiva.

Caso geral: Parte que resolve o problema chamando a função com um tamanho reduzido do problema, até o caso base.

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

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



chamada recursiva

```
int fatorial(int n) {  
    if( n == 0 ) {  
        return 1;    caso base ou trivial  
    } else {  
        return n * fatorial(n - 1) ; caso geral  
    }  
}
```

Vantagens da Recursividade

Simplicidade

Redução de código

Solução elegante

Facilita a solução de problemas complexos

Memória Principal

Área de Código do Programa

Programa/Função Principal

Variáveis da
Função Principal

código



Função 1

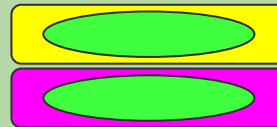
parâmetro 1

parâmetro 2

código

Pilha de Execução do Programa

Espaço de Execução da
Função 1



Espaço de Execução da
Função Principal