

# Curso de Extensão - INF1900

# Programação em C++



## Módulo 1

# Introdução à Programação em C++

## Aula 3

Profa. Dra. Esther Luna Colombini

[esther@ic.unicamp.br](mailto:esther@ic.unicamp.br)

Agosto de 2023

# Módulo 1: Introdução à Programação em C++ (10h)



Profa. Dra. Esther Luna Colombini



Familiarizar os alunos com os conceitos básicos da linguagem C++ e prepará-los para escrever programas simples, aprofundando o conhecimento em estruturas de dados e gerenciamento da memória em programas C++.

- Introdução à linguagem C++ e sua história
- Ambiente de desenvolvimento, configuração do compilador, pré-processador e link estático e dinâmico
- Estrutura básica de um programa em C++
- Tipos de dados, variáveis e constantes
- Conversões de tipos e value categories
- Operadores aritméticos, lógicos e relacionais e operadores bit a bit
- Controle de fluxo: estruturas condicionais e laços de repetição
- Funções e procedimentos em C++
- Funções lambda
- Manipulação de entradas e saídas
- Arrays e matrizes
- Strings e manipulação de cadeias de caracteres
- Ponteiros e referências
- Alocação dinâmica de memória
- Gerenciamento de memória e desalocação
- Estruturas de dados avançadas: listas, pilhas e filas

# Monitorias

- **Monitores do Módulo:**
  - Alana Correia
  - Iury Cleveston
- **Dia/Horário do atendimento dos monitores:**
  - A definir em conjunto
- **Dia/Horário do atendimento do professor:**
  - Quintas às 18:00h

# Calendário

**Aulas:** segunda-feira e quarta-feira

**Horário:** 8:00h às 10:00h



14/08/23	MÓDULO 1
16/08/23	MÓDULO 1
18/08/23	MÓDULO 1 - ATENDIMENTO
21/08/23	MÓDULO 1
23/08/23	MÓDULO 1
24/08/23	MÓDULO 1 - ATENDIMENTO

# Avaliação

- **Avaliação:**
  - Atividades práticas a serem realizadas em dupla liberadas todas às quartas
- **Média final:** média aritmética das notas

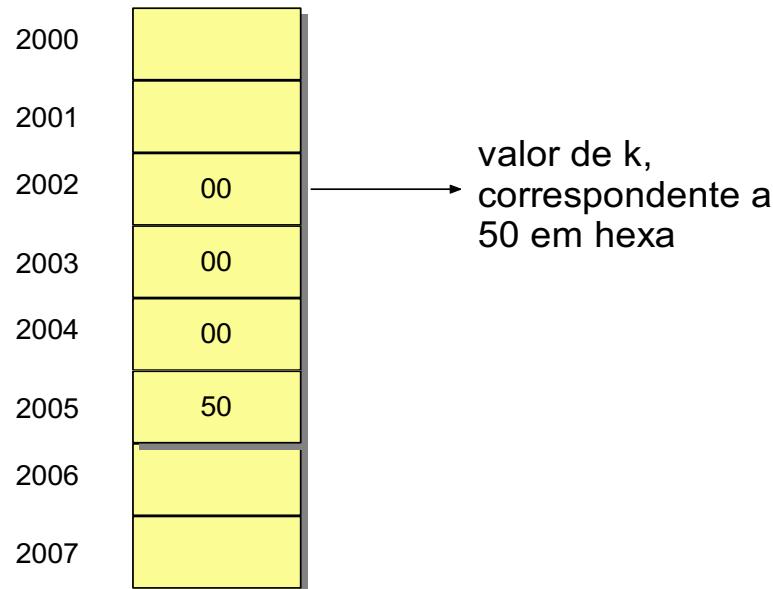
# Bibliografia

- Stroustrup, B. (2013). The C++ Programming Language (4th Edition). Addison-Wesley Professional.
- Josuttis, N. (2019). C++17 - The Complete Guide: First Edition. Leanpub.
- Schildt, H. (2017). C++: The Complete Reference (5th Edition). McGraw-Hill Education.

# Ponteiros

# Ponteiros

- No computador, tanto o SO quanto os programas compilados fazem um certo uso da memória RAM.
- Os dados de seu programa também são armazenados automaticamente na memória RAM.
- Quando declaramos uma variável, por exemplo “k”, inicializada, por exemplo com o valor inteiro 80, a mesma é armazenada na memória.



# Ponteiros

- O endereço em que uma variável é armazenada é normalmente escolhido pelo compilador
    - Este endereço é dado por um número;
  - Pode-se criar uma **variável especial para armazenar um endereço**;
    - Esta variável é chamada de **ponteiro**.
- Um **ponteiro** é uma **variável** que armazena o **endereço** de memória de **outra variável**.
- **Declaração** de um ponteiro:

```
int    *x    // x é um ponteiro para um int  
char   *p    // p é um ponteiro para char
```

# Ponteiros

- Por que informar o tipo de dado para o qual o ponteiro aponta?
  - Para saber quantos bytes deve-se buscar a partir do endereço base e como tratar aquela região de memória
- Operação:
  - &: o endereço de...
  - \*: no endereço...

```
#include <iostream>

int main() {
    int number = 42;
    int* ptr;           // Declaração de um ponteiro para inteiro

    ptr = &number;    // Atribuição do endereço da variável 'number' ao ponteiro

    std::cout << "Value of number: " << number << std::endl;
    std::cout << "Value pointed by ptr: " << *ptr << std::endl; // Desreferência do ponteiro

    return 0;
}
```

# Ponteiros

- Parâmetros de funções como ponteiros
  - A operação realizada na função afeta a variável original

```
#include <iostream>

void t(int *u)
{
    *u = *u + 1;
}
int main(void)
{
    int x;
    x = 10;
    t(&x);
    std::cout << "Value of x: " << x << std::endl;
    return 0;
}
```

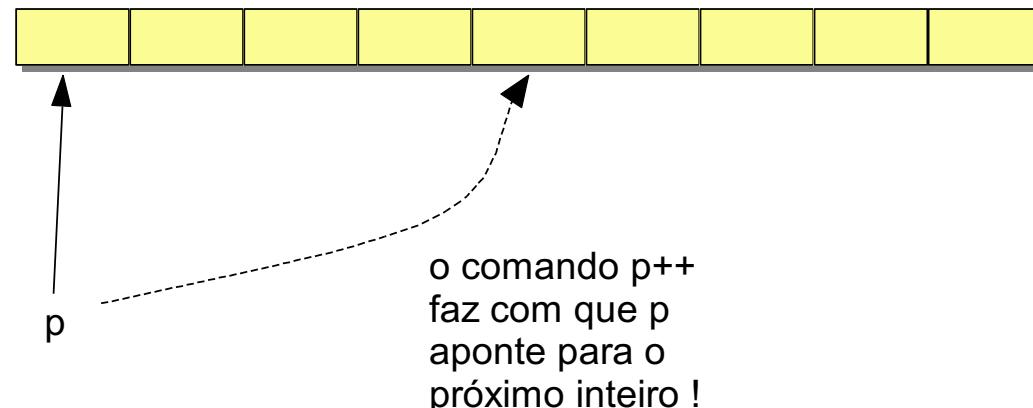
# Aritmética de Ponteiros

- A aritmética de ponteiros em C++ envolve o uso de operações aritméticas em ponteiros para mover sua posição de memória para acessar diferentes elementos em arrays ou estruturas.
- **Operadores:**
  - `++` incrementa o ponteiro para apontar para o próximo elemento, enquanto o operador
  - `--` decrementa o ponteiro para apontar para o elemento anterior
  - Também é possível usar os operadores `+=` e `-=` para mover o ponteiro para a frente ou para trás por um número específico de elementos.

# Aritmética de Ponteiros

- Exemplo:
  - Considerando que  $p$  é um ponteiro para um inteiro

```
int *p;
```
  - A operação  $p++$  faz com que  $p$  aponte para o próximo inteiro



- O que acontece se eu declararmos um ponteiro para `int` e atribuirmos a ele um endereço de memória de um vetor de `char`?

# Aritmética de Ponteiros

```
#include <iostream>

int main() {
    int numbers[] = {10, 20, 30, 40, 50};

    int* ptr = numbers; // O ponteiro ptr aponta para o primeiro elemento do array 'numbers'
    std::cout << "First element: " << *ptr << std::endl; // Saída: First element: 10

    // Aritmética de ponteiro: movendo para o próximo elemento
    ptr++; // Agora o ponteiro aponta para o próximo elemento do array
    std::cout << "Second element: " << *ptr << std::endl; // Saída: Second element: 20

    // Aritmética de ponteiro: movendo para o elemento anterior
    ptr--; // Agora o ponteiro volta a apontar para o primeiro elemento do array
    std::cout << "First element again: " << *ptr << std::endl; // Saída: First element again: 10

    // Aritmética de ponteiro: movendo para frente por duas posições
    ptr += 2; // Agora o ponteiro aponta para o terceiro elemento do array
    std::cout << "Third element: " << *ptr << std::endl; // Saída: Third element: 30

    // Aritmética de ponteiro: movendo para trás por uma posição
    ptr--; // Agora o ponteiro volta a apontar para o segundo elemento do array
    std::cout << "Second element again: " << *ptr << std::endl; // Saída: Second element again: 20

    return 0;
}
```

First element: 10  
Second element: 20  
First element again: 10  
Third element: 30  
Second element again: 20

# Alocação de memória

- A alocação de memória em C++ permite reservar e liberar blocos de memória durante a execução de um programa.
- Existem duas principais formas de alocação de memória em C++: **alocação estática e alocação dinâmica**.
  - **Alocação Estática:** A alocação estática ocorre quando você define variáveis comuns com um tamanho fixo em tempo de compilação.

```
int staticVariable; // Alocação estática
```

# Alocação de memória

- **Alocação Dinâmica:** permite alocar memória em tempo de execução, permitindo a criação de objetos cujo tamanho não é conhecido antecipadamente. A alocação dinâmica é feita usando operadores new e delete (ou new[] e delete[] para arrays) em C++.
  - new: Aloca memória para um único objeto.
  - new[]: Aloca memória para um array de objetos.
  - delete: Libera memória alocada usando new.
  - delete[]: Libera memória alocada usando new[].

```
#include <iostream>

int main() {
    int* dynamicVariable = new int; // Alocação dinâmica
    *dynamicVariable = 42;
    std::cout << "Value: " << *dynamicVariable << std::endl;

    delete dynamicVariable; // Liberação da memória alocada

    return 0;
}
```

# Alocação de memória

```
struct Pessoa {  
    std::string nome;  
    int idade;  
};
```

```
int main() {  
    Pessoa pessoal; // Alocação estática de uma struct  
    pessoal.nome = "João";  
    pessoal.idade = 30;  
  
    std::cout << "Nome: " << pessoal.nome << ", Idade: " << pessoal.idade << std::endl;  
  
    return 0;  
}
```

**Alocação estática**

```
int main() {  
    Pessoa* pessoa2 = new Pessoa; // Alocação dinâmica de uma struct  
    pessoa2->nome = "Maria";  
    pessoa2->idade = 25;  
  
    std::cout << "Nome: " << pessoa2->nome << ", Idade: " << pessoa2->idade << std::endl;  
  
    delete pessoa2; // Liberação de memória  
  
    return 0;  
}
```

**Alocação dinâmica**

# Alocação de memória

- Alocação dinâmica de memória exige que você libere manualmente a memória alocada quando não for mais necessária.



## Memory Leak

- Para evitar os desafios da alocação e liberação manual de memória, é recomendado o uso de **smart pointers**
  - std::unique\_ptr e std::shared\_ptr
    - automatizam a liberação de memória, reduzindo a probabilidade de vazamentos de memória.

# Referências

- Referências
  - forma de criar "apelidos" ou "sinônimos" para variáveis existentes
  - permitem que você se refira a uma variável usando um nome diferente, mas sem criar uma cópia da variável original.
  - as referências são frequentemente usadas para passar argumentos para funções por referência e para evitar cópias desnecessárias de dados.

- **Declaração:**

```
int originalVariable = 42;  
int& reference = originalVariable; // Declaração de uma referência
```

- **Uso:**
  - uma referência deve ser inicializada no momento da declaração e não pode ser reatribuída para se referir a outra variável.
  - uma vez que uma referência é criada, ela age como um alias para a variável original.
    - qualquer alteração na referência afeta diretamente a variável original.

# Referências

- **Passagem de Argumentos por Referência:**
  - Ao passar uma variável para uma função por referência, você pode modificar a variável original dentro da função.
  - Isso é feito declarando o parâmetro da função como uma referência:

```
void modifyValue(int& num) {  
    num = 100;  
}  
  
int main() {  
    int value = 42;  
    modifyValue(value);  
    std::cout << "Modified value: " << value << std::endl; // Saída: Modified value: 100  
    return 0;  
}
```

# Referências

- **Referências como Retorno de Função:**

- Funções podem retornar referências, permitindo que você retorne variáveis que estão fora do escopo da função.
- Tenha cuidado ao retornar referências para variáveis locais, pois elas podem se tornar inválidas quando a função terminar.
- Normalmente, referências retornadas de funções são usadas para permitir atribuições encadeadas ou para retornar elementos de contêineres.

```
#include <iostream>

// Função que retorna uma referência para um elemento em um array
int& getElement(int arr[], int index) {
    return arr[index];
}

int main() {
    int numbers[] = {10, 20, 30, 40, 50};

    int& ref = getElement(numbers, 2); // Obtendo uma referência para o terceiro elemento do array
    std::cout << "Original value: " << ref << std::endl; // Saída: Original value: 30

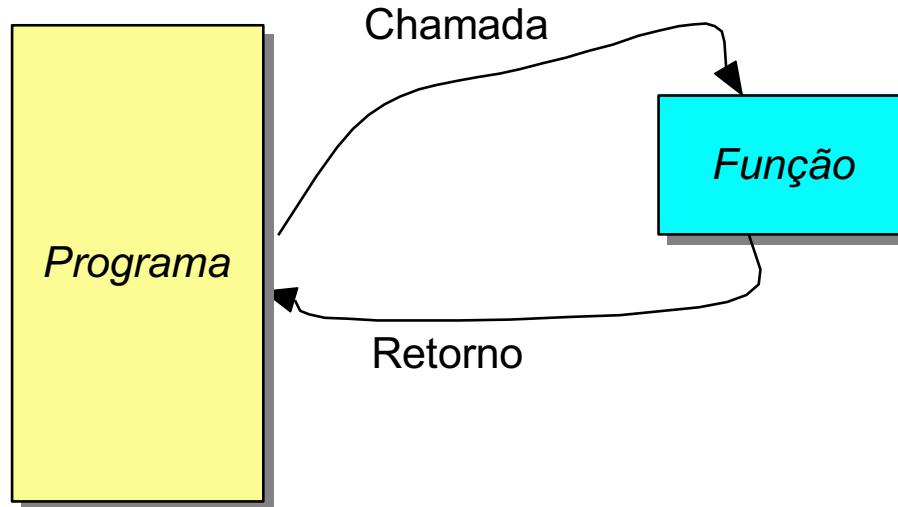
    ref = 100; // Modificando o valor através da referência
    std::cout << "Modified value: " << numbers[2] << std::endl; // Saída: Modified value: 100

    return 0;
}
```

# Funções

# Funções

- **Funções** contém um conjunto de instruções a fim de realizar uma tarefa específica.
- Dividir e conquistar
  - Construir um programa a partir de pedaços ou componentes menores
  - Cada pedaço é mais gerenciável do que o programa original



# Funções

- **Funções**
  - Escrito apenas uma vez
  - Essas instruções estão ocultas de outras funções.
  - Permitem que o programador modularize um programa
  - Permite a reusabilidade de código
  - Facilita a manutenção / teste
- **Variáveis locais**
  - Conhecidos apenas na função em que são definidos
  - Todas as variáveis declaradas nas definições de função são variáveis locais
- **Parâmetros**
  - Variáveis locais passadas quando a função é chamada que fornecem à função informações externas

# Funções: Definição

- Crie funções personalizadas para
  - Obter dados
  - Realizar operações
  - Retornar o resultado
- Formato para definição da função:

```
tipo_retorno nome_funcao(parametros) {  
    // Corpo da função  
    // ...  
    return valor_retorno; // Opcional, dependendo do tipo de retorno da função  
}
```

# Funções: Exemplo

```
#include <iostream>

// Função que recebe dois inteiros como parâmetros e retorna a soma deles
int somar(int a, int b) {
    int resultado = a + b;
    return resultado;
}

int main() {
    int x = 5;
    int y = 3;

    // Chamando a função somar e armazenando o resultado em uma variável
    int resultado_soma = somar(x, y);

    std::cout << "A soma de " << x << " e " << y << " é: " << resultado_soma <<
    std::endl;

    return 0;
}
```

# Protótipos da Função

- O protótipo de uma função em C++ é uma declaração antecipada da função, contendo:
  - **Nome da função**
  - **Parâmetros**
    - Informações que a função recebe
    - C++ é fortemente tipado – erro ao passar um parâmetro do tipo errado
  - **Tipo de retorno**
    - Tipo de informação que a função passa de volta para o chamador (int padrão)
    - **void** significa que a função não retorna nada
  - Necessário apenas se a definição da função vier após a chamada da função no programa
  - Particularmente útil quando há várias funções interdependentes

# Funções: Protótipo

! É uma prática recomendada usar protótipos de função, pois eles ajudam a tornar o código mais legível e organizado

```
// Protótipo da função
tipo_retorno nome_funcao(tipo_parametro1 nome_parametro1, tipo_parametro2
nome_parametro2, ...);

int main() {
    // ...
    // Chamada da função
    tipo_retorno resultado = nome_funcao(valor_parametro1, valor_parametro2,
...);
    // ...
    return 0;
}

// Definição da função
tipo_retorno nome_funcao(tipo_parametro1 nome_parametro1, tipo_parametro2
nome_parametro2, ...) {
    // Corpo da função
    // ...
    return valor_retorno;
}
```

# Arquivos de Cabeçalho

- **Arquivos de cabeçalho**

- Contêm protótipos de função para funções de bibliotecas
  - <cstdlib> , <cmath>, etc.
- Carregue com #include <nome do arquivo>
- Exemplo:

```
#include <cmath>
```

- **Arquivos de cabeçalho personalizados**

- Definido pelo programador
- Salvar como nome do **arquivo.h**
- Carregado no programa usando  

```
#include "arquivo.h"
```
- Busca no diretório do projeto

# Especificadores de Classe de Armazenamento

- São palavras-chave usadas para definir o escopo (tempo de vida) e o armazenamento de uma variável ou função em um programa.
- Existem quatro principais especificadores de classe de armazenamento em C++: **auto**, **register**, **static** e **extern**.
  - **auto**: foi redefinido em C++11 e tem um significado diferente do que tinha em C++98.
    - Em C++11 e versões posteriores, é usado principalmente em contextos de dedução de tipo automático, onde o compilador infere automaticamente o tipo da variável com base no valor inicial atribuído a ela.
  - **register**:
    - tenta colocar variáveis em registradores de alta velocidade
    - Só pode ser usado com variáveis e parâmetros locais
  - **static**: é usado para controlar o tempo de vida e o escopo de variáveis e funções.
    - aplicado a uma variável local dentro de uma função, faz com que a variável mantenha seu valor entre chamadas consecutivas da função.
    - aplicado a uma variável global, restringe o escopo da variável à unidade de tradução (arquivo de origem) em que ela é declarada.
    - aplicado a uma função global, restringe o escopo da função à unidade de tradução, tornando-a inacessível a outras unidades de tradução.
  - **extern**: é usado para declarar variáveis e funções que são definidas em outro lugar no programa, normalmente em um arquivo de origem separado.
    - Permite que você use variáveis e funções definidas em outros arquivos sem precisar redefini-las.
    - O **extern** informa ao compilador que a variável ou função será definida em algum outro lugar durante a ligação (linking) do programa final.

# Especificadores de Classe de Armazenamento

```
#include <iostream>

static int globalStaticVariable = 10; // Variável global com armazenamento
estático

void foo() {
    static int localStaticVariable = 5; // Variável local com armazenamento
estático
    localStaticVariable++;
    std::cout << "Local static variable: " << localStaticVariable << std::endl;
}

int main() {
    auto x = 42; // Uso de dedução automática de tipo

    register int y = 100; // Sugestão para armazenar em um registrador (pode
ser ignorado)

    foo();

    extern int globalVariableFromOtherFile; // Declarando variável definida em
outro arquivo

    return 0;
}
```

# Regras de Escopo

- Definem onde as variáveis, funções, classes e outros elementos do programa são visíveis e acessíveis. São elas:
  - 1. Escopo Global:** é o escopo mais amplo e abrange todo o programa. Variáveis globais e funções declaradas fora de qualquer função têm escopo global e podem ser acessadas de qualquer lugar no programa.
  - 2. Escopo de Função:** variáveis declaradas dentro de uma função não podem ser acessadas fora da função.
  - 3. Escopo de Bloco:** um bloco é um conjunto de instruções cercado por chaves {}. Variáveis declaradas dentro de um bloco têm escopo apenas dentro desse bloco. Inclui blocos dentro de funções e também blocos aninhados.
  - 4. Escopo de Classe:** membros (variáveis e funções) declarados dentro de uma classe têm escopo associado à classe. Isso significa que eles podem ser acessados por objetos (instâncias) da classe.

# Regras de Escopo

5. **Escopo de Namespace:** namespaces permitem organizar elementos em grupos separados para evitar conflitos de nomes. Os elementos declarados em um namespace têm escopo associado ao namespace.
6. **Escopo de Membro Estático:** membros estáticos de uma classe têm um escopo associado à classe, não às instâncias da classe. **Podem ser acessados usando o nome da classe, mesmo sem criar um objeto da classe.**
7. **Escopo de Parâmetro de Função:** parâmetros de função têm escopo dentro do corpo da função e são visíveis apenas nesse contexto.
8. **Escopo de Declaração:** em C++ mais moderno (a partir de C++17), é possível declarar variáveis em loops for, permitindo que essas variáveis tenham escopo limitado ao loop.

# Regras de Escopo

```
#include <iostream>

// Escopo de Classe
class MyClass {
public:
    int x = 100;

    void classFunction() {
        std::cout << "Class function: " << x << std::endl;
    }
};

// Escopo de Namespace
namespace MyNamespace {
    int x = 50;

    void namespaceFunction() {
        std::cout << "Namespace function: " << x << std::endl;
    }
}

// Escopo Global
int x = 10;
```

# Regras de Escopo

```
int main() {  
  
    std::cout << "Global variable: " << x << std::endl;  
  
    // Escopo de Função  
    int x = 20;  
  
    std::cout << "Main variable: " << x << std::endl;  
  
    // Escopo de Bloco  
    {  
        int x = 30;  
        std::cout << "Block variable: " << x << std::endl;  
  
    }  
  
    // Escopo de classe  
    MyClass myObject;  
    myObject.classFunction();  
  
    // Escopo de namespace  
    MyNamespace::namespaceFunction();  
  
    return 0;  
}
```

Global variable: 10  
Main variable: 20  
Block variable: 30  
Class function: 100  
Namespace function: 50

# Regras de Escopo

```
using namespace MyNamespace;

int main() {

    std::cout << "Global variable: " << x << std::endl;

    // Escopo de Função
    int x = 20;

    std::cout << "Main variable: " << x << std::endl;

    // Escopo de Bloco
    {
        int x = 30;
        std::cout << "Block variable: " << x << std::endl;

    }

    // Escopo de classe
    MyClass myObject;
    myObject.classFunction();

    // Escopo de namespace
    MyNamespace::namespaceFunction();

    return 0;
}
```

```
testes.cpp:27:41: error:
reference to 'x' is
ambiguous
    std::cout << "Global
variable: " << x <<
std::endl;
^

testes.cpp:22:5: note:
candidate found by name
lookup is 'x'
int x = 10;
^

testes.cpp:15:9: note:
candidate found by name
lookup is 'MyNamespace::x'
    int x = 50;
^

1 error generated.
```

# Operador Unário de Resolução de Escopo

- Operador de resolução de escopo unário (::)
  - Acessar variáveis globais se uma variável local tiver o mesmo nome
  - Não é necessário se os nomes forem diferentes

```
#include <iostream>
int x = 5; // Variável global
namespace MyNamespace {
    int x = 10; // Variável dentro do namespace
}
class MyClass {
public:
    static int x; // Variável de classe
};

int MyClass::x = 20; // Inicialização da variável de classe

int main() {
    int x = 15; // Variável local

    std::cout << "Local x: " << x << std::endl; // Imprime o valor da variável local
    std::cout << "Global x: " << ::x << std::endl; // Usa o operador de resolução de escopo unário
    para acessar a variável global
    std::cout << "Namespace x: " << MyNamespace::x << std::endl; // Acessa a variável no namespace
    std::cout << "Class x: " << MyClass::x << std::endl; // Acessa a variável de classe

    return 0;
}
```

# Referências e Parâmetros de Referência

- **Chamada por valor**

- Cópia dos dados passados para a função
- Alterações na cópia não alteram o original
- Usado para prevenir efeitos colaterais indesejados

- **Chamada por referência**

- Função pode acessar dados diretamente
- As alterações afetam o original

```
void change( int &variable ) {  
    variable += 3;  
}  
  
int main() {  
  
    int x = 20;  
    change(x);  
    std::cout << "Main variable: " << x << std::endl;  
    return 0;  
}
```

Main variable: 23

# Funções Recursivas

- São funções que chamam a si mesmas
  - Só pode resolver um caso base
  - Se não for o caso base, a função divide o problema em um problema um pouco menor, um pouco mais simples, que se assemelha ao problema original e
    - Lança uma nova cópia de si mesmo para trabalhar no problema menor, convergindo lentamente para o caso base
    - Faz uma chamada para si mesmo dentro da instrução de return
- Eventualmente, o caso base é resolvido e, em seguida, esse valor retorna para resolver todo o problema
- **Exemplo:** factorial
  - $n! = n * (n - 1) * (n - 2) * \dots * 1$
  - Relação recursiva ( $n! = n * (n - 1)!$ )
    - $5! = 5 * 4!$
    - $4! = 4 * 3! \dots$
  - Caso base ( $1! = 0! = 1$ )

# Funções Recursivas

```
#include <iostream>

// Função recursiva para calcular o fatorial
unsigned long long int fatorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * fatorial(n - 1);
    }
}

int main() {
    int numero;
    std::cout << "Digite um número inteiro não negativo: ";
    std::cin >> numero;

    if (numero < 0) {
        std::cout << "O número deve ser não negativo." << std::endl;
    } else {
        unsigned long long int resultado = fatorial(numero);
        std::cout << "O fatorial de " << numero << " é: " << resultado << std::endl;
    }
}

return 0;
}
```

# Funções com listas de parâmetros vazias

- Listas de parâmetros vazias
  - Escrever **void** ou deixar uma lista de parâmetros vazia indica que a função não aceita argumentos

```
void imprimir();
```

ou

```
void imprimir(void);
```
  - A função **imprimir** não aceita argumentos e não retorna nenhum valor

# Inline Functions

- inline functions
  - Reduz a sobrecarga da chamada de função
  - Solicita ao compilador que copie o código no programa em vez de usar uma chamada de função
  - O compilador pode ignorar inline
  - Deve ser usado com funções pequenas e usadas com frequência

```
#include <iostream>
// Função inline que retorna o quadrado de um número
inline int quadrado(int x) {
    return x * x;
}

int main() {
    int numero = 5;
    int resultado = quadrado(numero); // O corpo da função quadrado é inserido diretamente
    aqui

    std::cout << "O quadrado de " << numero << " é: " << resultado << std::endl;

    return 0;
}
```

# Argumentos Default

- Se o parâmetro da função for omitido, obtém o valor padrão
  - Podem ser constantes, variáveis globais ou chamadas de função
  - Se não forem especificados parâmetros suficientes, os parâmetros mais à direita vão para seus padrões
- Os valores default são definidos no protótipo da função

# Argumentos Default

```
#include <iostream>

// Função com argumento padrão
int calcularPotencia(int base, int expoente = 2) {
    int resultado = 1;
    for (int i = 0; i < expoente; ++i) {
        resultado *= base;
    }
    return resultado;
}

int main() {
    // Chama a função com ambos os argumentos especificados
    int resultado1 = calcularPotencia(2, 3);
    std::cout << "2^3 = " << resultado1 << std::endl;

    // Chama a função com apenas o primeiro argumento, usando o valor padrão
    // para o segundo
    int resultado2 = calcularPotencia(4);
    std::cout << "4^2 (default expoente) = " << resultado2 << std::endl;

    return 0;
}
```

# Expressões de Fold

- Introduzido na versão C++17, a dobragem (folding) de expressões é uma funcionalidade que permite combinar os valores de uma sequência de elementos usando um operador binário.
- O operador de fold (...) é usado em combinação com operadores binários para aplicar o operador repetidamente a todos os elementos em uma sequência.

```
#include <iostream>

template <typename... Args>
auto soma(Args... args) {
    return (args + ...); // Fold para somar todos os argumentos
}

int main() {
    int result = soma(1, 2, 3, 4, 5);
    std::cout << "Soma: " << result << std::endl; // Output: Soma: 15

    return 0;
}
```

# Sobrecarga de funções

- A sobrecarga de funções permite definir múltiplas funções com o mesmo nome, mas com diferentes tipos ou números de parâmetros
  - Deve executar tarefas semelhantes (ou seja, uma função **quadrado** para ints e uma função **quadrado** para floats).
    - `int quadrado(int x) {return x * x; }`
    - `float quadrado(float x) { return x * x; }`
  - O programa escolhe a função pela assinatura
    - assinatura determinada pelo nome da função e tipos de parâmetro
    - pode ter os mesmos tipos de retorno

# Sobrecarga de funções

```
#include <iostream>
// Função que recebe dois inteiros e retorna sua soma
int add(int a, int b) {
    return a + b;
}
// Função que recebe duas strings e concatena
std::string add(const std::string& s1, const std::string& s2) {
    return s1 + s2;
}
// Função que recebe três doubles e retorna sua soma
double add(double a, double b, double c) {
    return a + b + c;
}
int main() {
    int result1 = add(5, 7);
    std::string result2 = add("Hello, ", "world!");
    double result3 = add(1.5, 2.5, 3.5);

    std::cout << "Result 1: " << result1 << std::endl;
    std::cout << "Result 2: " << result2 << std::endl;
    std::cout << "Result 3: " << result3 << std::endl;

    return 0;
}
```

# Funções Lambda

- São uma forma concisa de criar funções anônimas diretamente em uma expressão.
- São úteis quando você precisa de uma função temporária ou quando deseja passar uma função como argumento para outra função.
- Fornecem uma maneira mais legível e conveniente de criar pequenas funcionalidades sem a necessidade de definir funções separadas.

# Funções Lambda

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

int main() {
    // Exemplo 1: Função lambda simples
    auto add = [] (int a, int b) {
        return a + b;
    };

    std::cout << "Sum: " << add(5, 3) << std::endl; // Saída: Sum: 8

    // Exemplo 2: Uso de funções lambda com algoritmos da STL
    std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    int count = std::count_if(numbers.begin(), numbers.end(), [] (int num) {
        return num > 4;
    });

    std::cout << "Count: " << count << std::endl; // Saída: Count: 6

    return 0;
}
```

# Ponteiro para função

- Um ponteiro de função em C++ é um ponteiro que aponta para o endereço de uma função, permitindo que você chame essa função através do ponteiro.
- Sintaxe:

```
tipo_de_retorno (*nome_do_ponteiro) (lista_de_tipos_de_parâmetros)
```

**tipo\_de\_retorno:** O tipo de dado que a função apontada retornará.

**(\*nome\_do\_ponteiro):** A declaração do próprio ponteiro de função. Note o uso de parênteses para garantir que o operador \* (ponteiro) seja aplicado ao nome do ponteiro.

**lista\_de\_tipos\_de\_parâmetros:** Uma lista dos tipos de parâmetros que a função apontada recebe.

# Ponteiro para função

```
#include <iostream>

// Função que retorna a soma de dois inteiros
int add(int a, int b) {
    return a + b;
}

// Função que retorna a diferença de dois inteiros
int subtract(int a, int b) {
    return a - b;
}

int main() {
    // Declarando um ponteiro de função que aponta para uma função que recebe dois inteiros e retorna
    // um int
    int (*funcPtr)(int, int);

    funcPtr = add; // Apontando para a função "add"
    int result1 = funcPtr(5, 3); // Chamando a função "add" através do ponteiro
    std::cout << "Result 1: " << result1 << std::endl;

    funcPtr = subtract; // Apontando para a função "subtract"
    int result2 = funcPtr(10, 4); // Chamando a função "subtract" através do ponteiro
    std::cout << "Result 2: " << result2 << std::endl;

    return 0;
}
```

- Na próxima aula
  - Arrays e matrizes
  - Strings e manipulação de cadeias de caracteres
  - Estruturas de dados avançadas: listas, pilhas e filas
  - Value categories