

# Curso de Extensão - INF1900

# Programação em C++



## Módulo 1

# Introdução à Programação em C++

## Aula 4

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

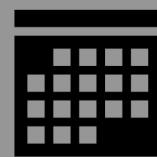
# Monitorias

- **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.

# Vetores e Matrizes

# Arrays

- Um array é uma coleção de elementos do mesmo tipo, acessados por um índice
  - A inclusão de índice facilita o tratamento destas variáveis em algoritmos
- Os índices de um array **começam em 0** e vão até o **tamanho do array menos um**
- **Declaração:**

```
int numeros[5]; // Declara um array de inteiros com 5 elementos
```

- **Inicialização:**

```
int numeros[5] = {1, 2, 3, 4, 5}; // Inicializa o array com valores
```

- **Acesso aos elementos:**

```
int terceiroNumero = numeros[2]; // Acessa o terceiro elemento (índice 2)
```

- **Tamanho do Array:**

```
int tamanho = sizeof(numeros) / sizeof(numeros[0]); // Calcula o tamanho do array
```

# Matrizes

- Uma matriz é um array multidimensional, organizado em linhas e colunas.
- Pode ser pensada como um array de arrays, onde cada elemento é acessado por dois índices.
- **Declaração:**

```
int matriz[3][4]; // Declara uma matriz 3x4 de inteiros
```

- **Inicialização:**

```
int matriz[3][4] = {{1, 2, 3, 4},  
                     {5, 6, 7, 8},  
                     {9, 10, 11, 12}}; // Inicializa a matriz com valores
```

# Alocação Dinâmica de vetores e matrizes

- A alocação dinâmica permite alocar memória em tempo de execução usando ponteiros.

- **Alocação Dinâmica de Arrays:**

```
int tamanho = 5;
int *ponteiroArray = new int[tamanho]; // Aloca um array de inteiros
delete[] ponteiroArray; // Libera a memória alocada quando não for mais necessário
```

- **Alocação Dinâmica de Matrizes:**

```
int linhas = 3, colunas = 4;
int **ponteiroMatriz = new int *[linhas];

for (int i = 0; i < linhas; i++) {
    ponteiroMatriz[i] = new int[colunas];
}

// Acesso aos elementos: ponteiroMatriz[linha][coluna]
for (int i = 0; i < linhas; i++) {
    delete[] ponteiroMatriz[i]; // Libera memória das linhas
}
delete[] ponteiroMatriz; // Libera memória do array de ponteiros
```

# Alocação Dinâmica de vetores e matrizes

- A alocação dinâmica requer gerenciamento cuidadoso da memória para evitar vazamentos.
- A partir do C++11, é recomendado o uso de `std::vector` para gerenciar arrays dinâmicos.

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

int main() {
    int numeros[5] = {1, 2, 3, 4, 5};
    cout << "Terceiro número: " << numeros[2] << endl;

    int matriz[3][3] = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
    cout << "Elemento na segunda linha e segunda coluna: " << matriz[1][1] << endl;

    return 0;
}
```

# Strings em C++

- Uma string em C++ é uma sequência de caracteres.
- C++ oferece uma maneira conveniente de lidar com strings usando a classe **std::string** da biblioteca padrão. A classe **std::string** fornece diversas funcionalidades para manipulação de strings de forma eficiente e segura.
- Em C++, o uso de uma string como uma expressão direta em um switch não é suportado.

```
#include <string>

int main() {

    std::string nome = "INF1900";

    std::string saudacao = "Olá, " + nome + "!";
    int tamanho = nome.length(); // Ou: int tamanho = nome.size();
    char primeiroChar = nome[0]; // Acesso aos Caracteres

    std::string mensagem = "C++ é divertido!";
    mensagem.append(" Vamos aprender mais.");
    mensagem.erase(9, 8); // Remove "divertido"
    mensagem.replace(0, 3, "C"); // Substitui "C++" por "C"
}
```

# Strings em C++

- **Comparação de Strings:**

```
std::string texto1 = "abc";
std::string texto2 = "def";
if (texto1 == texto2) {
    // Igual
} else if (texto1 < texto2) {
    // texto1 vem antes
} else {
    // texto2 vem antes
}
```

- **Busca em Strings:**

```
std::string frase = "A vida é bela.";
size_t pos = frase.find("vida"); // Retorna a posição da primeira ocorrência
if (pos != std::string::npos) {
    // Encontrou a substring
}
```

# Strings em C++

- Conversão de String para Inteiro e Vice-Versa:

```
std::string numTexto = "123";
int numero = std::stoi(numTexto); // String para int
std::string novoNumTexto = std::to_string(numero); // Int para string
```

- Uso de Iteradores:

```
std::string mensagem = "Iterando.";
for (auto it = mensagem.begin(); it != mensagem.end(); ++it) {
    char caractere = *it;
}
```

- Exemplo geral

```
#include <iostream>
#include <string>

int main() {
    std::string nome = "Alice";
    std::cout << "Olá, " << nome << "!" << std::endl;

    std::string frase = "C++ é poderoso.";
    std::cout << "Tamanho da frase: " << frase.length() << std::endl;

    return 0;
}
```

# Strings em C++

- A classe **std::string** oferece uma forma mais segura e flexível de trabalhar com strings em comparação com as strings do estilo C (char arrays). Ela realiza automaticamente a alocação e liberação de memória, facilitando o gerenciamento de strings.
- A conversão entre strings e vetores de caracteres pode ser feita de várias maneiras:
  - **De string para vetor de caracteres:**

```
#include <iostream>
#include <string>

int main() {
    std::string minhaString = "Hello, world!";

    // Usando c_str() e mantendo assim a integridade da string.
    const char* vetorDeCaracteres = minhaString.c_str();
    std::cout << vetorDeCaracteres << std::endl;

    // Usando data()
    const char* dadosDaString = minhaString.data();
    std::cout << dadosDaString << std::endl;

    return 0;
}
```

# Strings em C++

- **De vetor de caracteres para string:**

```
#include <iostream>
#include <string>

int main() {
    const char vetorDeCaracteres[] = "Hello, world!";

    // Criando uma string a partir do vetor de caracteres
    std::string minhaString(vetorDeCaracteres);
    std::cout << minhaString << std::endl;

    return 0;
}
```

- Strings em C++ são automaticamente gerenciadas em relação ao tamanho e ao armazenamento de memória
- Vetores de caracteres podem não ter essas vantagens, exigindo uma manipulação mais cuidadosa para evitar problemas de buffer overflow e vazamentos de memória.

# Categorias de Valor

# Categorias de Valor

- As **value categories** são usadas para classificar as expressões e determinar como essas expressões podem ser usadas em diferentes contextos.
- Em C++, existem três categorias de valores principais: lvalue, prvalue e xvalue.
  - **Lvalue (l-value - "left value"):**
    - Representa um objeto identificável na memória.
    - Pode aparecer no lado esquerdo de uma atribuição.
    - Tem um endereço no espaço de memória.
    - Exemplos: variáveis, elementos de um array e objetos com nome.
  - **Prvalue (p-rvalue - "pure r-value"):**
    - Representa um valor computado.
    - Geralmente não tem um endereço na memória.
    - Não pode ser usado diretamente como o alvo de uma atribuição.
    - Exemplos: literais numéricos, literais de string e resultados de expressões aritméticas.
  - **Xvalue (x-value - "expiring value"):**
    - Indica um objeto que está prestes a ser expirado (movido ou transferido).
    - Geralmente ocorre em situações como retornos de `std::move()` ou em algumas chamadas de funções.
    - Foi introduzido com o conceito de "rvalue references" para permitir transferência eficiente de recursos.
    - Geralmente usado para implementar a semântica de movimento (move semantics).
- O uso de referências lvalue e rvalue em C++11 e versões posteriores é fundamental para permitir a semântica de movimento eficiente e a implementação de técnicas como a troca (swap) eficiente.

# Categorias de Valor

- **L-value:**

```
int x = 10; // 'x' é um lvalue
int array[5]; // 'array' é um lvalue

int* ptr = &x; // endereço de 'x' é obtido, portanto 'ptr' é um lvalue
```

- **Prvalue (p-rvalue):**

```
int result = 20 + 30; // A expressão '20 + 30' é um prvalue
double value = 3.14; // O literal '3.14' é um prvalue
```

- **Xrvalue (x-rvalue):**

```
#include <utility>

int main() {
    int a = 5;
    int&& rv = std::move(a); // 'std::move(a)' retorna um xvalue, indicando que a está prestes a
    ser expirada ou transferida.
    return 0;
}
```

# Estruturas de Dados Avançadas em C++

# Vector

- É uma classe da biblioteca padrão do C++ que implementa um array dinâmico.
  - Fornece uma estrutura de dados sequencial que permite armazenar um conjunto de elementos em uma ordem contígua na memória.
  - `std::vector` pode crescer ou encolher dinamicamente durante o tempo de execução.
  - é otimizado para oferecer um bom desempenho para operações comuns, como acesso aleatório e inserção/remoção no final. No entanto, ele pode não ser a estrutura de dados mais eficiente para inserções/remoções frequentes no meio.

# Vector

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

int main() {
    std::vector<int> numeros;

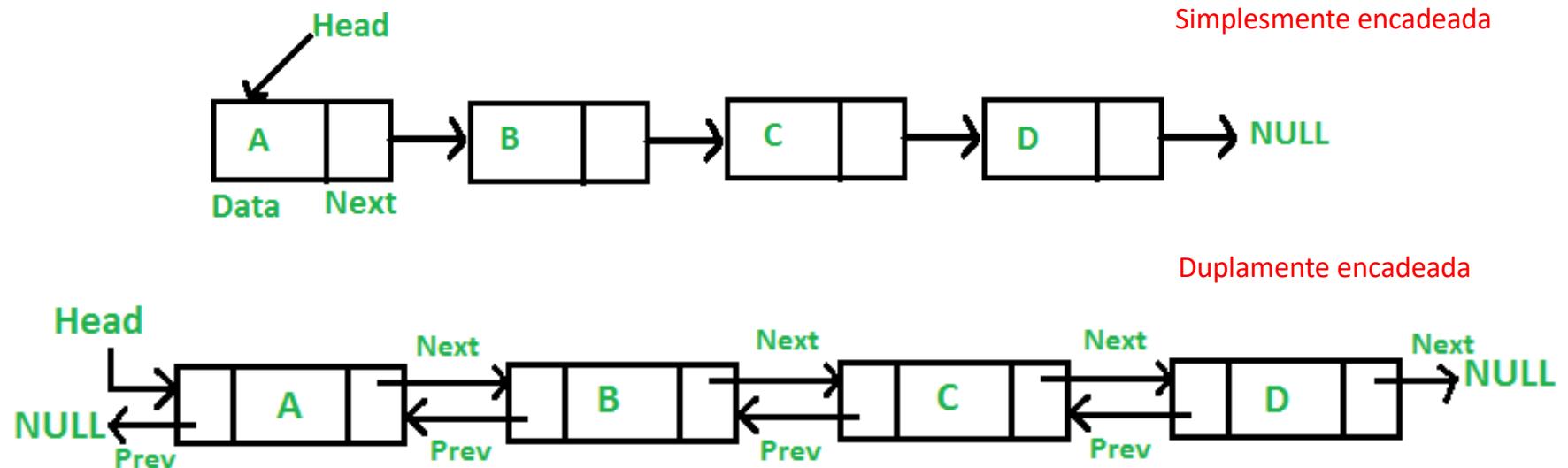
    numeros.push_back(10);
    numeros.push_back(20);
    numeros.push_back(30);

    for (const int& num : numeros) {
        std::cout << num << " ";
    }

    return 0;
}
```

# Listas

- **Lista (Linked List):** Uma lista é uma estrutura de dados composta por nós, onde cada nó contém um valor e um ponteiro para o próximo nó na sequência. Existem dois tipos principais de listas: lista encadeada simples e lista encadeada dupla.



Fonte: <https://www.geeksforgeeks.org/>

# Listas

- **Lista encadeada simples**

- Complexidade de inserção e deleção em uma determinada posição é  $O(n)$ .
- A complexidade da exclusão com um determinado nó é  $O(n)$ , porque o nó anterior precisa ser conhecido e a travessia leva  $O(n)$
- Preferida para pilhas.
- Menos memória.

# Listas

- **Lista duplamente encadeada**
  - A complexidade da inserção e exclusão em uma determinada posição é  $O(n/2) = O(n)$  porque a travessia pode ser feita do início ou do fim.
  - A complexidade da exclusão com um determinado nó é  $O(1)$  porque o nó anterior pode ser acessado facilmente
  - Usadas para construir heaps e pilhas, árvores binárias.
  - Mais memória
- **A classe `std::list` da biblioteca padrão do C++ implementa uma lista duplamente encadeada**

# Listas

```
#include <iostream>
#include <list>

int main() {
    std::list<int> myList;
    myList.push_back(5); // Adiciona 5 ao final da lista
    myList.push_front(2); // Adiciona 2 no inicio da lista
    myList.push_back(8); // Adiciona 8 ao final da lista
    // Exibe os elementos da lista
    for (const int& num : myList) {
        std::cout << num << " ";
    }
    std::cout << std::endl; // Output: 2 5 8
    myList.pop_front(); // Remove o elemento do inicio da lista
    std::cout << "After pop_front(): ";
    for (const int& num : myList) {
        std::cout << num << " ";
    }
    std::cout << std::endl; // Output: 5 8
    myList.push_back(10);
    std::cout << "After push_back(10): ";
    for (const int& num : myList) {
        std::cout << num << " ";
    }
}
```

# Listas

```
std::cout << std::endl; // Output: 5 8 10
myList.reverse(); // Inverte a ordem dos elementos
std::cout << "After reverse(): ";
for (const int& num : myList) {
    std::cout << num << " ";
}
std::cout << std::endl; // Output: 10 8 5

myList.sort(); // Ordena os elementos

std::cout << "After sort(): ";
for (const int& num : myList) {
    std::cout << num << " ";
}
std::cout << std::endl; // Output: 5 8 10

return 0;
}
```

# Pilha

- Uma pilha é uma estrutura de dados baseada no princípio LIFO (Last In, First Out), onde o último elemento inserido é o primeiro a ser removido.

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> stack;
    stack.push(5);
    stack.push(10);
    stack.push(3);

    std::cout << stack.top() << std::endl; // Output: 3

    stack.pop();

    std::cout << stack.top() << std::endl; // Output: 10

    return 0;
}
```

# Queue

- Uma fila é uma estrutura de dados baseada no princípio FIFO (First In, First Out), onde o primeiro elemento inserido é o primeiro a ser removido.

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> queue;
    queue.push(8);
    queue.push(15);
    queue.push(6);

    std::cout << queue.front() << std::endl; // Output: 8

    queue.pop();

    std::cout << queue.front() << std::endl; // Output: 15

    return 0;
}
```

- Próxima aula...
- Orientação a objetos