

IMD0030 - LINGUAGEM DE PROGRAMAÇÃO I (GRADUAÇÃO)

Professor: Jerffeson Gomes



Apresentação

Nome: Jerffeson Gomes Dutra

E-mail: jerffeson.gomes@imd.ufrn.br (Tarde)



Quem são vocês:

- Nome;
- Já sabia programação antes de entrar no curso;
- Como está indo o semestre? Difícil?



Disciplina

IMD0030 - LINGUAGEM DE PROGRAMAÇÃO I (GRADUAÇÃO)

Horário: 35T56

Local: A309

Faltas justificadas apenas mediante atestado



As regras do jogo

- Não será aceito uso de IA no desenvolvimento dos Trabalho ou Provas;
- Trabalhos copiados em todo ou em parte de outros estudantes ou da Internet receberão automaticamente nota zero
- Códigos que não compilam serão avaliados, porém com penalidade;
- Pontos extras serão adicionados em códigos com uso de boas práticas de programação (Se necessário para aprovação)



Avaliação

1. Prova escrita; (Replicada para 1º Unidade)
2. Trabalho Final Prático;

Cronograma

20/05	Apresentação e introdução da Disciplina
22/05	Estruturas de Controle e de Repetição
27/05	Debug, Makefile, GitLab e Struct
29/05	Funções com Sobrecarga e Argumentos Padrão
03/06	Tratamento de Exceções
05/06	Introdução à STL (Standard Template Library)
10/06	Classes e Objetos – Conceitos Básicos
12/06	Construtores e Destrutores
17/06	Prova Teórica
19/06	Feriado - Corpus Christi

24/06	Encapsulamento e Métodos de Acesso
26/06	Sobrecarga de Operadores
01/07	Membros Estáticos e Constantes de Classe
03/07	Herança e Classes Derivadas
08/07	Polimorfismo e Métodos Virtuais
10/07	Templates (Funções e Classes Genéricas)
15/07	Ajuda com os trabalhos
17/07	Apresentação Trabalho
22/07	Apresentação Trabalho
24/07	Reposição (Prova Teórica)



Dúvidas?





Como vamos abordar a disciplina?

Teoria



ou

Prática





Boas práticas de programação a serem avaliadas

Clareza e Legibilidade

1. Use nomes descritivos para variáveis e funções (`somaTotal`, `calcularMedia`).
2. Evite abreviações desnecessárias.
3. Use indentação e espaçamento consistentes.
4. Comente apenas quando necessário (explique o porquê, não o como).



Boas práticas de programação a serem avaliadas

Organização do Código

1. Separe o código em funções pequenas.
2. Evite repetições.
3. Uma função, uma responsabilidade.
4. Agrupe funções relacionadas em arquivos/modularização.

Faz sentido estudar C/C++

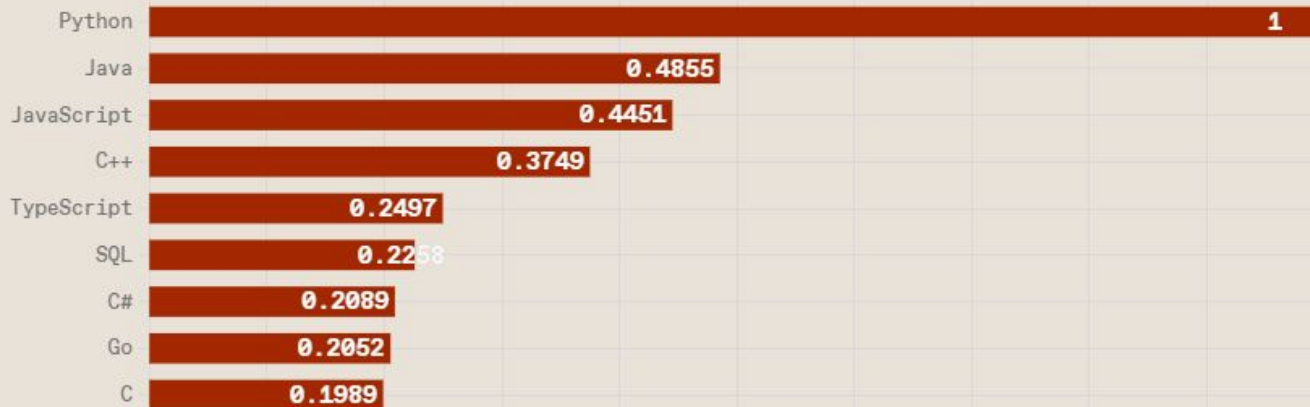
Top Programming Languages 2024

Click a button to see a differently weighted ranking

Spectrum

Trending

Jobs



Comparando C/C++

Linguagem C	Linguagem C++
Paradigma procedural	Multiparadigma (procedural e orientado a objetos)
Inteiro como valor booleano	Tipo <code>bool</code>
Variáveis devem ser declaradas no início de um bloco	Variáveis podem ser declaradas em qualquer parte de um bloco
<code>stdio.h</code> define canais de entrada e saída (<code>printf</code> e <code>scanf</code>)	<code>iostream</code> define canais de entrada e saída (<code>std::cout</code> e <code>std::cin</code>)
<i>String</i> como vetor de caracteres	Tipo <code>std::string</code>
Casts simples	Novos tipos de cast
Não suporta tipos de dados abstratos	Suporta tipos de dados abstratos
Desprovida de suporte a estruturas genéricas	Suporta estruturas de código parametrizadas ou genéricas (<i>templates</i>)



Compilador g++

Windows: <https://www.msys2.org/>

Talvez seja necessário adicionar o caminho C:\msys64\mingw64\bin nas variáveis de ambiente

Linux: Normalmente vem padrão

Se não, instalar: `sudo apt install build-essential`

Testar: `g++ -version`



Compilador online

<https://www.onlinegdb.com/>



Olá Mundo!

```
#include <iostream>

using namespace std;

int main() {

    cout << "Olá Mundo" << endl;

    return 0;

}
```




```
int main()
```

É a função principal de um programa em C++.

Toda vez que você executa um programa, o processo de execução começa dentro dessa função.

O C++ exige que o programa tenha uma função main, pois é nela que a execução do programa inicia.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout << "Olá Mundo" << endl;
```

```
    return 0;
```

```
}
```

#include <iostream>

É uma diretiva de pré-processador no C e C++ que indica ao compilador para incluir o conteúdo de um arquivo específico no código fonte durante o processo de compilação.

Em outras palavras, ela permite que você importar ou incluir arquivos externos, como bibliotecas padrão ou arquivos criados por você.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout << "Olá Mundo" << endl;
```

```
    return 0;
```

```
}
```



iostream

O iostream é um arquivo de cabeçalho que contém funcionalidades para ler dados de entrada (como do teclado) e escrever dados de saída (como na tela).

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout << "Olá Mundo" << endl;
```

```
    return 0;
```

```
}
```



return 0;

Indica o fim do programa.

Pode ser colocada em vários locais da minha aplicação.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout << "Olá Mundo" << endl;
```

```
    return 0;
```

```
}
```



cout

É utilizado para exibir informações no console

```
#include <iostream>

using namespace std;

int main() {

    cout << "Olá Mundo" << endl;

    return 0;

}
```



endl ou \n?

\n: Apenas insere uma quebra de linha.

endl: Insere uma quebra de linha e força o "flush" do buffer de saída,

Ou seja, garante que a saída seja realmente mostrada no terminal imediatamente.

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    cout << "Olá Mundo" << endl;
```

```
    return 0;
```

```
}
```



Comentários

Use // Para comentários de 1 linha

```
#include <iostream>

using namespace std;
```

Use /* */ Para comentários de várias linhas

```
int main() {

    // Comentários Simples

    /*
        Comentários de múltiplas linhas
    */

    return 0;

}
```

Variáveis

int	-2.147.483.648 a 2.147.483.647
double	$\pm 2.225 \times 10^{-308}$ a $\pm 1.798 \times 10^{308}$
char	'a' ou 'b'
string	"Hello Word"
bool	true ou false

```
int main() {  
  
    int numerointeiro = 5;  
    double numerosDecimais = 5.99;  
    char letra = 'D';  
    string textos = "Hello";  
    bool boleanos = true;  
    bool boleanos = false;  
  
    return 0;  
}
```




cin

O cin em C++ é usado para ler entradas do usuário no terminal — ele é o oposto do cout, que imprime saídas.

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

int main() {
    int idade;

    cout << "Digite sua idade: ";
    cin >> idade;

    cout << "Você tem " << idade << "
anos." << endl;

    return 0;
}
```



IF em C++

Serve para: executar um bloco de código condicionalmente

```
int x = 10;

if (x > 5) {
    cout << "Maior que 5" << endl;
} else if (x == 5) {
    cout << "Igual a 5" << endl;
} else {
    cout << "Menor que 5" << endl;
}
```



IF em C++

Serve para: executar um bloco de código condicionalmente

Em C++ é possível comparar strings com ==

```
string nome = "Ana";  
  
if (nome == "Ana") {  
    cout << "Olá, Ana!" << endl;  
}
```



switch em C++

Seleciona entre vários blocos de código com base no valor de uma variável.

Usado quando há muitas opções (casos).

break: termina o loop/switch

Se não colocar o break, a execução vai "cair" no próximo caso, mesmo que a condição não combine mais

```
switch (opcao) {  
    case 1:  
        cout << "Opção 1" << endl;  
        break;  
    case 2:  
        cout << "Opção 2" << endl;  
        break;  
    default:  
        cout << "Outra opção" << endl;  
        break;  
}
```



While em C++

Repetir um bloco de código enquanto a condição for verdadeira.

break: termina o loop/switch

continue: pula para a próxima iteração do loop

```
while (i < 5) {  
    cout << "i = " << i << endl;  
    continue;  
    i++;  
}
```



While em C++

Repete um bloco de código pelo menos uma vez e continuar enquanto a condição for verdadeira.

A condição é testada depois da execução.

break: termina o loop/switch

continue: pula para a próxima iteração do loop

```
do {  
    cout << "j = " << j << endl;  
    j++;  
} while (j < 5);
```



for em C++

Repete um bloco de código com controle de início, condição e incremento.

```
for (int i = 0; i < 5; i++) {  
    // Executa 5 vezes  
}
```

break: termina o loop/switch

continue: pula para a próxima iteração do loop



for em C++

Repete um bloco de código com controle de início, condição e incremento.

```
vector<int> numeros = {1, 2, 3, 4};  
  
for (int n : numeros) {  
    cout << n << endl;  
}
```




int[] ou vector<int>?

int[]: Um array estático é uma sequência de elementos de tamanho fixo que não pode ser alterado durante a execução do programa.

O array `nu[]` é de tamanho fixo, definido na hora da inicialização. Não podemos adicionar ou remover elementos depois.

```
for (int i = 0; i < 4; i++) {  
    cout << numeros[i] << endl;  
}
```



int[] ou vector<int>?

O `std::vector` é uma classe do C++ que fornece uma estrutura de dados dinâmica.

Você pode adicionar, remover e acessar elementos de maneira eficiente. Ele gerencia a memória automaticamente.

```
vector<int> numeros = {1, 2, 3, 4};
```

```
for (int n : numeros) {  
    cout << n << endl;  
}
```

```
nu.push_back(6); //Add no fim
```

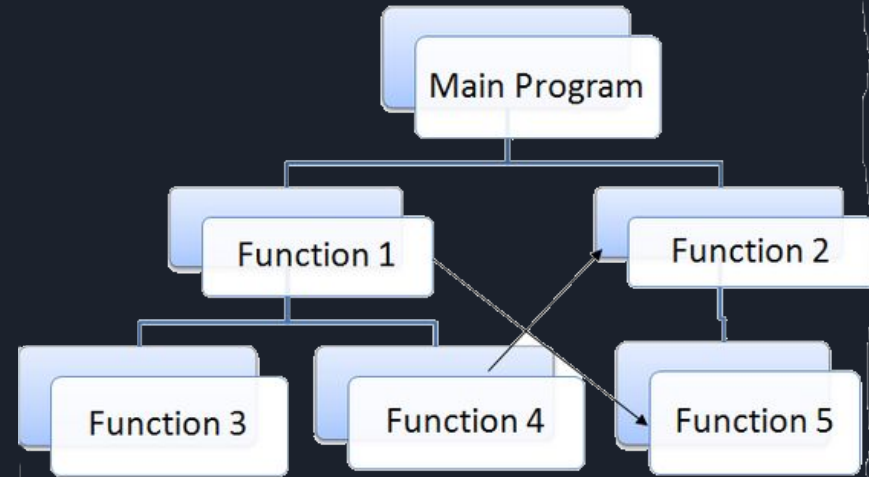
```
nu.pop_back(); //Remove no fim
```

Modularização

Estratégia para a construção de software complexo a partir de pequenas partes distintas, cada uma contendo responsabilidades específicas

Dividir para conquistar: dividir uma solução complexa em pequenas tarefas, cada uma sendo resolvida individualmente

Módulo: conjunto de instruções em um programa que possui responsabilidade bem definida e é o mais independente possível em relação ao resto do programa





Modularização

É possível organizar nosso código em funções.

Funções em C++ são blocos de código reutilizáveis que executam uma tarefa específica.

```
int subtracao(int a, int b) {  
    return a - b;  
}  
  
void nomeCalculadora() {  
    std::cout << "Calculadora LP 1";  
}
```



Modularização

É possível organizar nosso código em funções.

Funções em C++ são blocos de código reutilizáveis que executam uma tarefa específica.

```
void olaMundo() {  
    cout << "Olá, Mundo!";  
}  
  
int main() {  
  
    olaMundo();  
    return 0;  
}
```

Modularização

Por convenção da linguagem C++, a organização do código de um programa pode ser feita da seguinte maneira:

Arquivos de cabeçalho (.h) contêm declarações de estruturas, tipos, variáveis globais, protótipos de funções, constantes, etc. e não podem conter a função principal do programa (main)

Arquivos de corpo (.cpp) implementam ou fazem chamadas ao que é definido nos arquivos de cabeçalho



Modularização

matematica.h

```
#ifndef MATEMATICA_H
#define MATEMATICA_H

    int somar(int a, int b);
    int subtracao(int a, int b);

#endif
```

matematica.cpp

```
#include "matematica.h"

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

int subtracao(int a, int b) {
    return a - b;
}
```

Modularização

matematica.h

```
#pragma once
```

```
int somar(int a, int b);  
int subtracao(int a, int b);
```

matematica.cpp

```
#include "matematica.h"
```

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

```
int subtracao(int a, int b) {  
    return a - b;  
}
```




Modularização

#ifndef (if not defined) — verifica se uma macro NÃO foi definida antes.

#define — define uma macro para marcar que o arquivo já foi incluído.

`matematica.h`

```
#ifndef MATEMATICA_H
```

```
#define MATEMATICA_H
```

```
int somar(int a, int b);
```

```
int subtracao(int a, int b);
```

```
#endif
```



Como compilar?

```
g++ [arquivos.cpp] -o <nomePrograma>
```

```
g++ main.cpp matematica.cpp -o main
```



Opções de compilação?

Opções mais comuns para o compilador

- **-o <nome>** : especifica o nome do arquivo executável a ser criado a partir dos arquivos objeto (.o) já pré-compilados
- **-Wall** : diz ao compilador para indicar com um aviso (warning) qualquer instrução que possa levar a um erro

Por exemplo, ao habilitar esta opção, o compilador irá avisar sobre a instrução:

```
if (var = 5) { .. }.
```

- **-I<dir>** ("I" maiúscula): adiciona o diretório <dir> na lista de diretórios para a busca de arquivos incluídos (através do uso da diretiva #include), ou seja, indica ao compilador uma fonte extra de arquivos cabeçalho (.h)



Opções de compilação?

Opções mais comuns para o compilador

- **-o <nome>** : especifica o nome do arquivo executável a ser criado a partir dos arquivos objeto (.o) já pré-compilados
- **-Wall** : diz ao compilador para indicar com um aviso (warning) qualquer instrução que possa levar a um erro

Por exemplo, ao habilitar esta opção, o compilador irá avisar sobre a instrução:

```
if (var = 5) { .. }.
```

- **-I<dir>** ("I" maiúscula): adiciona o diretório <dir> na lista de diretórios para a busca de arquivos incluídos (através do uso da diretiva #include), ou seja, indica ao compilador uma fonte extra de arquivos cabeçalho (.h)



Calculadora simples modularizada

Criar um programa em C++ que realize operações matemáticas simples (adição, subtração, multiplicação e divisão) usando:

- Exibir as opções para o usuário 1 - Soma; 2 - Subtração; 3 - Divisão; 4 - Multiplicação; 0 - Sair;
- O programa deve para de executar quando o usuário digitar 0;
- Arquivos de cabeçalho para declarar as funções
- Arquivos .cpp para implementar as funções
- Um arquivo principal (main.cpp) que usa essas funções para interagir com o usuário



Depuração de Código

Depuração (debugging ou debug) é um procedimento para diagnóstico e correção de erros já detectados em um programa

Em geral, mais da metade do tempo gasto no desenvolvimento de software é gasto com depuração

A depuração é muito útil pois permite ao programador:

- Monitorar a execução de um programa (passo a passo).
- Ativar pontos de parada (linha, função, condição).
- Monitorar os valores das variáveis.



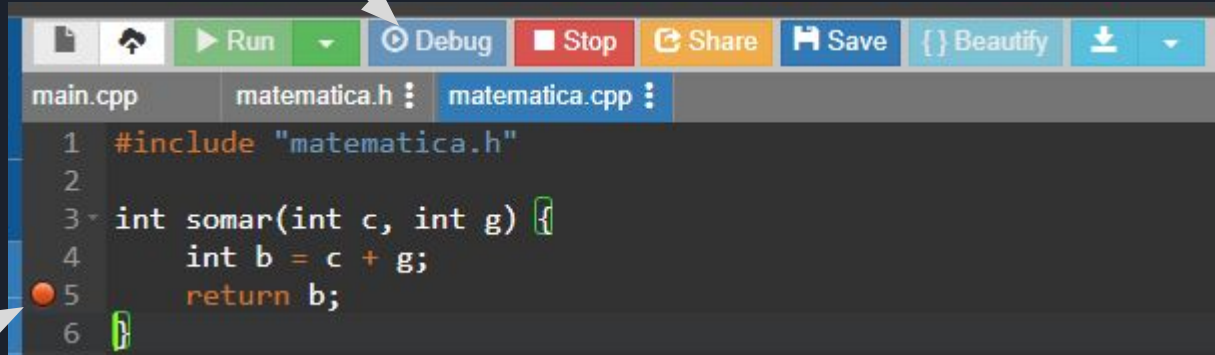
Depuração de Código

Apesar da boa utilidade, a depuração não é sempre a melhor alternativa

Certos tipos de programas não se dão muito bem com ela:

- Sistemas operacionais, sistemas distribuídos, múltiplos threads, sistemas de tempo real
- Não é possível em algumas linguagens e ambientes
- Pode variar muito de um ambiente para outro

Depuração de Código - onlinegdb



The screenshot shows the onlinegdb web interface. At the top, there is a toolbar with buttons for 'Run', 'Debug', 'Stop', 'Share', 'Save', 'Beautify', and a download icon. Below the toolbar, the file tabs show 'main.cpp', 'matematica.h', and 'matematica.cpp'. The code editor displays the following C++ code:

```
1 #include "matematica.h"
2
3 int somar(int c, int g) {
4     int b = c + g;
5     return b;
6 }
```

Two white arrows point to specific elements: one points to the 'Debug' button in the toolbar, and the other points to a red circular breakpoint icon on the left margin of line 5 in the code editor.

Depuração de Código - onlinegdb

The screenshot displays the onlinegdb web interface. The main editor shows a C++ file named `matematica.cpp` with the following code:

```
1 #include "matematica.h"
2
3 int somar(int c, int g) {
4     int b = c + g;
5     return b;
6 }
```

The interface includes a top toolbar with buttons for Run, Debug, Stop, Share, Save, Beautify, and Download. On the right side, there are panels for Call Stack, Local Variables, Display Expressions, and Breakpoints and Watchpoints.

The Breakpoints and Watchpoints panel is highlighted with a red dashed box. It contains a table with the following data:

	#	Description	
<input checked="" type="checkbox"/>	1	in somar(int, int) at matematica.cpp:5	✕

Below the code editor, there is an input field and a Debug Console. The Debug Console shows the output of the program: `Reading symbols from a.out...` and `(gdb)` .

Depuração de Código - onlinegdb

Botão	Descrição	Terminal
Continue	Continua a execução até o próximo breakpoint ou até o fim do programa	continue ou c
Step Over	Executa a próxima linha, mas pula o interior de funções chamadas	next ou c
Step Into	Executa a próxima linha e entra na função se houver chamada de função	step ou s
Step Out	Sai da função atual e retorna para a função chamadora	finish



Depuração de Código - no Terminal

Compilar com a propriedade -g

```
g++ -g main.cpp -o main
```

Executar com o gdb

```
gdb ./main
```

Depuração de Código - no Terminal

Comando	O que faz
<code>break main</code>	Coloca breakpoint na função main
<code>break nome.cpp:linha</code>	Breakpoint em uma linha específica
<code>run</code>	Inicia a execução do programa
<code>next (ou n)</code>	Executa a próxima linha (sem entrar em funções)
<code>step (ou s)</code>	Vai para dentro da próxima função chamada
<code>continue (ou c)</code>	Continua até o próximo breakpoint
<code>print variavel</code>	Mostra o valor de uma variável
<code>list</code>	Mostra o código-fonte ao redor

Depuração de Código - no Terminal

```
❖ jerfff@pc:~/projetos/ufrn/lpI/src$ gdb ./app
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./app...
(gdb) █
```



Executem a calculadora com debug

Use o debug para verificar o funcionamento da calculadora

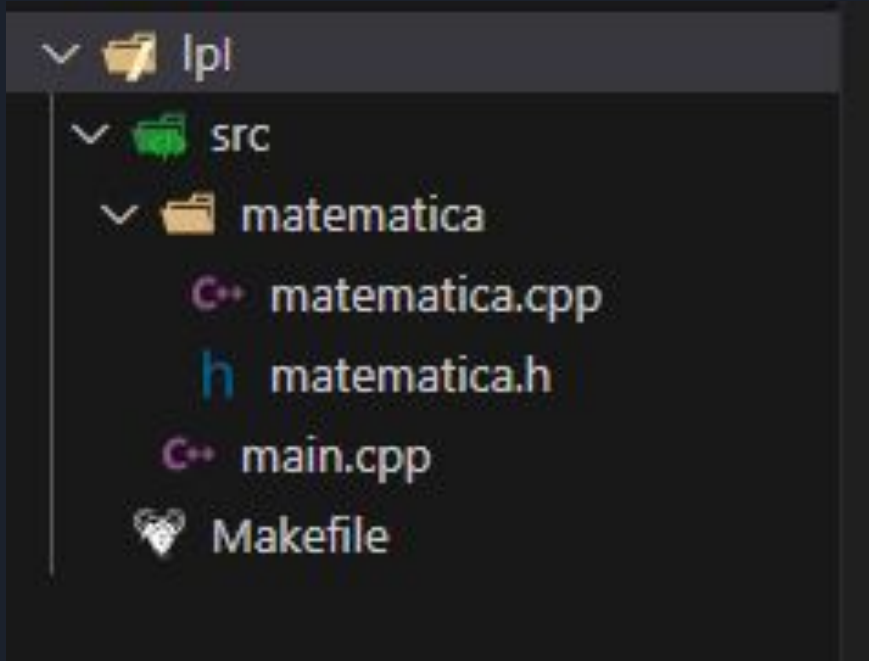


Make

Make é uma ferramenta de automação de compilação usada principalmente em projetos de programação em C e C++.

Ela lê um arquivo chamado Makefile, onde estão definidas as regras para compilar, organizar e construir um programa a partir de vários arquivos fonte.


Estrutura do nosso projeto



src /-> Todo o código fica nessa pasta

build/ -> os .o ficam nessa pasta

Makefile -> sem extensão, apenas o nome



```
TARGET = main
SRC_DIR = src
BUILD_DIR = build

CXX = g++
CXXFLAGS = -Wall -std=c++17 -I$(SRC_DIR)

# Pega todos os .cpp
SRCS = $(shell find $(SRC_DIR) -name '*.cpp')

# Função para trocar src/xxx/yyy.cpp em build/xxx/yyy.o
OBJS = $(patsubst $(SRC_DIR)/%.cpp,$(BUILD_DIR)/%.o,$(SRCS))

# Regra padrão
all: $(TARGET)

# Linka o executável
$(TARGET): $(OBJS)
    $(CXX) $^ -o $@

# Regra para compilar .cpp em .o no build/
# A receita cria a pasta build/ para cada .o, se precisar
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp
    @mkdir -p $(dir $@)
    $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
    rm -rf $(BUILD_DIR) $(TARGET)
```



Como compilar e executar?

Na raiz do nosso projeto:

```
make
```

E depois

```
./main
```

Para limpar os arquivos gerados

```
make clean
```



Struct em C++

Uma struct (estrutura) é um tipo de dado composto que permite agrupar várias variáveis sob um mesmo nome. Essas variáveis podem ser de tipos diferentes.

Organizar dados relacionados de forma lógica. É especialmente útil quando queremos representar entidades do mundo real, como uma pessoa, um carro, um livro etc

Struct em C++

Pessoa é o nome do novo tipo de dado.

nome, idade e altura são os membros da struct.

Cada instância de Pessoa terá essas três variáveis associadas a ela.

```
struct Pessoa {  
    string nome;  
    int idade;  
    float altura;  
};
```



Struct em C++

Em C o struct não pode conter métodos.

```
struct Pessoa {  
    string nome;  
    int idade;  
  
    void apresentar() {  
        cout << "Olá, meu nome é " << nome  
    }  
};
```



```
struct Pessoa {  
    string nome;  
    int idade;  
  
    void apresentar() {  
        cout << "Olá, meu nome é " << nome  
    }  
};  
  
Pessoa p;  
p.nome = "João";  
p.idade = 25;  
p.apresentar(); // ✅ Correto: chamando método
```

Struct em C++

Um construtor é uma função especial dentro da struct que é chamada automaticamente quando um objeto é criado.

Ele serve para inicializar os membros da struct com valores assim que o objeto é criado

```
struct Pessoa {  
    string nome;  
    int idade;  
  
    // Construtor  
    Pessoa(string n, int i) {  
        nome = n;  
        idade = i;  
    }  
}
```

Struct em C++

A **lista de inicialização** é uma forma de inicializar os membros de uma struct ou class no momento da construção do objeto, antes do corpo do construtor.

```
struct Pessoa {  
    string nome;  
    int idade;  
  
    // Lista de inicialização  
    Pessoa(string n, int i) : nome(n), idade(i) {  
        cout << "Construtor executado!" << endl;  
    }  
}
```


Struct em C++

O `this` serve para identificar atributos do nosso struct.

```
struct Pessoa {  
    string nome;  
  
    Pessoa(string nome) {  
        this->nome = nome;  
    }  
}
```

Struct em C++

O **destrutor** é uma função especial que é chamada automaticamente quando um objeto é destruído — ou seja, quando ele sai de escopo ou é deletado.

- Mesmo nome da struct, **precedido por ~ (til)**
- **Não recebe parâmetros**
- **Não tem tipo de retorno** (nem void)
- Só pode haver **um único destrutor**

```
~Pessoa() {  
    // Código de limpeza aqui  
}
```



Struct em C++

Como chamar o Destrutor?

```
Pessoa* p = new Pessoa("Lucas", 30);  
delete p; // Destrutor chamado aqui
```



Exercício

Crie, um programa que gerencia alunos de uma escola:

- Perguntar ao usuário quantos alunos deseja cadastrar (máximo de 10).
- Permitir o cadastro dos alunos, calculando a média final automaticamente (2 notas).
- Exibir todos os alunos cadastrados.
- Permitir buscar um aluno pelo nome e mostrar os dados dele.
- Exibir a média geral da turma.
- Usar struct para organizar os dados do aluno.

Struct + modularização

```
main.cpp Pessoa.h Pessoa.cpp
1  #pragma once
2
3  #include <string>
4  using namespace std;
5
6  struct Pessoa {
7      string nome;
8      int idade;
9
10     Pessoa(string nome, int idade);
11     void apresentar() const;
12 };
```

```
main.cpp Pessoa.h Pessoa.cpp
1  #include "Pessoa.h"
2  #include <iostream>
3
4  using namespace std;
5
6  Pessoa::Pessoa(string nome, int idade) {
7      this->nome = nome;
8      this->idade = idade;
9  }
10
11 void Pessoa::apresentar() const {
12     cout << "Olá, meu nome é " << nome << endl;
13 }
14
15
```

Struct + modularização

É possível implementar tudo em um arquivo .h

Mas não é uma boa prática.

```
1  #pragma once
2
3  #include <string>
4  using namespace std;
5
6  struct Pessoa {
7      string nome;
8      int idade;
9
10     Pessoa(string nome, int idade) {
11         this->nome = nome;
12         this->idade = idade;
13     }
14
15     void apresentar() const {
16         cout << "Olá, meu nome é " << nome
17     }
18 };|
```

Argumentos Padrão (Default Arguments)

Você pode fornecer valores padrão para parâmetros em funções. Isso permite que você chame a função com menos argumentos do que os declarados.

```
void saudacao(string nome = "Usuário", string saud = "Olá") {  
    cout << saud << ", " << nome << "!" << endl;  
}  
  
int main() {  
    saudacao(); // Olá, Usuário!  
    saudacao("Maria"); // Olá, Maria!  
    saudacao("Carlos", "Bem-vindo"); // Bem-vindo, Carlos!  
    return 0;  
}
```



Sobrecarga de métodos

A sobrecarga de funções permite que você defina várias funções com o mesmo nome, mas com diferentes listas de parâmetros (tipo, número ou ordem dos parâmetros).

O compilador escolhe automaticamente qual versão da função usar com base nos argumentos fornecidos na chamada.



Exemplo

Existe um método
`imprimir` para cada
tipo

```
void imprimir(int x) {  
    cout << "Inteiro: " << x << endl;  
}  
  
void imprimir(double x) {  
    cout << "Double: " << x << endl;  
}  
  
void imprimir(string x) {  
    cout << "String: " << x << endl;  
}  
  
int main() {  
    imprimir(10);  
    imprimir(3.14);  
    imprimir("Olá");  
    return 0;  
}
```



Cuidado com argumento padrão e Sobrecarga

Qual função exemplo será executada?

```
void exemplo(int x = 0) {  
    cout << "Função com int: " << x << endl;  
}  
  
void exemplo() {  
    cout << "Função sem parâmetros" << endl;  
}  
  
int main() {  
    exemplo(); // ✗ Erro: chamada ambígua  
    return 0;  
}
```

Exemplo com argumento padrão

É possível definir um valor padrão para métodos com sobrecarga

```
void mensagem(string texto, int repeticoes = 1) {  
    for (int i = 0; i < repeticoes; ++i)  
        cout << texto << endl;  
}  
  
void mensagem() {  
    cout << "Mensagem padrão!" << endl;  
}  
  
int main() {  
    mensagem("Oi", 3);  
    mensagem("Olá");    // Usa repeticoes = 1  
    mensagem();          // Chama versão sem parâmetros  
    return 0;  
}
```

Quando usar cada um?

Argumento padrão:

- Diferença é pequena na lógica
- Evitar duplicação de código
- Número de combinações de argumentos é pequena

```
void mensagem(string texto, int repeticoes = 1) {  
    for (int i = 0; i < repeticoes; ++i)  
        cout << texto << endl;  
}  
  
void mensagem() {  
    cout << "Mensagem padrão!" << endl;  
}  
  
int main() {  
    mensagem("Oi", 3);  
    mensagem("Olá"); // Usa repeticoes = 1  
    mensagem();      // Chama versão sem parâmetros  
    return 0;  
}
```

Quando usar cada um?

Sobrecarga

- Lógica é muito diferente
- Os tipos dos parâmetros variam, não a quantidade

```
void mensagem(string texto, int repeticoes = 1) {  
    for (int i = 0; i < repeticoes; ++i)  
        cout << texto << endl;  
}  
  
void mensagem() {  
    cout << "Mensagem padrão!" << endl;  
}  
  
int main() {  
    mensagem("Oi", 3);  
    mensagem("Olá"); // Usa repeticoes = 1  
    mensagem();      // Chama versão sem parâmetros  
    return 0;  
}
```



Vantagens da Sobrecarga de Métodos

- Código mais legível e intuitivo
- Reutilização de lógica com pequenas variações

Cuidados

- Ambiguidade (ex: chamadas que podem se encaixar em mais de uma versão)
- Sobrecargas excessivas que dificultam entender qual função será chamada

Exercício

Crie um sistema de pedidos, onde é possível adicionar vários tipos de itens ao pedido.

- Usar Struct para o Pedido e Itens;
- Item (nome, quantidade e valor unitários);
- Calcular o total do pedido;
- Criar um struct chamado `impressora` que tenha 2 métodos imprimir: Um imprime um Pedido (Com todos os seus itens) o outro imprime apenas um item;
- No método imprimir, adicionar um atributo opcional que define se manda a impressa imprimir em paisagem ou Retrato;

Tratamento de Exceções



Tratamento de Exceções

Tratamento de exceções é uma forma de lidar com erros de execução que podem ocorrer em um programa, como divisão por zero, acesso fora dos limites de um vetor, falha ao abrir um arquivo, entre outros.



O que é uma exceção?

Uma exceção é um mecanismo para lidar com erros de forma controlada, separando a lógica normal do tratamento de erro.

No C++, uma exceção é lançada com **throw** e capturada com **catch**.

Código

```
vector<int> numeros = {1, 2, 3};

try {
    // índice inválido
    cout << "Número: " << numeros.at(5) << endl;
}
catch (const out_of_range& e) {
    cout << "Exceção capturada: " << e.what() << endl;
}
```



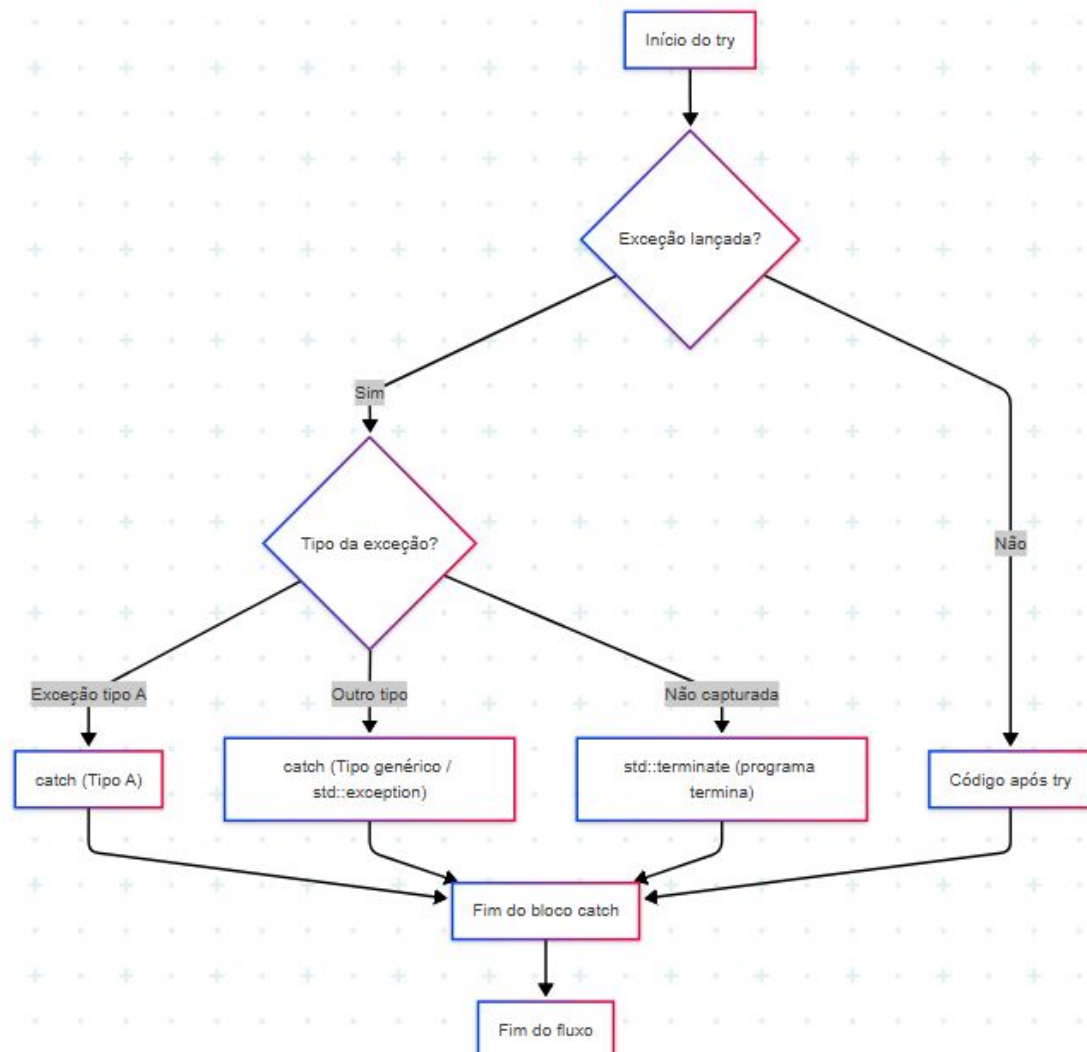
Tratamento de Exceções

try: onde o código que pode causar um erro é colocado.

catch: onde o erro é tratado.

throw: usado para lançar uma exceção.

.what(): retorna a mensagem do erro.



Destrutores e Exceções

Quando uma exceção é lançada, os destrutores dos objetos são chamados.

```
struct Pessoa {  
    ~Pessoa() {  
        cout << "Destrutor chamado";  
    }  
};  
  
int main() {  
    try {  
        Pessoa p;  
        throw invalid_argument("aaaaaaaaa");  
    } catch (const exception& e) {  
        cout << "Mensagem do erro: " << e.what() << endl;  
    }  
    return 0;  
}
```

Exercício

Permitir que o usuário digite 5 números inteiros e armazene em um `std::vector<int>`.

Em seguida, permitir o usuário digitar um índice para acessar um dos elementos.

Use `vector.at(indice)` para acessar o elemento e imprimir seu valor.

Use o bloco try-catch para capturar a exceção caso o índice esteja fora dos limites.

Como lançar exceção

O C++ permite você lançar exceções manualmente:

```
throw valor_ou_objeto("Mensagem de erro");
```

```
if (b == 0) {  
    throw invalid_argument("Divisão por zero.");  
}
```


Principais Exceções

1. Permitir que o usuário digite 5 números inteiros e armazene em um `std::vector<int>`.
2. Em seguida, permitir o usuário digitar um índice para acessar um dos elementos.
3. Se o usuário informar um valor maior do que o índice que existe, lançar a exceção: `out_of_range`
4. Use o bloco try-catch para capturar a exceção caso o índice esteja fora dos limites.

Principais Exceções

Exceção	Descrição
exception	Base de todas as exceções
runtime_error	Erros em tempo de execução
logic_error	Erros de lógica (pré-condições)
invalid_argument	Argumento inválido
out_of_range	Índice fora dos limites
overflow_error	Overflow numérico
underflow_error	Underflow numérico
bad_alloc	Falha na alocação de memória
bad_cast	Falha no cast dinâmico

Vários catch

Você pode ter vários blocos catch para tratar diferentes tipos de exceções:

```
try {  
    // código  
}  
catch (const out_of_range& e) {  
    cout << "Erro de acesso: " << e.what();  
}  
catch (const runtime_error& e) {  
    cout << "Erro em tempo de execução: " << e.what();  
}  
catch (...) {  
    cout << "Exceção desconhecida!";  
}
```

Exercício 01

Permitir ao usuário digitar uma quantidade infinita de números inteiros positivos até que o usuário digite -1.

- Se o valor for negativo e diferente de -1, lance uma exceção `invalid_argument`.
- Se for maior que 10000, lance uma exceção `overflow_error`.
- Ignorar os valores inválidos.
- Armazenar apenas os válidos em um vetor.

Ao final, exiba:

- A quantidade de valores válidos.
- A média dos valores válidos.
- O maior e o menor valor digitado.

Re-lançar uma exceção

Quando você captura uma exceção em um catch, pode decidir fazer algum tratamento parcial (como logar uma mensagem) e depois relançar a mesma exceção para ser tratada em outro lugar.

Se a exceção não for relançada, o `main` não vai saber que houve erro.

```
void funcaoIntermediaria() {
    try {
        throw invalid_argument("Erro na função intermediária");
    } catch (const exception& e) {
        cout << "Tratamento parcial: " << e.what() << endl;
        throw; // Re-lança a mesma exceção
    }
}

int main() {
    try {
        funcaoIntermediaria();
    } catch (const exception& e) {
        cout << "Tratamento final no main: " << e.what() << endl;
    }
    return 0;
}
```

Exercício 02

- Crie um cadastro de produtos nome, preço e quantidade.
- Criar um método `adicionarProduto`, nesse método verificar se:
- A cada cadastro, o programa deve:
- Verificar se:
 - O nome está vazio → lançar `invalid_argument`.
 - O preço é negativo → lançar `invalid_argument`.
 - A quantidade é negativa → lançar `out_of_range`.
 - Continuar pedindo o valor até ele ser válido;
- Continuar pedindo os dados do próximo produto até o usuário digitar o nome "FIM".



namespace personalizados

Um `namespace` é um espaço de nomes que serve para organizar código e evitar conflitos de nomes entre variáveis, funções, classes, etc.

Em projetos grandes, pode ter várias funções ou classes com nomes iguais e o namespace evita que esses nomes "colidam".

Você “coloca” essas funções dentro de um nome, para diferenciá-las.

Usando um namespace

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

int main() {

    int a = 10;

    cout << a;

    return 0;
}
```


```
#include <iostream>

int main() {

    int a = 10;

    std::cout << a;

    return 0;
}
```

E quando eu não defino um namespace e não uso ::

Todas as variáveis, funções e classes ficam “soltas” no escopo global.

Se houver 2 funções com mesmo nome, haverá conflitos de nome.

```
int soma(int a, int b) {  
    return a + b;  
}  
  
int soma(int a, int b) {  
    return a * b;  
}  
  
int main() {  
    cout << soma(2, 3); // Erro  
}
```

E quando eu não defino um namespace e não uso ::

Todas as variáveis, funções e classes ficam “soltas” no escopo global.

Se houver 2 funções com mesmo nome, haverá conflitos de nome.

```
namespace matematica {  
    int soma(int a, int b) {  
        return a + b;  
    }  
}  
  
int soma(int a, int b) {  
    return a * b;  
}  
  
int main() {  
    cout << soma(2, 3); // Retorna 6  
}
```

E quando eu não defino um namespace e não uso ::

Todas as variáveis, funções e classes ficam “soltas” no escopo global.

Se houver 2 funções com mesmo nome, haverá conflitos de nome.

```
namespace matematica {  
    int soma(int a, int b) {  
        return a + b;  
    }  
}  
  
int soma(int a, int b) {  
    return a * b;  
}  
  
int main() {  
    cout << matematica::soma(2, 3); // Retorna 6  
}
```

Se se eu modularizar?

matematica.h

```
#pragma once

namespace matematica {
    int soma(int a, int b);
}
```

matematica.cpp

```
#include "matematica.h"

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

Se se eu modularizar?

main.cpp

```
#include <iostream>
#include "matematica.h"

using namespace std;

int main() {
    cout << matematica::soma(2, 3);
}
```



Se se eu modularizar?

main.cpp

```
using namespace std;  
using namespace matematica;  
  
int main() {  
    cout << soma(2, 3);  
}
```

namespace com struct

matematica.h

```
namespace matematica {  
  
    struct Calculadora {  
  
        void soma(int a, int b);  
  
    };  
  
}
```

matematica.cpp

```
namespace matematica {  
  
    void Calculadora::soma(int a, int b){  
        cout << a + b;  
    }  
  
}
```

Exercício - Sistema de cadastro de funcionário.

Os dados devem ser armazenados em uma `struct` chamada `Funcionario(nome, idade e salario)`, que estará dentro de um `namespace` chamado `empresa`.

O sistema deve permitir que o usuário cadastre vários funcionários, mas apenas os maiores de idade ($idade \geq 18$) podem ser registrados.

Em um `namespace` `cadastrarFuncionario` criar uma função que recebe os dados do funcionário, verifica a idade (se inválida) lançar a exceção `invalid_argument`.

Use um loop para continuar cadastrando até o usuário digitar "n" para sair.

Permitir exibir todos os funcionários.



Introdução à STL (Standard Template Library)

Ela oferece um conjunto de classes e algoritmos genéricos prontos para uso, otimizando o tempo de desenvolvimento e aumentando a segurança e legibilidade do código.

vector: Vetor dinâmico (como um array que cresce).

list: Lista duplamente encadeada.

deque: Fila dupla (inserções no início e fim).

stack: Pilha (LIFO).

queue: Fila (FIFO).



list no C++

```
#include <list>
```

A `std::list` é uma lista duplamente encadeada.

Cada elemento da lista aponta para o anterior e o próximo.

Ao contrário do `std::vector`, os elementos não estão em posições contíguas na memória.

Permite inserções e remoções rápidas em qualquer parte da lista, especialmente no início e no meio.

`list` não permite acesso por índice igual a `vector: numero[i]`



list ou vector

Use `list` quando:

- Você precisa inserir ou remover elementos no meio ou no início com frequência.
- Não precisa de acesso por índice (como `v[2]`), que a `list` não oferece.

Use `vector` quando:

- Você precisa de acesso aleatório rápido.
- O foco é em performance de leitura linear ou ordenação



list no C++

```
#include <list>
```

```
list<int> numeros;
```

```
numeros.push_back(10); // Insere no final
```

```
numeros.push_front(5); // Insere no início
```

```
// Remoções
```

```
numeros.pop_back(); // Remove do final
```

```
numeros.pop_front(); // Remove do início
```



list no C++

```
#include <list>
```

```
list<int> numeros;
```

```
numeros.remove(10); // Remove o elemento com valor 10
```

```
numeros.sort(); // Ordena do menor para o maior
```

```
numeros.reverse(); // Reverte a ordem dos elementos
```

list no C++

Como `list` não permite acesso por índice direto, você precisa usar `advance` para mover o iterador até a posição desejada.

```
#include <iterator>
```


```
list<int> numeros = {10, 20, 30, 40};
```

```
auto it = numeros.begin();  
advance(it, 2);
```

```
numeros.insert(it, 25); // Insere 25 antes de 30
```

deque no C++

```
deque<int> d;  
  
// Inserções  
d.push_back(10); // Insere no fim  
d.push_front(5); // Insere no início  
  
// Acesso por índice  
cout << d[0] << " " << d[1] << endl;  
  
// Remoções  
d.pop_back(); // Remove do fim  
d.pop_front(); // Remove do início
```



```
list<Pessoa> pessoas;
```

```
Pessoa ana;  
ana.nome = "Ana";  
ana.idade = 25;
```

```
Pessoa carlos;  
carlos.nome = "Carlos";  
carlos.idade = 230;
```

```
Pessoa carlos2;  
carlos2.nome = "Carlos";  
carlos2.idade = 23;
```

```
// Adicionando pessoas  
pessoas.push_back(ana);  
pessoas.push_back(carlos);  
pessoas.push_back(carlos2);
```

```
auto it = find(pessoas.begin(), pessoas.end(), carlos);
```

```
if (it != pessoas.end()) {  
    cout << "Pessoa encontrada: " << it->nome << ", " << it->idade << " anos\n";  
} else {  
    cout << "Pessoa não encontrada\n";  
}
```

```
pessoas.remove(carlos);
```


Sobrescrever o operador ==

```
struct Pessoa {  
    std::string nome;  
    int idade;  
  
    // Sobrescrevendo o operador ==  
    bool operator==(const Pessoa& outra) const {  
        return nome == outra.nome && idade == outra.idade;  
    }  
};
```



Exercício 02

Adicionar o cpf no funcionário do exercício anterior.

Permite que o usuário remova um funcionário pelo CPF.

Sobrecarregue o operador ==



Introdução à STL (Standard Template Library)

Ela também fornece funções para trabalhar com essas estruturas:

- `sort()`
- `reverse()`
- `find()`
- `count()`
- `binary_search()`
- `copy()`
- `for_each()`



Programação Orientada a Objetos

Um programa é visto como um conjunto de objetos que se comunicam.

POO é o paradigma mais popular entre os devs, pois esse paradigma aproxima o mundo real do mundo virtual.



Abstração

- Ação de abstrair, de analisar isoladamente um aspecto, contido num todo, sem ter em consideração sua relação com a realidade.
- Não levar algo em consideração.



Classes, Objetos e atributos

Classe:

- Uma classe é um modelo ou um "template" para criar objetos. Ela define as propriedades e comportamentos que os objetos de determinado tipo terão.
- Em termos mais simples, você pode pensar em uma classe como um esboço para um objeto. Ela contém definições de métodos (funções) e atributos (variáveis) que descrevem o comportamento e as características dos objetos.

Objeto:

- Um objeto é uma instância específica de uma classe.
- Quando você cria um objeto, está criando uma cópia da classe, com seus próprios valores atribuídos aos atributos.
- Por exemplo, se você tem uma classe chamada "Carro", um objeto dessa classe seria um carro específico, como um Ford Fiesta ou um Toyota Corolla.

Atributo (estados):

- Os atributos são as características dos objetos, ou seja, as variáveis que pertencem a um objeto específico.
- Eles representam o estado do objeto e podem ter diferentes valores para cada instância da classe.
- Por exemplo, para a classe "Carro", os atributos poderiam ser "cor", "modelo", "ano", etc.

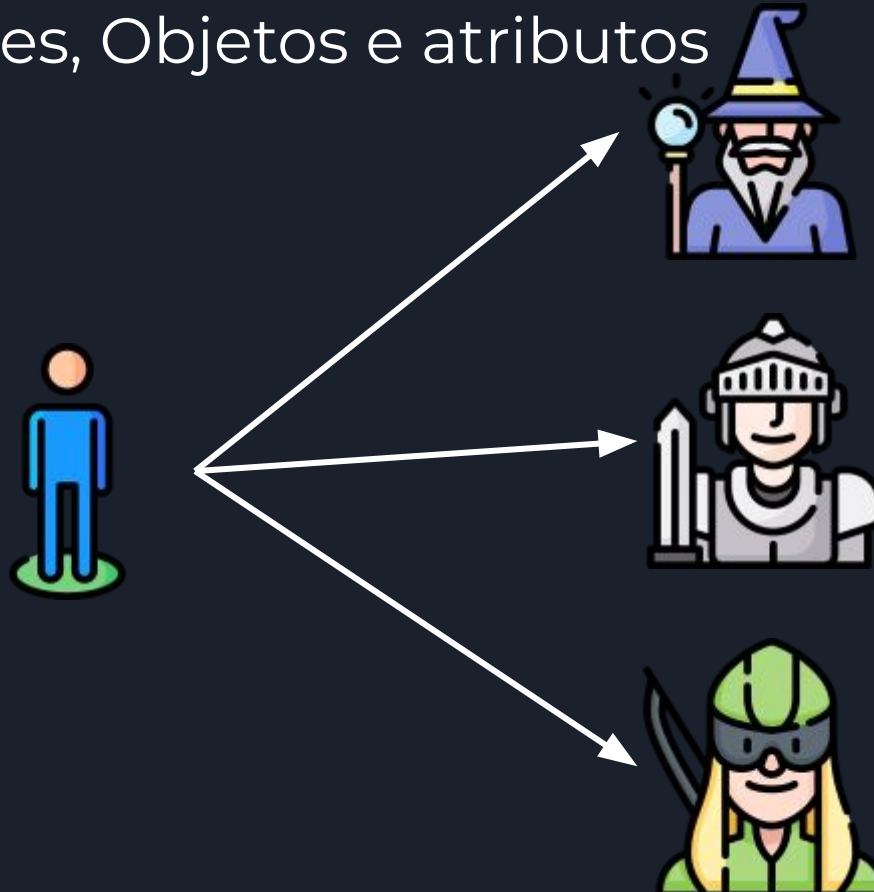
Classes, Objetos e atributos



Classes, Objetos e atributos



Classes, Objetos e atributos





Classes

Uma classe é um molde (modelo) para criar objetos. Ela agrupa dados (atributos) e funções (métodos) que trabalham juntos.

É o conceito central da programação orientada a objetos.

```
class Pessoa {  
  
};
```



Classes

Uma classe é um molde (modelo) para criar objetos. Ela agrupa dados (atributos) e funções (métodos) que trabalham juntos.

É o conceito central da programação orientada a objetos.

```
class Pessoa {  
    string nome;  
    int idade;  
  
};
```



O que acontece?

Implemente e verifiquem o que acontece?

```
class Pessoa {  
    string nome;  
    int idade;  
  
};  
  
int main()  
{  
    Pessoa p;  
    p.nome = "Jerfff";  
  
    cout << p.nome;  
  
    return 0;  
}
```



O que acontece?

Por padrão, todos os atributos de uma classe são `private`

```
class Pessoa {  
    string nome;  
    int idade;  
  
};  
  
int main()  
{  
    Pessoa p;  
    p.nome = "Jerfff";  
  
    cout << p.nome;  
  
    return 0;  
}
```



Modificadores de Acesso

Controlam **quem pode acessar** uma classe, método ou atributo.

Modificador	Visível para...
public	Todas as classes em qualquer pacote.
protected	Classes do mesmo pacote ou subclasses, mesmo que estejam em outros pacotes.
private (default)	Somente dentro da própria classe.



O que acontece?

Por padrão, todos os atributos de uma classe são `private`

```
class Pessoa {  
    public:  
        string nome;  
        int idade;  
};  
  
int main()  
{  
    Pessoa p;  
    p.nome = "Jerfff";  
  
    cout << p.nome;  
  
    return 0;  
}
```



O que acontece?

Por padrão, todos os atributos de uma classe são `private`

```
class Pessoa {  
    public:  
        string nome;  
        int idade;  
};  
  
int main()  
{  
    Pessoa p;  
    p.nome = "Jerfff";  
  
    cout << p.nome;  
  
    return 0;  
}
```




Classes

```
class Pessoa {  
    private:  
        // Atributos privados (encapsulados)  
        // Métodos privador  
  
    protected:  
        // Atributos privados (encapsulados)  
        // Métodos privador  
  
    public:  
        // Atributos privados (encapsulados)  
        // Construtor  
        // Destrutor  
        // Métodos públicos  
  
};
```



Mas não parece um struct?

Sim!

Struct e Classes são bem parecidas mas diferem quando falamos de acesso aos seus elementos.

Característica	Struct	Class
Modificador de acesso padrão	public	private
Herança Padrão	public	private

Como modularizar?

Pessoa.h

```
#pragma once

#include <iostream>
using namespace std;

class Pessoa {

    public:
        string nome;
        int idade;

        void apresentacao();

};
```

Pessoa.cpp

```
#include "Pessoa.h"

void Pessoa::apresentacao() {
    cout << nome << ": " << idade;
}
```

Exercício - 1p extra na prova (prazo: 22/06 23:59:59)

Gere uma aplicação para gerenciar os pedidos de um restaurante.

O restaurante tem um cardápio com vários itens predefinidos com nome e valor.

O restaurante tem 3 mesas com clientes e os clientes podem realizar vários pedidos.

O usuário do sistema recebe o pedido em mãos e atualiza o sistema.

Ele pode listar as mesas e mostrando se esta aberta e os itens pedidos (com o total vendido até o momento).

Ele pode lançar um pedido para a mesa (Passando o ID do prato solicitado).

Quando os clientes vão embora, deve ser possível puxar os itens pedidos e encerrar a conta da mesa, não sendo possível reabrir a mesa.

Apenas quando as 3 mesas estiverem fechadas, o usuário do sistema pode puxar o resultado das vendas, informando quantos itens do menu foram vendidos, o preço unitário e o valor total e ao final o total vendido.

Dividir as classes em arquivos .h e .cpp

Utilizar o makefile e estrutura padrão do projeto como mostrado em aulas anteriores.

Construtor e Destrutor



Construtor

- É um método especial chamado automaticamente quando um objeto é criado.
- Serve para inicializar atributos ou realizar preparações iniciais.
- Tem o mesmo nome da classe e não tem tipo de retorno (nem void).



Como declarar um Construtor

```
class Pessoa {  
  
    public:  
        string nome;  
        int idade;  
  
        Pessoa(string nome, int idade);  
  
};
```

```
Pessoa::Pessoa(string nome, int idade) {  
    this->nome = nome;  
    this->idade = idade;  
}
```

Sobrecarga e Parâmetros opcionais

```
int main()
{
    Pessoa p;
    Pessoa p2("Jerfff");
    Pessoa p3("Alexandre", 55);

    p.apresentacao(); // Ninguém: 0
    p2.apresentacao(); // Jerfff: 0
    p3.apresentacao(); // Alexandre: 55

    return 0;
}
```

```
class Pessoa {

    public:
        string nome;
        int idade;

        Pessoa();
        Pessoa(string nome, int idade=0);
        void apresentacao();

};
```




Construtor Padrão

É um construtor sem parâmetros (ou com todos os parâmetros com valor padrão). Ele é chamado automaticamente:

- Quando você instancia um objeto sem argumentos
- Quando um array de objetos é criado
- Em várias operações internas da linguagem (ex: inicialização de containers STL)

Construtor Padrão - Quando você instancia um objeto sem argumentos

```
class Pessoa {  
  
    public:  
        string nome;  
        int idade;  
  
        Pessoa();  
  
};
```

```
class Pessoa {  
  
    public:  
        string nome;  
        int idade;  
  
        Pessoa(string nome = "Fulano", int idade = 0);  
  
};
```

Construtor Padrão - Quando você instancia um objeto sem argumentos

Se não houver construtor padrão, não é possível criar um objeto sem informações.

É necessário criar um construtor padrão

```
int main()
{
    Pessoa p; ❌
    return 0;
}
```

```
class Pessoa {
    public:
        string nome;
        int idade;

        Pessoa(string nome, int idade);
};
```

Construtor Padrão - Quando ele é exigido

- Vetores de objetos.
- Classes que contêm objetos de outras classes.

```
Pessoa grupo[10]; // Requer construtor padrão
```

```
class Cliente {  
    Pessoa pessoa; // Pessoa precisa de construtor padrão  
};
```

UML



UML (Unified Modeling Language)

É uma linguagem visual padrão para modelar sistemas de software. Ela usa diagramas para representar:

- Estrutura (como as classes, objetos, relações)
- Comportamento (como processos, estados, interações)



Para que serve

- Visualizar a arquitetura de um sistema antes de programar.
- Documentar o funcionamento e a estrutura do sistema.
- Planejar funcionalidades, requisitos e fluxos de dados.
- Identificar erros ou melhorias durante o design.



Os principais diagramas da UML

- Diagramas Estruturais - Estrutura estática do sistema
 - Diagrama de Classes *
 - Diagrama de Objetos
 - Diagrama de Componentes
 - Diagrama de Pacotes
- Diagramas Comportamentais - Comportamento dinâmico do sistema
 - Diagrama de Casos de Uso
 - Diagrama de Sequência
 - Diagrama de Atividades

Diagrama de Classes

É um tipo de diagrama estrutural da UML que representa:

- As classes do sistema
- Seus atributos
(variáveis/estado)
- Seus métodos
(operações/comportamentos)
- E os relacionamentos entre as classes

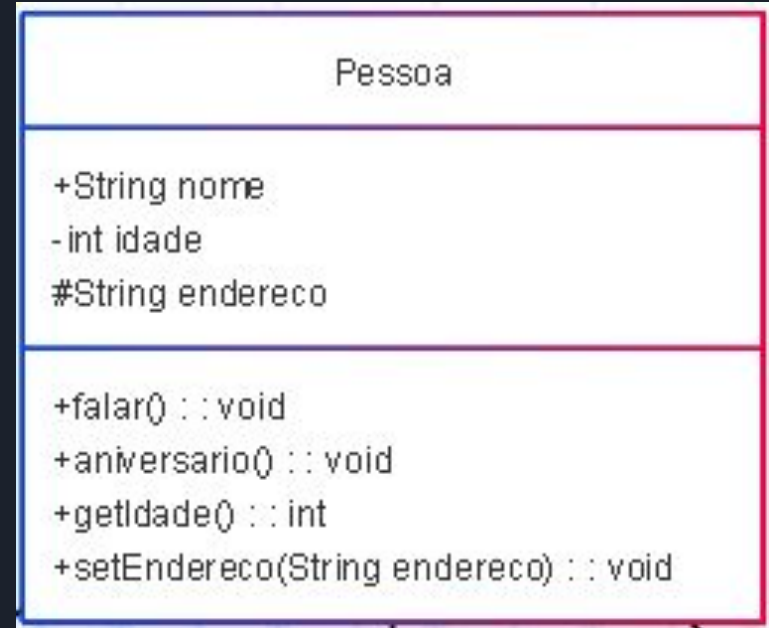


Diagrama de Classes - Modificadores de acesso

- + público
- - Privado
- # protegido

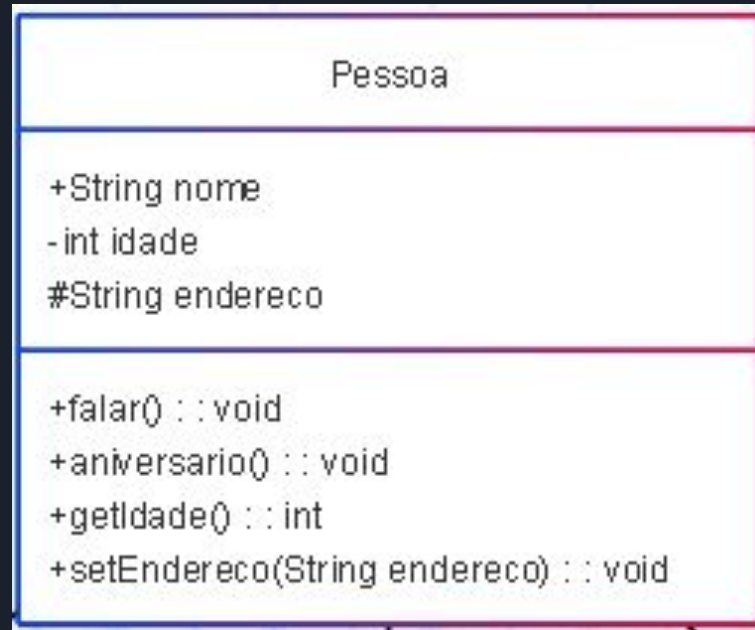


Diagrama de Classes - Retorno dos métodos

- nomeMetodo() :: tipoRetorno
 - void - Sem retorno
 - int - Retorna int

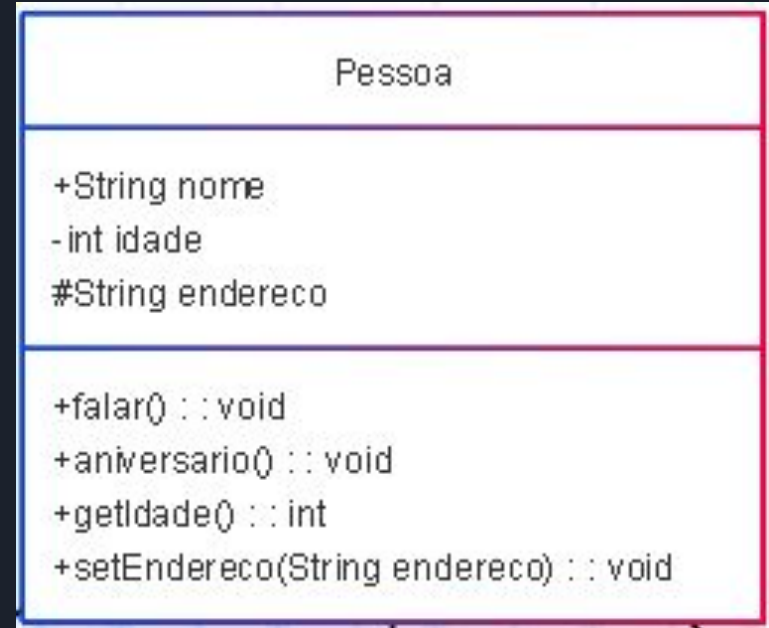
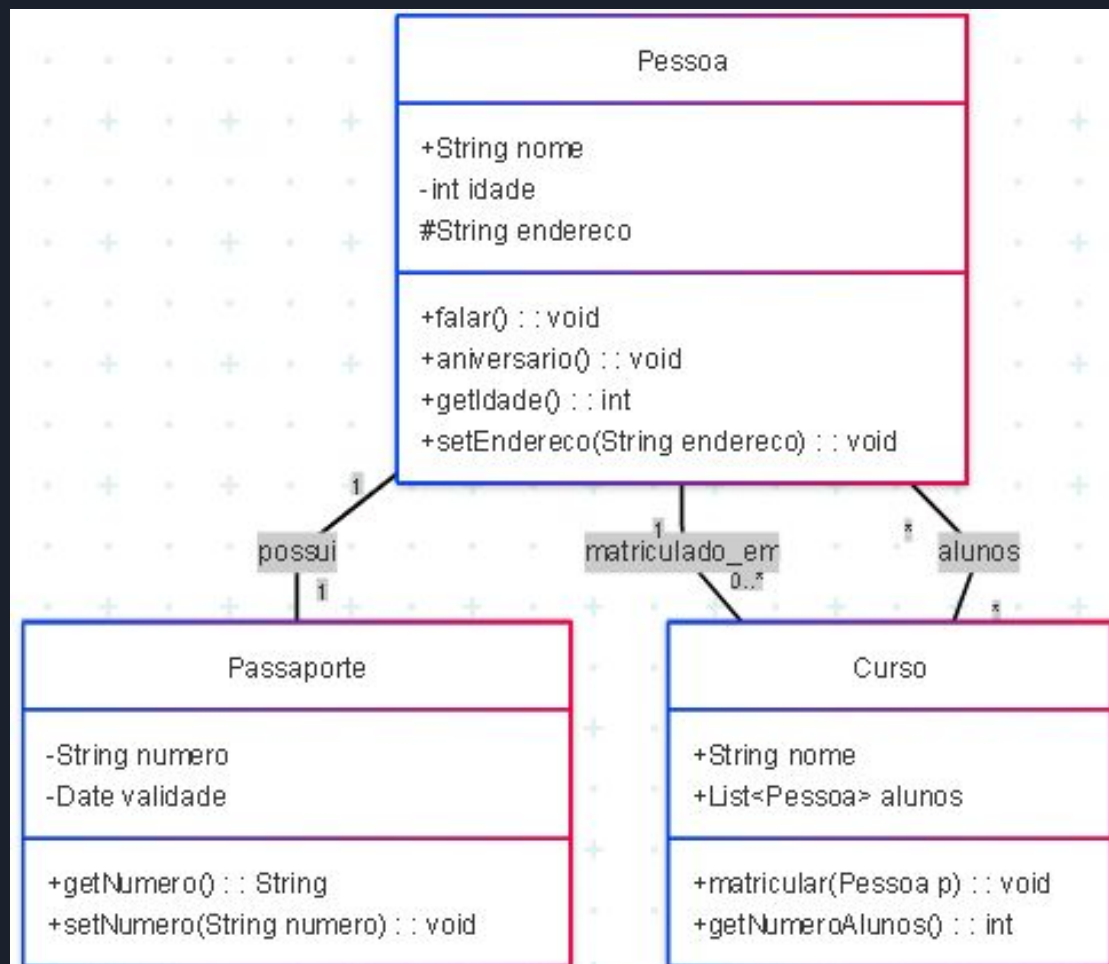




Diagrama de Classes - Relacionamentos

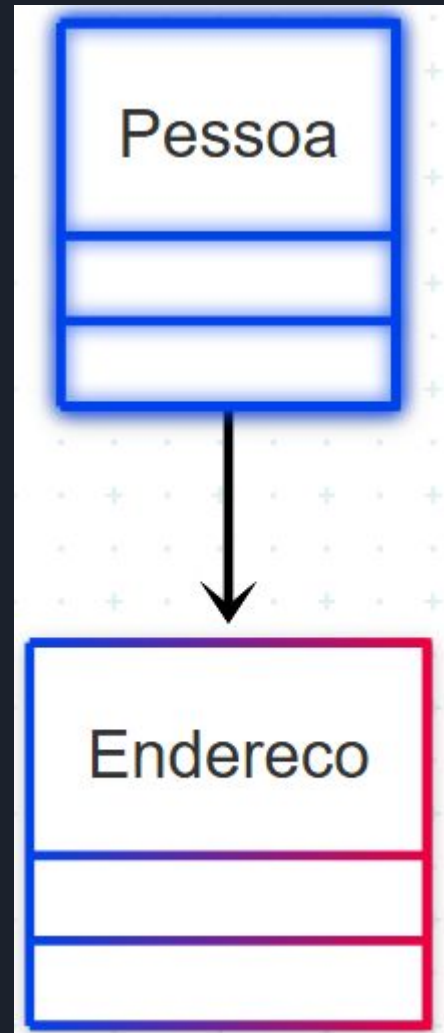
- 1:1 - Cada objeto de uma classe está associado a exatamente um objeto da outra classe, e vice-versa.
- 1:n - Um objeto de uma classe está associado a muitos objetos da outra classe, mas cada objeto da segunda classe está associado a apenas um da primeira.
- n:m - Um objeto de uma classe está associado a muitos objetos da outra classe, e vice-versa.



Mermaid - 1:1

No Mermaid para representar relacionamentos 1:1 basta usar a sintaxe:

```
Pessoa --> Endereco
```



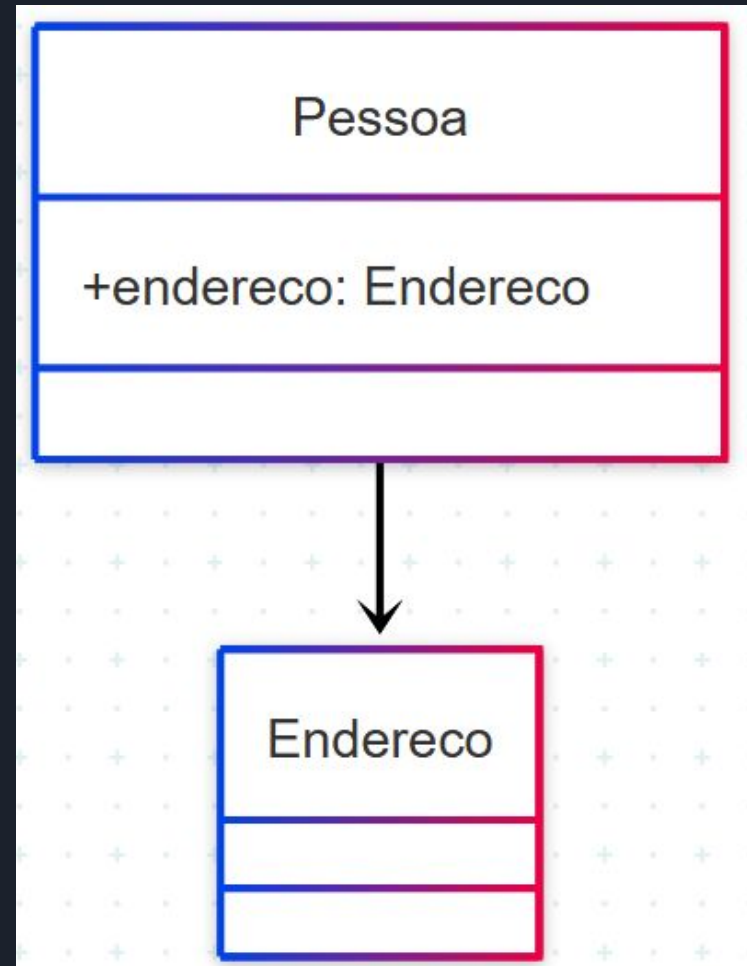
Mermaid - 1:1

No Mermaid para representar relacionamentos 1:1 basta usar a sintaxe:

```
Pessoa --> Endereco
```

Quando é indicada a associação, não é necessário adicionar o atributo na classe, pois a associação já indica a existência desse atributo.

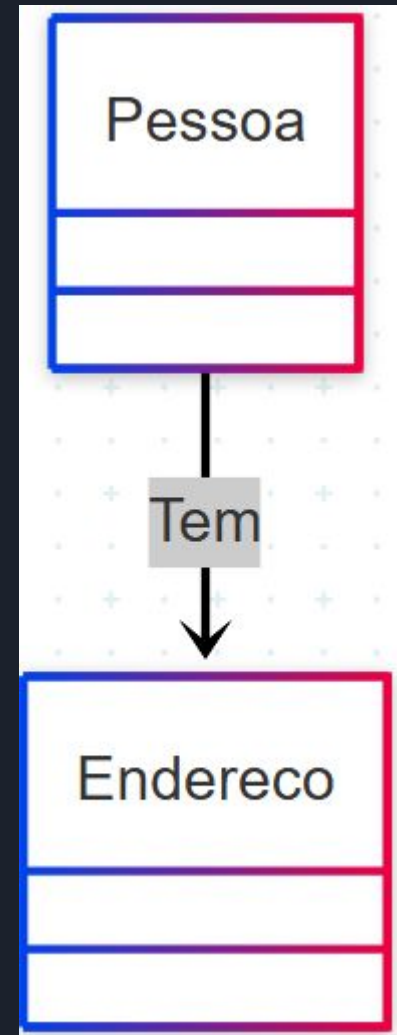
Mas eu gosto de adicionar.



Mermaid - 1:1

No Mermaid para representar relacionamentos 1:1 basta usar a sintaxe:

```
Pessoa --> Endereco : Tem
```





cardinalidade

Esses valores indicam quantas instâncias de uma classe podem estar associadas a instâncias de outra classe em um relacionamento.

Cardinalidade	Significado
0	Não é obrigatório
1	Tem que haver obrigatoriamente 1
*	Vários



cardinalidade

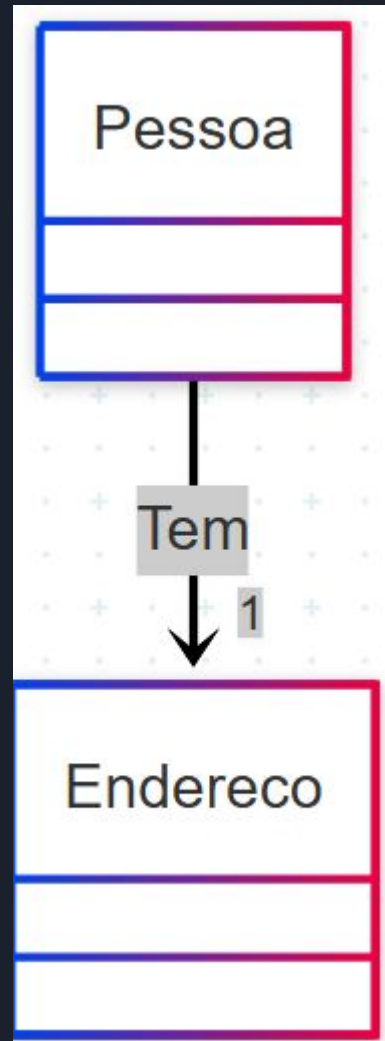
Esses valores indicam quantas instâncias de uma classe podem estar associadas a instâncias de outra classe em um relacionamento.

Cardinalidade	Significado
0..1	Não é obrigatório, mas tem no máximo 1 instância
1..1	Tem que haver obrigatoriamente 1 instância
0..*	Nenhum ou Várias instâncias (Ou seja, uma lista de objetos)
1.. *	uma ou Várias instâncias (Ou seja, uma lista de objetos)
x.. y	Intervalo específico (2..5; 5..10; 6..8)

Mermaid - 1:1

No Mermaid para representar relacionamentos 1:1 basta usar a sintaxe:

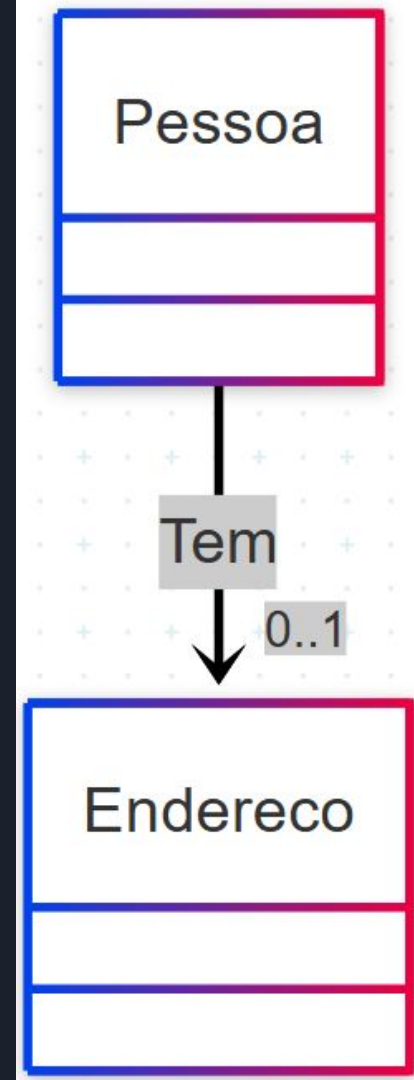
`Pessoa → "1" Endereco : Tem`



Mermaid - 1:n

No Mermaid para representar relacionamentos 0:1 basta usar a sintaxe:

Pessoa → "0..1" Endereco : Tem





Atividade - Sistema de Gerenciamento de Contas Bancárias

O sistema deve representar contas bancárias com dados públicos e fornecer operações que sigam regras de negócio realistas.

Criar a classe `Titular` (nome e CPF), `ContaBancaria` (`Titular`, `numeroConta`, `saldo` e `status`). Ao abrir uma conta (instanciar), o usuário deve informar qual o valor que quer iniciar a conta (> 1). Caso seja inserido um valor ≤ 0 lançar uma exceção `invalid_argument`.

Criar uma classe chamada `SistemaFinanceiro` que deve haver os métodos de depósito, saque e transferência. Além disso, deve ser permitido puxar o saldo e extrato da conta, as informações das operações.

No fim, permitir que o usuário possa encerrar a conta apenas quando estiver zerada.



Implementem o diagrama de classes usando o Mermaid do seguinte sistema

O sistema deve representar contas bancárias com dados públicos e fornecer operações que sigam regras de negócio realistas.

Criar a classe `Titular` (nome e CPF), `ContaBancaria` (`Titular`, `numeroConta`, `saldo` e `status`). Ai abrir uma conta (instanciar), o usuário deve informar qual o valor que quer iniciar a conta (> 1).

Caso seja inserido um valor ≤ 0 lançar uma exceção `invalid_argument`.

Criar uma classe chamada `SistemaFinanceiro` que deve haver os métodos de `deposito`, `saque` e `transferencia`. Além disso, deve ser permitido puxar o saldo e extrato da conta, as informações das operações.

No fim, permitir que o usuário possa encerrar a conta apenas quando estiver zerada.

Exibinda data e Hora

```
#include <iostream>
#include <ctime>
#include <iomanip>

using namespace std;

int main() {
    time_t agora = time(nullptr); // obtém o tempo atual
    tm* tempoLocal = localtime(&agora); // converte para data/hora local

    cout << "Data atual: ";
    cout << put_time(tempoLocal, "%d/%m/%Y %H:%M:%S") << endl;

    return 0;
}
```

Encapsulamento



Encapsulamento

Encapsulamento é o princípio de esconder os detalhes internos de uma classe, expondo apenas o necessário por meio de interfaces públicas (geralmente métodos chamados getters e setters).

Isso ajuda a proteger os dados e a manter o controle sobre como eles são acessados ou modificados.



Por que usar?

1. **Protege os dados:** A principal razão do encapsulamento é impedir o acesso direto a dados sensíveis ou internos de uma classe.
2. **Isola mudanças internas:** Se você mudar a implementação interna de uma classe (por exemplo, trocar `double` saldo por `std::string saldoEmTexto`), o resto do código continua funcionando — desde que a interface (`getSaldo`, `setSaldo`) permaneça igual.
3. **Adiciona validações e lógica:** Você pode adicionar lógica de negócio nos métodos `set`, como validar dados, aplicar regras ou disparar eventos.



Situações práticas

Sistema de RH (Funcionário, Pagamento)

- Impede que o salário seja alterado diretamente.
- Permite validar quanto será o pagamento.
- Evita fraudes e erros.

Sem encapsulamento:

Um programador poderia fazer `funcionario.salario = -100`.



Situações práticas

Jogos

- Evitar trapaças
- Permite aplicar lógica.

Sem encapsulamento:

Um programador poderia fazer personagem.HP = 999999.

Um programador poderia fazer personagem.HP = -100.



Get e Set

Métodos usados para controle de acesso.

Get: Retorna o valor de um atributo privado.

Set: Define ou atualiza o valor de um atributo privado, com validações se necessário.

```
class Pessoa {  
    private:  
        string nome;  
  
    public:  
        string getNome() const {  
            return this->nome;  
        }  
  
        void setNome(string nome) {  
            this->nome = nome;  
        }  
};
```



Get e Set para todos os privates?

Se todos os atributos são privados e são acessíveis com getters e setters simples, o encapsulamento é inútil.

- Deixe seus atributos privados por padrão;
- Crie os getters e setters apenas quando necessário;

Get e Set + Construtores

Você pode inicializar atributos com valores válidos direto no construtor, evitando erros:

```
class Pessoa {  
    private:  
        string nome;  
  
    public:  
  
        Pessoa(string nome) {  
            setName(nome);  
        }  
  
        string getNome() const {  
            return this->nome;  
        }  
  
        void setName(string nome) {  
            this->nome = nome;  
        }  
};
```



Get + const

Marcar getters como `const` significa que eles não alteram o estado do objeto.

Mas o `const` é opcional

error: passing 'const std::string' {aka 'const std::__cxx11::basic_string'} as 'this' argument discards qualifiers [-fpermissive]

```
class Pessoa {  
private:  
    string nome;  
  
public:  
  
    Pessoa(string nome) {  
        setName(nome);  
    }  
  
    string getName() const {  
        nome = nome + "!";  
        return nome;  
    }  
  
    void setName(string nome) {  
        this->nome = nome;  
    }  
};
```


Atividade 01 - Implemente um sistema de biblioteca com as classes

Implementar a classe Livro com as informações do título, autor e ISBN, além de estado que indica se está emprestado ou disponível.

A classe a Cliente com as informações nome do cliente, um identificador único e os livros que ele já pegou emprestado e os livros que tem atualmente. Adicione também um método que retorne a quantidade de livros que ele tem emprestados no momento.

Criar a classe Emprestimo, responsável por representar o ato de um cliente pegar um livro emprestado. Essa classe deve ter atributos para armazenar o cliente que fez o empréstimo, o livro emprestado, a data do empréstimo e a data da devolução.

Regras: 1 - Um livro só pode ser emprestado se estiver disponível; 2 - um cliente não pode pegar mais do que três livros ao mesmo tempo;

Todos os atributos devem ser privados e deve-se criar os get e set apenas quando necessários.

Sobrecarga de Operadores



Sobrecarga de Operadores

Permite redefinir o comportamento de operadores como `+`, `-`, `==`, `<<`, etc., para que funcionem com tipos definidos pelo usuário (como classes e structs).

- Facilita o código, deixando-o mais legível e intuitivo.
- Permite operações diretas entre objetos (ex: somar dois objetos).
- Mantém o encapsulamento ao mesmo tempo que oferece uma sintaxe simples.



Por que usar?

Imagine que você está programando um sistema de supermercado.

Sem sobrecarga você precisa iterar e acessar o atributo que contém o valor e a quantidade do item comprado.

```
void exibir() const {  
    cout << "Produto: " << nome << ", Preço: R$ " << preco << endl;  
}
```



Por que usar?

Imagine que você está programando um sistema de supermercado.

Com a sobrecarga, você lida com o objeto como se fosse um tipo primitivo.

```
int main() {  
    Produto p("Arroz", 5.50);  
    cout << p << endl; // Impressão com operador sobrecarregado  
    return 0;  
}
```



Quando usar?

Use sobrecarga de operadores quando quiser tornar o uso de objetos mais natural, como somar dois objetos, comparar, imprimir com `<<`, etc. Mas cuidado para não abusar: use apenas quando faz sentido.



Quais eu posso sobrecarregar?

Aritméticos: +, -, *, /, %

Relacionais: ==, !=, <, >, <=, >=

Lógicos: &&, ||, !

Atribuição: =, +=, -=, *= etc.

Incremento/decremento: ++, --

Subscripting: []

Chamada de função: ()

Fluxo de entrada/saída: <<, >>

Quais eu não posso sobrecarregar?

`.` → operador de acesso a membro (ex: `obj.atributo`)

`.*` → operador de acesso a membro por ponteiro (ex: `obj.*ptr`)

`::` → operador de resolução de escopo (ex: `std::cout`)

`sizeof` → operador que retorna o tamanho de um tipo/objeto

`typeid` → operador usado para RTTI (informações de tipo em tempo de execução)

`alignof` / `alignas` → usados para alinhamento de memória

`noexcept` → especificador de exceção

`static_cast`, `dynamic_cast`, `reinterpret_cast`, `const_cast` → operadores de casting

`new` e `delete` de forma global podem ser redefinidos, mas os operadores `new[]` e `delete[]` específicos da linguagem não podem ser sobrecarregados em contexto global diretamente com nova sintaxe.



Cuidados ao sobrecarregar operadores

1. Mantenha o comportamento esperado
2. Cuidado com operadores que alteram estado
 - a. ++
 - b. --
 - c. +=, -=, /=, *=
3. Tratamento de exceções e segurança
 - a. Garanta que seu operador lide corretamente com possíveis erros e não deixe o objeto em estado inválido.
4. Use `const` sempre que possível



Atenção para operações binárias

Binárias: São operadores que operam sobre dois operandos — ou seja, recebem dois valores para realizar a operação.

Exemplo: +, -, /, =, ., !, +

Não binárias: São operadores que operam sobre um único operando.

Exemplo: ++, --



De qual operando é a operação

A função de sobrecarga chamada é sempre o do operando da esquerda.

No exemplo abaixo, a operação `d = b + a` vai tentar chamar a função do `int` e não da classe `Numero`

```
Numero a(5);  
int b = 10;  
  
auto c = a + b; // funciona se oper+ é membro de Numero  
auto d = b + a; // não funciona se operador+ é membro, pois b não é Numero
```

Prefira operadores binários como funções não-membro

São funções que não estão declaradas dentro de nenhuma classe — ou seja, não são métodos de objeto. Elas existem "solitas" no escopo global (ou namespace).

>> e << devem ser declaradas como não membros ou declaradas na classe, mas implementadas fora da classe.

Ou declarados com `friend`

```
class Numero {  
public:  
    int valor;  
    Numero(int v) : valor(v) {}  
};  
  
// função não-membro que soma um int com um Numero  
Numero operator+(int lhs, const Numero& rhs) {  
    return Numero(lhs + rhs.valor);  
}
```

Atividade - Sistema de Bancário

Criar uma classe Conta (numero, titulo e saldo) que represente uma conta bancária simples, permitindo realizar depósitos e transferências de forma intuitiva, usando sobrecarga de operadores.

Sobrecarga do operador += para realizar um depósito, somando o valor passado ao saldo da conta.

Exemplo: conta += 500.00;

Sobrecarga do operador - para permitir a transferência de saldo entre duas contas:

contaDestino = contaOrigem - 200.00;

Se o saldo for insuficiente, deve lançar exceção.

Sobrecarga do operador << para imprimir as informações da conta formatadas: número, titular e saldo.



Sobrecarga do operador +

```
Produto Produto::operator+(const Produto& outro) const {  
    if (codigo != outro.codigo)  
        throw invalid_argument("Não é possível somar produtos com códigos diferentes.");  
    return Produto(codigo, nome, preco, quantidade + outro.quantidade);  
}
```

Sobrecarga do operador ==

```
bool Produto::operator==(const Produto& outro) const {  
    return codigo == outro.codigo;  
}
```

Sobrecarga do operador ++

Em C++, dentro de qualquer método membro de uma classe, existe um ponteiro oculto chamado `this`. Ele aponta para o objeto atual — ou seja, o objeto que está executando aquele método.

Quando você escreve `*this`, está desreferenciando o ponteiro `this`, ou seja, acessando o objeto em si.

```
Produto& Produto::operator++() {  
    quantidade++;  
    return *this;  
}
```


Sobrecarga do operador ==

```
bool Produto::operator==(const Produto& outro) const {  
    return codigo == outro.codigo;  
}
```

Sobrecarga do operador <<

```
ostream& operator<<(ostream& os, const Produto& p) {  
    os << "Código: " << p.codigo << "\n"  
    << "Nome: " << p.nome << "\n"  
    << "Preço: R$" << p.preco << "\n"  
    << "Quantidade: " << p.quantidade;  
    return os;  
}
```

Sobrecarga do operador >>

```
// include <string> -> stoi; stof; stod
istream& operator>>(istream& is, Pessoa& p) {

    string idade_str;

    cout << "Digite a idade: ";
    getline(is, idade_str);
    p.idade = stoi(idade_str);

    cout << "Digite o nome: ";
    getline(is, p.nome);

    return is;
}
```

Membros Estáticos e Constantes de Classe



Membros estáticos

Membros estáticos são membros da classe (sejam variáveis ou funções) que não pertencem a uma instância específica da classe, mas sim à própria classe.

Ou seja, não importa quantos objetos da classe você crie, todos eles compartilham a mesma instância do membro estático.



Características

1. Membros estáticos podem ser variáveis ou funções.
2. São acessíveis sem a necessidade de um objeto, ou seja, podem ser acessados diretamente pela classe, usando o operador ::
3. Variáveis estáticas devem ser inicializadas fora da classe
4. Membros estáticos são compartilhados por todas as instâncias da classe. Ou seja, se um membro estático for alterado em uma instância, essa mudança será refletida em todas as outras instâncias.
5. Membros estáticos não têm acesso a membros não estáticos da classe. Apenas membros estáticos ou funções externas podem acessá-los.



Podem ser variáveis ou funções

Para definir um membro como estático basta usar o modificador `static`.

```
class Pessoa {  
public:  
  
    static int contagem; // Membro estático  
  
};
```



Podem ser variáveis ou funções

Para definir um membro como estático basta usar o modificador `static`.

```
class Pessoa {  
public:  
  
    static void mostrarContagem() {  
        cout << "Contagem: " << contagem << endl;  
    }  
  
};
```




São acessados diretamente pela classe, usando ::

Você não precisa criar um objeto para acessar membros estáticos, basta utilizar o padrão: NomeClasse::membroEstático.


Contudo, objetos podem acessar membros estáticos, mas o membro continua com seu valor ou comportamento sendo estático.

```
class Pessoa {  
public:  
    static int contagem;  
  
    static void mostrarContagem() {  
        cout << "Contagem: " << contagem << endl;  
    }  
};  
  
int main() {  
    Pessoa p1;  
  
    Pessoa::mostrarContagem();  
    p1.mostrarContagem();  
    return 0;  
}
```

Variáveis estáticas devem ser inicializadas fora da classe

Para inicializar um membro estático, você deve fazer isso fora da definição da classe, geralmente em um arquivo .cpp.

```
class Pessoa {  
public:  
    static int contagem;  
  
    static void mostrarContagem() {  
        cout << "Contagem: " << contagem << endl;  
    }  
};  
  
int Pessoa::contagem = 0;
```



Membros estáticos são compartilhados por todas as instâncias da classe.

Se um membro estático for alterado em uma instância, essa mudança será refletida em todas as outras instâncias.

```
class Pessoa {  
public:  
    static int contagem;  
};  
  
int Pessoa::contagem = 0;  
  
int main() {  
    Pessoa p1;  
    Pessoa p2;  
  
    p1.contagem++;  
    p2.contagem++;  
  
    cout<< Pessoa::contagem; // 2  
    return 0;  
}
```

Membros estáticos não têm acesso a membros não estáticos da classe.


Métodos estáticos só conseguem acessar valores estáticos da classe.

```
class Pessoa {  
public:  
    static int contagem;  
  
    static void mostrarContagem() {  
        cout << "Contagem: " << contagem << endl;  
    }  
};
```



```
class Pessoa {  
public:  
    int contagem;  
  
    static void mostrarContagem() {  
        cout << "Contagem: " << contagem << endl;  
    }  
};
```





Membros não estáticos têm acesso a membros estáticos da classe.

Métodos não estáticos conseguem acessar membros estáticos das classes

```
class Pessoa {  
public:  
    static int contagem;  
  
    void getContagem() {  
        cout << Pessoa::contagem;  
    }  
};  
  
int Pessoa::contagem = 0;  
  
int main() {  
    Pessoa p1;  
    Pessoa p2;  
  
    p1.contagem++;  
    p2.contagem++;  
  
    p1.getContagem(); // 2  
    return 0;  
}
```



Constantes de classe

As constantes de classe são variáveis estáticas com valores constantes, ou seja, que não podem ser alterados depois de serem inicializadas.

```
class Circulo {  
public:  
    static const double PI; // Declaração de constante de classe  
    static const int raioMaximo = 100; // Inicializada diretamente  
};  
  
// Inicialização fora da classe  
const double Circulo::PI = 3.14159;
```

Atividade - Sistema de Reservas de Hotel

A aplicação deve simular as operações de um hotel, como cadastrar quartos, realizar reservas, calcular preços de diárias e aplicar descontos dependendo do tipo de cliente e do quarto. O sistema deve ter as classes: Hotel, Quarto, Cliente e Reserva.

A classe `Hotel` mantém os quartos disponíveis e gerenciar as reservas feitas pelos clientes, além de um membro estático para manter o número total de reservas. Adicionar também uma constante que representa o preço base da diária dos quartos (por exemplo, R\$200) que será usada como valor inicial para o cálculo dos preços de todas as diárias.

A classe `Cliente` armazena o seu nome e tipo (VIP ou Comum).

A classe `Quarto` representa os quartos disponíveis para reserva. Cada quarto tem um número de identificação e um preço base. Além disso, ela define um preço adicional para quartos de luxo. Cada quarto tem um método que calcula o preço final da diária, levando em consideração o tipo de cliente.

A classe `Reserva` é responsável por registrar uma reserva feita por um cliente. Cada reserva está associada a um cliente e a um quarto específico, e contém o valor da diária final calculado após o possível desconto. Ela deve ter um método para exibir os detalhes da reserva: nome do cliente, o número do quarto e o valor final a ser pago.

O preço final da diária é calculado com base no tipo de cliente (VIP tem 10% de desconto) e no tipo de quarto (de Luxo tem 20% de desconto).

Se a reserva for bem sucedida mostra o valor final da diária.

Herança e Classes Derivadas



Herança

É o mecanismo da orientação a objetos que permite que uma classe herde características (atributos e métodos) de outra.

Evitar repetição de código, promover reutilização, facilitar manutenção e organização do projeto.

estabelecer uma relação hierárquica entre classes, onde a classe derivada "é um tipo de" classe base.

```
class Animal {  
    public:  
        string nome;  
};  
  
class Gato : public Animal {  
  
};
```




Herança

Mesmo que Gato não tenha um atributo chamado nome, é possível acessar pois Gato herda esse atributo de animal.

Assim, a classe derivada Gato tem acesso aos mesmo atributos da classe Animal.

```
class Animal {  
public:  
    string nome;  
};  
  
class Gato : public Animal {  
};  
  
int main()  
{  
    Gato gatinho;  
  
    gatinho.nome = "frajola";  
  
    cout << gatinho.nome;  
  
    return 0;  
}
```



E se eu tiver um atributo de mesmo nome?

Se uma classe derivada declarar um atributo com o mesmo nome de um atributo da classe base, o ****atributo** da derivada "esconde" (ou oculta) o da base — esse comportamento é chamado de name hiding (ocultação de nome).

```
class Animal {  
public:  
    string nome;  
};  
  
class Gato : public Animal {  
public:  
    string nome;  
};
```

E se eu tiver um atributo de mesmo nome?

Gato tem um membro nome, que oculta o membro nome herdado de Animal e por padrão, ao acessar g.nome, o compilador acessa o membro da classe derivada.

Se você quiser acessar o nome da classe base, precisa usar a qualificação de escopo: g.Animal::nome

```
class Animal {
public:
    string nome = "Animal";
};

class Gato : public Animal {
public:
    string nome = "Gato";
};

int main() {
    Gato g;
    cout << g.nome << endl;           // imprime "Gato"
    cout << g.Animal::nome << endl; // imprime "Animal"
    return 0;
}
```



Como chamar um construtor

Construtores não são herdades e para passar as informações da classe derivada para a classe base você deve usar a seguinte estrutura.

```
class Gato : public Animal {  
    int idade;  
  
    Gato(string nome, int i) : Animal(nome) {  
        idade = i;  
    }  
};
```



Modificadores de acesso

Em C++, você pode definir a visibilidade da herança, modificando o acesso aos atributos herdados.

Modificador	Efeito
<code>public</code>	Membros <code>public</code> e <code>protected</code> da base permanecem acessíveis na derivada
<code>protected</code>	Membros <code>public</code> e <code>protected</code> da base se tornam <code>protected</code> na derivada
<code>private</code>	Membros <code>public</code> e <code>protected</code> da base se tornam <code>private</code> na derivada

Modificadores de acesso

Modificador	Acesso na própria classe	Acesso nas classes derivadas	Acesso fora da classe
public	✓	✓	✓
protected	✓	✓	✗
private	✓	✗	✗

Herança Múltipla

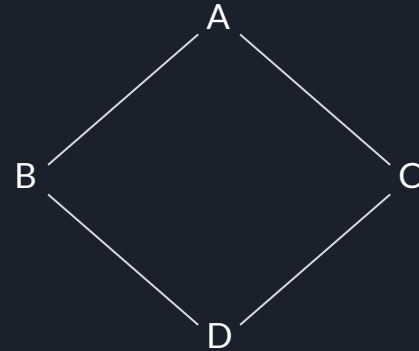
C++ permite que uma classe herde de duas ou mais classes ao mesmo tempo. Ou seja, uma classe derivada pode herdar atributos e métodos de várias classes base.

```
class Triatleta : public Caminhante, public Nadador {  
public:  
    void competir() {  
        cout << "Competindo no triatlo\n";  
    }  
};
```


Herança Múltipla

Para evitar ambiguidades no código, como o problema do "Diamond Problem" (problema do losango), onde duas superclasses definem o mesmo método e a linguagem não saberia qual usar.

O problema do losango acontece quando uma subclasse herda de duas superclasses, e essas superclasses herdam de uma mesma superclasse base.





Tipos estáticos e dinâmicos

O tipo estático é o tipo da variável conforme determinado no código-fonte, ou seja, o tipo que o compilador vê quando o código é compilado. Em termos de herança, esse é o tipo da variável na declaração, ou seja, a classe que você especifica ao criar a variável.

O tipo dinâmico é o tipo real do objeto no momento da execução, ou seja, o tipo do objeto para o qual a variável aponta.

Tipo Estático

O tipo estático é o tipo da variável conforme determinado no código-fonte, ou seja, o tipo que o compilador vê quando o código é compilado. Em termos de herança, esse é o tipo da variável na declaração, ou seja, a classe que você especifica ao criar a variável.

```
int main() {  
  
    Animal* a = new Animal(); // Tipo Estático: Animal  
  
    Animal* g = new Gato(); // Tipo Estático: Animal  
  
    return 0;  
}
```



Tipo Dinâmico

O tipo dinâmico é o tipo real do objeto no momento da execução, ou seja, o tipo do objeto para o qual a variável aponta.

```
int main() {  
  
    Animal* a = new Animal(); // Tipo Dinâmico: Animal  
  
    Animal* g = new Gato(); // Tipo Dinâmico: Gato  
  
    return 0;  
}
```

Tipos estáticos e dinâmicos - Para que serve?

Permite tratar todos os objetos de mesma família de forma igual.

Se existir as classes Gato e Cachorro que herdam Animal, então é possível tratar Gato e Cachorro como Animal, sem distinção.

```
int main() {  
    vector<Animal*> animais;  
  
    // Adicionando animais no vetor  
    animais.push_back(new Gato("Frajola"));  
    animais.push_back(new Cachorro("Rex"));  
    animais.push_back(new Gato("Garfield"));  
    animais.push_back(new Cachorro("Bolt"));  
  
    // Percorrendo o vetor e chamando falar()  
    for (Animal* a : animais) {  
        a->falar(); // Polimorfismo: chama o método da classe real  
    }  
  
    // Limpando memória  
    for (Animal* a : animais) {  
        delete a;  
    }  
  
    return 0;  
}
```

Tipos estáticos e dinâmicos - Para que serve?

Permite tratar todos os objetos de mesma família de forma igual.

Se existir as classes `Gato` e `Cachorro` que herdam `Animal`, então é possível tratar `Gato` e `Cachorro` como `Animal`, sem distinção.

```
void bincarComAnimal(Animal& a) {  
    cout << a.nome << endl;  
}  
  
int main() {  
  
    Animal* a = new Animal("Frajola");  
    Gato b("Mingal");  
  
    bincarComAnimal(*a);  
    bincarComAnimal(b);  
  
    return 0;  
}
```



Como transformar um Animal em Gato?

Basta usar o `dynamic_cast` para converter um tipo estático para o tipo dinâmico.

```
Animal* a = new Gato("Frajola");  
  
(dynamic_cast<Gato*>(a))->ronronar();
```

Atividade - Cadastro de Veículos

Criar um sistema de cadastro de veículos:

A classe `Veículo` deve possuir dois atributos: `marca` e `modelo`, além de um método chamado `exibirInformacoes`, que deve mostrar a `marca` e o `modelo` do veículo.

A classe `Carro` deve herdar da classe `Veiculo`. Ela deve incluir um novo atributo chamado `numeroPortas`. Além disso, ela deve possuir um método chamado `exibirInformacoesCarro`, que deve chamar o método `exibirInformacoes` da superclasse para mostrar a `marca` e o `modelo`, e depois imprimir o número de portas.

A classe `Moto` também deve herdar de `Veiculo`. Ela deve possuir um atributo chamado `cilindrada` e um método chamado `exibirInformacoesMoto`, que chama o método `exibirInformacoes` da superclasse e, em seguida, exibe a cilindrada da moto.

O sistema deve contar com uma classe de controle chamada `SistemaVeiculos`, que deve conter uma lista de objetos do tipo `Veiculo`. Essa classe deve implementar o CRUD completo da lista de veículos.

Polimorfismo e Métodos Virtuais



Polimorfismo

A palavra "Polimorfismo" vem do grego e significa "muitas formas" e é um conceito na programação orientada a objetos que significa que um objeto pode fazer a mesma coisa de formas diferentes.

Em C++, isso se refere ao fato de que você pode usar um único método ser implementado de diferentes formas por cada uma das subclasses.



Sobrecarga e sobrescrita

É possível se utilizar do polimorfismo de duas maneiras:

- Sobrecarga de métodos
- Sobrescrita de métodos

Sobrecarga e sobrescrita

Sobrecarga (ou Overloading): Acontece quando você tem vários métodos com o mesmo nome, mas diferentes parâmetros (quantidade ou tipo de argumento).

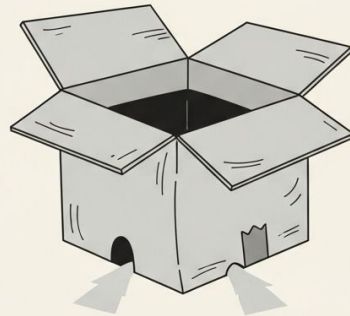
```
class Animal {  
public:  
    virtual void emitirSom() { cout << "Animal emitiu um som." << endl; }  
    virtual void emitirSom(string nome) { cout << nome << " emitiu um som." << endl; }  
};
```

Sobrecarga e sobrescrita

Sobrescrita(ou Overriding): Isso acontece quando uma subclasse redefine (implementa sua própria versão) um método da sua superclasse.

Em C++ apenas métodos virtuais podem ser sobrescritos

```
class Animal {  
public:  
    virtual void emitirSom() { cout << "Animal emitiu um som." << endl; }  
};  
  
class Cachorro : public Animal {  
public:  
    void emitirSom() override { cout << "Au AU." << endl; }  
};
```



O animal emitiu um som



Miau



Au, Au



ko ko

ai



Requisitos para polimorfismo de sobreescrita

- A subclasse deve herdar da superclasse.
- O método da superclasse deve ser acessível (normalmente public ou protected).



O que não é polimorfismo

- Métodos com nomes iguais mas em classes diferentes e sem relação entre si.



Chamada em construtores

Quando você instancia um objeto em C++, o processo de construção/destruição é feito em fases:

- Da base até a derivada (construção) e
- Da derivada até a base (destruição).

Portanto, se você chamar um método virtual dentro do construtor ou destrutor, o C++ chama a versão da classe atual.

```

29
30 class Base {
31 public:
32     Base() {
33         cout << "Base construindo...\n";
34         virtualFunc();
35     }
36     virtual void virtualFunc() { cout << "Base::virtualFunc\n"; }
37     virtual ~Base() {
38         cout << "Base destruindo...\n";
39         virtualFunc();
40     }
41 };
42
43 class Derivada : public Base {
44 public:
45     Derivada() {
46         cout << "Derivada construída\n";
47         virtualFunc();
48     }
49     void virtualFunc() override { cout << "Derivada::virtualFunc\n"; }
50     ~Derivada() {
51         cout << "Derivada destruída\n";
52         virtualFunc();
53     }
54 };
55
56 int main()

```

```

Base::virtualFunc
Derivada construída
Derivada::virtualFunc
Derivada destruída
Derivada::virtualFunc
Base destruindo...
Base::virtualFunc

```

Fase de inicialização separada

Em vez de depender do construtor, você define um método como `init()` que é chamado fora do construtor, quando o objeto já foi completamente construído.

```
class Base {  
public:  
    virtual void inicializar() = 0;  
    virtual ~Base() {}  
};  
  
class Derivada : public Base {  
public:  
    void inicializar() override {  
        cout << "Inicializando Derivada!" << endl;  
        // qualquer lógica polimórfica aqui  
    }  
};
```



Métodos virtuais puros

Significa que as subclasses são obrigadas a implementar o método

```
class Animal {  
public:  
    virtual void fazerSom() = 0; // método virtual puro  
};
```

Classe Abstrata

Quando uma classe possui pelo menos um método virtual puro (= 0), ela é chamada de classe abstrata.

E classes abstratas não podem ter objetos criados diretamente — ou seja, não podem ser instanciadas.

```
class Animal {  
public:  
  
    virtual void correr() = 0;  
  
};
```

```
int main()  
{  
  
    Animal a;  
  
    return 0;  
}
```



E se eu não colocar o =0

Seu código vai compilar e vai executar normalmente desde que essa função não seja chamada.



Para que serve?

Elas são usadas quando:

- Você quer definir um comportamento comum base, mas deixar detalhes para as subclasses.
- Há alguns métodos que todas as subclasses devem obrigatoriamente implementar.
- Você quer evitar que alguém instancie diretamente uma classe que só faz sentido como base.

Atividade

Desenvolva uma aplicação em C++ para simular o processo de autenticação de usuários do Discord, onde o usuário pode fazer login de 3 formas diferentes: login e senha, Google ou Qr code.

Creia classe abstrata Autenticacao com os métodos autenticar() que executa o processo de autenticação. Nesse método são executado os passos de coletar dados, validar dados e iniciar a sessão do usuário.

Implemente a classe LoginSenha, que herda de Autenticacao, e solicite do usuário um nome de login e uma senha. A autenticação deve ser considerada válida apenas se o login tiver mais de 4 letras e a senha for tiver mais de 8 dígitos.

Em seguida, implemente a classe GoogleAuth, que simula a autenticação via Google. Nesse caso, o sistema deve perguntar o e-mail e validar se ele termina com “[@gmail.com](mailto:)”.

Por fim, crie a classe QRCodeAuth, que simula a leitura de um código QR. O sistema deve solicitar um código textual e validá-lo. O código só será aceito se for igual a “QR12345”.

Os métodos de coletar os dados e validar os dados devem ser abstratos na classe Autenticacao implementados em cada subclasse.

Templates



Templates em C++

Templates são um recurso da linguagem que permite escrever código genérico.

Ou seja, você pode escrever uma única função ou classe que funciona com vários tipos de dados, sem repetir código.

Evitar a duplicação de código para funções ou classes que fazem a mesma lógica, mas com tipos diferentes.

Como funciona

Você define que seu método ou classe vai trabalhar com um tipo abstrato `T` que vai ser definido depois.

Você define que existe um tipo genérico `T` que vai ser informado usando `<>` ao lado do nome da classe

```
template <typename T>
class Animal {
public:

    T identificador;

};

int main()
{
    Animal<string> a;

    a.identificador = "987546213";

    cout << "Identificador do animal: " << a.identificador ;

    return 0;
}
```

Como funciona

Você define que seu método ou classe vai trabalhar com um tipo abstrato `T` que vai ser definido depois.

Você define que existe um tipo genérico `T` que vai ser informado usando `<>` ao lado do nome da classe

```
template <typename T>
class Animal {
public:

    T identificador;

};

int main()
{
    Animal<int> a;

    a.identificador = 1025;

    cout << "Identificador do animal: " << a.identificador ;

    return 0;
}
```

Como funciona

Você define que seu método ou classe vai trabalhar com um tipo abstrato `T` que vai ser definido depois.

Você define que existe um tipo genérico `T` que vai ser informado usando `<>` ao lado do nome da classe

```
template <typename T>
class Animal {
public:

    T identificador;

};

int main()
{
    Animal<int> a;

    a.identificador = 1025;

    cout << "Identificador do animal: " << a.identificador ;

    return 0;
}
```

Como funciona

Você define que seu método ou classe vai trabalhar com um tipo abstrato `T` que vai ser definido depois.

Você define que existe um tipo genérico `T` que vai ser informado usando `<>` ao lado do nome da classe

```
template <typename T>
class Animal {
public:

    T identificador;

};

class Identificador {
public:
    string head = "fsfsa";
    string payload = "78965";
    string signature = "98gfds7fd";
};

int main()
{
    Animal<Identificador> a;
    Identificador id;

    a.identificador = id;

    cout << "Identificador do animal: " << a.identificador.head
        << a.identificador.payload << a.identificador.signature;

    return 0;
}
```



Herança + Polimorfismo vs Generics

Use herança + polimorfismo quando você precisa de comportamentos diferentes entre objetos.

Use templates quando você precisa da mesma lógica para múltiplos tipos de dados.

Restrição de Tipo

É possível restringir o tipo genérico para que seja apenas membros de uma determinada hierarquia de herança.

```
// Método 2: Usando conceitos (C++20)
template <typename T>
concept AnimalType = std::is_base_of<Animal, T>::value;

template <AnimalType T>
class PetShop {
public:
    void banhoETosa(const T& animal) {
        std::cout << "Realizando banho e tosa...\n";
        animal.emitirSom();
    }
};

int main() {
    Cachorro dog;
    Gato cat;

    // Funciona - Cachorro é Animal
    ClinicaVeterinaria<Cachorro> vetDog;
    vetDog.realizarExame(dog);

    // Funciona - Gato é Animal
    PetShop<Gato> petShopCat;
    petShopCat.banhoETosa(cat);
}
```




Exemplo prático

Imagine que você deseja criar uma interface para uma API comum. A interface que definir que todo endpoint precisa dos métodos:

- Criar;
- Listar;
- Editar;
- Deletar;

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Classes básicas Pessoa e Produto
struct Pessoa {
    int id;
    string nome;
};

struct Produto {
    int id;
    string descricao;
};

// Classe abstrata template EndPoint<T>
template <typename T>
class EndPoint {
public:
    virtual ~EndPoint() = default;

    virtual void criar(const T& item) = 0;
    virtual vector<T> listar() const = 0;
    virtual void editar(int id, const T& item) = 0;
    virtual void deletar(int id) = 0;
};

```

```

// Implementação concreta para Pessoa
class PessoaEndPoint : public EndPoint<Pessoa> {
private:
    vector<Pessoa> pessoas;
public:
    void criar(const Pessoa& p) override {
        pessoas.push_back(p);
        cout << "Pessoa criada: " << p.nome << "\n";
    }

    vector<Pessoa> listar() const override {
        return pessoas;
    }

    void editar(int id, const Pessoa& p) override {
        for (auto& pessoa : pessoas) {
            if (pessoa.id == id) {
                pessoa.nome = p.nome;
                cout << "Pessoa editada: " << pessoa.nome << "\n";
                return;
            }
        }
        cout << "Pessoa com ID " << id << " não encontrada.\n";
    }

    void deletar(int id) override {
        pessoas.erase(
            remove_if(pessoas.begin(), pessoas.end(),
                [id](const Pessoa& p) { return p.id == id; }),
            pessoas.end());
        cout << "Pessoa deletada: ID " << id << "\n";
    }
};

```

// Implementação concreta para Produto

```
class ProdutoEndPoint : public EndPoint<Produto> {
private:
    vector<Produto> produtos;
public:
    void criar(const Produto& p) override {
        produtos.push_back(p);
        cout << "Produto criado: " << p.descricao << "\n";
    }

    vector<Produto> listar() const override {
        return produtos;
    }

    void editar(int id, const Produto& p) override {
        for (auto& produto : produtos) {
            if (produto.id == id) {
                produto.descricao = p.descricao;
                cout << "Produto editado: " << produto.descricao << "\n";
                return;
            }
        }
        cout << "Produto com ID " << id << " não encontrado.\n";
    }

    void deletar(int id) override {
        produtos.erase(
            remove_if(produtos.begin(), produtos.end(),
                [id](const Produto& p) { return p.id == id; }),
            produtos.end());
        cout << "Produto deletado: ID " << id << "\n";
    }
};
```

```
int main() {
    PessoaEndPoint pessoasApi;
    ProdutoEndPoint produtosApi;

    Pessoa p1{1, "João"};
    Pessoa p2{2, "Maria"};

    pessoasApi.criar(p1);
    pessoasApi.criar(p2);

    Produto pr1{1, "Teclado"};
    Produto pr2{2, "Mouse"};


    produtosApi.criar(pr1);
    produtosApi.criar(pr2);

    // Listar pessoas
    cout << "Pessoas cadastradas:\n";
    for (const auto& p : pessoasApi.listar()) {
        cout << "- " << p.id << ": " << p.nome << "\n";
    }

    // Editar pessoa
    pessoasApi.editar(1, Pessoa{1, "João Silva"});

    // Deletar produto
    produtosApi.deletar(2);

    return 0;
}
```



Por que usar templates e não apenas classes abstrata

Não seria possível, para cada classe de endpoint, definir o tipo de cada que os métodos devem receber.