PROGRAMAÇÃO EM C++
PROJETO FINAL

INF 1900

Prof. Dr. Bruno B. P. Cafeo

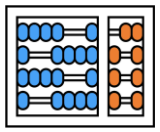
Institute of Computing

University of Campinas

Comentários Significativos

```
// Calcula a média de uma lista de números
float calcularMedia(const std::vector<int>& numeros);
```

- Comentários claros e significativos ajudam a entender a finalidade de uma função ou trecho de código. No exemplo, o comentário descreve o propósito da função calcularMedia.

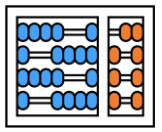


UNICAMP

Nomes Significativos para Variáveis

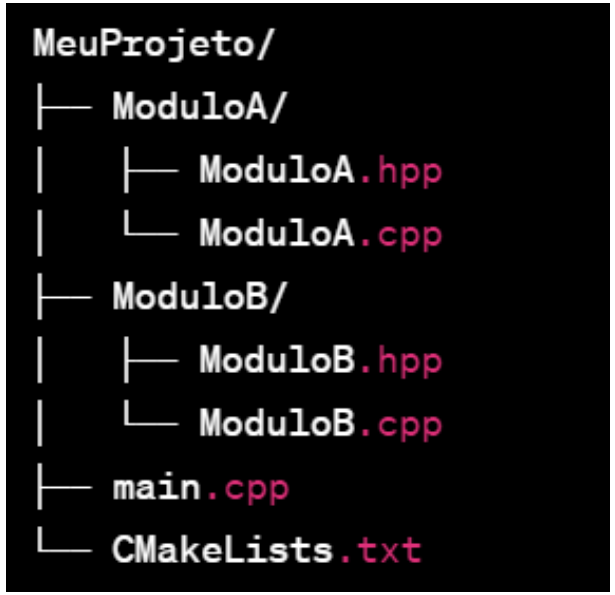
```
int numeroDeAlunos;
```

- Escolher nomes significativos para variáveis torna o código mais legível. Neste exemplo, a variável numeroDeAlunos é claramente descrita.

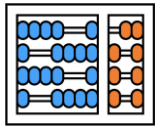


UNICAMP

Organização em Diretórios



- Organizar um projeto C++ em pacotes usando diretórios facilita a manutenção e a compreensão. Cada módulo (ModuloA, ModuloB) tem sua própria pasta.

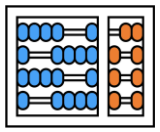


UNICAMP

Uso de Include Guards

```
// ModuloA.hpp
#pragma once
namespace MeuProjeto::ModuloA {
    void funcaoModuloA();
}
```

- O uso de Include Guards (`#pragma once`) evita a inclusão múltipla do mesmo cabeçalho em diferentes partes do código.



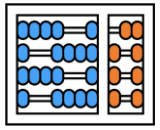
UNICAMP

Organização dos Cabeçalhos em C++

```cpp
// ModuloA.hpp
namespace MeuProjeto::ModuloA {
    void funcaoModuloA();
}

// ModuloA.cpp
#include "ModuloA.hpp"
void MeuProjeto::ModuloA::funcaoModuloA() { /* Implementação */ }
```

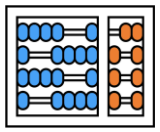
- Evitar a implementação de lógica nos cabeçalhos e movê-la para os arquivos de implementação, como demonstrado no exemplo, melhora a modularidade do código.



Inclusão Mínima Necessária

```
// ModuloA.hpp
#ifndef MODULOA_H
#define MODULOA_H
namespace MeuProjeto::ModuloA {
    void funcaoModuloA();
}
#endif
```

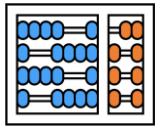
- Incluir apenas o necessário nos cabeçalhos, usando Include Guards, ajuda a otimizar o tempo de compilação e evita erros de inclusão múltipla.



Análise Estática e Ferramentas de Linting

- A análise estática é uma prática essencial para identificar potenciais problemas no código antes mesmo da execução.

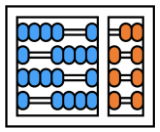
- Ferramentas de linting, como o Clang ou CppCheck, podem ser integradas ao fluxo de desenvolvimento para realizar verificações automáticas.



Exemplo de Análise Estática - CppCheck

```cpp
// ModuloA.cpp
int MeuProjeto::ModuloA::funcaoComBug() {
    int x;
    return x; // Erro: 'x' é usado sem ser inicializado
}
```

- Ferramentas de linting, como o CppCheck, podem detectar erros estáticos, como o uso de variáveis não inicializadas, contribuindo para código mais seguro.

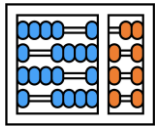


UNICAMP

Exemplo de Análise Estática - Clang Static Analyzer

```
// ModuloA.cpp
int MeuProjeto::ModuloA::funcaoComLeak() {
    int* ptr = new int;
    return *ptr; // Aviso: Possível vazamento de memória
}
```

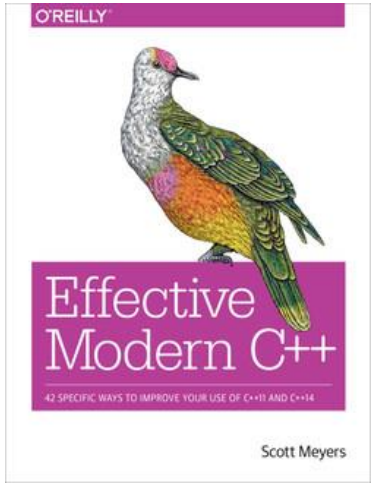
- O Clang Static Analyzer pode identificar potenciais vazamentos de memória, auxiliando na prevenção de problemas comuns.

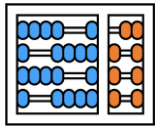


UNICAMP

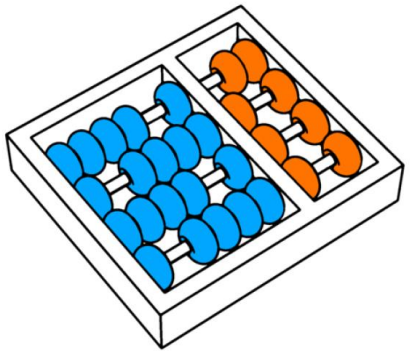
Links e referências de boas práticas

- ISO
- Sutter e Stroustrup
- ROS
- Linux
- Google
- Microsoft
- CERN
- GCC
- ARM
- LLVM





UNICAMP



Prof. Dr. Bruno B. P. Cafeo

Sala 04
Instituto de Computação - Unicamp
Av. Albert Einstein, 1251
Cidade Universitária
Campinas – SP
13083-852

https://ic.unicamp.br/~cafeo/
cafeo@ic.unicamp.br