Introdução a Técnicas de Programação

Organização do código

Prof. André Campos
DIMAp/UFRN

Organizar... pra quê?

“Hábito saudável” de qualquer profissão.
Economia cognitiva

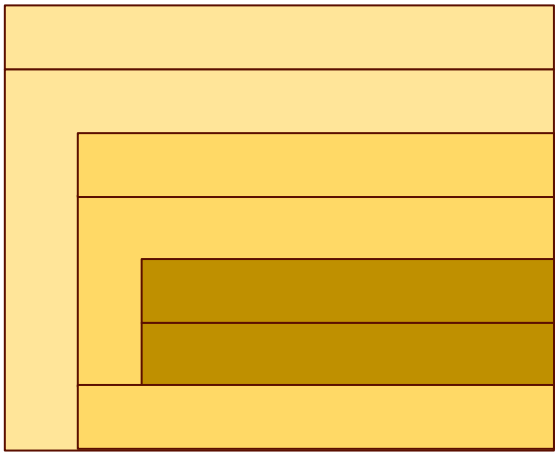
Em programação:

- Para facilitar a leitura e consequente manutenção do código (por você ou terceiros);
- Para dividir o trabalho (organização lógica de tarefas);
- Para não repetir a mesma tarefa e reutilizar algo já feito;
- ...

Organização visual

Indentação

- Termo dado ao espaço no início dos parágrafos
- Em programação, é uma forma de organizar o código para ajudar a leitura
- É fácil identificar quais blocos de instruções são subconjuntos de quais outros blocos.



Organização visual

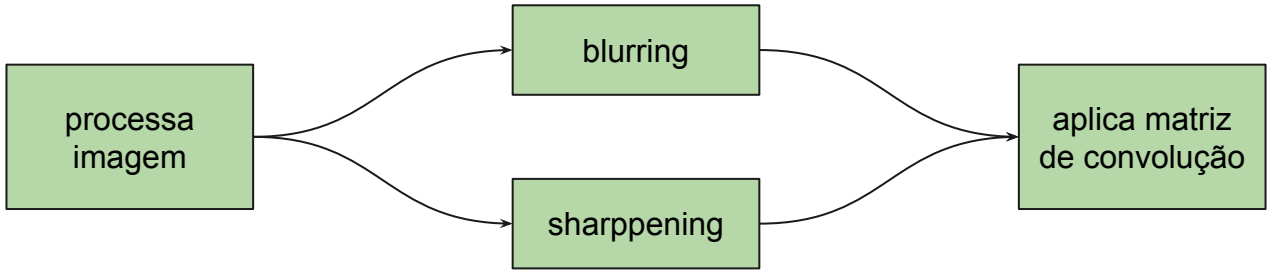
Padronização do estilo

- Útil quando se trabalha em equipe (+1 pessoas)
- Convenção mútua: sabe-se o que se espera quando lê o código de outros
- Exemplos
  - Constantes em maiúsculas
  - Nomes de variáveis e funções em minúsculas
  - Uso de _ para separar nomes (ex: num_pessoas) ou iniciais maiúsculas (ex: numPessoas)
  - Número de espaços nas identações
  - Início da definição de blocos na mesma linha (ex: `if(...) {`) ou na linha seguinte
  - Funções pequenas (que dê para ler toda sem precisar “rolar” a tela do editor)
  - ...

Organização lógica

Modularização

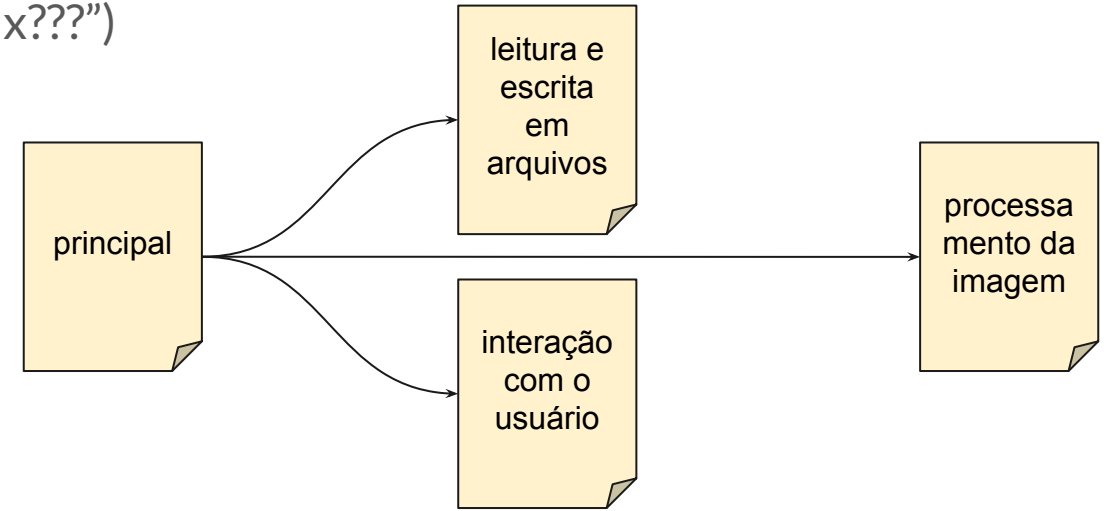
- Divisão do programa em partes lógicas (cada uma com sua função, classe, ...)
- Reuso de código



Organização lógica

Modularização com múltiplos arquivos

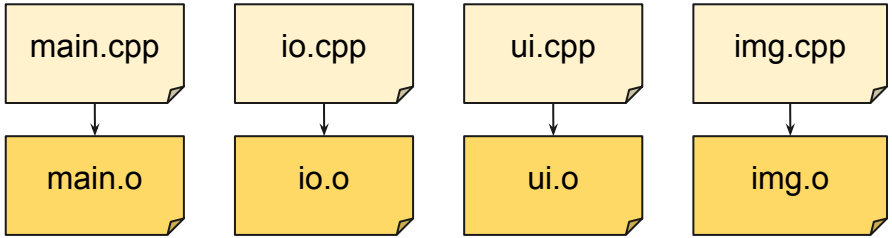
- Agrupamento de funcionalidades de um programa
- Facilita manutenção (projetos com +500 funcionalidades: “onde está a classe x???”)



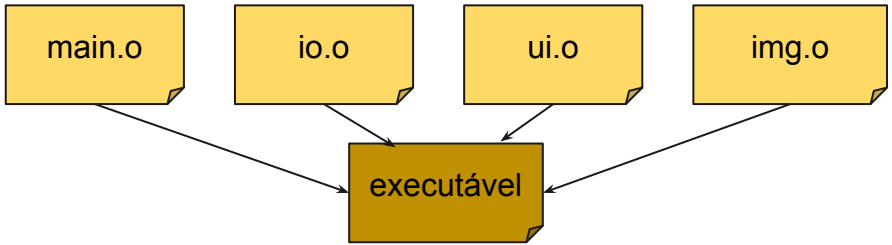
Entendendo a compilação de vários arquivos

A transformação de código C++ (arquivo .cpp) em código de máquina (binário executável) consiste em 2 etapas:

1. **Compilação**: transforma cada arquivo .cpp em um binário equivalente

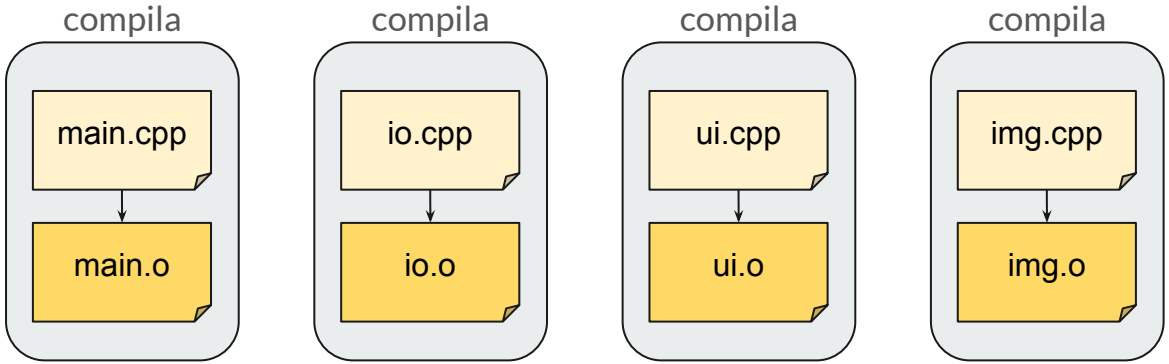


2. **Linkedição**: junta todos os binários em um executável



Entendendo a compilação de vários arquivos

O compilador processa cada arquivo `.cpp` separadamente
Então, cada `.cpp` deve ser "autocontido" (conhece todas as definições que usa)



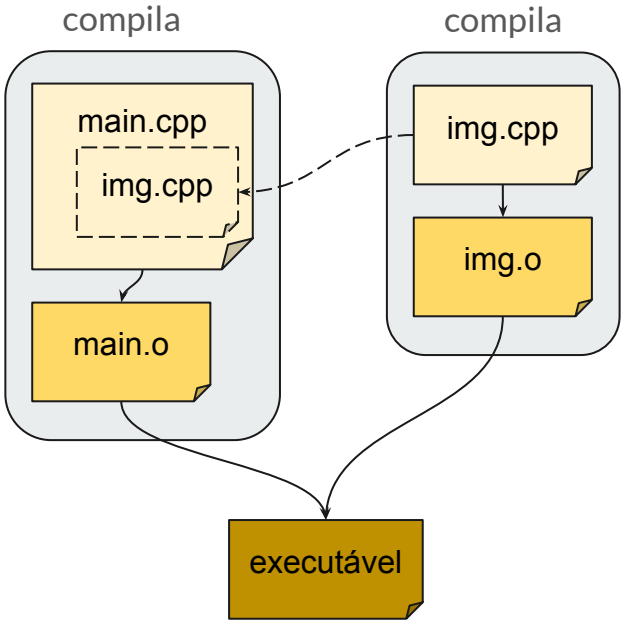
Mas alguns arquivos vão precisar da definição que se encontra em outros
Uso do `#include`

Include de arquivos locais

A diretiva `#include` inclui o conteúdo de um arquivo em outro.

Na hora de juntar (linkedição), algumas definições podem ser duplicadas

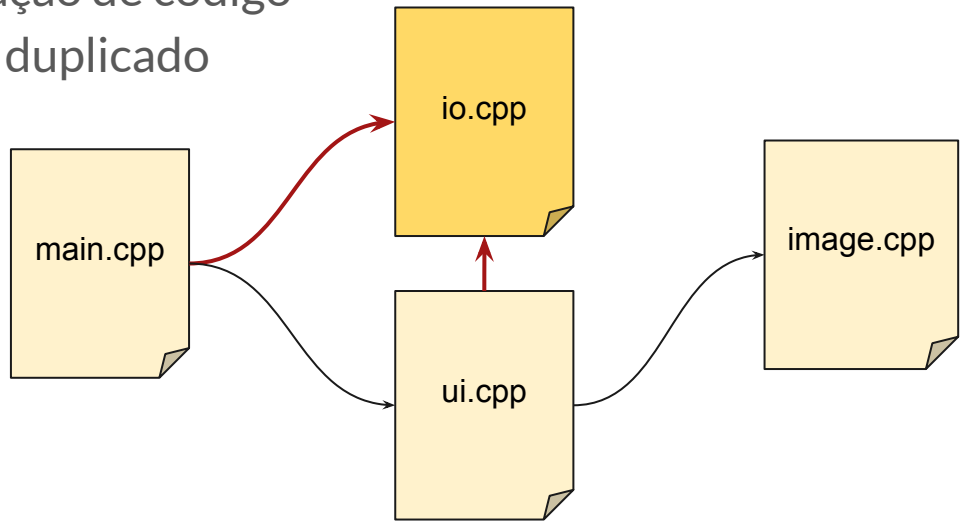
No exemplo ao lado, as definições em `img.cpp` estarão duplicadas e o compilador apresentará erro.



Separação dos arquivos em C++

As definições (classes, constantes, ...) relacionadas entre si devem estar implementadas em um único arquivo `.cpp`

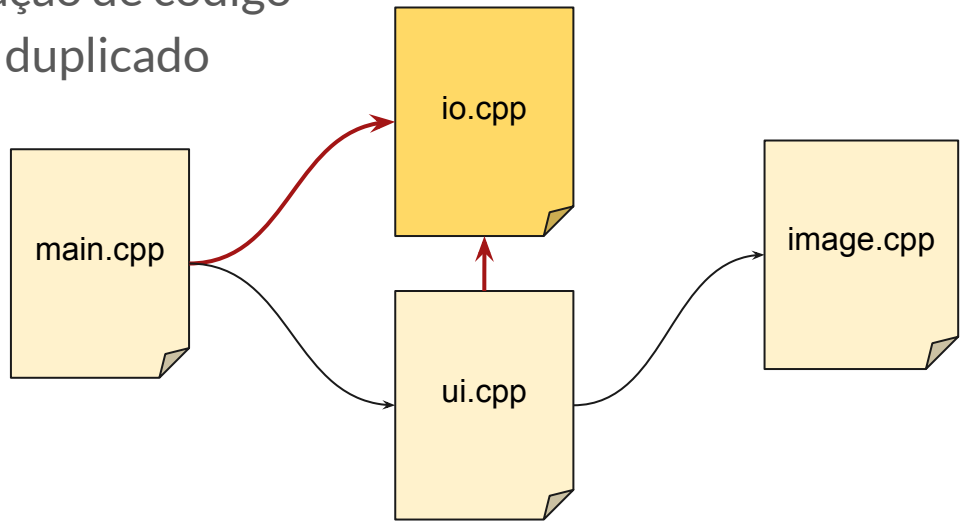
- Porém, não é uma boa prática incluir o `.cpp` através do #include, pois pode haver duplicação de código
- Ex: código `io.cpp` duplicado



Separação dos arquivos em C++

As definições (classes, constantes, ...) relacionadas entre si devem estar implementadas em um único arquivo `.cpp`

- Porém, não é uma boa prática incluir o `.cpp` através do #include, pois pode haver duplicação de código
- Ex: código `io.cpp` duplicado

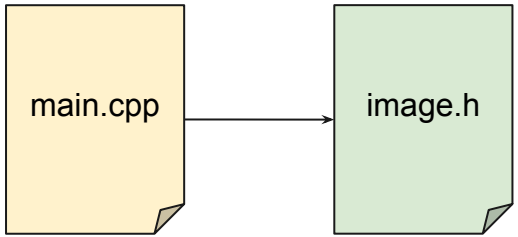


Solução: arquivos cabeçalhos (.h)

Arquivos que possuem apenas definições (e não implementações):

- Assinatura de funções
- Definição de tipos (struct, enum, class, ...)
- Constantes

Obs: há como assegurar que o arquivo será processado uma única vez (mais adiante)



O include deve usar as “” para incluir arquivos locais

```
#include "image.h"
```

Exemplo

image.h

```
class Image {
  int width, height;

public:
  Image(int w = 0, int h = 0): width(w), height(h) {}
  int get_width() { return width; }
  int get_height() { return height; }
  void read(string filename);
};
```

main.cpp

```
#include <iostream>
#include "image.h"
using namespace std;

int main() {
  Image img;
  img.read(“arquivo.ppm”);
  cout << img.get_width() << " x " << img.get_height() << endl;
  return 0;
}
```

Evitando duplicação de definições

Exemplo 2

main.cpp

```
#include <iostream>
#include “point.h”
using namespace std;
void main() {
  Point p1 = { 5.3, 7.8 };
  Point p2 = { -8.2, 9.1 };
  cout << distance(p1, p2) << endl;
}
```

point.h

```
struct Point {
  float x, y;
};

float distance(Point a, Point b);
```

point.cpp

```
#include <cmath>
#include “point.h”

float distance(Point a, Point b) {
  float dx = a.x - b.x;
  float dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}
```

Compilação

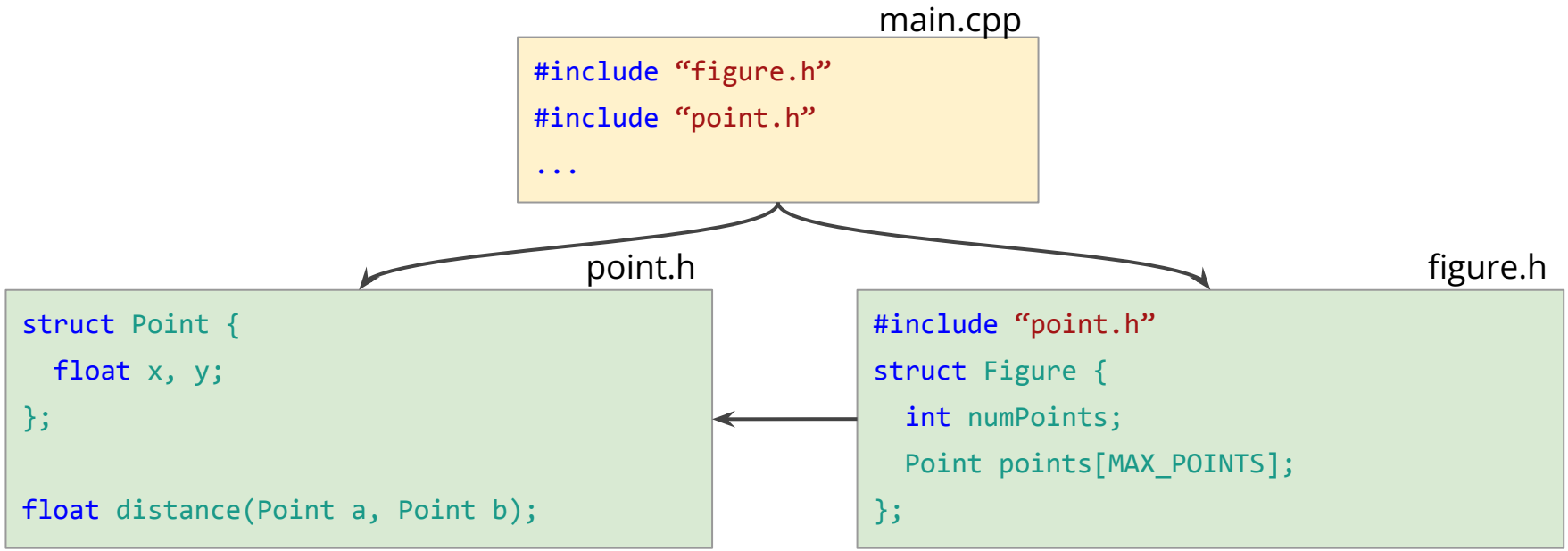
Não há ligação entre os .cpp, então como saber que eles fazem parte do mesmo programa????

⇒ No momento da compilação! Compila-se apenas os arquivos .cpp

```
$ g++ main.cpp modulo1.cpp modulo2.cpp -o executavel
```

Mais um problema...

E se um arquivo .h precisar de outro arquivo .h e um .cpp incluir os dois?
No exemplo, o tipo `Point` será definido duas vezes.



Solução: include guard

Uso de diretivas para o compilador não processar um arquivo.

- `#ifndef` → verificar se uma macro foi definida
- `#define` → define uma macro
- `#endif` → termina um bloco de código iniciado por `#ifndef` (ou `#ifdef`)

Na 1ª vez que for processar, `POINT_H` não foi definido, então `#ifndef POINT_H` é verdadeiro e o bloco é processado.

Na 2ª vez, `POINT_H` já foi definido, então `#ifndef` é falso e o bloco não é processado.

```
#ifndef POINT_H
#define POINT_H

typedef struct {
  float x, y;
} Point;

float distance(Point a, Point b);

#endif
```