

# Programação e Desenvolvimento de Software 2

## TAD - Modularização

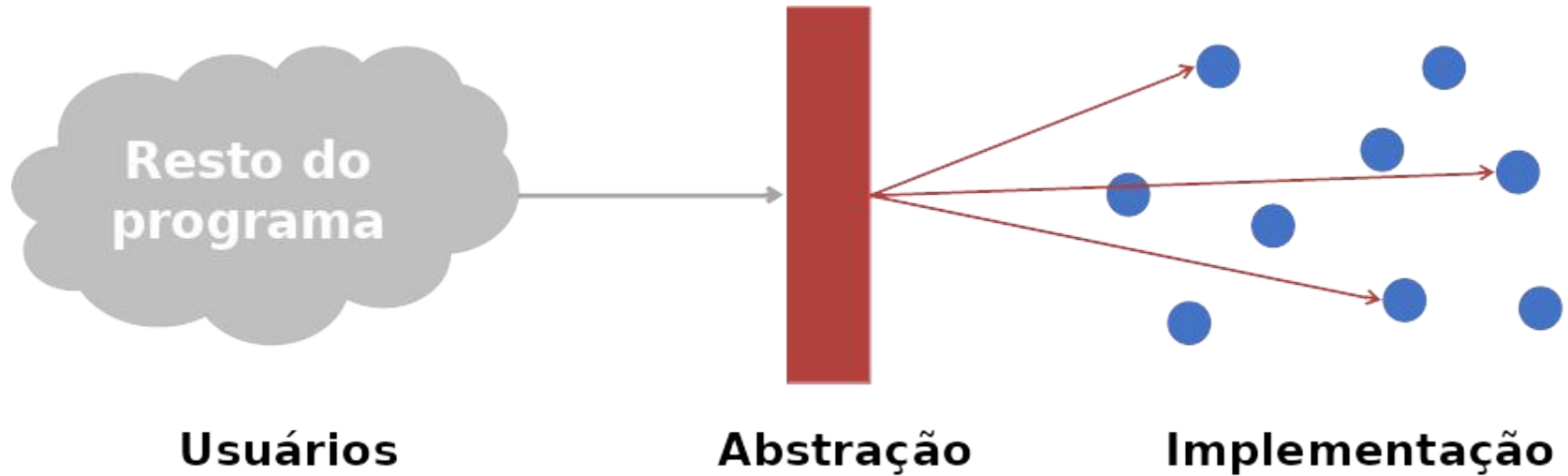
---

Camila Laranjeira  
[camilalaranjeira@ufmg.br](mailto:camilalaranjeira@ufmg.br)

# Projeto Modular

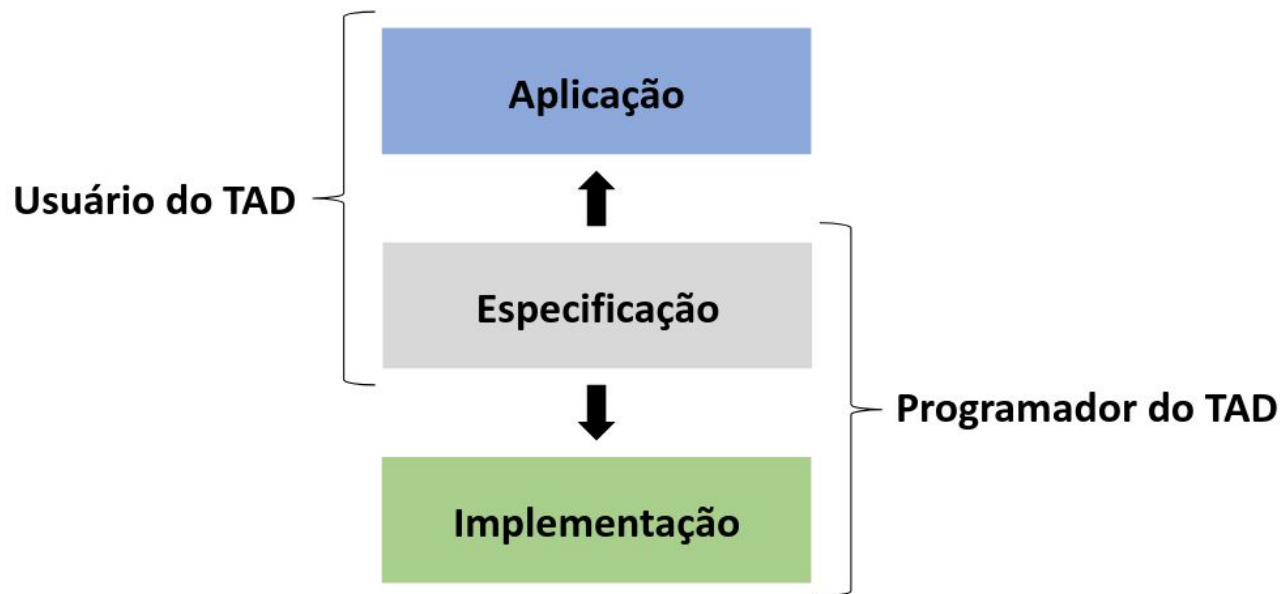
---

# Lembrando do nosso objetivo



Com TADs queremos que o resto do programa seja cliente, ou seja, apenas use as operações.

# Lembrando do nosso objetivo



Com TADs queremos que o resto do programa seja cliente, ou seja, apenas use as operações.

# Projeto Modular

## Propriedades

- Decomposição
- Composição
- Significado fechado
- Continuidade
- Proteção

# Projeto Modular

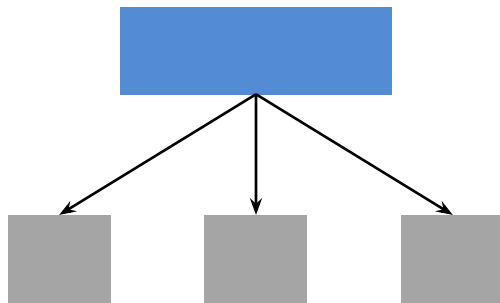
## Decomposição

- **Nível de Projeto**
  - Capaz de separar uma tarefa em subtarefas, que podem ser abordadas separadamente
- **Nível de Software**
  - Capaz de trabalhar em cada um dos módulos do software independente dos outros módulos
- **O que pode prejudicar a decomposição?**

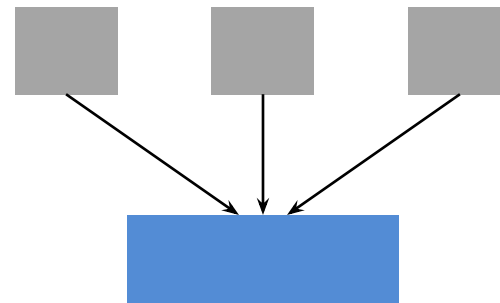
# Projeto Modular

## Composição

- Capacidade de conseguir combinar de forma livre diferentes elementos de software



Decomposição

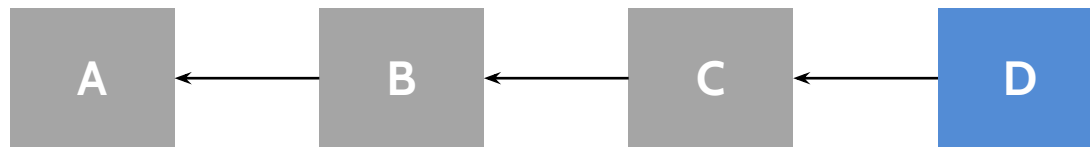


Composição

# Projeto Modular

## Significado fechado

- O programa deve ser compreendido por um leitor (usuário) que não possui acesso a outras (ou todas) partes do sistema



Problema: dependência sequencial.



# Projeto Modular

## Continuidade

- Alterações em parte da especificação demandam alterações em poucos módulos
- Bom exemplo
  - Utilização de constantes
- Mau exemplo
  - Dependência forte de um único módulo

# Projeto Modular

## Proteção

- Situações anormais em tempo de execução não são propagadas para outros módulos
  - Erros não detectados em outras partes
- Extensibilidade
- Validação dos dados nos módulos
  - Tipos, asserções, exceções

# Modularizando um TAD simples

---

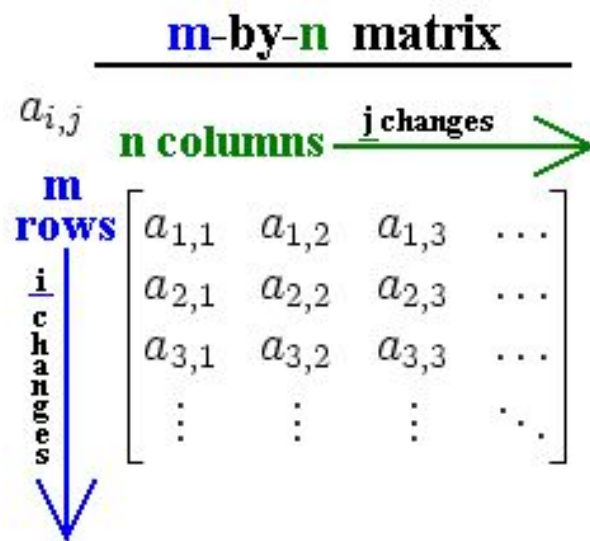
# Cabeçalhos

- Em C++, usamos arquivos de cabeçalhos
- Os mesmos descrevem os módulos

# Problema I

## Matriz

- Vamos criar um módulo matriz
- A mesma representa uma matriz que vai ser alocada dinamicamente



# Iniciando do .h

```
#ifndef PDS2_MATRIZ_H
#define PDS2_MATRIZ_H
struct Matriz {
    // Dados
    int **_dados;
    int _n_linhas;
    int _n_colunas;

    // Construtor
    Matriz(int n_linhas, int n_colunas);
    // Destrutor
    ~Matriz();

    // Métodos
    void seta(int i, int j, int v); //  $M[i][j] = v$ 
    int valor(int i, int j); // retorna valor  $i, j$ 
    Matriz soma(Matriz &outra); // soma duas matrizes
};
#endif
```


Header guard, evitar erros

Note que não temos código

Já será explicado

# Header Guards

```
#ifndef PDS2_MATRIZ_H
#define PDS2_MATRIZ_H
struct Matriz {
    // Omitindo implementação
};
#endif
```



Header guard, evitar erros

<https://www.learncpp.com/cpp-tutorial/header-guards/>

# Iniciando do .h

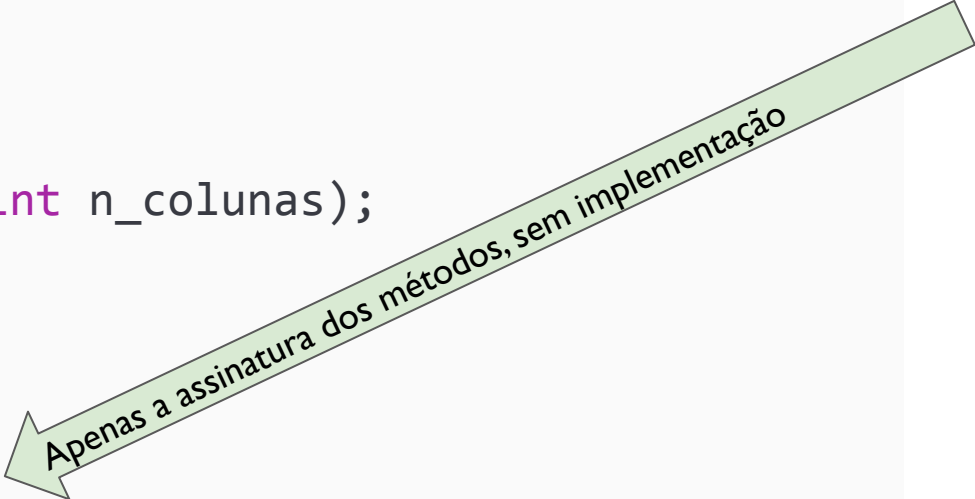
```
#ifndef PDS2_MATRIZ_H
#define PDS2_MATRIZ_H

struct Matriz {
    // Dados
    int **_dados;
    int _n_linhas;
    int _n_colunas;

    // Construtor
    Matriz(int n_linhas, int n_colunas);
    // Destrutor
    ~Matriz();

    // Métodos
    void seta(int i, int j, int j); //  $M[i][j] = v$ 
    int valor(int i, int j); // retorna valor  $i, j$ 
    Matriz soma(Matriz &outra); // soma duas matrizes
};

#endif
```



Apenas a assinatura dos métodos, sem implementação



# Implementando o .cpp

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

Note o include do módulo matriz

```
void Matriz::seta(int i, int j, int v) {
    _dados[i][j] = v;
}
```

```
int Matriz::valor(int i, int j) {
    return _dados[i][j];
}
```

```
Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```

# Implementando o .cpp

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

```
void Matriz::set(int i, int j, int v) {
    _dados[i][j] = v;
}
```

Retorne int

Classe Matriz

Método valor

Params

```
int Matriz::valor(int i, int j) {
    return _dados[i][j];
}
```

```
Matriz Matriz::soma(Matriz &outra) {
    Matriz retorno = Matriz(_n_linhas, _n_colunas);
    for (int i = 0; i < _n_linhas; i++) {
        for (int j = 0; j < _n_colunas; j++) {
            retorno._dados[i][j] = valor(i, j) + outra.valor(i, j);
        }
    }
    return retorno;
}
```

# Arquivo main

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

int main(void) {
    Matriz m1(2, 2);
    Matriz m2(2, 2);

    std::cout << m1.valor(0, 0) << std::endl;
    std::cout << m2.valor(0, 0) << std::endl;

    m1.seta(0, 0, 1);
    std::cout << m1.valor(0, 0) << std::endl;

    m2.seta(0, 0, 2);
    std::cout << m2.valor(0, 0) << std::endl;

    Matriz m3 = m1.soma(m2);
    std::cout << m3.valor(0, 0) << std::endl;
}
```

# Arquivo main

- Faz uso dos módulos
- Não se preocupa como a matriz é implementada, cliente do módulo

Note que não implementamos tudo aqui nos slides. Erro de compilação na prática. Estamos indo por partes!

# Compilando

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

- Note que passamos dois arquivos
- O do main e o da matriz

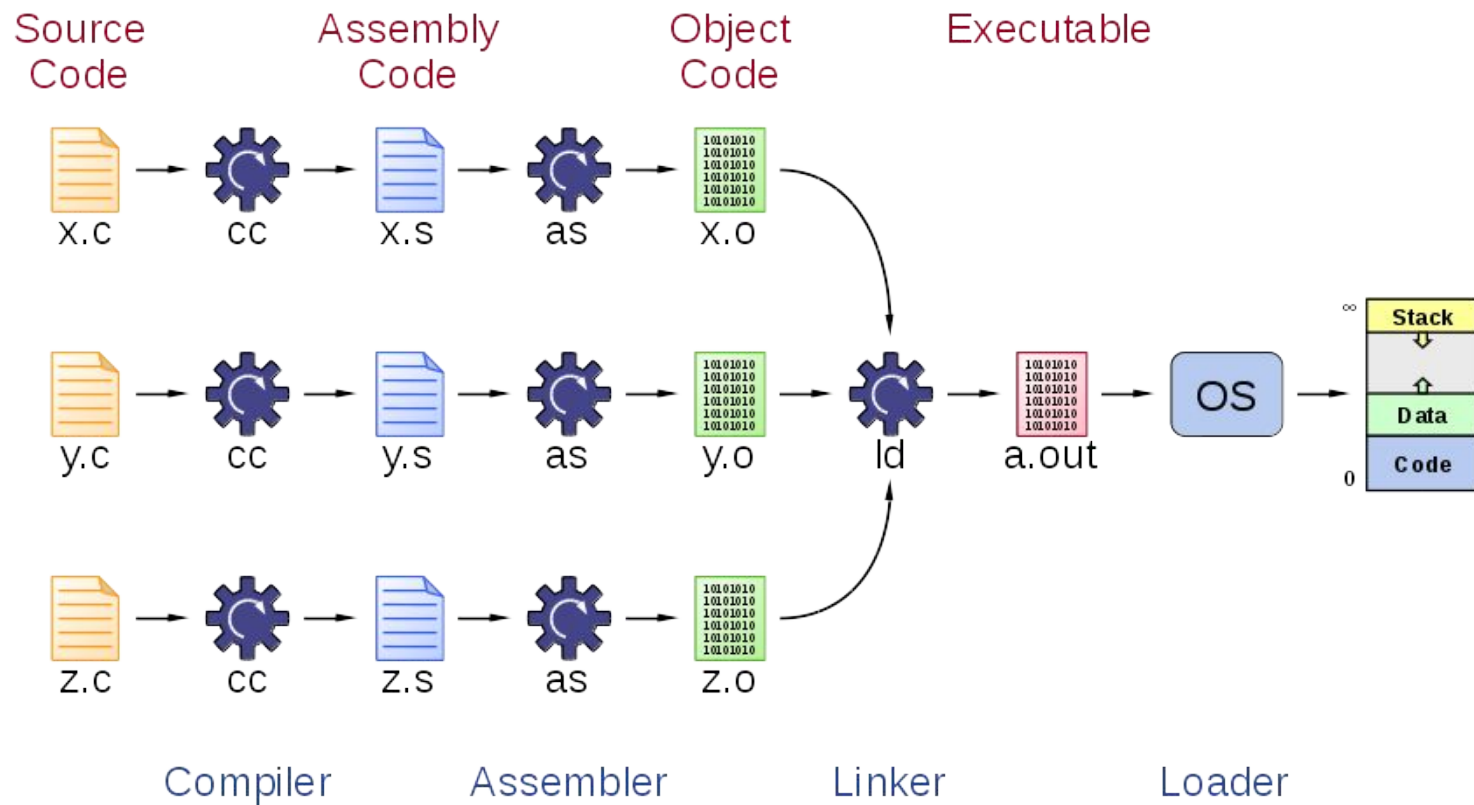
# Compilando

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

- Note que passamos dois arquivos
- O do main e o da matriz
- Ou usamos o \* para compilar todos os .cpp daquele diretório

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

# Compilação



# Construtores v. Destrutores

O nome diz tudo

- Procedimentos de inicialização
  - Usados apenas na criação de um novo objeto
- Procedimentos de destruição
  - Usados para liberar os recursos adquiridos na criação e utilizados por um certo objeto



# Construtores v. Destrutores

O nome diz tudo

- Destrutores tem um papel similar
- Liberar toda a memória que o objeto pode ter alocado
  - isto é, chamadas para **new**
- Também é útil para fechar recursos
  - Arquivos
  - Dentre outros

# Aquisição de Recurso é Inicialização

- Qual o motivo do destrutor?
- Uma boa prática é que todo objeto cuide da memória que o mesmo alocou

# Código Construtor e Destrutor

```
#include <iostream>
#include "matriz.h"
Matriz::Matriz(int n_linhas, int n_colunas) {
    std::cout << "Construindo uma matriz" << std::endl;
    _dados = new int*[n_linhas]();
    for (int i = 0; i < n_linhas; i++) {
        _dados[i] = new int[n_colunas];
    }
    _n_linhas = n_linhas;
    _n_colunas = n_colunas;
}
```

# Código Construtor e Destrutor


```
#include <iostream>
#include "matriz.h"
/...
Matriz::~Matriz() {
    std::cout << "Destruindo uma matriz" << std::endl;
    for (int i = 0; i < _n_linhas; i++) {
        delete[] _dados[i];
    }
    delete[] _dados;
}
```

# Código

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

Matriz::Matriz(int n_linhas, int n_colunas) {
    std::cout << "Construindo uma matriz" << std::endl;
    _dados = new int*[n_linhas]();
    for (int i = 0; i < n_linhas; i++) {
        _dados[i] = new int[n_colunas];
    }
    _n_linhas = n_linhas;
    _n_colunas = n_colunas;
}

Matriz::~Matriz() {
    std::cout << "Destruindo uma matriz" << std::endl;
    for (int i = 0; i < _n_linhas; i++) {
        delete[] _dados[i];
    }
    delete[] _dados;
}
```



Tipo \*\*.  
Similar à malloc(n\*sizeof(int\*));

# Código

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

Matriz::Matriz(int n_linhas, int n_colunas) {
    std::cout << "Construindo uma matriz" << std::endl;
    _dados = new int*[n_linhas]();
    for (int i = 0; i < n_linhas; i++) {
        _dados[i] = new int[n_colunas];
    }
    _n_linhas = n_linhas;
    _n_colunas = n_colunas;
}

Matriz::~Matriz() {
    std::cout << "Destruindo uma matriz" << std::endl;
    for (int i = 0; i < _n_linhas; i++) {
        delete[] _dados[i];
    }
    delete[] _dados;
}
```

Lembrando, cada new → 1 delete

# Exemplificando Destrutores

```
#include "matriz.h"
```



```
int main(void) {  
    Matriz *matriz = new Matriz(100, 100);  
    delete matriz;  
  
    Matriz matriz2(100, 100);  
    return 0;  
}
```

```
$ ./main
```

```
Construindo matriz
```

# Exemplificando Destrutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;

    Matriz matriz2(100, 100);
    return 0;
}
```

```
$ ./main
Construindo matriz
Destruindo matriz
```



# Exemplificando Destrutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;
    Matriz matriz2(100, 100);
    return 0;
}
```

```
$ ./main
Construindo matriz
Destruindo matriz
Construindo matriz
```

# Exemplificando Destrutores

```
#include "matriz.h"

int main(void) {
    Matriz *matriz = new Matriz(100, 100);
    delete matriz;

    Matriz matriz2(100, 100);
    return 0;
}
```



```
$ ./main
Construindo matriz
Destruindo matriz
Construindo matriz
Destruindo matriz
```

# Destrutores

São chamados sempre que o objeto é desalocado

- Destrutores são chamados tanto para:
  - Objetos no heap
    - Depois de um delete
  - Objetos no stack
    - Depois que a função termina

# Destrutores

São chamados sempre que o objeto é desalocado

Lembrando que

- O computador cuida do stack
- Você cuida do heap
- Por isso fazemos o destrutor, a matriz é alocada dinamicamente no heap!