



PCS 3111

Laboratório de Programação Orientada a Objetos para Engenharia Elétrica

Aula 4: Encapsulamento

Escola Politécnica da Universidade de São Paulo

Agenda

1. Encapsulamento
 - Métodos e Atributos Públicos e Privados
2. Métodos Setters/Getters
3. Organização de Arquivos
 - Processo de Compilação e Ligação
 - Diretivas de Compilação
4. Coesão e Acoplamento
5. Vetor de objetos

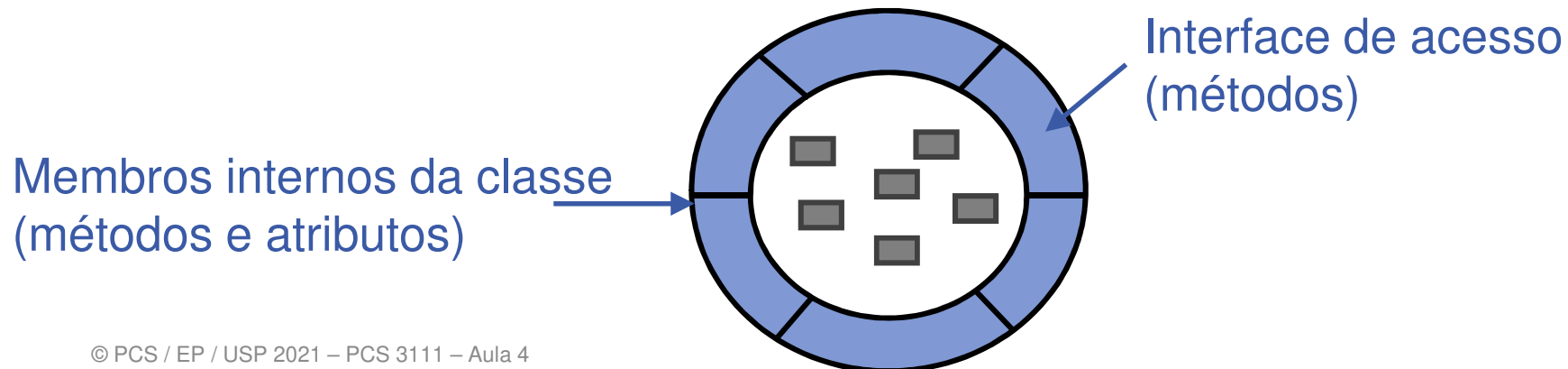
Encapsulamento

Introdução

- Uma boa prática no desenvolvimento de software é quebrar a solução em **módulos**
 - Aula passada
- Um dos princípios para modularização é a **ocultação da informação**
 - Esconde **detalhes** do módulo de outros módulos
 - Separa o **propósito** de sua **implementação**
 - Especifica um módulo do **ponto de vista externo**
 - Restringe o acesso a detalhes de implementação e a qualquer estrutura de dados do módulo

Encapsulamento

- Na OO a ocultação de informação é definida como **encapsulamento**
 - Capacidade de ocultar detalhes de outras classes que manipulam a classe
 - Esconde detalhes **não essenciais** para compreender as características da classe
- Permite o acesso a membros **internos** apenas por uma **interface** bem definida



Encapsulamento em C++

- Modos de visibilidade público e privado
 - Forma de restrição de acesso a *membros*
 - Atributos e métodos
 - Modos
 - public: objetos de qualquer classe podem acessar
 - private: acesso restrito a apenas objetos da classe
 - (Veremos um outro modo de visibilidade futuramente)

```
class A {  
    private:  
        // Atributos e método privados  
  
    public:  
        // Atributos e método públicos  
};
```

Encapsulamento em C++

▪ *Exemplo*

Membros privados

Membros públicos

```
...  
4  class Relogio {  
5  private:  
6      int hora = 0;  
7      int minuto = 0;  
8  public:  
9      void inicializar(int hora, int minuto);  
10     void imprimir();  
11 };  
EX01
```

▪ O modo de visibilidade *default* é o privado

```
class X {  
    int y();  
};
```

equivalente a



```
class X {  
private:  
    int y();  
};
```

• *Exemplo*

```
X *x = new X;
```

```
int resultado = x->y();
```



Erro de compilação

Métodos *Setters* e *Getters*

Boas práticas de programação

- Atributos *devem* ser privados
 - Vantagens
 - Verificação dos dados
 - Independência da implementação
 - Tipo usado, codificação usada, forma de persistência etc.
 - Evita que mudanças nos atributos afetem as demais classes
- Métodos *auxiliares* *devem* ser privados
 - Não refletem um comportamento externo relevante

Getters / Setters

- Se for necessário acessar atributos externamente, podem-se criar métodos específicos
 - *Accessors* (ou *getters*): para recuperar o valor
 - Prefixo **get**
 - Não precisa de argumentos
 - *Em geral* retorna o mesmo tipo do atributo
 - *Mutators* (ou *setters*): para alterar o valor
 - Prefixo **set**
 - Não precisa retornar um valor
 - O parâmetro é o valor a ser colocado no atributo

Nem todo atributo precisa de *getter* e *setter*!
Somente devem ser criados **se necessário**.

Exemplo

EX02

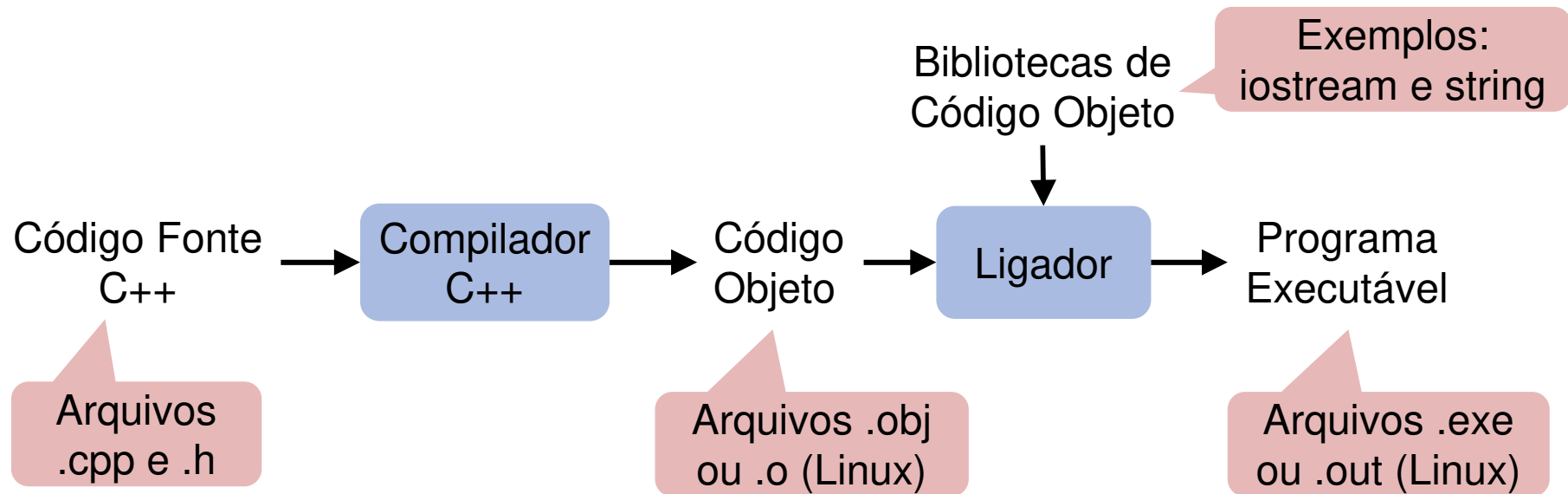
```
...
4  class Aluno {
5  private:
6      string nome;
7      int numeroUsp;
8  public:
9      void setNome (string nome);
10     string getNome();
11     void setNumeroUsp(int numeroUsp);
12     int getNumeroUsp();
13 };
```

Atributos privados

```
15 void Aluno::setNome(string nome) {
16     this->nome = nome;
17 }
18
19 string Aluno::getNome() {
20     return nome;
21 }
22
23 void Aluno::setNumeroUsp(int numeroUsp) {
24     this->numeroUsp = numeroUsp;
25 }
26
27 int Aluno::getNumeroUsp() {
28     return numeroUsp;
29 }
30
31 int main() {
32     Aluno *a = new Aluno;
33     a->setNome("Ana");
34     a->setNumeroUsp(987654321);
35     cout << a->getNome() << endl;
36     return 0;
37 }
```

Organização de Arquivos

Processo de compilação



- O compilador transforma o código fonte em código objeto
- O ligador liga o código objeto com outros códigos objeto e com bibliotecas já existentes
 - *Exemplo:* bibliotecas de entrada/saída

Processo de compilação

- Um projeto pode ter vários códigos objeto
 - Cada código objeto é um **módulo**
 - Só precisa ser **recompilado** se
 - Módulo for alterado
 - Módulos dos quais ele depende tiverem a **interface** alterada
- **Boa prática**
 - Crie um arquivo **separado** para cada definição e para cada implementação
 - Cada classe se torna um módulo

Definição

- Arquivo de cabeçalho: ".h"
 - *Exemplo:* Relogio.h

```
1  class Relogio {  
2  private:  
3      int hora = 0;  
4      int minuto = 0;  
5  public:  
6      void inicializar (int hora, int minuto);  
7      int getHora();  
8      int getMinuto();  
9      void imprimir();  
10 };
```

EX03

Implementação

- Arquivo ".cpp"
 - Necessário fazer um `#include` do ".h"
 - *Exemplo*: `Relogio.cpp`
 - Necessário implementar **todos** os métodos

```
1  #include "Relogio.h"
2  #include <iostream>
3
4  using namespace std;
5
6  void Relogio::inicializar (int hora, int minuto) {
7      if (hora < 0 || hora > 23) this->hora = 0;
8      else this->hora = hora;
9
10     if (minuto < 0 || minuto > 59) this->minuto = 0;
11     else this->minuto = minuto;
12 }
...
```

← Incluindo a definição
Para usar
o cout

EX03

Implementação

- Arquivo main.cpp

```
1  #include "Relogio.h"
2
3  int main() {
4      Relogio *r = new Relogio;
5      r->inicializar (10, 5);
6      r->imprimir();
7      return 0;
8  }
```

EX03

- Para compilar e *ligar* manualmente

```
g++ -c -std=c++11 Relogio.cpp -o Relogio.o
g++ -c -std=c++11 main.cpp -o main.o
g++ Relogio.o main.o -o main.exe
```

Diretivas de Compilação

- Se iniciam com o símbolo '#' em C++
 - *Exemplo:* #include, #define e #ifndef
- São utilizadas em uma etapa inicial de compilação
 - Pré-processamento
 - Nessa etapa, tais diretivas são processadas e incluem ou removem código fonte do arquivo compilado

Diretivas de Inclusão

- Indica para o compilador *onde um arquivo necessário* está
 - O arquivo pode estar em pastas diferentes
 - Quando se inclui cabeçalhos da biblioteca padrão, usa-se o "<" e ">"
 - O compilador procura o arquivo onde as bibliotecas *ficam*

```
#include "Relogio.h" ← Do projeto  
#include <iostream> ← Da biblioteca padrão
```

- Arquivos de cabeçalho também podem fazer inclusões
 - *Exemplo:* Lampada.h inclui o Relogio.h

Diretiva `ifndef`

- *Problema:* múltipla inclusão de um cabeçalho
 - *Exemplo:* tanto a `Lampada` quanto o `main` incluem o `Relogio`
 - Gera um erro de compilação!
 - (Compiladores de outras linguagens evitam esse erro)
- *Solução:* diretiva `#ifndef`

```
#ifndef RELOGIO_H  
#define RELOGIO_H
```

} Se `RELOGIO_H` não estiver
definido, define-o

```
class Relogio {  
    ...  
};
```

```
#endif
```

← Fim do `ifndef`

Coesão e Acoplamento

Princípios básicos de projeto

- Existem dois princípios básicos para qualquer projeto (*design*) de software
 - Independem do paradigma de programação
 - Importantes para programação **modular**
 - Acoplamento e Coesão

Coesão

- Grau em que as responsabilidades de um único componente formam uma **unidade significativa**
 - Membros estão relacionados entre si
 - Estão relacionados a um tema comum e tem o mesmo objetivo
 - Métrica
 - Alta: membros fortemente relacionados
 - Baixa: membros pouco relacionados

Dica: cada classe deve tratar de um único conceito do mundo real

Coesão

■ Exemplo

```
...  
7  class Aluno {  
8  private:  
9      string nome;  
10     double p1, p2, p3;  
11  
12     public:  
13         void setNome(string nome);  
14         string getNome();  
15         void setNotas(double p1,  
16                        double p2, double p3);  
17         double getMedia();  
18     };  
19
```

EX04



```
...  
7  class Aluno {  
8  private:  
9      string nome;  
10  
11     public:  
12         void setNome(string nome);  
13         string getNome();  
14     };  
15  
...  
4  class Avaliacao {  
5  private:  
6      double p1, p2, p3;  
7  public:  
8      void setNotas(double p1,  
9                     double p2, double p3);  
10     double getMedia();  
11 };  
12
```

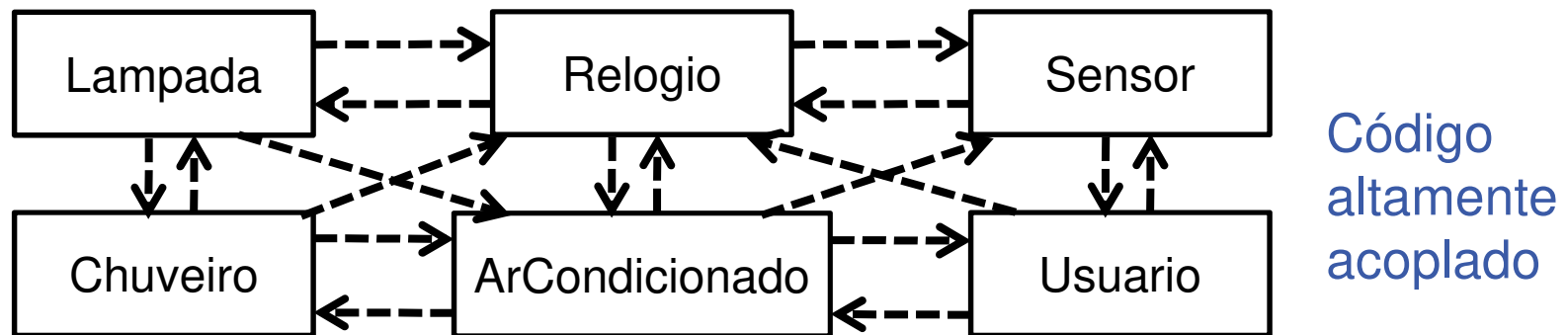
EX05

Dois conceitos

- Aluno
- Avaliação do aluno

Acoplamento

- Grau de inter-relacionamento entre as classes
 - Métrica: alto e baixo acoplamento
 - Existem vários tipos de dependências
 - “Força” diferente
 - Em C++ são evidentes pelo `#include`



- Deve-se **minimizar o acoplamento**
 - A classe deve depender o **mínimo possível** de outras classes

Exemplo

- Lâmpada: ligar se for de noite

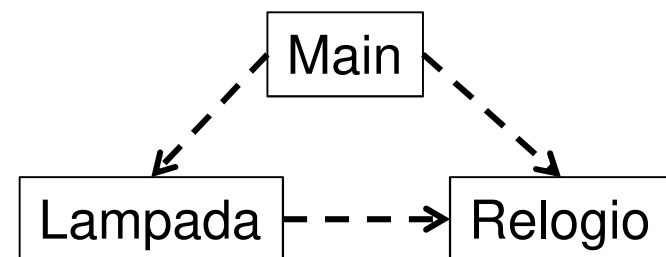
EX06

```
...
4  #include "Relogio.h"
5
6  class Lampada {
7  private:
8      bool acesa;
9  public:
10     void ligar();
11     void desligar();
12     void imprimir();
13     void ligarSeNoite(Relogio *horaAtual);
14 };
```

Depende do Relogio

```
1  #include <iostream>
2  #include "Lampada.h"
3  #include "Relogio.h"
4
5  using namespace std;
6
7  int main() {
8      Lampada *sala = new Lampada;
9      Relogio *r = new Relogio;
10     r->inicializar(10, 30);
11     sala->ligarSeNoite(r);
12
13     ...
19 }
```

```
...
12 void Lampada::ligarSeNoite(Relogio *horaAtual) {
13     int hora = horaAtual->getHora();
14     if (hora >= 18) ligar();
15     else desligar();
16 }
```



Exemplo

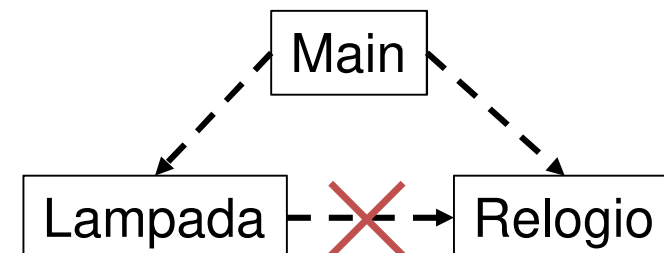
- Uma forma de diminuir o acoplamento

EX07

```
4 class Lampada {
5 private:
6     bool acesa;
7 public:
8     void ligar();
9     void desligar();
10    void imprimir();
11    void ligarSeNoite(bool ehNoite);
12 };
```

```
4 class Relogio {
5 private:
6     int hora = 0;
7     int minuto = 0;
8 public:
9     void inicializar (int hora, int minuto);
10    int getHora();
11    int getMinuto();
12    bool ehNoite();
13    void imprimir();
14 };
```

```
1 #include <iostream>
2 #include "Lampada.h"
3 #include "Relogio.h"
4
5 using namespace std;
6
7 int main() {
8     Lampada *sala = new Lampada;
9     Relogio *r = new Relogio;
10    r->inicializar(10, 30);
11    sala->ligarSeNoite(r->ehNoite());
12    sala->imprimir();
13    ...
19 }
```



Vetor de objetos

Vetor de objetos

- Em C++ é possível criar vetores de qualquer tipo
 - Inclusive de classes
 - Sintaxe: <Tipo> *nome[tamanho]
 - Note que é um **vetor de ponteiros** do tipo
 - *Exemplo*

EX08

```
...
6  int main() {
7      Relogio *relogios[2];
8      relogios[0] = new Relogio;
9      relogios[0]->inicializar(10, 20);
10     relogios[1] = new Relogio;
11     relogios[1]->inicializar(18, 30);
12
13     for (int i = 0; i < 2; i++)
14         relogios[i]->imprimir();
15
16     return 0;
17 }
```

Vetor de tamanho 2

Colocando um objeto na posição 1

Bibliografia

- **STROUSTRUP B. Princípios e práticas em programação com C++. Bookman. 2012.**
- **BUDD, T. An Introduction to Object-Oriented Programming. 3rd Edition. Addison-Wesley. 2001.**
- **LARMAN, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition. Addison-Wesley, 2005**