



PCS311

**Laboratório de Programação
Orientada a Objetos para
Engenharia Elétrica**

Aula 5: Construtores e Destrutores

Escola Politécnica da Universidade de São Paulo

Agenda

1. Construtores
2. Destrutores
3. Escopo e gerenciamento de memória
4. Constantes

Construtores

Construtores

- O que fazer para usar uma variável
 - *Criação da variável*: alocação de espaço na memória e ligação deste espaço a um nome
 - *Inicialização da variável*: colocar os valores iniciais e assegurar as condições iniciais necessárias para sua correta utilização

Construtores

- Usualmente a **criação** se dá através de um comando de declaração

Criação →

```
int maximo;
```

- A **inicialização** se dá através de um ou mais comando(s) de atribuição

Inicialização →

```
maximo = 10;
```

- Problemas ocorrem se o usuário esquece de inicializar uma variável
 - Ou se o faz mais de uma vez!

```
while (i < maximo) {...}
```

Construtores

- São métodos chamados automaticamente quando um objeto é criado
 - Contém as instruções para a sua correta inicialização
 - Inicializar atributos com os valores adequados
 - Eventualmente, criar objetos internos ...
 - São métodos como outros quaisquer
 - Com alguns detalhes...
- Garante-se que um objeto *nunca* será utilizado sem que esteja pronto para tal

Construtores em C++

- Declaração do construtor
 - Método com o mesmo nome da classe
 - Não tem tipo de retorno
 - Pode ter parâmetros

Construtor

```
6  class Sensor {  
7  public:  
8  → Sensor(int numero, Residencia *residencia);  
9      void detectarAcao();  
10  
11     int getNumero();  
12 private:  
13     int numero;  
14     Residencia *residencia;  
15 };
```

EX01

Construtores em C++

- Os parâmetros muitas vezes tem o mesmo nome de alguns dos atributos
 - Eles inicializam os atributos
- Como implementar o construtor?
 - Existem duas opções

Construtores em C++

- Implementação do construtor
 - Opção 1

```
6  Sensor::Sensor(int numero, Residencia *residencia) {  
7      this->numero = numero;  
8      this->residencia = residencia;  
9  }
```

EX02

- Diferencia o **atributo** do **parâmetro** usando `this`
 - Permite nomes mais adequados

Construtores em C++

- Implementação do construtor
 - Opção 2 (recomendada)

```
13 Sensor::Sensor (int numero, Residencia *residencia) :  
14     numero (numero), residencia (residencia) {  
15 }
```

EX01

Atributo

Parâmetro

Atributo

Parâmetro

- Ainda usa os nomes dos atributos como parâmetros
 - Facilita a leitura para quem usa o construtor

Construtores em C++

- Chamada do construtor

```
8  Residencia *r = new Residencia (); // Construtor sem parâmetros
9
10 Sensor *s1 = new Sensor(1, r); // Construtor com parâmetros
11 Sensor *s2 = new Sensor(2, r);
```

EX01

- O que ocorreria neste caso?

```
Residencia *r;

Sensor *s1 = new Sensor(1, r);
Sensor *s2 = new Sensor(2, r);
```

NULL

- Representa um ponteiro que aponta para nenhum valor
 - Explicado na Aula 2

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main (int argc, char** argv) {
...
10     int *p2; // endereço indefinido
11     p2 = NULL; // nenhum valor
12
13     if (p2 == NULL) { É possível testar
14         cout << "Null" << endl;
15     }
...
18 }
```

Necessário fazer um
include para usar o **NULL**

(Está definido em
várias bibliotecas)

EX03 – Aula 2

Construtores em C++

- Se quisermos criar um Sensor fora de uma Residência
 - Indicar um objeto vazio usando NULL

```
Sensor *s1 = new Sensor(1, NULL);  
Sensor *s2 = new Sensor(2, NULL);
```

EX02

Construtores em C++

- Se um construtor não for especificado, o C++ cria um construtor padrão (sem parâmetros)

```
4 class Residencia {  
5 public:  
6     Residencia();  
7 };
```

EX02

Equivalente a



```
4 class Residencia {  
5 };
```

EX01

```
7 Residencia::Residencia() {  
8 }
```

- Usa-se um construtor padrão quando não houver nada a ser feito na criação

Observação: se for declarado um construtor sem parâmetros, ele precisará ser implementado!

Destrutores

Destrutores

- Em alguns casos é útil realizar algo ao término do ciclo de vida de um objeto
 - Quando ele deve ser *destruído*
- Analogamente aos construtores, existem métodos chamados **destrutores**
 - São chamados *automaticamente* quando os objetos são destruídos

Destrutores

- Ele deve liberar os *recursos* usados
- Declaração de um destrutor em C++
 - Não tem parâmetros e nem retorno
 - Nome da classe com ~

Destrutor

```
6  class Sensor {  
7  public:  
8      Sensor (int numero, Residencia *residencia);  
9  → ~Sensor();  
    ...  
16 };
```

EX03

Destrutores em C++

- Implementação do destrutor

```
9  Sensor::~~Sensor() {  
10      cout << "Destruído" << endl;  
11  }
```

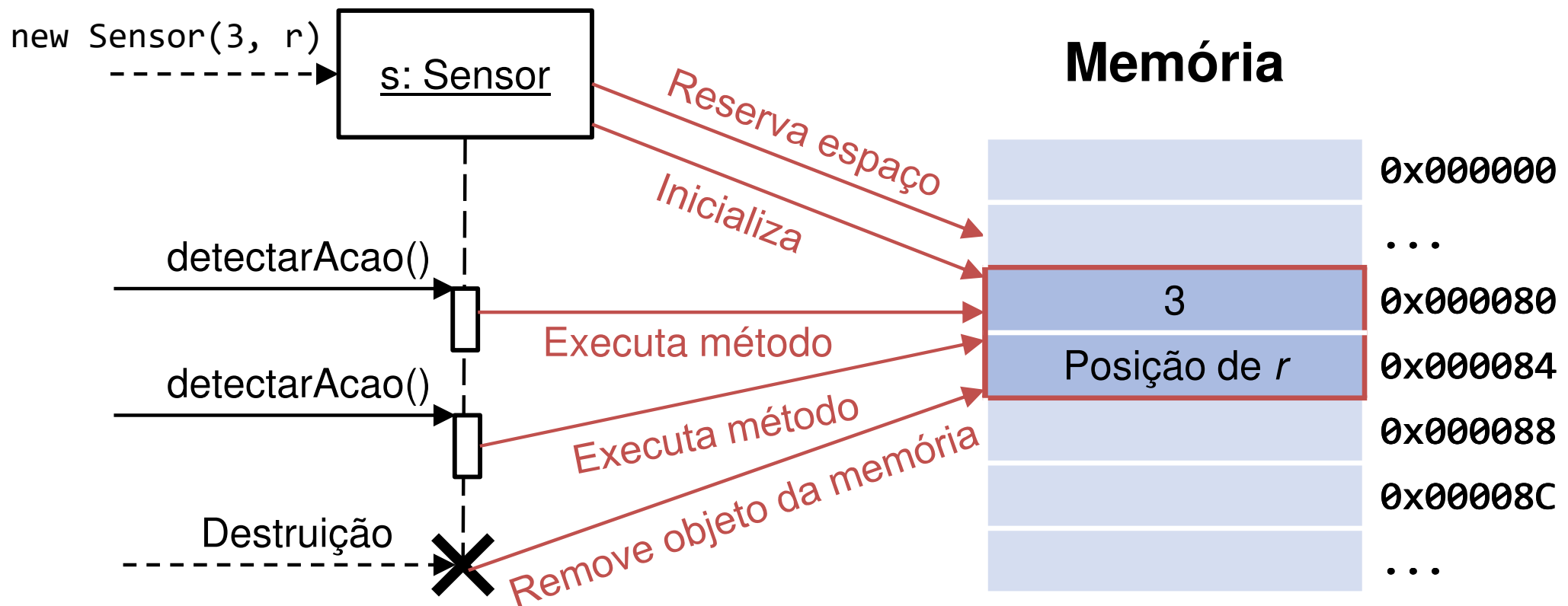
EX03

- Quando um objeto é destruído?
- Quando um destrutor é executado?

Escopo e Gerenciamento de Memória

Ciclo de vida de um objeto

- Criação, uso e destruição



Escopo

- Um bloco define o escopo de uma variável
 - **Bloco**: conjunto de comandos entre "{" e "}"
 - A **variável** e o **objeto** existem naquele bloco

```
1  if (i < maximo) {  
2      int j = maximo;  
3  }  
4  
5  if (j > 2) ...;
```

Aqui a variável j é desalocada da memória

Erro de compilação.

A variável "j" não está declarada neste escopo!

Alocação de memória

- Normalmente o **compilador** gerencia a memória
 - **Alocação estática**
 - Ele cria o *objeto* na declaração da variável
 - Ele destrói o *objeto* no fechamento do escopo
 - Problemas
 - Nem sempre se sabe **em tempo de compilação** quantos objetos são necessários
- ```
1 int quantidade;
2 cin >> quantidade;
3 int vetor[quantidade];
```
- **Problema: qual o tamanho?**
- Nem sempre se quer que o *objeto* seja destruído ao se fechar o escopo
    - *Exemplo:* quando o objeto criado foi guardado em um atributo de um outro objeto

# Alocação de memória

- C++ permite realizar alocação dinâmica de memória
  - Usa uma área especial de memória
    - Chamada *heap* / *free store* / memória dinâmica
- Programador deve gerenciar a memória
  - Alocar o elemento → **new**
  - Desalocar o elemento → **delete**

**Observação:** Em algumas linguagens o *ambiente* gerencia a destruição dos objetos

- *Exemplo:* Java, Perl, Python e C#
- Compromisso entre **controle** X **facilidade**

# Alocação de memória

- **New**: alocação no *heap*
  - Retorna um identificador para o elemento criado
  - Chama o construtor

```
8 int *p = new int;
9 Residencia *r = new Residencia (); // Construtor sem parâmetros
10 Sensor *s = new Sensor(5, r); // Construtor com parâmetros
```

EX04

- **Delete**: desalocação do elemento no *heap*
  - Chama o destrutor

```
8 int *p = new int;
...
12 delete p;
```

```
10 Sensor *s = new Sensor(5, r);
...
14 delete s;
```



# Alocação de memória

## ■ Alocação dinâmica de vetores

```
17 int maximo;
18 cin >> maximo;
19
20 int *inteiros = new int[maximo];
21 Sensor **sensores = new Sensor*[maximo];
22
23 for (int i = 0; i < maximo; i++) {
24 inteiros[i] = i + 1;
25 sensores[i] = new Sensor(i, NULL);
26 }
```

EX04

## ■ Desalocação de vetores: delete[]

```
33 delete[] inteiros;
34 delete[] sensores;
```

EX04

# Gerenciamento de memória

- Regra geral

Toda a vez que algo for **alocado** (new), ele deve ser em algum momento **desalocado** (delete)!

- Problemas comuns

- Objetos criados (**new**) mas não apagados (**delete**)
  - *Memory leak*
- Objeto desalocado prematuramente
  - A área de memória pode ter outro uso!
- Objeto desalocado mais de uma vez
  - Não se sabe o que será apagado na segunda vez

# Constantes

# Constantes

- É possível definir constantes em C++
  - Modificador **const**
    - O valor da constante deve ser atribuído na declaração da variável

```
17 const int x = 0; // x não pode ser alterado
18 const Sensor *s = new Sensor(1, new Residencia()); // s não pode
19 // ser alterado
20 x = 3;
21 s->setNumero(4);
```

EX05

**Erro de compilação.**  
**Inválido alterar uma constante.**

O `#define` não cria uma constante. Ele é um comando para o pré-processador fazer uma substituição de um **texto** por **outro**. Isso pode gerar problemas!

# Constantes

- Um parâmetro pode ser uma constante
  - Não pode ser alterado pelo método
  - Verificação de erros em tempo de compilação
  - *Exemplo*

```
8 void processar(const Sensor *s) {
9 s->setNumero(4);
10 }
```

EX05

**Erro de compilação.**  
**Objeto não pode ser alterado**

# Constantes

- O retorno de uma função / método também pode ser uma constante

```
12 const int* criarVetor() {
13 return new int[3];
14 }
```

EX05

→ **Compilador obriga que a variável seja um const int \***

```
29 const int *vetor = criarVetor();
30 vetor[0] = 3;
```

EX05

**Erro de compilação.  
Não pode ser alterado**

# Constantes

- É possível definir que um método não pode alterar o objeto

```
6 class Sensor {
7 public:
8 Sensor(int numero, Residencia *residencia);
9 ~Sensor();
10 void detectarAcao() const;
...
17 };
```

EX05

Não pode alterar os atributos

```
22 void Sensor::detectarAcao() const {
23 this->numero = 0;
24 }
```

Erro em tempo de compilação

# Bibliografia

- BUDD, T. **An Introduction to Object-Oriented Programming**. Addison-Wesley, 3rd ed. 2002. Capítulo 5.
- LAFORE, R. **Object-Oriented Programming in C++**. Sams, 4th ed. 2002. Capítulo 6.
- SAVITCH, W. **C++ Absoluto**. Pearson, 1st ed. 2003. Seções 7.1 e 10.3.