

# IMD0030

# LINGUAGEM DE PROGRAMAÇÃO I

Aula 09 – Construtores, Destrutores e Membros Estáticos  
(material baseado nas aulas do Prof. Silvio Sampaio)

---

# Objetivos desta aula

- Introduzir os conceitos de construtores, destrutores e membros estáticos em classes
  - Para isso, estudaremos:
    - Como implementar construtores, destrutores e membros estáticos utilizando a linguagem C++
  - Ao final da aula, espera-se que o aluno seja capaz de:
    - Compreender a importância da criação de construtores e destrutores
    - Compreender a utilização de membros estáticos
    - Implementar classes com construtores, destrutores e membros estáticos em C++
-

---

# Aulas anteriores

- Vimos como **criar classes e instanciar objetos** utilizando a linguagem C++
  - Classes possuem atributos e métodos como membros
  - A instanciação estática de um objeto de uma classe é feita de forma similar à declaração de uma variável em C++; no caso dinâmico, utiliza-se o operador **new**.

```
class Retangulo {  
    private:  
        int largura;  
        int altura;  
  
    public:  
        int getLargura();  
        void setLargura(int l);  
        int getAltura();  
        void setAltura(int a);  
};
```

```
#include <iostream>  
#include "retangulo.h"  
  
int main() {  
    Retangulo r;  
    r.setLargura(10);  
    r.setAltura(5);  
    std::cout << "Largura = " << r.getLargura();  
    std::cout << ", Altura = " << r.getAltura();  
  
    return 0;  
}
```

---

---

# Problema

- Como garantir que os atributos de um objeto sejam devidamente inicializados?
  - Solução inicial:
    - Instanciar o objeto
    - Invocar os respectivos métodos *setters* para atribuir os valores de cada atributo
    - Ainda há o problema de garantir que todos os métodos *setters* sejam corretamente invocados
  - Solução definitiva: implementação e uso de **construtores**
-

---

# Construtores

- Métodos **invocados automaticamente quando um objeto é instanciado**, garantindo que este seja iniciado em um estado considerado consistente
  - Um construtor **deve ter o mesmo nome da classe e não possuir qualquer retorno**
    - Método declarado com **visibilidade pública**
  - Dois tipos básicos de construtor:
    - **Construtor padrão**: sem parâmetros
      - Criado automaticamente pelo compilador quando não se define qualquer construtor para a classe
    - **Construtores parametrizados**: recebem como parâmetros valores que irão inicializar os atributos do objeto a ser instanciado
      - Os nomes dos parâmetros devem ser diferentes dos nomes dos atributos da classe
      - **Importante**: Se a classe contiver pelo menos um construtor parametrizado, o construtor padrão não é criado automaticamente pelo compilador, tornando-se portanto a única forma de instanciar objetos
-

# Construtores: exemplo 1

```
class Retangulo {  
    private:  
        int largura;  
        int altura;  
  
    public:  
        Retangulo();  
        Retangulo(int l, int a);  
};  
  
Retangulo::Retangulo() {  
    largura = 0;    // setLargura(0)  
    altura = 0;    // setAltura(0)  
}  
  
Retangulo::Retangulo(int l, int a) {  
    largura = l;    // setLargura(l)  
    altura = a;    // setAltura(a)  
}
```

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

```
int main() {  
    Retangulo r(10, 5);  
    std::cout << "Largura = " << r.getLargura();  
    std::cout << ", Altura = " << r.getAltura();  
  
    Retangulo q;  
    std::cout << "Largura = " << q.getLargura();  
    std::cout << ", Altura = " << q.getAltura();  
  
    return 0;  
}
```

criação de objeto  
**Retangulo**  
com largura 10 e altura 5  
(construtor parametrizado)

criação de objeto  
**Retangulo**  
com largura 0 e altura 0  
(construtor padrão)

# Construtores: exemplo 2 – lista de inicializadores

```
class Retangulo {  
    private:  
        int largura;  
        int altura;  
  
    public:  
        Retangulo();  
        Retangulo(int l, int a);  
};  
  
Retangulo::Retangulo() {  
    largura = 0;        // setLargura(0)  
    altura = 0;         // setAltura(0)  
}  
  
Retangulo::Retangulo(int l, int a) :  
    largura(l), altura(a) {}
```

Lista de inicializadores  
de atributos

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

```
int main() {  
    Retangulo r(10, 5);  
    std::cout << "Largura = " << r.getLargura();  
    std::cout << ", Altura = " << r.getAltura();  
  
    Retangulo q;  
    std::cout << "Largura = " << q.getLargura();  
    std::cout << ", Altura = " << q.getAltura();  
  
    return 0;  
}
```

criação de objeto  
**Retangulo**  
com largura 10 e altura 5  
(construtor parametrizado)

criação de objeto  
**Retangulo**  
com largura 0 e altura 0  
(construtor padrão)

---

# Construtor cópia

- Construtor especial que recebe como parâmetro uma referência para um objeto da mesma classe e **cria um novo objeto como cópia do primeiro**
    - O compilador cria automaticamente construtores cópia para uma classe, porém às vezes tal construtor não é suficiente para realizar essa operação efetivamente (“cópia rasa”, do Inglês *shallow copy*)
  - Implementação: **cópia de membro por membro da classe**
    - A utilização de um construtor cópia é particularmente útil quando a classe faz manipulação de recursos alocados dinamicamente, para evitar que dois objetos apontem para o mesmo recurso inadvertidamente
-



---

# Construtores: exemplo 3 - construtor cópia

```
class Retangulo {  
    private:  
        int largura;  
        int altura;  
  
    public:  
        Retangulo();  
        Retangulo(int l, int a);  
        Retangulo(Retangulo &r);  
};  
  
Retangulo::Retangulo() {  
    // Construtor padrao  
}  
  
Retangulo::Retangulo(int l, int a) {  
    largura = l;    // setLargura(l)  
    altura = a;    // setAltura(a)  
}
```

```
Retangulo::Retangulo(Retangulo &r) {  
    largura = r.getLargura();  
    altura = r.getAltura();  
}
```



**Construtor  
cópia**

---

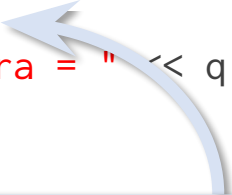
# Construtores: utilizando um construtor cópia

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

int main() {
    Retangulo r(10, 5);
    std::cout << "Largura = " << r.getLargura() << ", Altura = " << r.getAltura();

    Retangulo q(r);
    std::cout << "Largura = " << q.getLargura() << ", Altura = " << q.getAltura();

    return 0;
}
```



criação de novo objeto  
Retangulo  
a partir do primeiro que foi criado  
(construtor cópia)

---

## Outro problema

- Imagine uma classe que possui como atributo um vetor alocado em memória de forma dinâmica
  - Vimos que é sempre necessário liberar memória dinamicamente alocada para evitar diversos problemas
- Quando o objeto está prestes a ser liberado da memória, como liberar essa memória alocada para o vetor?
- Solução: implementação de **destrutores**

---

# Destrutor

- Método invocado **automaticamente** quando um objeto está prestes a ser liberado da memória
    - Útil quando a classe utiliza recursos previamente alocados e que precisam ser liberados ao término da execução, tais como memória alocada dinamicamente ou arquivos abertos
    - **Não se invoca explicitamente um destrutor**
  - Um destrutor deve
    - **deve ter o mesmo nome da classe, precedido por um til (~)**
    - **não deve possuir parâmetros**
    - **não deve possuir retorno**
  - Uma classe pode ter mais de um construtor, **porém apenas um destrutor**
-

# Destrutor: exemplo

```
class Retangulo {  
    private:  
        int largura;  
        int altura;  
  
    public:  
        Retangulo();  
        Retangulo(int l, int a);  
        ~Retangulo();  
};  
  
Retangulo::~~Retangulo() {  
    // Destrutor padrao  
}
```

```
#include <iostream>  
#include "retangulo.h"  
  
int main() {  
    Retangulo r(10, 5);  
    std::cout << "Largura = " << r.getLargura();  
    std::cout << ", Altura = " << r.getAltura();  
  
    return 0;  
}
```

criação de objeto  
**Retangulo**  
com largura 10 e altura 5

Aqui o objeto Retangulo  
perde o escopo e o seu destrutor é  
invocado **implicitamente!!!**

---

# Membros estáticos

- Uma instância de um objeto de uma classe armazena de forma autocontida todas as suas informações
  - Mas... e se quisermos armazenar de forma global informações referentes à classe como um todo?
    - Exemplo: um contador de objetos de uma classe
  - Solução: implementação de **membros estáticos**
-

---

# Membros estáticos

- Atributos e/ou métodos estáticos terão **uma instância única** independente do número de objetos criados
  - **Não é necessário instanciar um objeto** para poder acessar um atributo ou método estático: é possível acessá-lo utilizando o operador de resolução de escopo (`::`)
  - A declaração de membros estáticos se dá por meio da palavra-chave `static`
    - **Atributo estático:** `static <tipo_atributo> <nome_atributo>`
      - Só pode ser inicializado fora da definição da classe
    - **Método estático:** `static <tipo_retorno> <nome_metodo>(<parametros>)`
      - Normalmente implementado para manipular atributos estáticos da classe
      - Não é necessário adicionar a palavra-chave `static` se o método for implementado fora da classe
-

# Membros estáticos

```
class Retangulo {  
    private:  
        int largura;  
        int altura;  
  
    public:  
        Retangulo(int l, int a);  
        static int total;  
        static int getTotal();  
};  
  
Retangulo::Retangulo(int l, int a) {  
    largura = l;    // setLargura(l)  
    altura = a;    // setAltura(a)  
    total++;  
}  
  
int Retangulo::getTotal() {  
    return total;  
}
```

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

```
int Retangulo::total = 0;
```

```
int main() {  
    Retangulo r(10, 5);  
    std::cout << "Largura = " << r.getLargura();  
    std::cout << ", Altura = " << r.getAltura();  
  
    std::cout << "Numero de instancias: ";  
    std::cout << Retangulo::getTotal() << std::endl;  
  
    return 0;  
}
```

inicialização de atributo  
estático da classe  
**Retangulo** (cpp)

criação de objeto  
**Retangulo**  
com largura 10 e altura 5

retorna o numero de objetos da  
classe **Retangulo** instanciados



---

# Trabalhando com valores randômicos no C++

- Funções pré-definidas para a geração de números pseudo-randômicos do C podem ser usadas para criar eventos imprevisíveis pelo usuário (por exemplo, eventos em jogos)
  - Necessário incluir a biblioteca `<cstdlib>`
  - `srand(seed)` – inicializa o gerador de números randômicos, necessita ser executada uma única vez no programa e não retorna valor
    - `seed` – Um valor inteiro a ser usado como semente
  - `rand()` – retorna um novo número randômico inteiro
    - O número retornado varia de **0** to **MAX\_RANDOM** – uma constante definida em `<cstdlib>`
    - Para obter um número em um intervalo específico:

*// Exemplo, gera um numero randômico entre 0 e 9*

**int valor = rand() % 10;**

*// Exemplo, gera um numero randômico entre 7 e 25*

**int valor = 7+ rand() % 25;**

---

---

# Trabalhando com valores randômicos no C++11

- A especificação C++11 amplia as possibilidades para a geração de números randômicos
    - Necessário incluir `<random>`
    - `std::random_device rd;` // Instancia o gerador de números pseudo randômicos não-determinístico
      - `std::mt19937 gen(rd());` // Gerador de números randômicos pré-definido. Há vários algoritmos já implementados. mt19937 implementa 32-bit Mersenne Twister by Matsumoto and Nishimura, 1998.
    - Utilizando uma distribuição:
      - `std::uniform_real_distribution<> real_dis (3.1, 7.5);` // Distribuição real com valores entre 3.1 e 7.5
      - `std::normal_distribution<> normal_dis (500,25);` // Distribuição normal, com média 500 e desvio padrão de 25
    - Obtendo um valor:
      - `double valor = real_dis (rd);` // Obtem um valor real entre 3.1 e 7.5 usando o gerador padrão
      - `double valor = real_dis (gen);` // Obtem um valor real entre 3.1 e 7.5 usando o gerador pré-definido
      - `int valor = std::round( normal_dis (gen));` // Obtem um valor inteiro de uma normal(500,25)
  - Muito mais em: <http://en.cppreference.com/w/cpp/numeric/random>
-

---

# Exemplo: A classe dado

```
#ifndef _DADO_H_
#define _DADO_H_

#include <random>

class dado {
    private:
        int valor;
        std::random_device rd;
        std::default_random_engine gen;
        std::uniform_int_distribution<> dis;
    public:
        dado ();
        int jogar();
        int getValor();
};

#endif
```

---

---

# Exemplo: A classe dado

```
#include <random>
```

```
#include "dado.h"
```

```
dado::dado(): gen(rd()), dis(1, 6) {  
    valor = std::round(dis(gen));  
}
```

```
int  
dado::jogar() {  
    valor = std::round(dis(gen));  
    return valor;  
}
```

```
int  
dado::getValor() {  
    return valor;  
}
```

---

---

# Exemplo: Testando a classe dado

```
#include <iostream>
#include <cstdlib>

#include "dado.h"

int main(int argc, char const *argv[])
{
    dado meu_dado;
    for (int i=0; i<atoi(argv[1]); ++i) {
        std::cout << meu_dado.jogar() << std::endl;
    }
    return 0;
}
```

?



---

# Exercício

## Corrida de Sapos

1) Implemente uma classe chamada Sapo contendo

- Atributos privados: identificador, distância percorrida, quantidade de pulos dados
- Atributo estático público: distância total da corrida
- Métodos públicos:
  - *getters e setters*
  - pular:
    - incrementa distância percorrida de forma randômica entre 1 e o máximo que o sapo pode saltar
    - Incrementa o número de pulos dados em uma unidade

---

# Exercício

## Corrida de Sapos

2) Implemente um programa que simule a Corrida de Sapos

- Criar três sapos
- Cada sapo deve executar o seu pulo por vez
- O sapo que primeiro alcançar a distância da corrida será o vencedor
  - Imprimir a quantidade de pulos dados pelo sapo vencedor e a distância por ele percorrida