

# IMD0030 LINGUAGEM DE PROGRAMAÇÃO I

Aula 04 – Introdução à orientação a objetos: Classes e objetos, métodos de acesso, construtores e destrutores (Parte I).

---

# Objetivos desta aula

- Introduzir o paradigma de Programação Orientada a Objetos (POO)
  - Conceitos fundamentais: classes e objetos
  - Introduzir os conceitos de construtores, destrutores e membros estáticos em classes
- Para isso, estudaremos:
  - Como o usuário pode criar tipos por meio de classes
  - Como implementar construtores, destrutores e membros estáticos utilizando a linguagem C++
  - Como manipular tipos criados utilizando a linguagem C++
- Ao final da aula, espera-se que o aluno seja capaz de:
  - Entender os conceitos básicos de POO
  - Implementar programas utilizando classes e objetos em C++

---

# Contextualização

- Tipos estrutura (*structs*) representam uma maneira através da qual um usuário podem criar seus próprios tipos nas linguagens C e C++ a partir de tipos já existentes ou outros por ele criados
  - Estruturas são também chamadas de **tipos compostos**
- Exemplo: modelagem de um retângulo e funções para manipular variáveis desse tipo

```
struct Retangulo {  
    int largura;  
    int altura;  
};
```

```
int area(struct Retangulo r) {  
    return r.largura * r.altura;  
}  
  
int perimetro(struct Retangulo r) {  
    return (2 * r.largura + 2 * r.altura);  
}
```

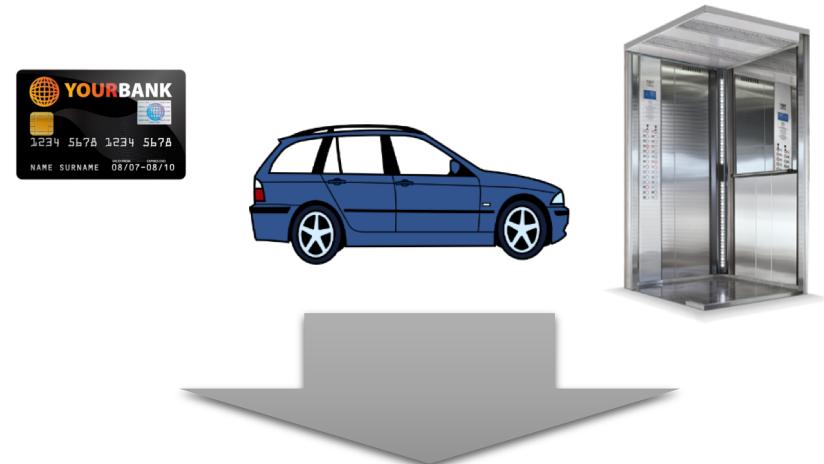
---

# Programação Orientada a Objetos (POO)

- **Paradigma de programação** que surgiu na década de 1980 com o objetivo principal de **facilitar o desenvolvimento de programas**
  - aproveitando as melhores ideias da programação estruturada (1960-)
  - agregando novos conceitos para a representação do mundo real de forma mais intuitiva
  - procurando melhorar produtividade e qualidade
- A solução de problemas utilizando POO é baseada na **abstração**
  - das **entidades** do mundo real a serem representadas no programa
  - dos **dados** associados a tais entidades
  - das **operações** que podem ser realizadas por tais entidades

# Abstração

- É um dos conceitos fundamentais em POO
- Utiliza-se a técnica de abstração para mapear entidades do mundo real, dando origem ao modelo de classes a ser implementado
- Toma-se como **entidade** tudo o que é real, tendo em consideração as suas características e ações
  - De forma geral, cada **entidade** será mapeada como **classe**
  - As **características** serão mapeadas como **atributos** da classe
  - As **ações** serão mapeadas como **métodos** da classe



| Entidade          | Características                             | Ações                          |
|-------------------|---|--------------------------------|
| Carro             | Cor, peso, modelo, ano, placa               | Ligar, desligar, mover, freiar |
| Elevador          | Tamanho, capacidade em pessoas, peso máximo | Subir, descer, escolher andar  |
| Cartão de Crédito | Titular, numero, saldo, senha, limite       | Debitar, sacar, gerar extrato  |

---

# Abstração

- Através de um modelo abstrato, pode-se concentrar nas características relevantes e ignorar as irrelevantes
- **Abstração é fruto do raciocínio**
- Através da abstração é possível controlar a complexidade

---

# Encapsulamento

- Esconder os detalhes da implementação de um objeto é chamado **encapsulamento**
  - Conceito que indica que os dados contidos em um objeto **somente** poderão ser acessados e/ou modificados através de seus métodos
  - Não se deve permitir acesso direto aos atributos de uma classe
  - Assegura que toda a comunicação com o objeto seja realizada por um conjunto pré-definido de operações
- Benefícios
  - O código cliente pode usar apenas a interface para a operação
  - A implementação do objeto pode mudar, por exemplo, para corrigir erros ou aumentar desempenho, sem que seja necessário modificar o código do cliente
  - A manutenção é mais fácil e menos custosa
  - Cria um programa legível e bem estruturado

---

# Desenvolvimento Orientado a Objetos

## Análise Orientada a Objetos

É o processo de construção de modelos do domínio do problema, identificando e especificando um conjunto de objetos que interagem e comportam-se conforme os requisitos estabelecidos para o sistema.

## Projeto Orientado a Objetos

É o processo de geração de uma especificação detalhada do software a ser desenvolvido, de tal forma que esta especificação possa levar a direta implementação no ambiente alvo.

## Programação Orientada a Objetos

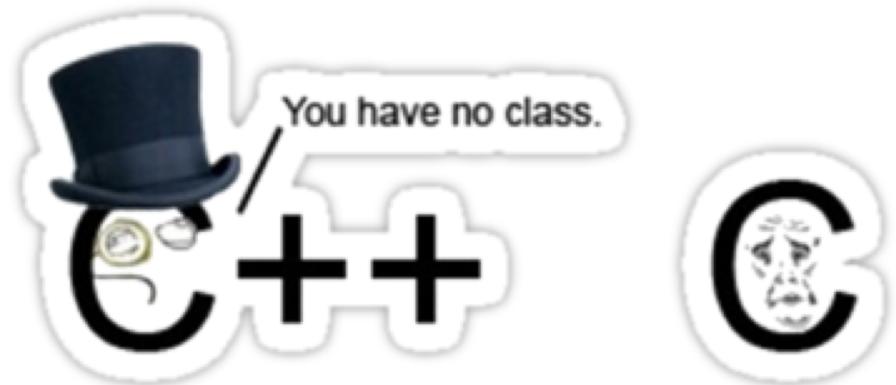
É um modelo de programação que baseia-se em conceitos como classes, objetos, herança, etc. Seu objetivo é a resolução de problemas baseada na identificação de objetos e o processamento requerido por estes objetos, e então na interação entre estes objetos.

---

# Classes e objetos

**Classe:** mecanismo provido pelo C++ (e por outras linguagens de programação) para criar tipos compostos

- Apresenta similaridades com os tipos estrutura, porém representa **um conceito novo e mais genérico**
- A introdução de POO na linguagem C++ representa um dos maiores saltos em comparação à linguagem C



---

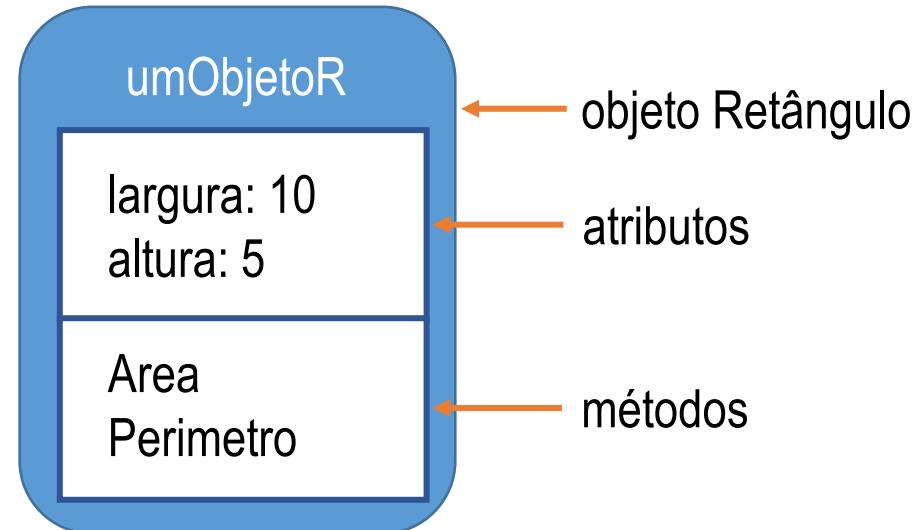
# Classes e objetos

**Objeto:** instância de uma classe

- **Instanciar** um objeto = **criar** um objeto a partir de sua definição (ou seja, de sua classe)
- Um objeto pode ser entendido como uma variável em memória autocontida, responsável pelo seu conteúdo definido na classe
  - Uma classe pode ser também entendida como o tipo de um objeto
- Um objeto tem uma **identidade**, um **estado** e um **comportamento**
  - Identidade: endereço do objeto em memória
  - Estado: valores dos dados armazenados pelo objeto
  - Comportamento: descrito por meio de operações que manipulam os dados gerenciados pelo objeto

# Classes e objetos

- Classes definem **membros** para um objeto
  - **Atributos (data members)**: informações a serem armazenadas por um objeto
  - **Métodos (function members)**: funções que podem ser chamadas por um objeto e manipulam suas informações



---

# Criando classes em C++

- Utilização da palavra-chave **class**, seguida do nome a ser dado à classe
- A definição (corpo) da classe, delimitada por chaves, pode conter
  - uma sequência de **atributos**, declarados de maneira similar a uma variável, e
  - uma sequência de **métodos**, declarados de maneira similar a uma função (métodos *inline*)
- Atributos podem ser acessados por qualquer método da classe

# Criando classes em C++ (1)

Exemplo: modelagem de um retângulo

```
class Retangulo {  
    int largura;  
    int altura;  
  
    int area() {  
        return largura * altura;  
    }  
  
    int perimetro() {  
        return (2 * largura + 2 * altura);  
    }  
};
```

atributos

métodos (*inline*)

---

# Criando classes em C++

- A implementação dos métodos de uma classe pode ser feita **em separado** à sua definição (interface), o que é uma **boa prática de programação**
  - Maior modularidade, uma vez que basta fazer a inclusão apenas do cabeçalho que define a interface
  - Melhor legibilidade, principalmente para métodos mais longos
  - Conhecimento apenas da interface, sem revelar detalhes de sua implementação
- Uso do **operador de resolução de escopo** `::` para fazer referência a um método de uma classe, quando sua implementação estiver sendo feita fora dela
  - Sintaxe: `<nome da classe>::<nome do método>`

---

# Criando classes em C++ (2)

Exemplo: modelagem de um retângulo

retangulo.h

```
class Retangulo {  
    int largura;  
    int altura;  
  
    int area();  
    int perimetro();  
};
```

retangulo.cpp

```
#include "retangulo.h"  
  
int Retangulo::area() {  
    return largura * altura;  
}  
  
int Retangulo::perimetro() {  
    return (2 * largura + 2 * altura);  
}
```

---

# Criando objetos em C++

- A **criação de um objeto** de uma classe é praticamente idêntica à declaração de uma variável em C++
- Exemplo: criação de um objeto da classe Retangulo

```
int main() {  
    Retangulo r;  
  
    return 0;  
}
```

---

# Criando objetos em C++

- A **criação de um objeto** de uma classe pode também ser feita **de forma dinâmica**, de maneira similar à criação de ponteiros
  - Útil quando algum atributo da classe representa um ponteiro a ser alocado dinamicamente
  - Uso do operador de alocação dinâmica em memória `new`
- Exemplo: criação dinâmica de um objeto da classe `Retangulo`

```
int main() {
    Retangulo* r = new Retangulo();

    return 0;
}
```

---

# Visibilidade

- A **visibilidade** de um membro (atributo ou método) de uma classe define a partir de onde ele pode ser acessado
- Três níveis de visibilidade
  - **Pública (public)**: acesso direto a partir de qualquer lugar, seja dentro ou fora da classe
  - **Privada (private)**: acesso exclusivo a métodos da própria classe
  - **Protegida (protected)**: acesso apenas por métodos da própria classe ou de suas sub-classes
- **Por padrão, todo e qualquer membro de uma classe é privado**
  - É possível especificar diferentes visibilidades respectivamente por meio das palavras-chave **public**, **private** e **protected**, seguidas de : (dois-pontos)

---

# Criando classes em C++ (3)

Exemplo: modelagem de um retângulo

```
class Retangulo {  
public:  
    int largura;  
    int altura;  
    int area();  
    int perimetro();  
};
```

---

# Visibilidade

- O **acesso a atributos públicos** de uma classe pode ser feito por meio do **operador de acesso .** (ponto), de maneira similar ao acesso em um tipo estrutura
  - Exemplo: `umRetanguloR.largura = 5`
- O **acesso a métodos públicos** de uma classe também é feito por meio do **operador de acesso .** (ponto)
  - Exemplo: `umRetanguloR.area()`
  - Métodos privados só podem ser chamados por outros métodos públicos da mesma classe → geralmente são métodos auxiliares
- Se um objeto foi criado de maneira dinâmica, o operador de acesso a ser utilizado é o mesmo usado para acessar campos de um ponteiro para estrutura, ->
  - Exemplo: `ptrRetanguloR->area()`

---

# Criando classes em C++ (4)

Exemplo: modelagem de um retângulo

retangulo.h

```
class Retangulo {  
public:  
    int largura;  
    int altura;  
  
    int area();  
    int perimetro();  
};
```

main.cpp

```
#include "retangulo.h"  
  
int main() {  
    Retangulo r;  
    r.largura = 10;  
    r.altura = 5;  
  
    return 0;  
}
```

---

# Criando classes em C++ (5)

Exemplo: modelagem de um retângulo

retangulo.h

```
class Retangulo {  
public:  
    int largura;  
    int altura;  
  
    int area();  
    int perimetro();  
};
```

main.cpp

```
#include "retangulo.h"  
  
int main() {  
    Retangulo* r = new Retangulo;  
    r->largura = 10;  
    r->altura = 5;  
  
    return 0;  
}
```

← alocação  
dinâmica

---

# Visibilidade

- **Não é possível acessar atributos privados de uma classe por meio do operador de acesso . (ponto)**
  - Conceito de **encapsulamento**: restringir o acesso aos atributos da classe
    - Encapsulamento é a técnica que faz com que detalhes internos do funcionamento dos métodos de uma classe permaneçam ocultos para os objetos
    - Com isso, o conhecimento a respeito da implementação interna da classe é desnecessário do ponto de vista do objeto, visto que isso passa a ser responsabilidade dos métodos da classe
  - Vantagens:
    - Boa prática de programação
    - Eliminar a necessidade de conhecer a estrutura interna da classe
    - Evitar possíveis inicializações e modificações ilegais, o que poderia acontecer com os atributos públicos (e com variáveis de tipo estrutura)

---

# Visibilidade

- Para acessar atributos privados de uma classe, é necessário **criar métodos públicos que os accessem** (métodos acessors)
  - **Métodos getters:** retornam o valor do atributo
    - Convenção: <tipo de retorno> get<nome do atributo>()
  - **Métodos setters:** modificam o valor do atributo atribuindo outro valor recebido como parâmetro
    - Convenção: **void** set<nome do atributo>(<parâmetro>)

---

# Criando classes em C++ (6)

Exemplo: modelagem de um retângulo

retangulo.h

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

retangulo.cpp

```
#include "retangulo.h"  
  
int Retangulo::getLargura() {  
    return largura;  
}  
  
void Retangulo::setLargura(int l) {  
    largura = l;  
}
```

---

# Criando classes em C++ (6)

Exemplo: modelagem de um retângulo

main.cpp

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

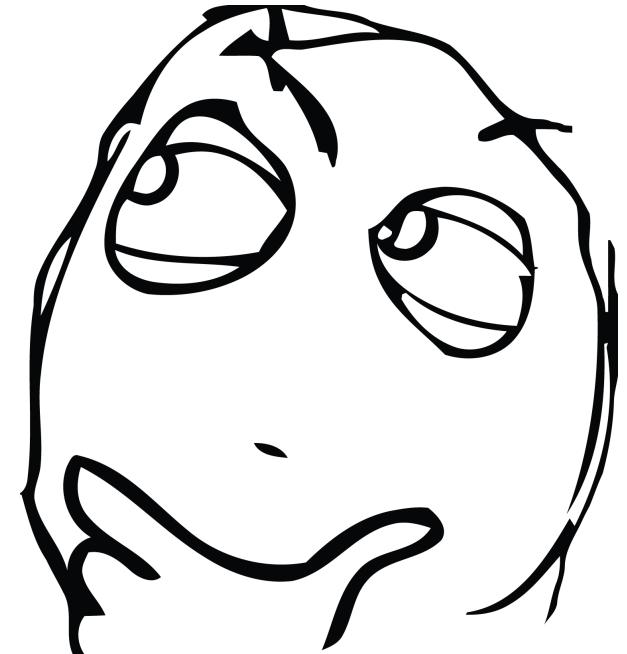
int main() {
    Retangulo r;
    r.setLargura(10);
    r.setAltura(5);
    std::cout << "Largura = " << r.getLargura() << ", Altura = " << r.getAltura();

    return 0;
}
```

---

# Um 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();

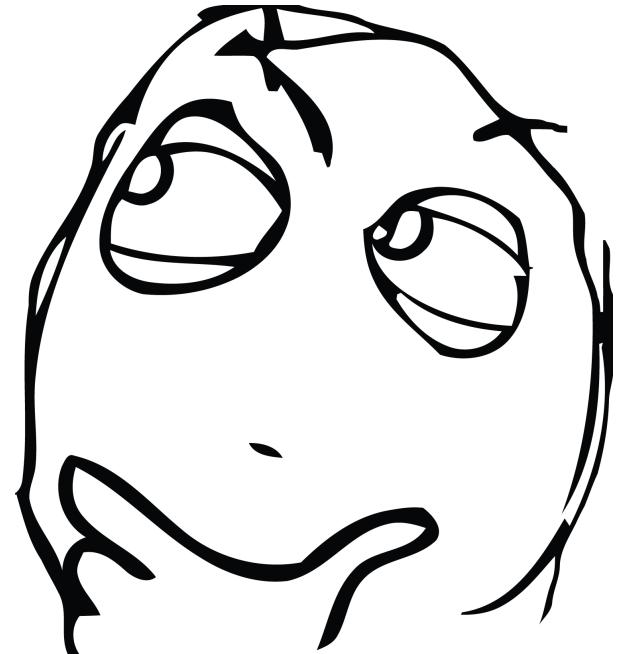
    return 0;
}
```

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**



---

# Destruitor

- 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**

# Destruitor: exemplo

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

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

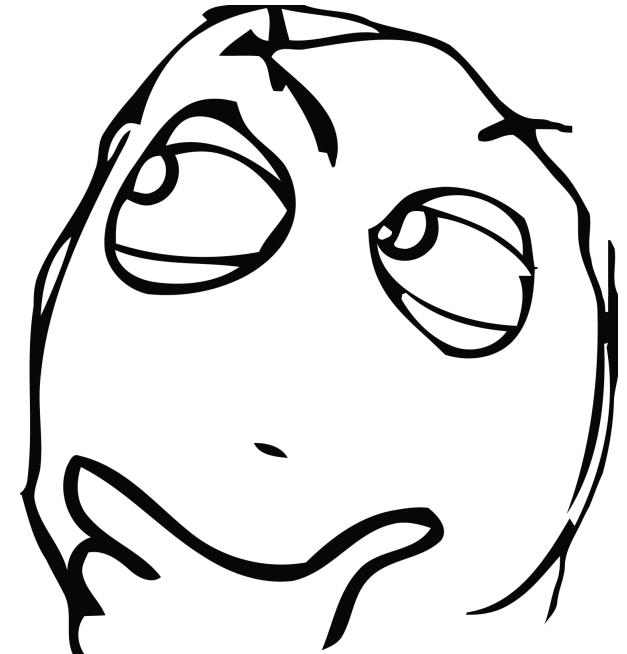
criação de objeto  
**Retangulo**  
com largura 10 e altura 5

Aqui o objeto Retangulo  
deixa de ser utilizado (a variável  
perde o escopo) e o seu destrutor  
é invocado **implicitamente!!!**

---

## Um *plus*

- 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 acessa-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**

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++

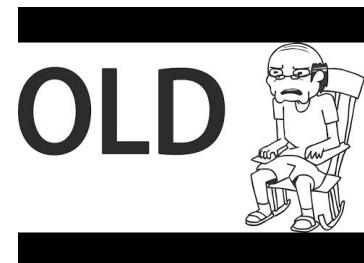
- Funções pré-definidas para a geração de números pseudo-randômicos do C podem ser usadas para criar eventos imprevisíveis pelo usuário (por exemplo, eventos em jogos)
  - Necessário incluir a biblioteca `<cstdlib>`
  - `srand(seed)` – inicializa o gerador de números randômicos, necessita ser executada 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 RAND` – 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 algorítmos 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():rd(), 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;
}
```

---

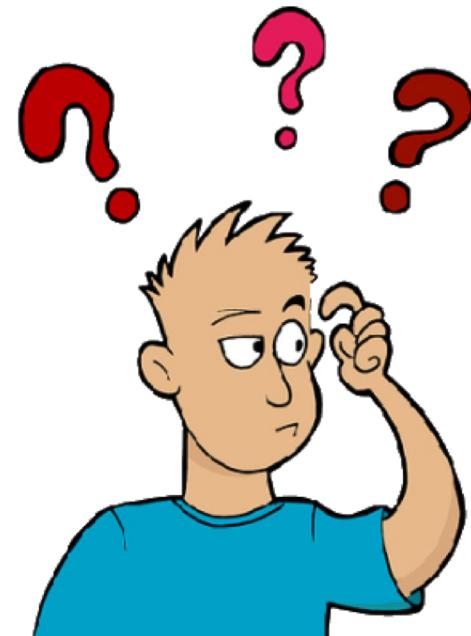
# Exemplo: Testando a classe dado

```
#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;
}
```

# Alguma Questão?



---

# Exercícios de Aprendizagem

1. Implemente uma classe **Aluno** que represente um aluno e do IMD e seus dados mais comuns. Abstraia os atributos que considerar importantes (pelo menos 5), definindo a sua visibilidade de acordo. Defina e implemente os métodos *getters* e *setters* para cada atributo, quando julgar necessário. Crie um programa para instanciar e testar um conjunto de alunos.
2. Implemente uma classe **Turma** que represente uma turma do IMD e seus dados mais comuns. Considere que toda turma tem um nome, a identificação do componente curricular à qual está associada, uma lista de alunos e uma quantidade de alunos matriculados. Defina os atributos e sua visibilidade. Defina e implemente ainda os métodos *getters* e *setters* para cada atributo, quando julgar necessário. Defina também métodos que permitam a inclusão e exclusão de alunos na turma. Crie um programa para instanciar e testar uma turma.

---

# Exercício de Aprendizagem

## 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 de Aprendizagem

## Corrida de Sapos

2) Implemente um programa que simule a Corrida de Sapos

- Definir globalmente a distância da corrida
- 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

