

# Programação e Desenvolvimento de Software 2

## Programação Orientada a Objetos (Classes)

---

Prof. Luiz Chaimowicz  
(slides adaptados do Prof. Douglas Macharet)

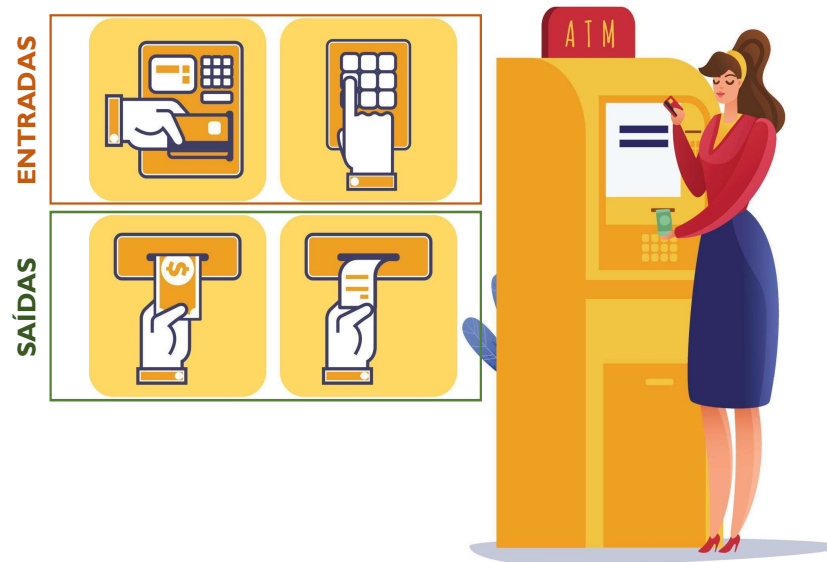
# Classes

- Utilizadas para a criação de TADs
- Suportam/utilizam os conceitos de
  - Abstração, Encapsulamento, Herança e Polimorfismo
- E as Structs (C++)?
  - Possuem comportamento semelhante, mas com “encapsulamento mais fraco” (não tem “modificadores de acesso”).
  - A partir de agora vamos utilizar apenas para armazenamento (apenas atributos, sem métodos)

# Classes

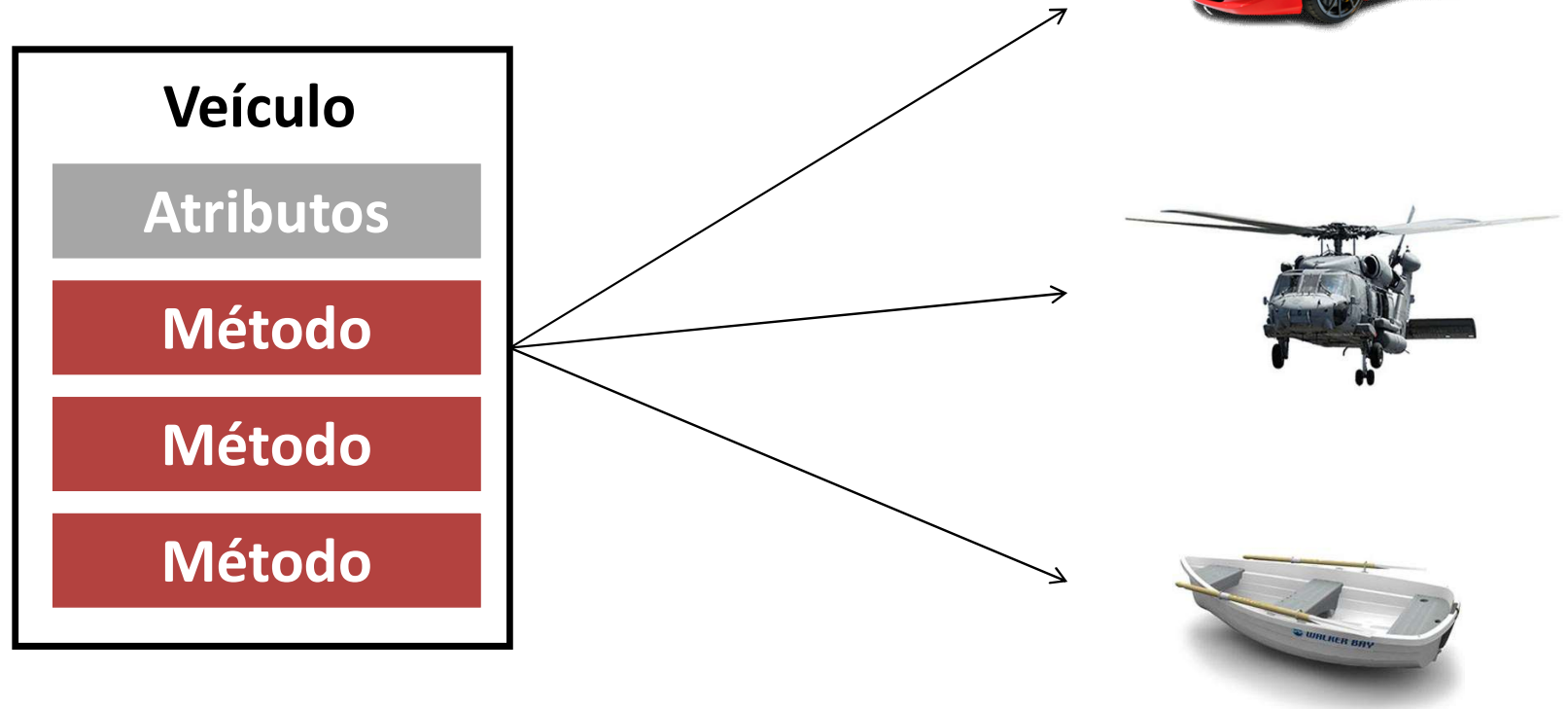
## Abstração

- Simplificação → Separação entre conceitos e detalhes
  - Ignorar o que não é relevante
- Utilizar algo complexo sem entender o funcionamento

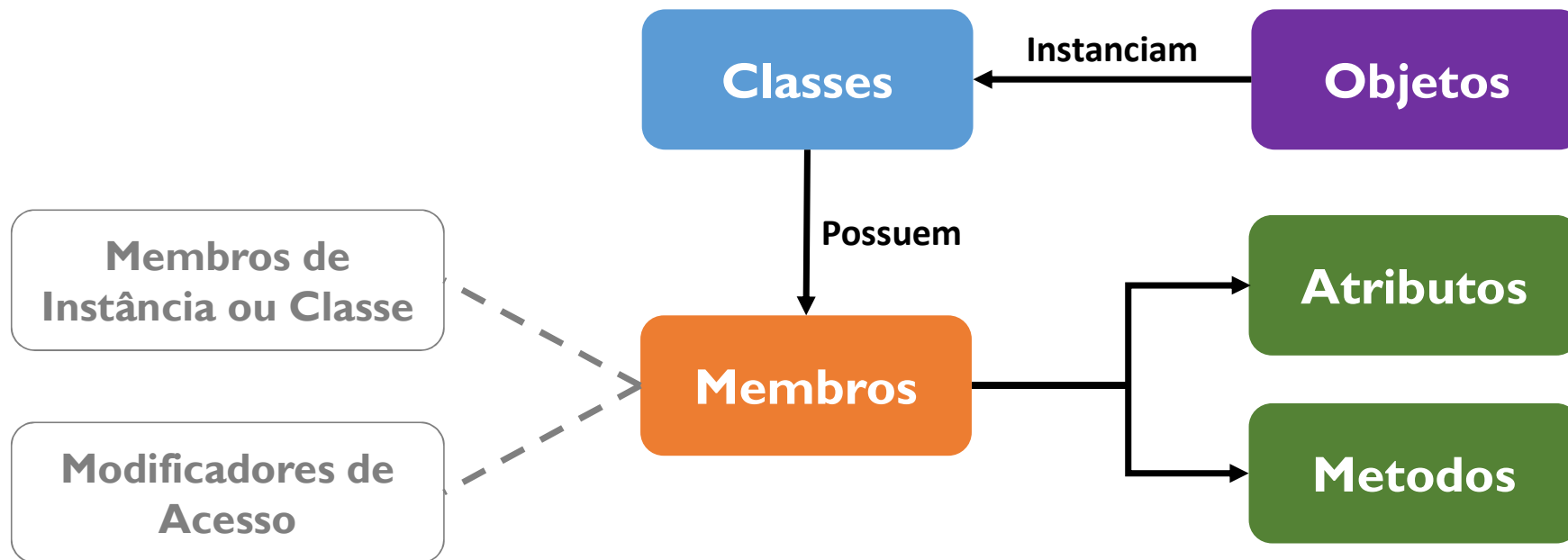


# Classes

## Abstração



# Classes



# Classes

## Membros


- Membros de **instância**
  - Espaço de memória alocado para cada Objeto
  - Somente são acessados através do Objeto
- Membros de **classe** (estáticos)
  - Espaço de memória único para todos Objetos
  - Podem ser chamados mesmo sem um Objeto

# Classes

## Atributos de instância

- Vão definir o Estado de cada objeto através do valor que está armazenado

Modificador de Acesso  
(Próxima aula!)



```
class Casa {  
    public:  
        int numero;  
        string cor;  
};
```

# Classes

## Atributos de instância

```
int main() {  
  
    Casa c1;  
    c1.numero = 77;  
    c1.cor = "verde";  
  
    Casa c2;  
    c2.numero = 55;  
    c2.cor = "vermelho";  
  
    Casa c3;  
    c3.numero = 11;  
    c3.cor = "amarelo";  
  
    return 0;  
}
```

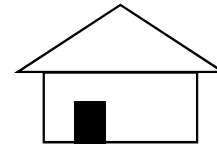


# Classes

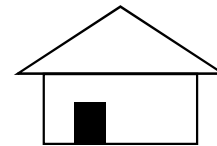
## Atributos de instância



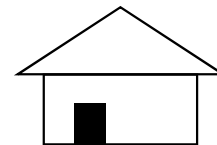
**Casa c1;**



**Casa c2;**



**Casa c3;**

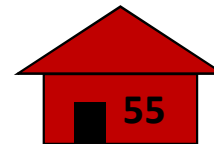


# Classes

## Atributos de instância



```
c1.numero = 77;  
c1.cor = "verde";
```



```
c2.numero = 55;  
c2.cor = "vermelho";
```




```
c3.numero = 11;  
c3.cor = "amarelo";
```

# Classes

## Atributos de instância

### Inicialização:

- Valores padrão
  - Tipos numéricos: valor 0 (zero)
  - Tipo boolean: valor 0 (false)
  - Atenção: não confiar cegamente nessa inicialização! 
- Demais atributos?
  - Não são “automaticamente” inicializados
  - Ponteiros → Lixo (pode levar à *segmentation fault*)

# Classes

## Métodos de instância

- Definem as operações a serem feitas sobre os objetos

```
class Casa {  
    public:  
        int numero;  
        string cor;  
  
        float CalculaIPTU();  
};
```

# Classes

## Métodos de instância

```
float Casa::CalculaIPTU() {  
    float valor;  
  
    if (this->numero > 100)  
        valor = 100;  
    else  
        valor = 50;  
    if (this->cor == "verde")  
        valor *= 0.9;  
  
    return valor;  
}
```

```
int main() {  
  
    Casa c1;  
    c1.numero = 77;  
    c1.cor = "verde";  
  
    Casa c2;  
    c2.numero = 55;  
    c2.cor = "vermelho";  
  
    cout << c1.CalculaIPTU();  
    cout << c2.CalculaIPTU();  
  
    return 0;  
}
```

# Classes

## Métodos de instância

- O operador `this` serve para referenciamento
- Utilizado dentro de qualquer método não estático para acessar a posição de memória do Objeto (instância) atual
- Principais utilizações
  - Informar a referência para o Objeto atual
  - Evitar conflitos de nome
  - Facilitar a compreensão do código!



# Classes

## Métodos de instância

```
class Aluno; ← DECLARAÇÃO do tipo.

class Curso {
public:
    string nome; vector<Aluno*> alunos;
    void inserir_aluno(Aluno *aluno) {
        this->alunos.push_back(aluno);
    }
};

class Aluno { ← DEFINIÇÃO do tipo.
public:
    int matricula; string nome;
    void fazer_inscricao(Curso &curso) {
        curso.inserir_aluno(this);
    }
};
```

# Classes

## Métodos de instância

```
int main() {  
    Curso curso;  
    curso.nome = "PDS2";  
  
    Aluno aluno;  
    aluno.matricula = 2020111222;  
    aluno.nome = "Douglas";  
  
    aluno.fazer_inscricao(curso);  
  
    cout << "Alunos de " << curso.nome << ": " << endl;  
    for (Aluno* aluno : curso.alunos)  
        cout << aluno->matricula << "\t" << aluno->nome << endl;  
  
    return 0;  
}
```



# Classes

## Métodos sobrecarregados

- **Sobrecarga (overload)**
  - Dois ou mais métodos com mesmo nome → Polimorfismo
  - Lista de parâmetros (tipos) deve ser diferente!
    - A ordem dos tipos dos parâmetros é importante
  - Melhora a legibilidade/usabilidade de partes do código
  - Resolvido em tempo de compilação
- Não são diferenciáveis pelo tipo de retorno
  - Podem possuir diferentes tipos de retorno desde que possuam diferentes parâmetros de entrada

# Classes

## Métodos sobrecarregados

```
class Ponto {  
    public:  
        int x;  
        int y;  
  
        void setarXY(int x, int y) {  
            this->x = x;  
            this->y = y;  
        }  
        void setarXY(int xy) {  
            this->x = xy;  
            this->y = xy;  
        }  
};
```

```
int main() {  
    Ponto p;  
  
    p.setarXY(10, 20);  
    cout << p.x << endl;  
    cout << p.y << endl;  
  
    p.setarXY(50);  
    cout << p.x << endl;  
    cout << p.y << endl;  
  
    return 0;  
}
```

# Classes

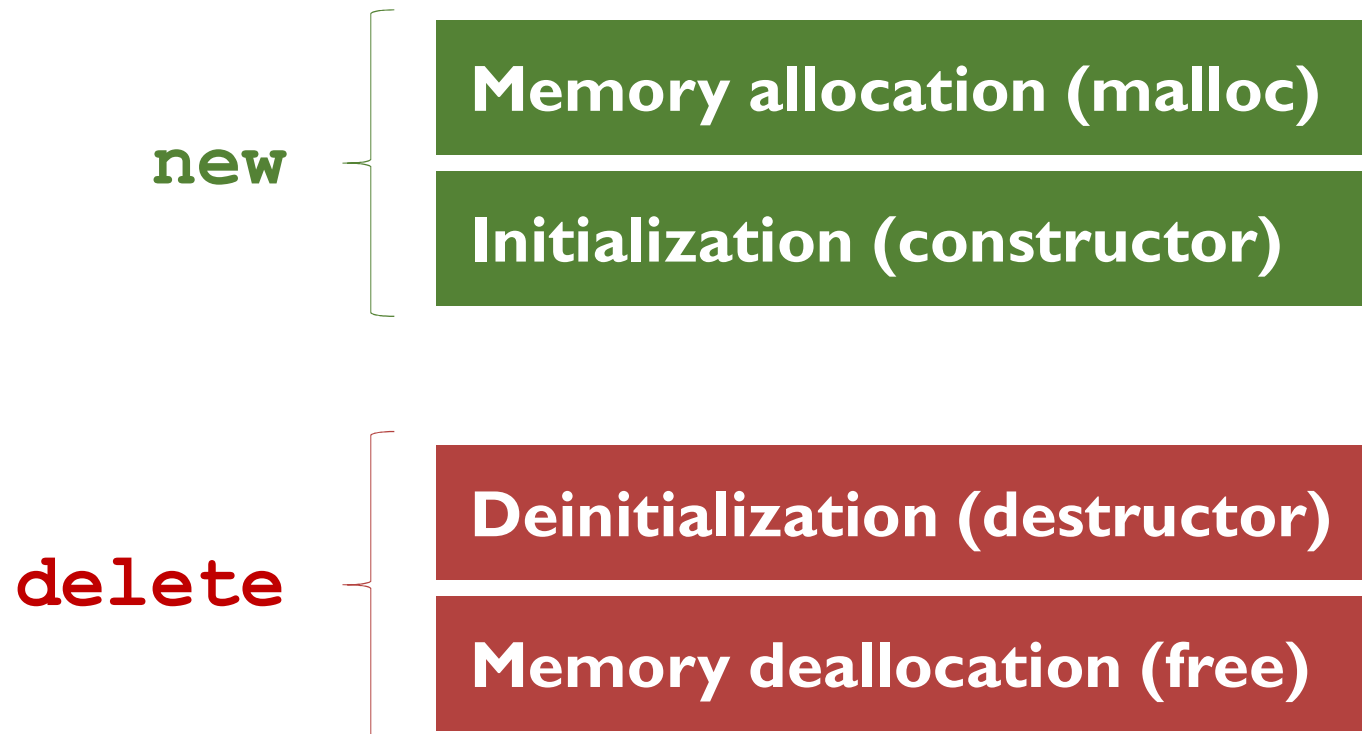
- **Construtor**

- Usados apenas na criação de um novo objeto
- Responsável pela inicialização

- **Destrutor**

- Usado no momento da remoção de um objeto
- Responsável por liberar os recursos adquiridos na criação

# Classes



# Classes

## Construtores

- Método chamado durante a instanciação de um objeto
  - Você pode criar um ou mais construtores
  - **Se não criar**, possui um construtor padrão (sem parâmetros)
- Devem possuir o mesmo nome da Classe
  - Selecionados através da lista de parâmetros (sobrecarga)
- Nunca declaram tipo de retorno
  - Por que?

# Classes

## Construtores

```
class Ponto {  
    public:  
        int x;  
        int y;  
  
    Ponto(int x, int y) {  
        this->x = x;  
        this->y = y;  
    }  
  
    Ponto(int xy) {  
        this->x = xy;  
        this->y = xy;  
    }  
};
```

Quando um novo construtor com parâmetros é declarado, o **padrão** não é mais acessível!  
(se quiser um padrão, tem que criar)

```
int main() {  
  
    Ponto p1;  
    Ponto p2(50, 50);  
    Ponto* p3 = new Ponto(50);  
  
    return 0;  
}
```

Ponto p2 = Ponto(50, 50);

# Classes

## Construtores

- Qual o problema com esse código?
  - Código duplicado!
- Como seria possível melhorar?
  - Reutilizar construtores!
  - Delegar partes específicas

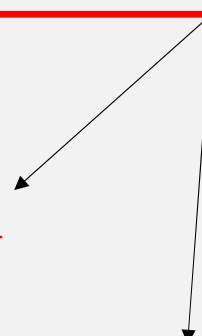
```
class Ponto {  
    public:  
        int x;  
        int y;  
  
    Ponto() {  
        this->x = -1;  
        this->y = -1;  
    }  
  
    Ponto(int xy) {  
        this->x = xy;  
        this->y = xy;  
    }  
  
    Ponto(int x, int y) {  
        this->x = x;  
        this->y = y;  
    }  
};
```

# Classes

## Construtores

```
class Ponto {  
    public:  
        int x;  
        int y;  
  
    Ponto() : Ponto(-1, -1) {}  
  
    Ponto(int xy) : Ponto(xy, xy) {}  
  
    ➔ Ponto(int x, int y) {  
        this->x = x;  
        this->y = y;  
    }  
};
```

Além de chamar o outro construtor, pode incluir algum código específico se quiser



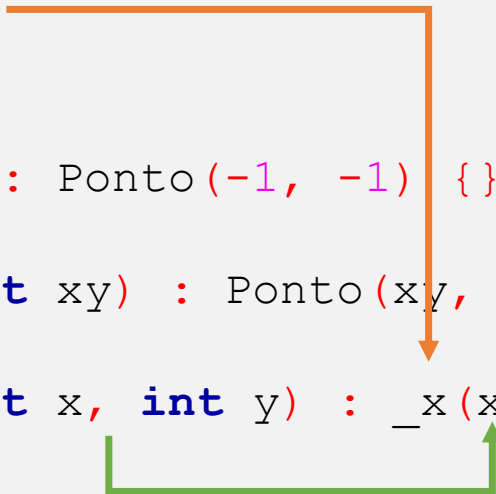
<https://docs.microsoft.com/pt-br/cpp/cpp/delegating-constructors?view=msvc-170>



# Classes

## Construtores (listas de inicialização de membros)

```
class Ponto {  
  
    public:  
        int _x;  
        int _y;  
  
        Ponto() : Ponto(-1, -1) {}  
  
        Ponto(int xy) : Ponto(xy, xy) {}  
  
        Ponto(int x, int y) : _x(x), _y(y) {}  
  
};
```



Inicialização mais eficiente, especialmente se o atributo for difícil/pesado de inicializar.

Variáveis que não estão na lista de inicialização o compilador irá atribuir um valor padrão antes de entrar no corpo do construtor, tipo assim:

```
Classe(int valor) : campo("valor default")  
{  
    campo = valor;  
}
```

Member initializer lists: <http://www.learncpp.com/cpp-tutorial/8-5a-constructor-member-initializer-lists/>

# Classes

## Destrutores

Observe o código abaixo. Há algum problema?

```
class Teste {  
    public:  
        int *x;  
  
    Teste(int x) {  
        this->x = new int(x);  
    }  
};
```

```
#include<iostream>  
  
int main() {  
    Teste *p1 = new Teste(10);  
  
    delete p1;  
  
    return 0;  
}
```

Área de memória alocada no construtor para o atributo **x** continua existindo

# Classes

## Destrutores

- Método chamado para a finalização de um objeto
  - Libera os recursos alocados na execução
  - Invocados quando o *lifetime* de um objeto chega ao fim
    - Heap → Após um delete
    - Stack → Após a saída do escopo
- Especialmente importantes quando:
  - O objeto aloca memória dinamicamente: (**você** deve desalocar!)
  - Você quer fazer alguma operação ao término do objeto
- Devem possuir o mesmo nome da Classe (precedido por ~)

# Classes

## Destrutores

```
class Teste {  
    public:  
        int *x;  
  
        Teste(int x) {  
            this->x = new int(x);  
        }  
  
        ~Teste(){  
            delete this->x ;  
        }  
};
```

```
#include<iostream>  
  
int main() {  
    Teste *p1 = new Teste(10);  
  
    delete p1;  
  
    return 0;  
}
```

# Classes

## Destrutores

```
class TestObject {  
    public:  
        int atributo;  
  
    TestObject(int valor) {  
        atributo = valor;  
    }  
  
    ~TestObject() {  
        cout << "~TestObject" << atributo << endl;  
    }  
};
```

Qual será a saída?

~TestObject2  
~TestObject1

```
int main() {  
  
    TestObject o1(1);  
    TestObject* o2 = new TestObject(2);  
  
    delete o2;  
    return 0;  
}
```

# Classes

## Destrutores

Qual será a saída?

Antes

~TestObject0

~TestObject1

~TestObject2

~TestObject3

~TestObject4

Depois

```
int main() {  
  
    cout << "Antes" << endl;  
  
    int i = 0;  
    while (i < 5) {  
        TestObject o(i);  
        i++;  
    }  
  
    cout << "Depois" << endl;  
  
    return 0;  
}
```

[Wandbox](#)

## (Escopo de Variáveis)

- Escopo de uma variável
  - Região dentro da qual uma variável “existe”
  - Pode ser referenciada pelo seu nome
- Relação com a memória?
  - Define quando o sistema aloca e libera memória para a variável
  - Variáveis alocadas na pilha são liberadas ao sair do escopo
  - Variáveis alocadas no heap continuam lá mesmo fora do escopo

# (Escopo de Variáveis)

- Quais escopos temos aqui?
  - Classe, Método, If
- O que acontece quando:
- Entra/sai do if?
  - result entra/sai da pilha
- O método termina?
  - param, x e y saem da pilha
- A classe é destruída?
  - var1 e var2 saem da pilha

```
class ClasseTeste {  
    public:  
        int var1;  
        std::string var2;  
  
        void metodo(int param) {  
            int x = 1;  
            int y = 9;  
  
            if (param % 2 == 0) {  
                int result = 12;  
            }  
        }  
};
```



# Classes

## Atributos estáticos

- Não estão associados a uma instância (objeto)
  - Atributos de **Classe**
- Atributos (valores) compartilhados pelas instâncias
  - Ocupam um espaço único na memória durante a execução
- Geralmente utilizados para constantes

# Classes

## Atributos estáticos

```
class AtributoEstatico {  
  
    public:  
  
    static int numero;  
  
    AtributoEstatico() {  
        AtributoEstatico::numero++;  
    }  
  
    void imprimirNumero () {  
        cout << AtributoEstatico::numero << endl;  
    }  
};  
  
int ClasseAtributoEstatico::numero = 0;
```

Acesso pelo nome da **CLASSE**.

O this não faz sentido aqui, por quê?

A inicialização deve ser feita no arquivo .cpp!

# Classes

## Atributos estáticos

O que será impresso na tela?

```
int main () {
```

```
    ClasseAtributoEstatico c1;
```

```
    c1.imprimirNumero();
```

→ 1

```
    ClasseAtributoEstatico c2;
```

```
    c2.imprimirNumero();
```

→ 2

```
    c1.imprimirNumero();
```

→ 2

```
    cout << ClasseAtributoEstatico::numero << endl;
```

→ 2

```
    return 0;
```

```
}
```

[Wandbox](#)

# Classes

## Métodos estáticos

- Parecidos com funções globais
  - Não demandam uma instância da Classe
- Acessados diretamente pela Classe
- Muito utilizados em classes do tipo “Util”
  - Classes com funções relacionadas
  - Por exemplo, funções matemáticas

# Classes

## Métodos estáticos

- Resolvidos em tempo de compilação
  - Não dinamicamente como em métodos de instância que são resolvidos baseados no tipo do objeto em tempo de execução
- Não podem ser sobrescritos (herança)
- Dentro de métodos *static* (de classe) não se consegue acessar/usar atributos e métodos que são de instância!
  - Por que?

# Classes

## Métodos estáticos

```
class MathUtils {  
  
    public:  
        static double calcularMedia(double a, double b) {  
            return (a + b) / 2;  
        }  
};  
  
int main() {  
  
    cout << MathUtils::calcularMedia(10, 20) << endl;  
  
    return 0;  
}
```

[Wandbox](#)

# Classes

## Modularização

- Da mesma forma que foi feito com as *structs*, é interessante separar o código dos TADs em diferentes arquivos
  - Melhor organização
  - Permite esconder o código fonte
  - Permite compilar separadamente
- TAD.hpp com o “contrato”, TAD.cpp com a implementação
- Main.cpp inclui o TAD.hpp

# Classes

## Modularização

### ClasseTeste.hpp

```
#ifndef CLASSETESTE_H
#define CLASSETESTE_H

#include <string>

using namespace std;

class ClasseTeste {

    public:
        int      _atributo1;
        string    _atributo2;

        ClasseTeste ();
        ClasseTeste (double, string);

        void metodoA ();
        void metodoB (double);

};

#endif
```



# Classes

## Modularização

### ClasseTeste.cpp

```
#include "ClasseTeste.hpp"
#include <iostream>

ClasseTeste::ClasseTeste() : ClasseTeste(0.0, "") {}

ClasseTeste::ClasseTeste(double a1, string a2) {
    _atributo1 = a1;
    _atributo2 = a2;
}

void ClasseTeste::metodoA() {
    cout << _atributo1 << "\t" << _atributo2 << endl;
}

void ClasseTeste::metodoB(double i) {
    _atributo1 += i;
}
```

# Classes

## Modularização

### main.cpp

```
#include "ClasseTeste.hpp"

int main() {

    ClasseTeste c1(10, "Joao da Silva");

    c1.metodoA();
    c1.metodoB(50);
    c1.metodoA();

    ClasseTeste c2;
    c2.metodoA();

    return 0;
}
```

[Wandbox](#)

**Compilando junto:** \$ g++ -std=c++11 -Wall main.cpp ClasseTeste.cpp -o main

**Compilando separado:** \$ g++ -std=c++11 -Wall -c ClasseTeste.cpp -o ClasseTeste.o

\$ g++ -std=c++11 -Wall main.cpp ClasseTeste.o -o main

# Princípio da Responsabilidade Única (SRP)

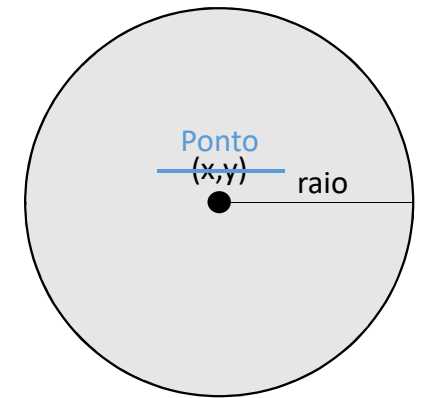
- Classe deve ser responsável por um “serviço”
- Existe apenas um “motivo” para mudar
- Classes com várias responsabilidades
  - Haverá mais de uma razão para alterá-la
    - Responsabilidade → Dimensão de mudança
  - Mais frágil torna-se o projeto
    - Cada alteração pode acabar introduzindo erros



[https://en.wikipedia.org/wiki/Single-responsibility\\_principle](https://en.wikipedia.org/wiki/Single-responsibility_principle)

## Exercício

- Como reimplementar o TAD Circunferência agora utilizando esses novos conceitos?
- Quais as possíveis classes envolvidas?
  - Ponto, Circunferencia
- Quais atributos e métodos?
  - Ponto  $\rightarrow x, y$
  - Circunferencia  $\rightarrow$  pontoCentro, raio, calcularArea
- Por onde começar?
  - Definição dos contratos  $\rightarrow$  .hpp



# Exercício

## Ponto.hpp

```
#ifndef PONTO_H
#define PONTO_H

class Ponto {
public:
    double _x;
    double _y;

    Ponto();
    Ponto(double x, double y);
};

#endif
```

## Circunferencia.hpp

```
#ifndef CIRCUNFERENCIA_H
#define CIRCUNFERENCIA_H

#include <cmath>
#include "Ponto.hpp"

class Circunferencia {

public:
    Ponto _centro;
    double _raio;

    Circunferencia(double raio);
    Circunferencia(Ponto centro, double raio);
    double calcularArea();
};

#endif
```

# Exercício

## Ponto.cpp

```
#include "Ponto.hpp"

Ponto::Ponto() : Ponto(0.0, 0.0) {}

Ponto::Ponto(double x, double y) : _x(x), _y(y) {}
```

## Circunferencia.cpp

```
#include "Circunferencia.hpp"

Circunferencia::Circunferencia(double raio) : Circunferencia(Ponto(), raio) {}

Circunferencia::Circunferencia(Ponto centro, double raio) : _centro(centro),
    _raio(raio) {}

double Circunferencia::calcularArea() {
    return M_PI * pow(this->_raio, 2);
}
```

# Exercício

main.cpp

```
#include <iostream>

#include "Ponto.hpp"
#include "Circunferencia.hpp"

using namespace std;

int main() {

    Circunferencia* c1 = new Circunferencia(10);
    cout << c1->calcularArea() << endl;

    Ponto p(5.0, 5.0);
    Circunferencia* c2 = new Circunferencia(p, 20);
    cout << c2->calcularArea() << endl;

    delete c1;
    delete c2;

    return 0;
}
```