

DCC004 - Algoritmos e Estruturas de Dados II

Programação Orientada a Objetos

Renato Martins

Email: renato.martins@dcc.ufmg.br

<https://www.dcc.ufmg.br/~renato.martins/courses/DCC004>

Material adaptado de PDS2 - Douglas Macharet e Flávio Figueiredo

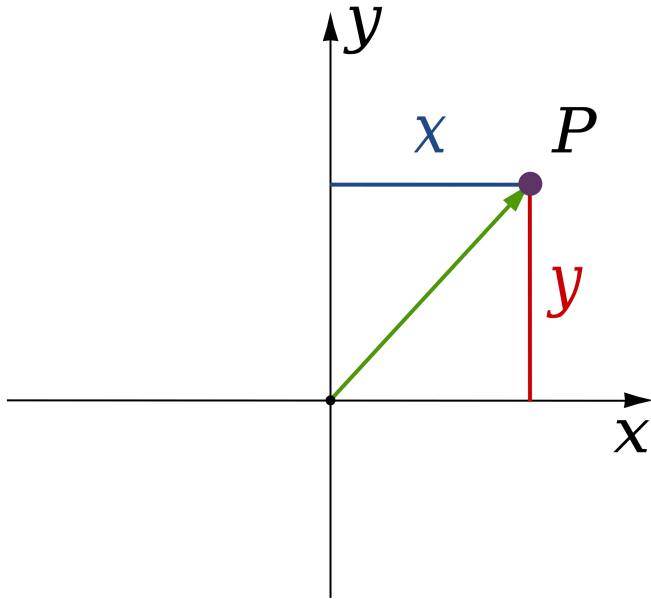
Revisando Structs

Structs

- Em C/C++ podemos criar novos tipos
- Úteis para representar conceitos mais complexos

Struct Ponto

apenas x e y



```
struct ponto_t {  
    double x;  
    double y;  
};
```

Structs em C++

- Um pouco mais simples do que em C
- Por enquanto, não precisamos de typedef

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

Structs em C++

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

| nome | end(&) | val(*) |
|------|--------|--------|
| main | 0x0048 | |
| | 0x0044 | |
| | 0x0040 | |
| | 0x003c | |
| | 0x0038 | |
| | 0x0034 | |
| | 0x0030 | |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

Structs em C++

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | ?? |
| ponto_b.y | 0x0040 | ?? |
| | 0x003c | |
| | 0x0038 | |
| | 0x0034 | |
| | 0x0030 | |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

Structs em C++

```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 7 |
| ponto_b.y | 0x0040 | ?? |
| | 0x003c | |
| | 0x0038 | |
| | 0x0034 | |
| | 0x0030 | |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

Structs em C++

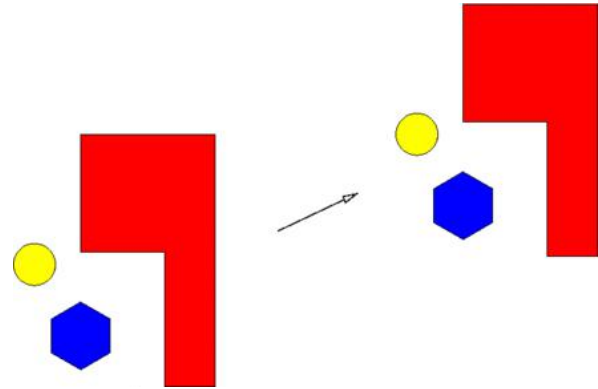
```
#include <iostream>
struct ponto_t {
    float x;
    float y;
};

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}
```

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 7 |
| ponto_b.y | 0x0040 | 9 |
| | 0x003c | |
| | 0x0038 | |
| | 0x0034 | |
| | 0x0030 | |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

Passagem por referência de structs

- Vamos implementar um procedimento de translação



```
void translacao(ponto_t &ponto, float dx, float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```

#include <iostream>

struct ponto_t {
    double x;
    double y;
};

void translacao(ponto_t &ponto, float dx,
               float dy) {
    ponto.x += dx;
    ponto.y += dy;
}

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}

```

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 7 |
| ponto_a.y | 0x0040 | 9 |
| | 0x003c | |
| | 0x0038 | |
| | 0x0034 | |
| | 0x0030 | |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

```

#include <iostream>

struct ponto_t {
    double x;
    double y;
};

void translacao(ponto_t &ponto, float dx,
               float dy) {
    ponto.x += dx;
    ponto.y += dy;
}

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}

```

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 7 |
| ponto_a.y | 0x0040 | 9 |
| transl... | 0x003c | |
| &ponto | 0x0038 | 0x0044 |
| dx | 0x0034 | 3 |
| dy | 0x0030 | 1 |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

```
#include <iostream>
```

```
struct ponto_t {  
    double x;  
    double y;  
};
```

Referência

```
void translacao(ponto_t &ponto, float dx,  
               float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

Observe que podemos
usar .

```
int main() {  
    ponto_t ponto_a;  
    ponto_a.x = 7;  
    ponto_a.y = 9;  
    translacao(ponto_a, 3, 1);  
    std::cout << ponto_a.x << std::endl;  
    std::cout << ponto_a.y << std::endl;  
    return 0;  
}
```

Sem &. Referência

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 10 |
| ponto_a.y | 0x0040 | 10 |
| transl... | 0x003c | |
| &ponto | 0x0038 | 0x0044 |
| dx | 0x0034 | 3 |
| dy | 0x0030 | 1 |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

```

#include <iostream>

struct ponto_t {
    double x;
    double y;
};

void translacao(ponto_t &ponto, float dx,
               float dy) {
    ponto.x += dx;
    ponto.y += dy;
}

int main() {
    ponto_t ponto_a;
    ponto_a.x = 7;
    ponto_a.y = 9;
    translacao(ponto_a, 3, 1);
    std::cout << ponto_a.x << std::endl;
    std::cout << ponto_a.y << std::endl;
    return 0;
}

```

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 10 |
| ponto_a.y | 0x0040 | 10 |
| | 0x003c | |
| | 0x0038 | |
| | 0x0034 | |
| | 0x0030 | |
| | 0x002c | |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

Erros

- Qual o problema com esta chamada?

```
void translacao(ponto_t ponto, float dx, float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```
#include <iostream>
```

```
struct ponto_t {  
    double x;  
    double y;  
};
```

```
void translacao(ponto_t ponto, float dx,  
               float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```
int main() {  
    ponto_t ponto_a;  
    ponto_a.x = 7;  
    ponto_a.y = 9;  
    translacao(ponto_a, 3, 1);  
    std::cout << ponto_a.x << std::endl;  
    std::cout << ponto_a.y << std::endl;  
    return 0;  
}
```

Passando cópia

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 7 |
| ponto_a.y | 0x0040 | 9 |
| transl... | 0x003c | |
| ponto.x | 0x0038 | 7 |
| ponto.y | 0x0034 | 9 |
| dx | 0x0030 | 3 |
| dy | 0x002c | 1 |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |


```
#include <iostream>
```

```
struct ponto_t {  
    double x;  
    double y;  
};
```

```
void translacao(ponto_t ponto, float dx,  
               float dy) {  
    ponto.x += dx;  
    ponto.y += dy;  
}
```

```
int main() {  
    ponto_t ponto_a;  
    ponto_a.x = 7;  
    ponto_a.y = 9;  
    translacao(ponto_a, 3, 1);  
    std::cout << ponto_a.x << std::endl;  
    std::cout << ponto_a.y << std::endl;  
    return 0;  
}
```

Passando cópia

| nome | end(&) | val(*) |
|-----------|--------|--------|
| main | 0x0048 | |
| ponto_a.x | 0x0044 | 7 |
| ponto_a.y | 0x0040 | 9 |
| transl... | 0x003c | |
| ponto.x | 0x0038 | 10 |
| ponto.y | 0x0034 | 10 |
| dx | 0x0030 | 3 |
| dy | 0x002c | 1 |
| | 0x0028 | |
| | 0x0024 | |
| | 0x0020 | |
| | 0x001c | |
| | 0x0018 | |
| | 0x0014 | |
| | 0x0010 | |
| | 0x000c | |
| | 0x0008 | |
| | 0x0004 | |
| | 0x0000 | |

Alocando structs no heap

- Fazemos uso de new e delete
- Similar a uma variável
- Uso comum para implementar listas etc.

```
ponto_t *p = new ponto_t;  
delete p;
```

Programação Orientada a Objetos

Introdução

- Programação Estruturada
 - Instruções que mudam o estado do programa
 - Programas imperativos (ações)
- Programação Orientada a Objetos
 - Dados e procedimentos encapsulados
 - Composto por diversos objetos
 - Interação/comunicação entre os objetos

Introdução

Programação Estruturada

- Como resolver problemas muito grandes?
 - Construí-lo a partir de partes menores
- Módulos compiláveis
 - Solucionam uma parte do problema
 - Dados x Manipulação
 - Abstração fraca para problemas mais complexos

Programação Orientada a Objetos

- Sistemas maiores e mais complexos
 - Aumentar a produtividade no desenvolvimento
 - Diminuir a chance de problemas
 - Facilitar a manutenção/extensão
- Programação Orientada a Objetos
 - Tem apresentado bons resultados
 - Não é uma bala de prata!

Programação Orientada a Objetos

História

- Desenvolvimento de hardware
 - Pedacos simples de hardware (chips) unidos para se montar um hardware mais complexo
- Amadurecimento dos conceitos
 - Simula (60's)
 - Smalltalk (70's)
 - C++ (80's)

Programação Orientada a Objetos

PE vs. POO

- Programação Estruturada
 - Procedimentos implementados em blocos
 - Comunicação pela passagem de dados
 - Execução → Acionamento de procedimentos
- Programação Orientada a Objetos
 - Dados e procedimentos encapsulados
 - Execução → Comunicação entre objetos

Programação Orientada a Objetos

Novo paradigma de programação

- Programação Estruturada
 - Dados acessados via funções
 - Representação de tipos complexos com struct
- Programação Orientada a Objetos
 - Dados são dotados de certa inteligência
 - Sabem realizar operações sobre si mesmos
 - É preciso conhecer a implementação?

Programação Orientada a Objetos

Benefícios

- Maior confiabilidade
 - Maior reaproveitamento de código
 - Facilidade de manutenção
 - Melhor gerenciamento
 - Maior robustez
- ...

Problemas

Sistema Bancário

- Como modelar um sistema bancário?
Temos que representar: **Clientes, Agências, Contas, Operações, Extratos etc.**

Problemas

Simulador de Vírus

Como modelar um sistema biológico? Temos que representar: **Pacientes, Vírus e Conexões**. Em cada instante de **Tempo**, um **Vírus** pode infectar um **Paciente**.

Classes vs Objetos

- Classe
 - Representa uma unidade de compilação
 - Um módulo, um tipo
 - Conceito, ideia, abstração (representação)
- Imagine uma classe como um struct **turbinado!**
- Na verdade, é justamente isto por baixo na memória

Classes vs Objetos

- Classes representam a **forma** da memória
- Objetos são **instâncias** de classes

Analogia

- Classes → formas
- Memória → massa
- Objetos → cookies



Classes vs. Objetos

Classe

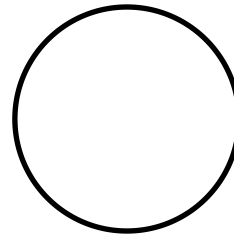
- Descrição de propriedades em comum de um grupo de objetos (conjunto)
- Um conceito
- Faz parte de um programa
- Exemplo: Pessoa
- Exemplo: Carro

Objeto

- Representação das propriedades de uma única instância (elemento)
- Um fenômeno (ocorrência)
- Faz parte de uma execução
- Exemplo: João da Silva
- Exemplo: Ferrari

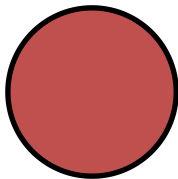
Classes vs. Objetos

CLASSE

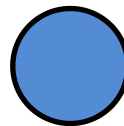


Peso
Raio
Cor

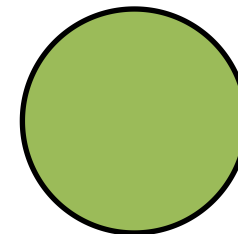
OBJETOS



Peso: 100 g
Raio: 25 cm
Cor:
Vermelha



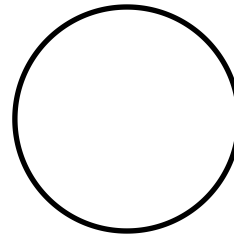
Peso: 50 g
Raio: 10 cm
Cor: Azul



Peso: 200 g
Raio: 30 cm
Cor: Verde

Classes vs. Objetos

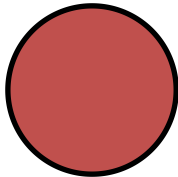
CLASSE



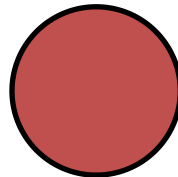
Peso
Raio
Cor



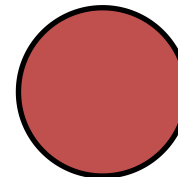
OBJETOS



Peso: 100 g
Raio: 25 cm
Cor:
Vermelha



Peso: 100 g
Raio: 25 cm
Cor:
Vermelha



Peso: 100 g
Raio: 25 cm
Cor:
Vermelha

Exemplo de Classe

```
class Ponto {
private:
    float _x;
    float _y;
public:
    Ponto(float x, float y) {
        _x = x;
        _y = y;
    }
    float get_x() {
        return _x;
    }
    float get_y() {
        return _y;
    }
    void translacao(double dx, double dy) {
        _x += dx;
        _y += dy;
    }
};
```

Exemplo de Classe

```
class Ponto {  
private:  
    float _x;  
    float _y;  
public:  
    Ponto(float x, float y) {  
        _x = x;  
        _y = y;  
    }  
    float get_x() {  
        return _x;  
    }  
    float get_y() {  
        return _y;  
    }  
    void translacao(double dx, double dy) {  
        _x += dx;  
        _y += dy;  
    }  
};
```

Atributos da classe

Construtor

Getters

Nossa função de translação

Usando o objeto

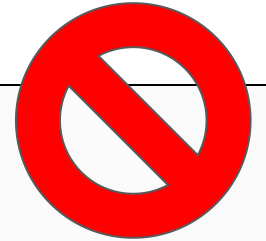
- Para utilizar o objeto usamos os métodos
 - Funções

```
#include <iostream>
// ... Declaracao do Ponto aqui em cima ... //
int main() {
    Ponto ponto(7, 9);
    std::cout << "Valor de x: " << ponto.get_x() << std::endl;
    std::cout << "Valor de y: " << ponto.get_y() << std::endl;
    ponto.translacao(3, 1);
    std::cout << "Valor de x: " << ponto.get_x() << std::endl;
    std::cout << "Valor de y: " << ponto.get_y() << std::endl;
}
```

Usando o objeto

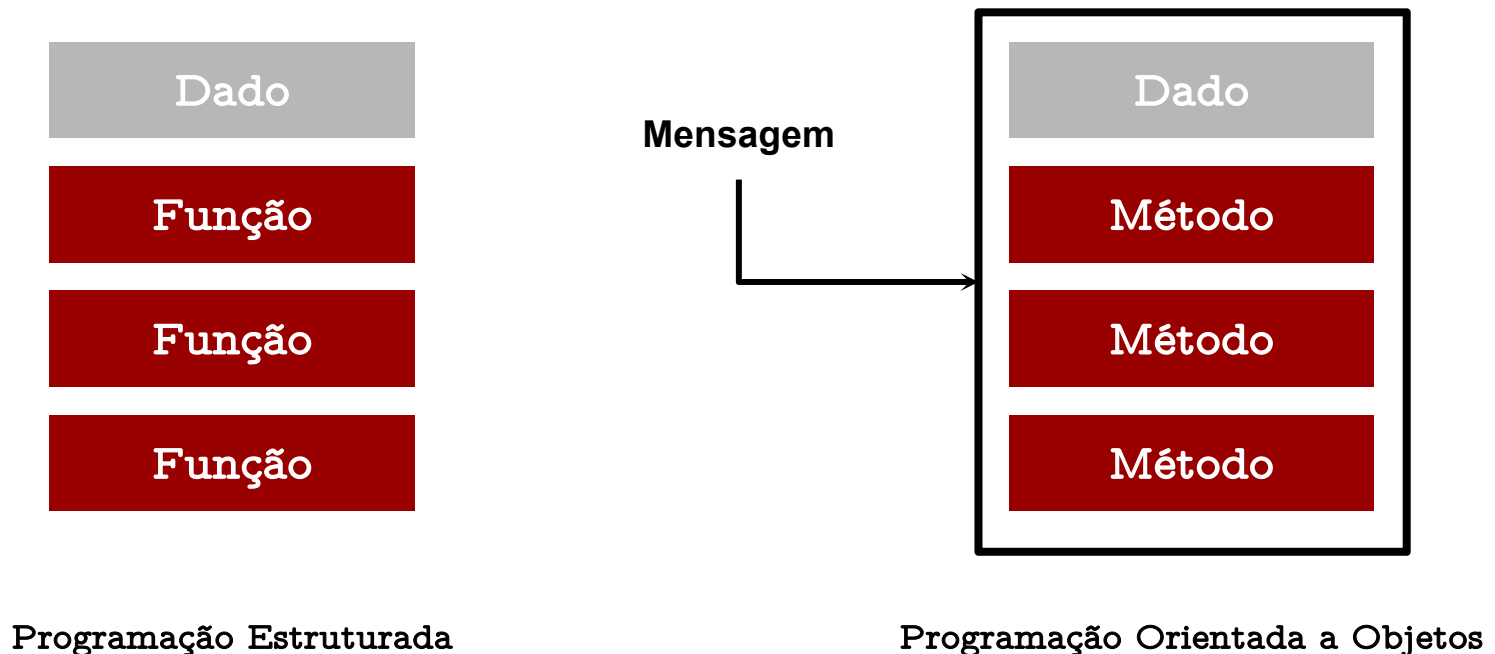
- Não temos acesso direto aos atributos
- Erro de compilação

```
#include <iostream>
// ... Declaracao do Ponto aqui em cima ... //
int main() {
    Ponto ponto(7, 9);
    std::cout << "Valor de x: " << ponto._x << std::endl;
    std::cout << "Valor de y: " << ponto._y << std::endl;
}
```

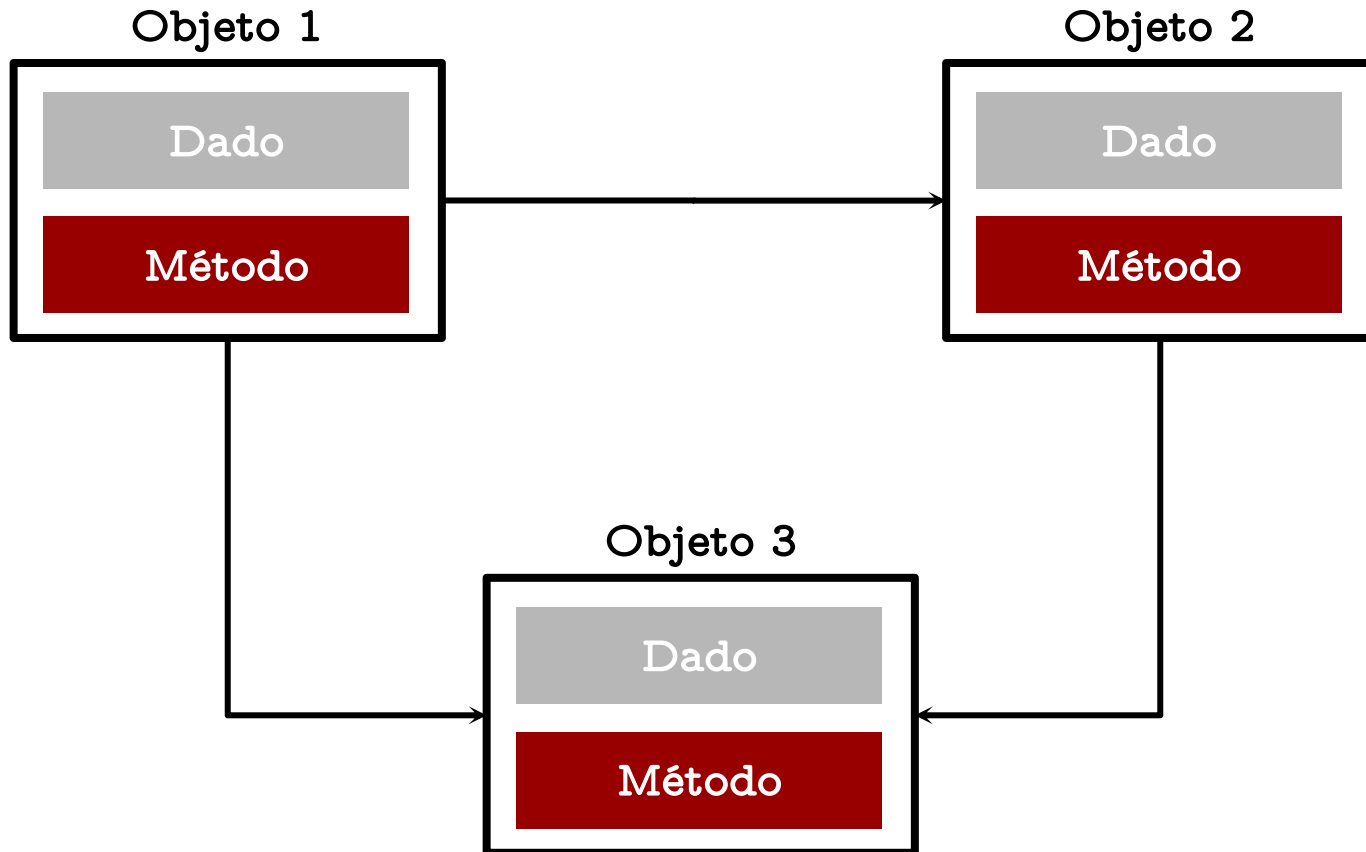


Objetos

- Dados ocultos do “mundo externo”
- Acessíveis somente via métodos internos



Objetos se comunicam por mensagens



Garantido a restrição de acesso

```
class Ponto {  
private:  Só pode ser acessado dentro da  
         classe!  
    float _x;  
    float _y;  
public:  Acesso de fora da classe!  
    Ponto(float x, float y) {  
        _x = x;  
        _y = y;  
    }  
    float get_x() {  
        return _x;  
    }  
    float get_y() {  
        return _y;  
    }  
    void translacao(double dx, double dy) {  
        _x += dx;  
        _y += dy;  
    }  
};
```

Limitando o acesso

private vs public

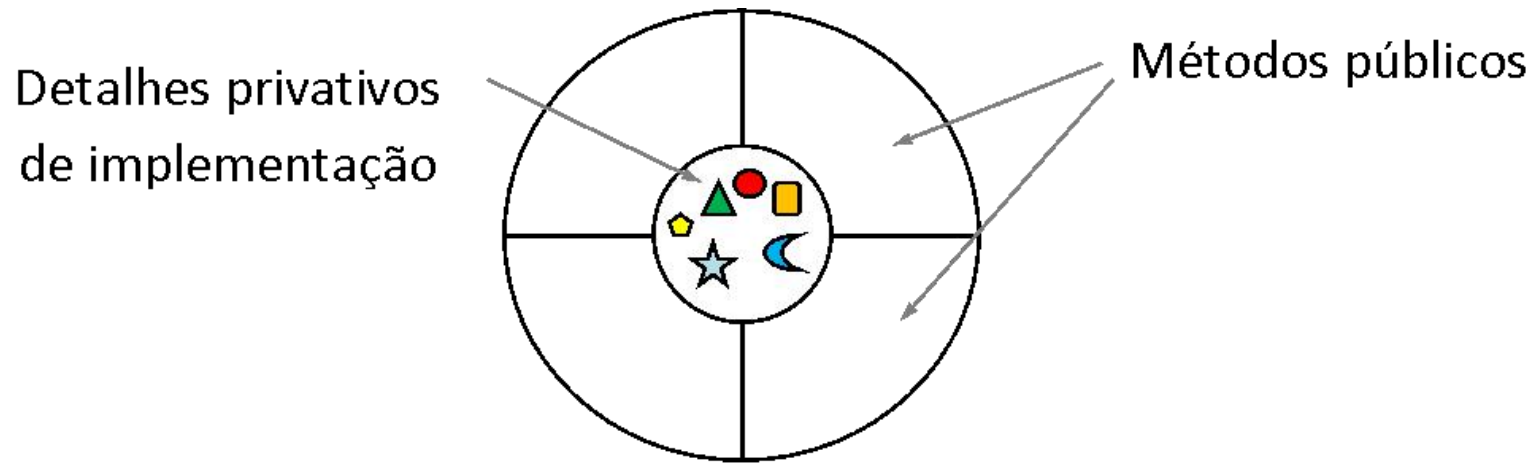
- A palavra **private** garante que ninguém "de fora" bagunce seu objeto
- Para fazer uso do mesmo, tem que chamar os métodos **public**
- Qual a vantagem?

Limitando o acesso

private vs public

- A palavra **private** garante que ninguém "de fora" bagunce seu objeto
- Para fazer uso do mesmo, tem que chamar os métodos **public**
- Qual a vantagem?
 - Resto do código não pode bagunçar a memória
 - Software feito em pequenos pedaços

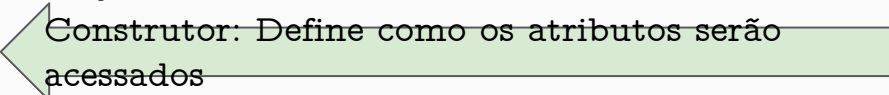
Encapsulamento



Construindo objetos

```
class Ponto {
private:
    float _x;
    float _y;
public:
    Ponto(float x, float y) {
        _x = x;
        _y = y;
    }
    float get_x() {
        return _x;
    }
    float get_y() {
        return _y;
    }
    void translacao(double dx, double dy) {
        _x += dx;
        _y += dy;
    }
};
```

Construtor: Define como os atributos serão acessados



Construindo objetos

Também podemos usar o Heap

```
// . . . Declaração do ponto aqui em cima //  
int main() {  
    Ponto *ponto = new Ponto(7, 9);  
    std::cout << "Valor de x: " << ponto->get_x() << std::endl;  
    std::cout << "Valor de y: " << ponto->get_y() << std::endl;  
    ponto->translacao(3, 1);  
    std::cout << "Valor de x: " << ponto->get_x() << std::endl;  
    std::cout << "Valor de y: " << ponto->get_y() << std::endl;  
    delete ponto;  
}
```

Construindo objetos

Acesso por ->

```
// . . . Declaração do ponto aqui em cima //  
int main() {  
    Ponto *ponto = new Ponto(7, 9);  
    std::cout << "Valor de x: " << ponto->get_x() << std::endl;  
    std::cout << "Valor de y: " << ponto->get_y() << std::endl;  
    ponto->translacao(3, 1);  
    std::cout << "Valor de x: " << ponto->get_x() << std::endl;  
    std::cout << "Valor de y: " << ponto->get_y() << std::endl;  
    delete ponto;  
}
```

Conceitos que vamos aprender

Programação Orientada a Objetos

Princípios

- Abstração
- Encapsulamento
- Herança
- Polimorfismo
- Modularidade
- Mensagens

Princípios
fundamentais

Programação Orientada a Objetos

Princípios - Abstração

- Modelagem de um domínio
 - Identificar artefatos de software
 - Ignorar aspectos não relevantes
 - Representação de detalhes relevantes do domínio do problema na linguagem de solução
- Classes são abstrações de conceitos

Programação Orientada a Objetos

Princípios - Encapsulamento

- Agrupamento dos dados e procedimentos correlacionados em uma mesma entidade
- Um sistema orientado a objetos baseia-se no contrato, não na implementação interna
- Proteção da estrutura interna (integridade)

Programação Orientada a Objetos

Princípios - Herança

- Permite a hierarquização das classes
- Classe especializada (subclasse, filha)
 - Herda as propriedades (atributos e métodos)
 - Pode sobrescrever/estender comportamentos
- Auxilia no reuso de código

Programação Orientada a Objetos

Princípios - Polimorfismo

- Tratar tipos diferentes de forma homogênea
- Classes distintas com métodos homônimos
- Diferentes níveis na mesma hierarquia
- Um método assume “diferentes formas”
- Apresenta diferentes comportamentos

Programação Orientada a Objetos

Princípios - Mensagens

- Comunicação entre objetos
 - Envio/recebimento de mensagens
 - Forma de invocar um comportamento
- Informação contida na mensagem
 - Utiliza o contrato firmado entre as partes