

INF 112 - Programação II

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
- ...

Classes vs Objetos vs TADs

- Classe/Struct

- Representa uma unidade de compilação
- Um módulo, um tipo
- Pode implementar um TAD
- Em C++, classes e structs são quase iguais

- TAD

- Conceito, ideia, abstração (representação)

Classes vs Objetos

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

Analogia

- Classes → fôrmas
- Memória → massa
- Objetos → cookies



Classes vs. Objetos

▪ Classe

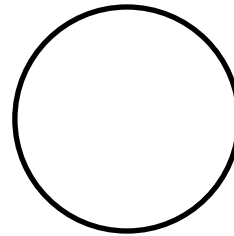
- Descrição de propriedades em comum de um grupo de objetos (conjunto)
- Um conceito
- Faz parte de um programa
- Exemplo: Pessoa
- Exemplo: Carro

▪ Objeto

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

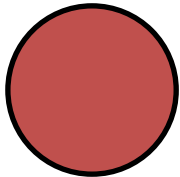
Classes vs. Objetos

CLASSE

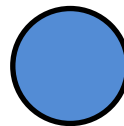


Peso
Raio
Cor

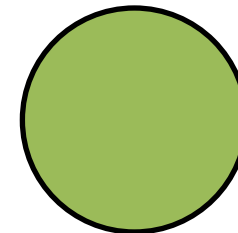
OBJETOS



Peso: 100 g
Raio: 25 cm
Cor: Vermelha



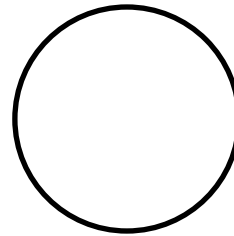
Peso: 50 g
Raio: 10 cm
Cor: Azul



Peso: 200 g
Raio: 30 cm
Cor: Verde

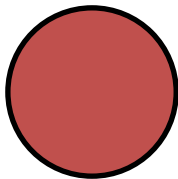
Classes vs. Objetos

CLASSE

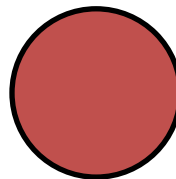


Peso
Raio
Cor

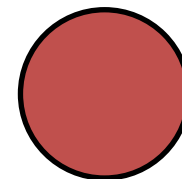
OBJETOS



Peso: 100 g
Raio: 25 cm
Cor: Vermelha



Peso: 100 g
Raio: 25 cm
Cor: Vermelha



Peso: 100 g
Raio: 25 cm
Cor: Vermelha

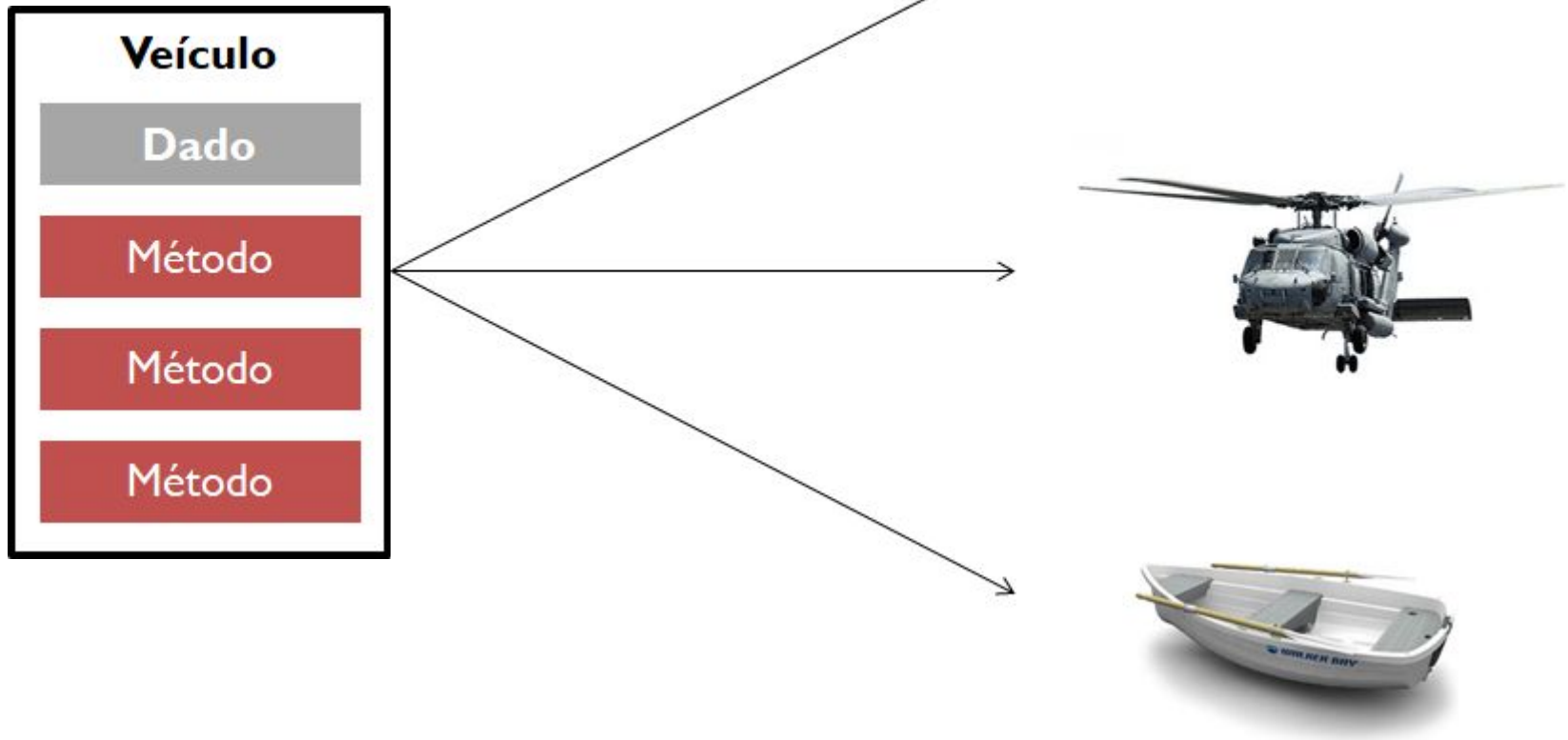
Classes

Abstração



Classes

Abstração



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

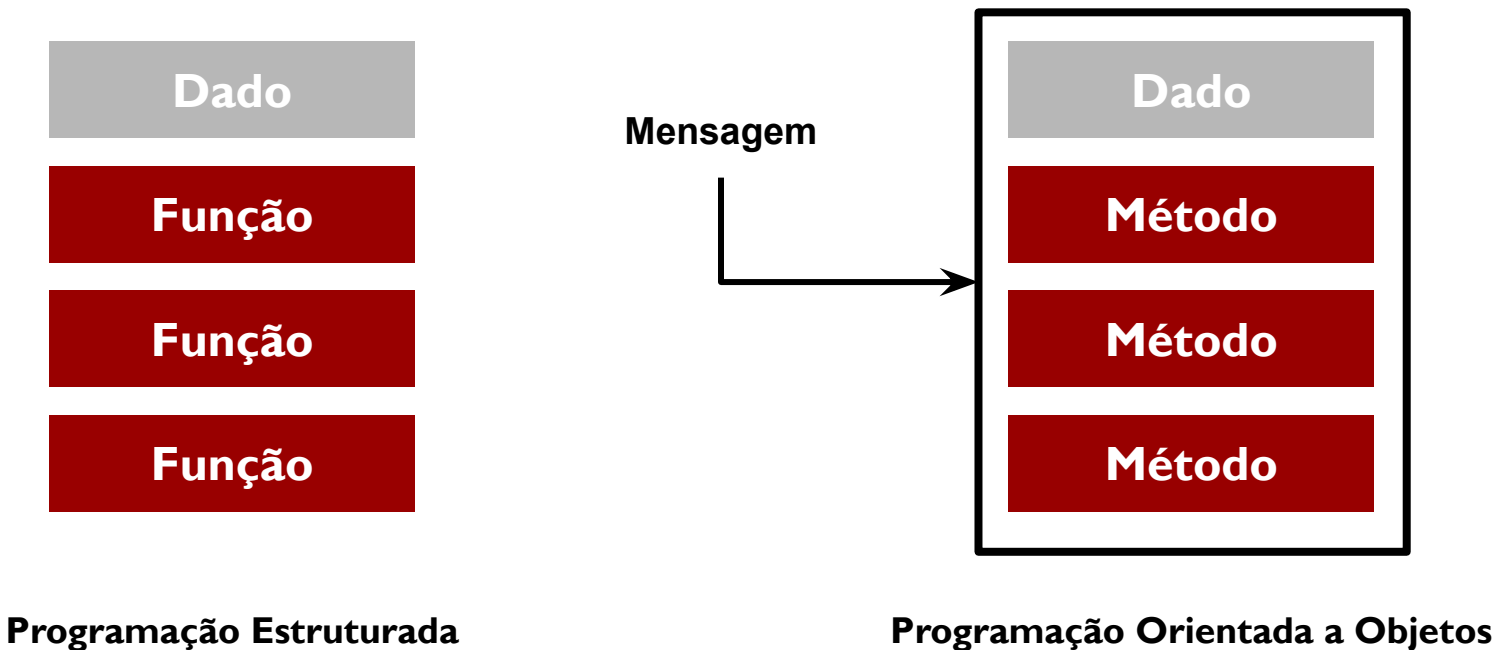
- Não temos acesso direto aos atributos
- Erro de compilação

```
#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;
}
```

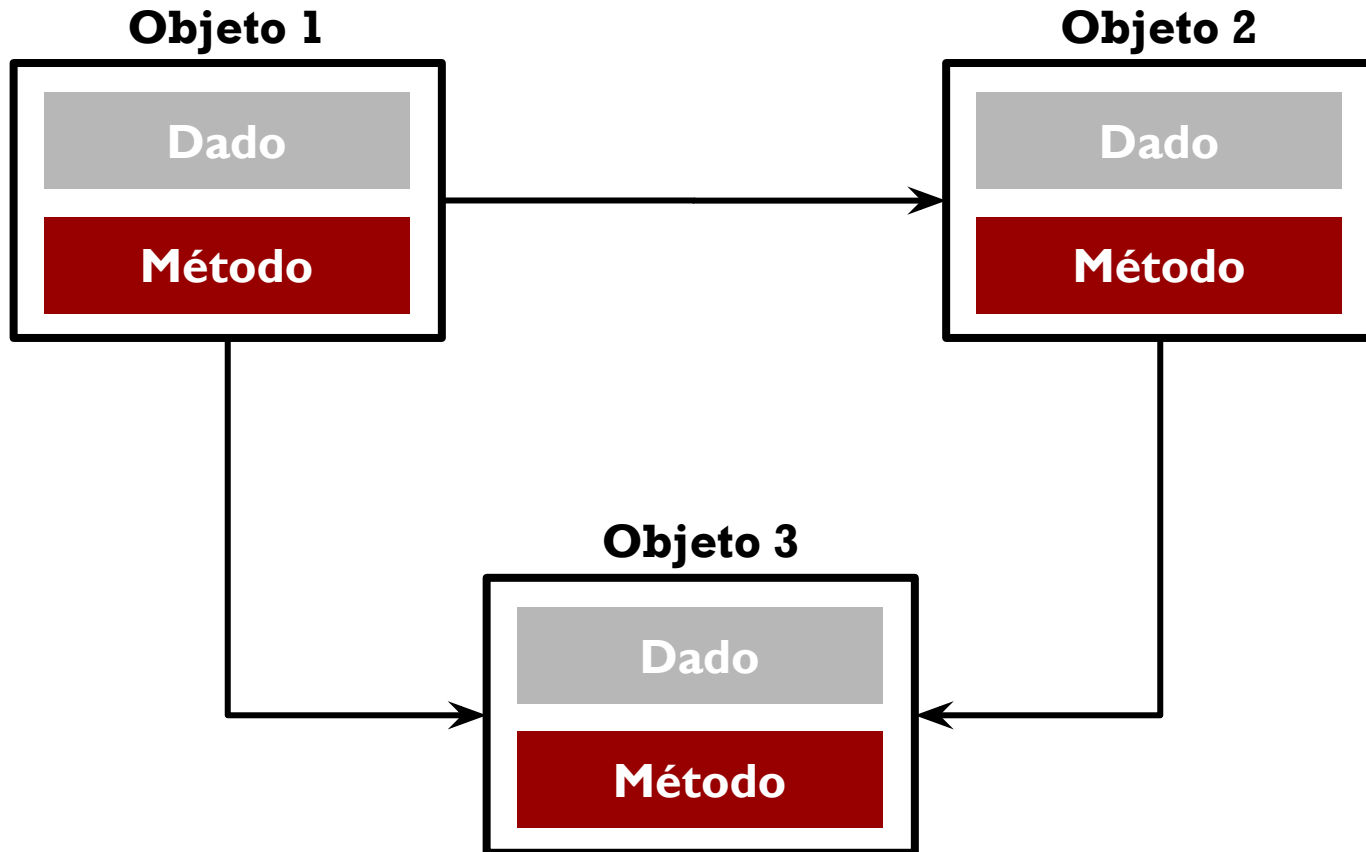


Objetos

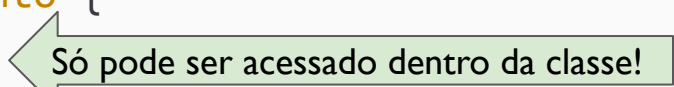
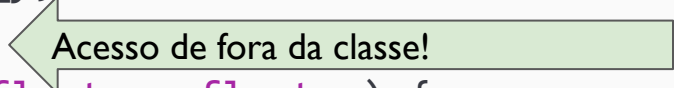
- Dados ocultos do “mundo externo”
- Acessíveis somente via métodos internos



Objetos se comunicam por mensagens



Garantido a restrição de acesso

```
class Ponto {  
private:    
    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;  
    }  
};
```

Limitando o acesso

private vs public

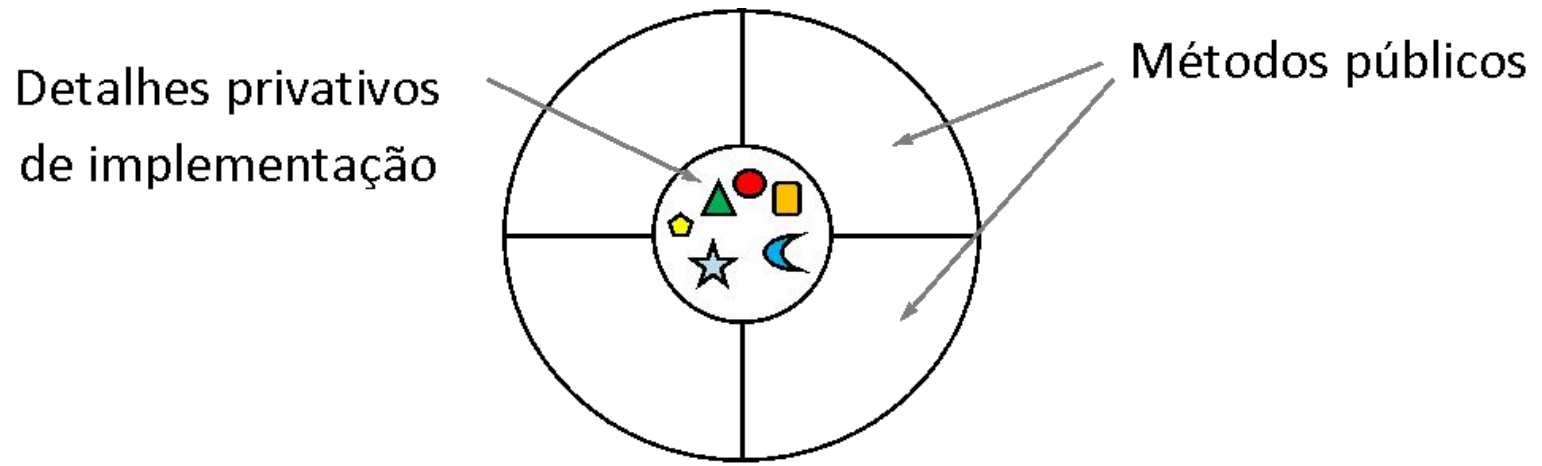
- A palavra **private** garante que ninguém "de fora" bagunce seu objeto
- Para fazer uso do mesmo, tem que chamar os métodos **public**
- Qual a vantagem?

Limitando o acesso

private vs public

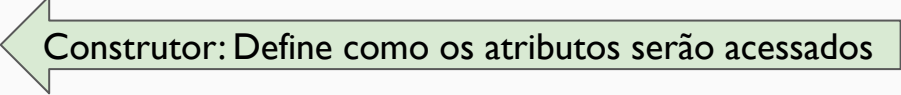
- A palavra **private** garante que ninguém "de fora" bagunce seu objeto
- Para fazer uso do mesmo, tem que chamar os métodos **public**
- Qual a vantagem?
 - Resto do código não pode bagunçar a memória
 - Software feito em pequenos pedaços

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

Exemplo do Banco

**Código usa conceitos mais avançados.
Motivar uso, não implementação!**

Exemplo de um main

```
#include <iostream>
#include <string>

#include "agencia.h"
#include "banco.h"
#include "cliente.h"
#include "conta.h"

int main(void) {
    // Mesma coisa de Banco banco(1, "Banco do Brasil")
    Banco banco = Banco(1, "Banco do Brasil");
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667", "Pampulha",
                                          "Belo Horizonte", 3217901);


    // Adicionando um novo cliente
    agencia.adiciona_cliente(1, "Julio Reis");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
    conta.depositar(200);
    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
}
```

Exemplo de um main

```
#include <iostream>
#include <string>

#include "agencia.h"
#include "banco.h"
#include "cliente.h"
#include "conta.h"
```



Módulos que vou utilizar

```
int main(void) {
    // Mesma coisa de Banco banco(1, "Banco do Brasil")
    Banco banco = Banco(1, "Banco do Brasil");
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667", "Pampulha",
                                           "Belo Horizonte", 3217901);
    // Adicionando um novo cliente
    agencia.adiciona_cliente(1, "Julio Reis");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
    conta.depositar(200);
    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
}
```




Objetos

Como fazer uso dos módulos?

```
#include <iostream>
#include <string>
```


```
#include "agencia.h"
#include "banco.h"
#include "cliente.h"
#include "conta.h"
```



Módulos que vou utilizar

```
int main(void) {
    // Mesma coisa de Banco banco(1, "Banco do Brasil")
    Banco banco = Banco(1, "Banco do Brasil");
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667", "Pampulha",
                                           "Belo Horizonte", 3217901);
    // Adicionando um novo cliente
    agencia.adiciona_cliente(1, "Julio Reis");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
    conta.depositar(200);
    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
}
```



Objetos

Chamada de função em POO

Cada objeto tem um estado, a função opera em cima do mesmo



```
int main(void) {  
    // Mesma coisa de Banco banco(1, "Banco do Brasil")  
    Banco banco = Banco(1, "Banco do Brasil");  
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667",  
                                           "Pampulha",  
                                           "Belo Horizonte", 3217901);  
}
```

Chamada de função em POO


Neste momento entramos no construtor

```
int main(void) {  
    // Mesma coisa de Banco banco(1, "Banco do Brasil")  
    Banco banco = Banco(1, "Banco do Brasil");  
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667",  
                                           "Pampulha",  
                                           "Belo Horizonte", 3217901);  
}
```

Chamada de função em POO

Neste momento entramos no construtor

```
int main(void) {  
    // Mesma coisa de Banco banco(1, "Banco do Brasil")  
    Banco banco = Banco(1, "Banco do Brasil");  
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667",  
                                           "Pampulha",  
                                           "Belo Horizonte", 3217901);  
}
```




```
Banco::Banco(int numero, std::string nome) {  
    _numero = numero;  
    _nome = nome;  
    _num_agencias = 0;  
}
```

Chamada de função em POO

Objeto é criado na memória

```
int main(void) {  
    // Mesma coisa de Banco banco(1, "Banco do Brasil")  
    Banco banco = Banco(1, "Banco do Brasil");  
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667",  
                                           "Pampulha",  
                                           "Belo Horizonte", 3217901);  
}
```

```
Banco::Banco(int numero, std::string nome) {  
    _numero = numero;  
    _nome = nome;  
    _num_agencias = 0;  
}
```



Memória

Um objeto é complicado

- Atributos + métodos
- Vamos representar de forma abstrar
 - Atributos apenas
 - Mora na pilha/stack do main neste caso

	_numero	_nome	_num_agencias
main::banco (stack)	1	"Banco do Brasil"	0

Chamada de função em POO

Chamando uma função de um objeto criado

```
int main(void) {  
    // Mesma coisa de Banco banco(1, "Banco do Brasil")  
    Banco banco = Banco(1, "Banco do Brasil");  
    ➔ Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667",  
                                             "Pampulha",  
                                             "Belo Horizonte", 3217901);  
}
```

Chamada de função em POO

Entramos no método do objeto

```
int main(void) {  
    // Mesma coisa de Banco banco(1, "Banco do Brasil")  
    Banco banco = Banco(1, "Banco do Brasil");  
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667",  
                                           "Pampulha",  
                                           "Belo Horizonte", 3217901);  
}
```

```
Agencia &Banco::cria_agencia(std::string logradouro, std::string bairro,  
                             std::string cidade, int cep) {  
    int numero = ++_num_agencias;  
    // Resto do código omitido por clareza  
}
```



Chamada de função em POO

Faz uso do estado atual **deste** objeto

	<code>_numero</code>	<code>_nome</code>	<code>_num_agencias</code>
<code>main::banco (stack)</code>	1	"Banco do Brasil"	0

```
Agencia &Banco::cria_agencia(std::string logradouro, std::string bairro,  
                             std::string cidade, int cep) {  
    int numero = ++_num_agencias;  
    // Resto do código omitido por clareza  
}
```




Chamada de função em POO

Faz uso do estado atual **deste** objeto

	<code>_numero</code>	<code>_nome</code>	<code>_num_agencias</code>
<code>main::banco (stack)</code>	1	"Banco do Brasil"	1

```
Agencia &Banco::cria_agencia(std::string logradouro, std::string bairro,
                             std::string cidade, int cep) {
    int numero = ++_num_agencias;
    // Resto do código omitido por clareza
    }
```



Múltiplos objetos

Nada impede de termos $n > 1$ objetos com estados diferentes

```
int main(void) {  
    Banco bb = Banco(1, "Banco do Brasil");  
    Banco bradesco = Banco(2, "Bradesco");  
}
```

	_numero	_nome	_num_agencias
main::bb (stack)	1	"Banco do Brasil"	0
main::bradesco (stack)	2	"Bradesco"	0

Múltiplos objetos

Nada impede de termos $n > 1$ objetos com estados diferentes


```
int main(void) {  
    Banco bb = Banco(1, "Banco do Brasil");  
    Banco bradesco = Banco(2, "Bradesco");  
    Agencia &agencia = bb.cria_agencia("Antonio Carlos, 6667",  
                                        "Pampulha",  
                                        "Belo Horizonte", 3217901);  
}
```

	_numero	_nome	_num_agencias
main::bb (stack)	1	"Banco do Brasil"	1
main::bradesco (stack)	2	"Bradesco"	0

Os objetos deste programa moram aonde?

```
#include <iostream>
#include <string>

#include "agencia.h"
#include "banco.h"
#include "cliente.h"
#include "conta.h"
```



Módulos que vou utilizar

```
int main(void) {
    // Mesma coisa de Banco banco(1, "Banco do Brasil")
    Banco banco = Banco(1, "Banco do Brasil");
    Agencia &agencia = banco.cria_agencia("Antonio Carlos, 6667", "Pampulha",
                                           "Belo Horizonte", 3217901);
    // Adicionando um novo cliente
    agencia.adiciona_cliente(1, "Julio Reis");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
    conta.depositar(200);
    std::cout << "Saldo de Julio " << conta.get_saldo() << std::endl;
}
```



Objetos

Múltiplas Classes em um Projeto

Uma Classe

Modelando um vírus infectando pacientes

Um **Vírus** tem um:

nome (string)

Uma força de infecção:

força (double)

Infecta um paciente quando:

força > resistência (do paciente)

Modularizando o código

Passo I

- Criar o cabeçalho/header (.h)
- O cabeçalho é o contrato da sua classe
 - Os usuários vão ler o mesmo
 - Não precisam entender como é implementado
- Aprender a separar:
 - Contratos de Comportamento

Cabeçalho (virus.h)

```
#ifndef INF112_VIRUS_H
#define INF112_VIRUS_H

#include <string>

class Virus {
private:
    std::string _nome;
    double _forca;
public:
    Virus(std::string nome, double forca);
    std::string get_nome();
    double get_forca();
};

#endif
```


Cabeçalho (virus.h)

```
#ifndef INF112_VIRUS_H  
#define INF112_VIRUS_H
```

Guarda de segurança. Evita módulos com o mesmo nome

```
#include <string>
```

```
class Virus {  
private:
```

```
    std::string _nome;
```

```
    double _forca;
```

Atributos Privado

```
public:
```

```
    Virus(std::string nome, double forca);
```

Construtor

```
    std::string get_nome();
```

```
    double get_forca();
```

Métodos públicos

```
};
```

```
#endif
```

Fim da guarda!

Note que não temos o corpo dos métodos

```
#ifndef INF112_VIRUS_H  
#define INF112_VIRUS_H
```

Guarda de segurança. Evita módulos com o mesmo nome

```
#include <string>
```

```
class Virus {  
private:
```

```
    std::string _nome;
```

```
    double _forca;
```

Atributos Privado

```
public:
```

```
    Virus(std::string nome, double forca);
```

Construtor

```
    std::string get_nome();
```

```
    double get_forca();
```

Métodos públicos

```
};
```

```
#endif
```

Fim da guarda!

Classes

Membros

- Tipos de componentes
 - Membros de instância
 - Membros de classe (estáticos)
 - Assunto futuro
 - Procedimentos de inicialização
 - Procedimentos de destruição

Implementando o .cpp

Passo 2

- [Geralmente] Cada .h tem um .cpp
 - Existem exceções
 - Exceção no exemplo do banco (endereco.h)
- No .cpp vai o código

Arquivo .cpp. Implementa os métodos

```
#include "virus.h"
```

← Tem que incluir o .h

```
Virus::Virus(std::string nome, double forca) {  
    _nome = nome;  
    _forca = forca;  
}
```

```
std::string Virus::get_nome() {  
    return _nome;  
}
```

```
double Virus::get_forca() {  
    return _forca;  
}
```

← Implementação do método

Arquivo .cpp. Implementa os métodos

```
#include "virus.h"
```

Tem que incluir o .h

```
Virus::Virus(std::string nome, double forca) {  
    _nome = nome;  
    _forca = forca;  
}
```

```
std::string Virus::get_nome() {  
    return _nome;  
}
```

Indica de qual classe pertence o método.

```
double Virus::get_forca() {  
    return _forca;  
}
```

Implementação do método

Boas práticas de .h

- É possível ter mais de uma classe por .h
- Por isso o uso de ::

Boas práticas de .h

- Possível de fazer, porém **evitar**

```
#ifndef INF112_DUAS_CLASSES_H
#define INF112_DUAS_CLASSES_H

class Class1 {
private:
    int _atributo;
public:
    int get_atributo();
};

class Class2 {
private:
    int _atributo;
public:
    int get_atributo();
};

#endif
```

```
#include "duasclasses.h"

int Class1::get_atributo() {
    return _atributo;
};

int Class2::get_atributo() {
    return _atributo;
};
```

Mesmo nome porém de classes diferentes

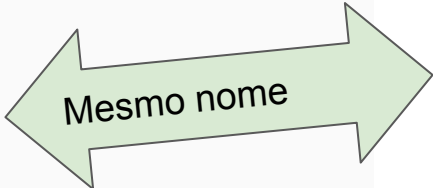
this

```
#ifndef INF112_VIRUS_H
#define INF112_VIRUS_H
```

```
#include <string>
```

```
class Virus {
private:
    std::string nome;
    double forca;
public:
    Virus(std::string nome,
          double forca);
    std::string get_nome();
    double get_forca();
};
```

```
#endif
```



Mesmo nome

```
#include "virus.h"
```

```
Virus::Virus(std::string nome, double forca) {
    this->nome = nome;
    this->forca = forca;
}
```

```
std::string Virus::get_nome() {
    return this->nome;
}
```

```
double Virus::get_forca() {
    return this->forca;
}
```

this

"Função identidade" de um objeto

- Em linguagens OO é comum ter um atributo implícito
- Em C++ o nome do mesmo é **this**

this

"Função identidade" de um objeto

- **this** é um ponteiro para o próprio objeto
- Útil quando os atributos têm o mesmo nome dos parâmetros do método

```
#include "virus.h"
```

```
Virus::Virus(std::string nome, double forca) {  
    this->nome = nome;  
    this->forca = forca;  
}
```

this

Quando usar

- Tem pessoas que preferem sempre usar
- Fica a seu critério
 - O uso de `_nome` antes de atributos é apenas um atalho para evitar `this`.

```
#include "virus.h"
```

```
Virus::Virus(std::string nome, double forca) {  
    this->nome = nome;  
    this->forca = forca;  
}
```

this

Não seria melhor uma referência?

- Lembrando que ponteiros e referências são quase iguais
- Porém o ponteiro veio antes
 - Então this é um ponteiro. Uso de ->

```
#include "virus.h"
```

```
Virus::Virus(std::string nome, double forca) {  
    this->nome = nome;  
    this->forca = forca;  
}
```

Classe Paciente

Seguimos o mesmo exemplo da classe Virus

- Criar um .h
- Criar um .cpp

paciente.h

```
#ifndef INF112_PACIENTE_H
#define INF112_PACIENTE_H

#include <string>

#include "virus.h"

class Paciente {
private:
    std::string _nome;
    double _resistencia;
    bool _infectado;
    Virus *_virus;
public:
    Paciente(std::string nome, double resistencia);
    Paciente(std::string nome, double resistencia, Virus *virus);
    bool esta_infectado();
    Virus *get_virus();
    std::string get_nome();
    void contato(Paciente &contato);
    void curar();
};

#endif
```

Construtor

```
#ifndef INF112_PACIENTE_H  
#define INF112_PACIENTE_H
```

```
// . . .
```

```
class Paciente {  
private:
```

```
    // . . .
```

```
    Virus *_virus;
```

```
    // . . .
```

```
// . . .
```

```
#endif
```



Lembrando que virus é um ponteiro

```
#include "paciente.h"
```

```
Paciente::Paciente(std::string nome, double resistencia) {
```


```
    _nome = nome;
```

```
    _resistencia = resistencia;
```

```
    _infectado = false;
```

```
    _virus = nullptr;
```

```
}
```



Iniciamos para nullptr

Construtor

```
#ifndef INF112_PACIENTE_H
#define INF112_PACIENTE_H
```

```
// . . .
```

```
class Paciente {
private:
```

```
    // . . .
```

```
    bool _infectado;
```

```
    Virus *_virus;
```

```
    // . . .
```

```
// . . .
```

```
#endif
```

Duas variáveis mantêm o estado. Podemos simplificar no futuro

```
#include "paciente.h"
```

```
Paciente::Paciente(std::string nome, double resistencia) {
```

```
    _nome = nome;
```

```
    _resistencia = resistencia;
```

```
    _infectado = false;
```

```
    _virus = nullptr;
```

```
}
```


Setamos para false

Segundo Construtor

- Podemos ter várias formas de construir o mesmo objeto
- Basta que tenha parâmetros diferentes
 - Overload de funções
 - Pode ser feito para qualquer método

```
#include "paciente.h"
```

```
Paciente::Paciente(std::string nome, double resistencia, Virus *virus) {  
    _nome = nome;  
    _resistencia = resistencia;  
    _infectado = true;  
    _virus = virus;  
}
```



Paciente já infectado

Lembrando

Métodos

- Procedimentos que podem modificar ou apenas acessar os valores dos atributos
- Controle de visibilidade
 - Determinar membros disponíveis para acesso
- Sobrecarga (overloading)
 - Dois ou mais métodos com mesmo nome
 - Lista de parâmetros (tipos) deve ser diferente!

Métodos

Focando no mais complicado


- Temos um método que:
 - Recebe um outro objeto do mesmo tipo
 - Usa **this** para diferenciar o local da memória

```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

Métodos

Focando no mais complicado

- Temos um método que:
 - Recebe um outro objeto do mesmo tipo
 - Usa **this** para diferenciar o local da memória



```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

Métodos

Quando chamamos Paciente::contato

```
void main(void) {  
→ Virus *virus = new Virus("V1", 0.8);  
  Paciente p1("John", 0.2, virus);  
  Paciente p2("Paul", 0.3);  
  p2.contato(p1);  
  delete virus;  
}
```

Stack

Heap

Métodos

Quando chamamos Paciente::contato

```
void main(void) {  
    Virus *virus = new Virus("V1", 0.8);  
    Paciente p1("John", 0.2, virus);  
    Paciente p2("Paul", 0.3);  
    p2.contato(p1);  
    delete virus;  
}
```

Stack		
nome	End.	Valor
main::virus	0x0044	0x0016

Heap		
tipo	End.	Valor
Virus	0x0016	{"V1", 0.8}

Métodos

Quando chamamos `Paciente::contato`

```
void main(void) {  
    Virus *virus = new Virus("V1", 0.8);  
    Paciente p1("John", 0.2, virus);  
    Paciente p2("Paul", 0.3);  
    p2.contato(p1);  
    delete virus;  
}
```

Stack		
nome	End.	Valor
main::virus	0x0044	0x0016
main::p1	0x0036	{"John", 0.2, true, *_virus=0x0016}
main::p2	0x0028	{"Paul", 0.3, false, nullptr}

Heap		
tipo	End.	Valor
Virus	0x0016	{"V1", 0.8}

Métodos

Quando chamamos Paciente::contato

```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

Stack		
nome	End.	Valor
main::virus	0x0044	0x0016
main::p1	0x0036	{"John", 0.2, true, *_virus=0x0016}
main::p2	0x0028	{"Paul", 0.3, false, nullptr}

Heap		
tipo	End.	Valor
Virus	0x0016	{"V1", 0.8}

Métodos

contato = p2; this = p1;

```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

Stack		
nome	End.	Valor
main::virus	0x0044	0x0016
main::p1	0x0036	{"John", 0.2, true, *_virus=0x0016}
main::p2	0x0028	{"Paul", 0.3, false, nullptr}

Heap		
tipo	End.	Valor
Virus	0x0016	{"V1", 0.8}

Métodos

contato = p2; this = p1;

```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

Stack		
nome	End.	Valor
main::virus	0x0044	0x0016
main::p1	0x0036	{"John", 0.2, true, *_virus=0x0016}
main::p2	0x0028	{"Paul", 0.3, false, nullptr}

Heap		
tipo	End.	Valor
Virus	0x0016	{"V1", 0.8}

Métodos

contato = p2; this = p1;

```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

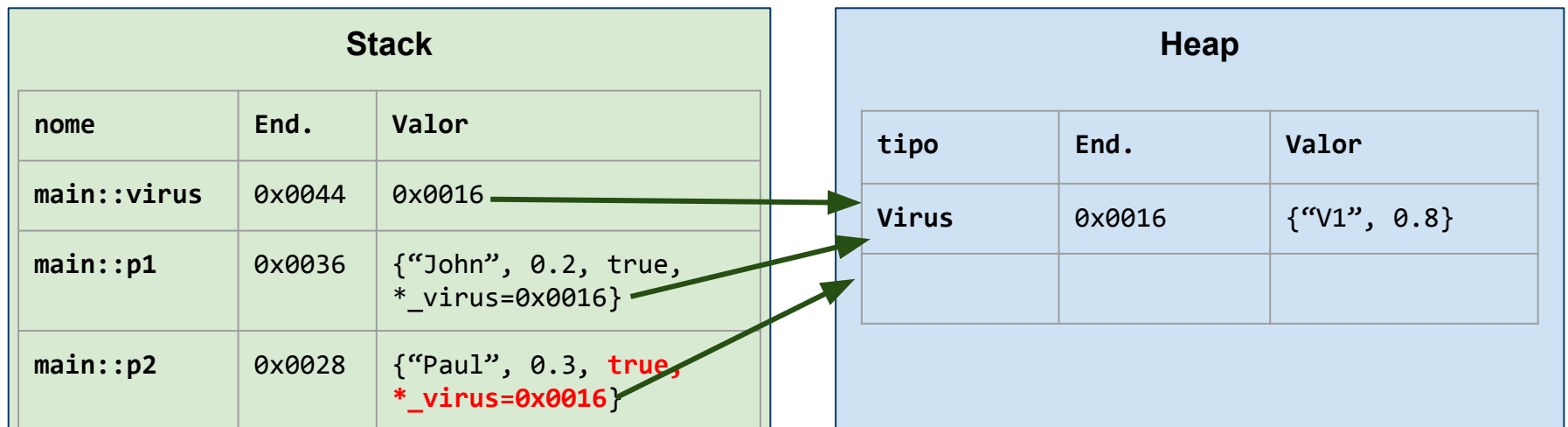
Stack		
nome	End.	Valor
main::virus	0x0044	0x0016
main::p1	0x0036	{"John", 0.2, true, *_virus=0x0016}
main::p2	0x0028	{"Paul", 0.3, false, nullptr}

Heap		
tipo	End.	Valor
Virus	0x0016	{"V1", 0.8}

Métodos

contato = p2; this = p1;

```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```



Aquisição de Recurso é Inicialização

Se o main chamou **new** o main chama **delete**

```
void main(void) {  
    Virus *virus = new Virus("V1", 0.8);  
    Paciente p1("John", 0.2, virus);  
    Paciente p2("Paul", 0.3);  
    p2.contato(p1);  
    delete virus;  
}
```

