

DCC004 - Algoritmos e Estruturas de Dados II

Classes e Uso de 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

Introdução

- Objeto
 - Identidade (referência única)
 - Estado
 - Comportamento
- Representação de um elemento no domínio

Classes

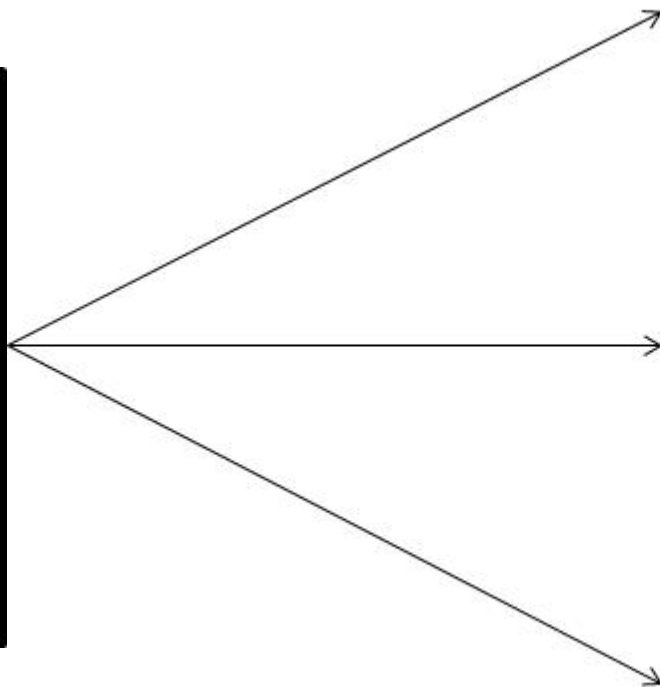
- Representam uma categoria de elementos
 - O Objeto representa um item em particular
 - Não existem no contexto da execução!
- Definem uma lógica estática
 - Relacionamentos entre classes não mudam
 - Relacionamentos entre objetos são dinâmicos

Classes

- Suportam os conceitos de
 - Encapsulamento
 - Herança
 - Polimorfismo

Classes

Abstração



Classes

Convenções

```
class ClasseTeste {  
    public:  
        int minhaVariavel;  
};
```

- Classes
 - Primeira letra maiúscula (UpperCamelCase)
- Atributos, variáveis locais e métodos
 - Primeira letra minúscula (lowerCamelCase)

Classes

Membros

- Membros \Rightarrow Componentes de uma classe
 - Atributos
 - Métodos
- Classe é uma estrutura:
 - É importante seguir um padrão lógico!
 - Atributos primeiro, Métodos depois
 - Agrupar responsabilidades
 - Atenção à questão de inicialização

Classes

Membros

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

Classes

Membros

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

Classes

Membros

- Procedimentos de inicialização
 - Usados apenas na criação de um novo objeto
- Procedimentos de destruição
 - Usados para liberar os recursos adquiridos na criação e utilizados por um certo objeto

Classes

Referência

- Todo objeto possui uma referência
 - Utilizada para acessar atributos e métodos
- Instanciação
 - Criação de um novo Objeto
 - Alocação de memória
 - Cria e retorna a referência do novo Objeto

Exemplo do Banco (POO na Prática)

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, "Andrei Markov");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

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

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, "Andrei Markov");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

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

Módulos que vou
utilizar

Objetos

Como fazer uso dos módulos?

```
#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, "Andrei Markov");
    // Criando uma conta (no momento, 1 conta por cliente)
    Conta &conta = agencia.cria_conta(1);

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

Módulos que vou utilizar

Objetos

Relembrando: Classes

- Representam uma categoria de elementos
 - O Objeto representa um item em particular
 - Não existem no contexto da execução!
- Definem uma lógica estática
 - Relacionamentos entre classes não mudam
 - Relacionamentos entre objetos são dinâmicos


Classes

Abstração



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 abstrata
 - 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?

- Em um código bem feito:
 - Não vai importar para quem **usa** os objetos
- Cada objeto deve se responsabilizar por:
 - Memória alocada
 - Arquivos abertos
 - Recursos adquiridos

Aquisição de Recurso é Inicialização

- Em um código bem feito:
 - Não vai importar para quem **usa** os objetos
- Cada objeto deve se responsabilizar por:
 - Memória alocada
 - Arquivos abertos
 - Recursos adquiridos
- Se o objeto cria (**new**)
o objeto deve liberar (**delete**)

Classes

Convenções

```
class ClasseTeste {  
public:  
    int variavel_da_classe;  
};
```

- Classes
 - Primeira letra maiúscula (UpperCamelCase)
 - Atributos, variáveis locais e métodos
 - Separados por __ (meu_estilo_pessoal)
 - Casa com o estilo de C++ (STL)

Sobre Estilos

É uma questão de cada projeto/pessoa

Underscore

```
class ClasseTeste {  
private:  
    int valor_da_conta;  
public:  
    int get_valor();  
    int set_valor();  
};
```

CamelCase

```
class ClasseTeste {  
private:  
    int valorDaConta;  
public:  
    int getValor();  
    int setValor();  
};
```


Classes

Membros

- Membros \Rightarrow Componentes de uma classe
 - Atributos
 - Métodos
- Classe é uma estrutura “não ordenada”
 - É importante seguir um padrão lógico!
 - Atributos primeiro, Métodos depois
 - Agrupar responsabilidades
 - Atenção à questão de inicialização

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 1

- 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 DCC004_VIRUS_H
#define DCC004_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 DCC004_VIRUS_H  
#define DCC004_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 DCC004_VIRUS_H  
#define DCC004_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 DCC004_DUAS_CLASSES_H
#define DCC004_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

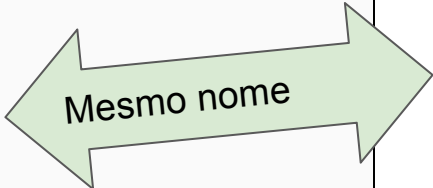
this

```
#ifndef DCC004_VIRUS_H  
#define DCC004_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 DCC004_PACIENTE_H
#define DCC004_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 DCC004_PACIENTE_H
#define DCC004_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 DCC004_PACIENTE_H
#define DCC004_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;
}
```

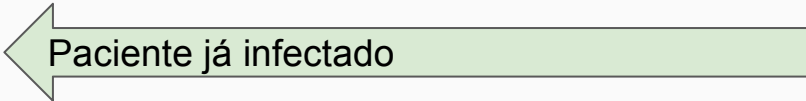
← Setamos para false

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;
}
```



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

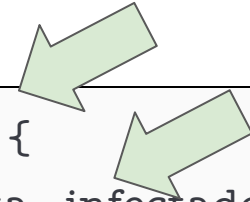
```
void Paciente::contato(Paciente &contato) {  
    if (contato.esta_infectado() && !this->esta_infectado()) {  
        if (contato.get_virus()->get_forca() > _resistencia) {  
            _infectado = true;  
            _virus = contato.get_virus();  
        }  
    }  
}
```

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();  
        }  
    }  
}
```

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

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

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;  
}
```

→

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

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

Classes

Membros

- Procedimentos de inicialização
 - Usados apenas na criação de um novo objeto
- Procedimentos de destruição
 - Usados para liberar os recursos adquiridos na criação e utilizados por um certo objeto

Classes

Referência

- Todo objeto possui uma referência
 - Utilizada para acessar atributos e métodos
- Instanciação
 - Criação de um novo Objeto
 - Alocação de memória
 - Cria e retorna a referência do novo Objeto

Classes

```
class Ponto {  
    public:  
        int x;  
        int y;  
};
```

```
int main() {  
    Ponto p1;  
    // Stack  
    Ponto *p2 = new Ponto(); // Heap  
  
    cout << p1.x << endl;  
    cout << p2->y << endl;  
  
    delete p2;  
    // Liberar  
    return 0;  
}
```

Instanciação (criação)
de dois Objetos do
tipo Ponto.

Classes



Classes

Atributos de instância

- Valores padrão
 - Tipos numéricos: valor 0 (zero)
 - Tipo boolean: valor 0 (false)
 - Atenção: não confiar nessa inicialização!
- Demais atributos:
 - Não são “automaticamente” inicializados
 - Ponteiros \Rightarrow Lixo (segmentation fault)

Classes

Atributos de instância

```
class Ponto {  
    public:  
        int x = 99;  
        int y;  
};
```

```
int main() {  
    Ponto p1;  
    Ponto *p2 = new Ponto();
```

```
p1.y = 123;  
p2->y = 123;
```

Acessando atributos

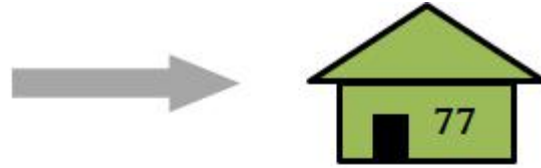
```
cout << p1.x << "\t" << p1.y << endl;  
cout << p2->x << "\t" << p2->y << endl;
```

```
delete p2;  
return 0;
```

```
}
```

Classes

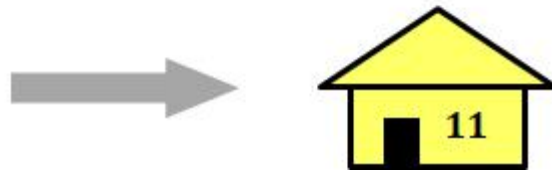
Atributos de instância



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



```
Casa c2;  
c2.numero = 55;  
c2.cor = "vermelha";
```



```
Casa c3;  
c3.numero = 11;  
c3.cor = "amarela";
```

Classes

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!

Classes

Métodos

- A palavra `this` serve para referenciamento
- Utilizada internamente à qualquer método não estático para referenciar o Objeto atual
- Principais utilizações
 - Passar uma referência para o Objeto atual
 - Evitar conflitos de nome
 - Facilita a compreensão do código!

Classes

Métodos sobrecarregados

- Diferenciáveis pela lista de parâmetros
 - Ordem dos tipos de parâmetros é importante
- Não são diferenciáveis pelo tipo de retorno
 - Podem possuir diferentes tipos de retorno desde que possuam diferentes parâmetros
 - Co/contra-variância no tipo de retorno (!)

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

Construtores

- Método chamado durante a instanciação
 - Classe declara zero ou mais construtores
 - Possui o construtor padrão (sem parâmetros)
- Devem possuir o mesmo nome da Classe
 - Selecionados através da lista de parâmetros
- 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!

```
int main() {
Ponto p1;
    Ponto p2(10, 20);
    Ponto p3(50);

    return 0;
}
```


Classes

Construtores

```
class Ponto {
    public:
        int x;
        int y;

    Ponto() {
        this->x = -1;
        this->y = -1;
    }

    Ponto(int x, int y) {
        this->x = x;
        this->y = y;
    }

    Ponto(int xy) {
        this->x = xy;
        this->y = xy;
    }
};
```

- Algum problema?
- Como seria possível melhorar o código?

Classes

Construtores

```
class Ponto {
    public:
        int x;
        int y;

        Ponto() : Ponto(-1, -1) {}

        Ponto(int x, int y) {
            this->x = x;
            this->y = y;
        }

        Ponto(int xy) : Ponto(xy, xy) {}
};
```

- Delegating constructors: C++11
- <https://isocpp.org/wiki/faq/ctors#init-methods>

Classes

Destrutores

- Método chamado para a finalização
 - Quando o *lifetime* de um objeto chega ao fim
 - Libera os recursos alocados na execução
- Devem possuir o mesmo nome da Classe
 - Semelhante aos construtores
 - Devem ser precedidos por ‘~’

Classes

Destrutores

```
class Object {  
  
    int campo;  
  
    public:  
        Object(int valor) : campo(valor) { }  
  
        ~Object() {  
            cout << "~Object" << valor << endl;  
        }  
};
```

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

Qual será a saída?

Classes

Destrutores

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

Qual será a saída?

Classes

Atributos estáticos

- Não estão associados a uma instância
- Atributos de Classe
- Atributos compartilhados pelas instâncias
- Ocupam uma posição única na memória
- Geralmente utilizados para constantes

Classes

Atributos estáticos

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

Classes

Atributos estáticos

- que será impresso na tela?

```
int main() {
```

```
    ClasseAtributoEstatico c1;
```

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



?

```
    ClasseAtributoEstatico c2;
```

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



?

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



?

```
    return 0;
```

```
}
```


Classes

Atributos estáticos

- que será impresso na tela?

```
int main() {
```

```
    ClasseAtributoEstatico c1;
```

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



1

```
    ClasseAtributoEstatico c2;
```

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



2

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



2

```
    return 0;
```

```
}
```

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 métodos de instância que são resolvidos baseados no tipo do objeto em tempo de execução
- Não podem ser sobrescritos
- Dentro de métodos *static* só podem ser acessados atributos e métodos *static*!

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;  
}
```

Classes abstratas

- Uma Classe que não pode ser instanciada
- Define um conjunto de métodos
 - Totalmente implementados
 - Parcialmente implementados (contrato)
- Usadas na implementação de outra classe
- Pelo menos uma função virtual pura
 - Quando é usado um especificador-puro (= 0)
 - Função virtual \Rightarrow Herança/Polimorfismo

Classes abstratas

```
class Ponto {
    public:
        int x, y, z;

        virtual void imprimirInfo() = 0;
};

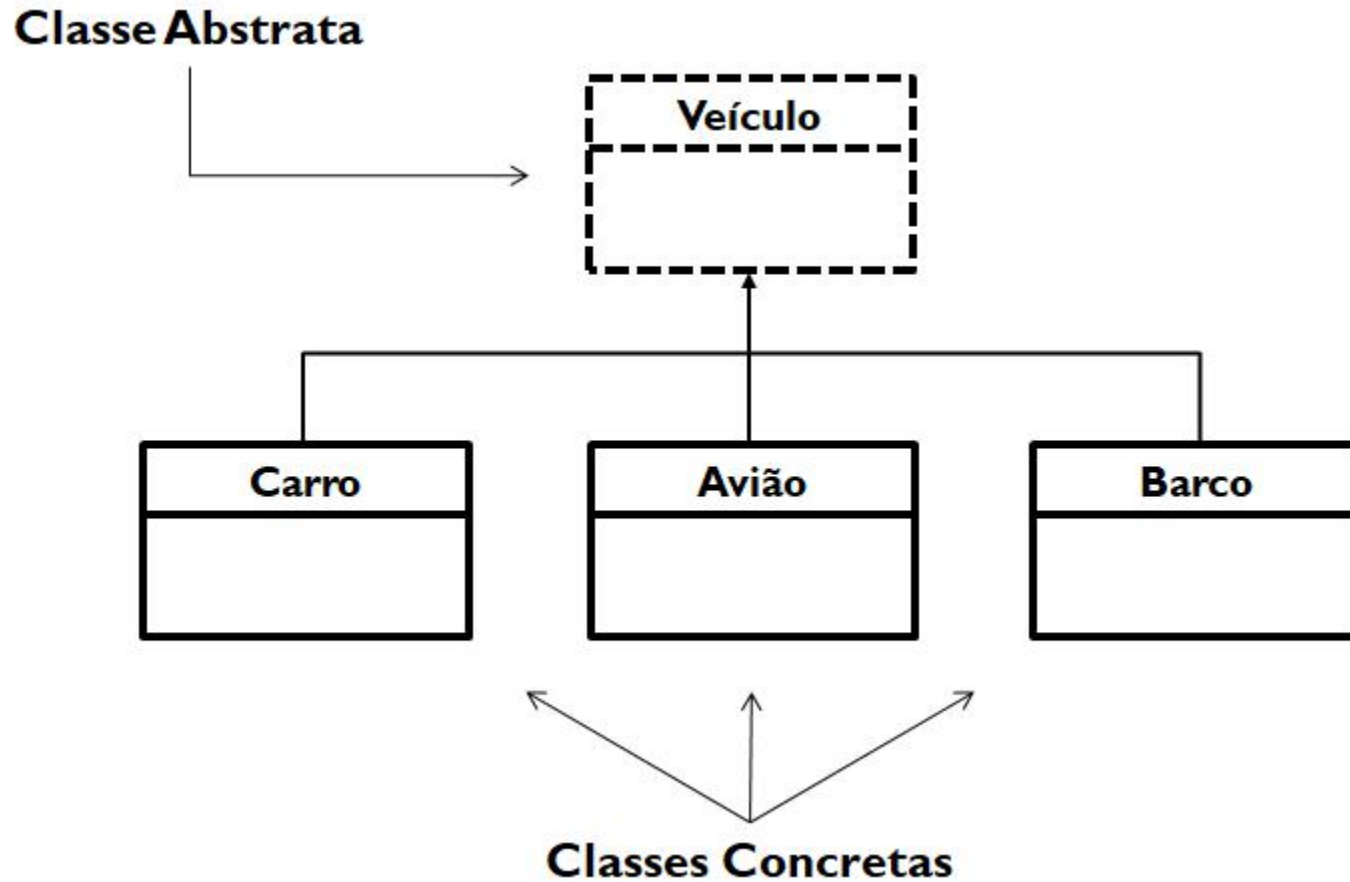
class Ponto2D : public Ponto {
    public:
        Ponto2D(int x, int y) {
            this->x = x;
            this->y = y;
        }

        void imprimirInfo() {
            cout << "Ponto 2D [" << x << ", " << y << "]" << endl;
        }
};
```

Classes abstratas

```
int main() {  
  
    //Ponto p;  
    //Erro  
  
    Ponto2D p(10, 20);  
    p.imprimirInfo();  
  
    return 0;  
}
```

Classes abstratas



Interfaces

- Possui unicamente o papel de um contrato
 - Garantia de que uma determinada Classe irá implementar um certo grupo de métodos
- C++ não implementa explicitamente
 - Outras linguagem sim, por exemplo, Java
 - Declarada da mesma forma que uma Classe, possui métodos que são puramente virtuais

Interfaces

```
class ClasseInterface
{
    public:
        //Contrato
        virtual void method() = 0;
};

class ClasseConcreta : public ClasseInterface {

    public:
        void method() {
            cout << "Comportamento definido" << endl;
        }
};
```