UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

# Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Classes)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

## Introdução

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

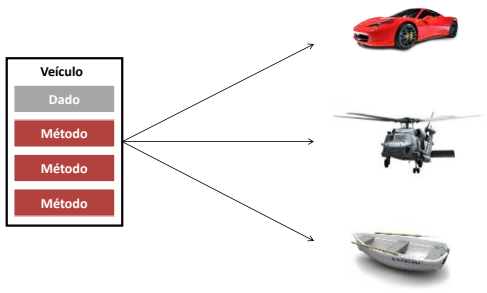


## Classes

- Representam uma categoria de elementos
  - "Não existem" no contexto da execução!
  - Objetos representam itens em particular
- Definem uma lógica estática
  - Relacionamentos entre classes não mudam
  - Relacionamentos entre objetos são dinâmicos



## Classes
### Abstração






## Classes

- Suportam os conceitos de
  - Encapsulamento
  - Herança
  - Polimorfismo
- Structs (C++)
  - Possuem comportamento semelhante
  - Utilizar apenas para guardar dados (atributos)
  - Caso contrário, usar classes (métodos)



## Classes
### Convenções

```
class ClasseTeste {

  public:
    int minhaVariavel;

};
```

```
class ClasseTeste {

  public:
    int minha_variavel;

};
```

- Classes
  - Primeira letra maiúscula (UpperCamelCase)
- Atributos, variáveis locais e métodos
  - Primeira letra minúscula (lowerCamelCase)
  - Separados por "_" (estilo da STL)

## Classes

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



## Classes
### Membros

- Membros
  - 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!



## Classes
### Membros

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



## Classes
### Componentes

- 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
  - Criação/retorno da referência do novo Objeto



## Classes

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

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

  delete p2;                   ← LIBERAR
  return 0;
}
```

Instanciação (criação) de dois Objetos do tipo Ponto.

## Classes

```
Casa c1;
Casa c2;
Casa c3;
```



## 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 ⇒ Lixo (segmentation fault)
- Estado do Objeto definido pelos atributos



## 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 = 321;

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

  return 0;
}
```

Acessando atributos

https://wandbox.org/permlink/31vb5CaszIovQ985



## Classes
### Atributos de instância

```
Casa c1;
c1.numero = 77;
c1.cor = "verde";
Casa c2;
c2.numero = 55;
c2.cor = "vermelho";
Casa c3;
c3.numero = 11;
c3.cor = "amarelo";
```



## Classes
### Métodos

- Procedimentos que podem modificar ou apenas acessar os valores dos atributos
  - Instância
  - Classe (estáticos)
- Controle de visibilidade
  - Determinar os métodos disponíveis para acesso
  - Permite definir o contrato da classe
  - Relacionado ao encapsulamento da classe



## Classes
### Métodos de instância

- 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
  - Facilitar a compreensão do código!

## Classes
### Métodos sobrecarregados

- Sobrecarga (overloading)
  - Dois ou mais métodos com mesmo nome
    - Polimorfismo
  - Lista de parâmetros (tipos) deve ser diferente!
    - A ordem dos tipos dos parâmetros é importante
- Não são diferenciáveis pelo tipo de retorno
  - Podem possuir diferentes tipos de retorno desde que possuam diferentes parâmetros
  - C++: 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;
}
```

https://wandbox.org/permlink/JDbMWX7NBJQhqGBZ



## 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(50, 50);
  Ponto* p3 = new Ponto(50);

  return 0;

}
```

https://wandbox.org/permlink/vTI9FIEqqjmxFL31



## Classes
### Construtores

```
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;
    }
};
```

- Qual o problema?
  - Código duplicado!
- Como seria possível melhorar o código?



## 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;
    }
};
```

Delegating constructors: C++11: https://en.cppreference.com/w/cpp/language/initializer_list

# Classes
## Construtores

```
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) {}
};
```

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



# Classes
## Destrutores

- Método chamado para a finalização
  - Libera os recursos alocados na execução
  - Quando o *lifetime* de um objeto chega ao fim
    - Heap → Após um delete
    - Stack → Após o término da função
- Devem possuir o mesmo nome da Classe
  - Semelhante aos construtores
  - Devem ser precedidos por '~'



# Classes
## Destrutores

```
class TestObject {

  int atributo;

  public:
    TestObject(int valor) : atributo(valor) {}

    ~TestObject() {
       cout << "~TestObject" << atributo << endl;
    }
};
```

```
int main() {

  TestObject o1(1);
  TestObject* o2 = new TestObject(2);

  delete (o2);
  return 0;

}
```

https://wandbox.org/permlink/ORCoSqATEwVqQrFT

**Qual será a saída?**

~TestObject2
~TestObject1



# Classes
## Destrutores

```
int main() {

  cout << "Antes" << endl;

  int i = 0;
  while(i<5) {
    TestObject o(i);
    i++;
  }

  cout << "Depois" << endl;

  return 0;

}
```

https://wandbox.org/permlink/FdoOCWIRzztzSQ02

**Qual será a saída?**

Antes
~TestObject0
~TestObject1
~TestObject2
~TestObject3
~TestObject4
Depois



# Classes
## Atributos estáticos

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



# Classes
## Atributos estáticos

```
class ClasseAtributoEstatico {

  public:

    static int numero;

    ClasseAtributoEstatico() {
      ClasseAtributoEstatico::numero++;
    }

    void imprimirNumero () {
      cout << ClasseAtributoEstatico::numero << endl;
    }
};

int ClasseAtributoEstatico::numero = 0;
```

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

  return 0;
}
```

https://wandbox.org/permlink/r2JciAZvGv2Ue2L1



## 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 <u>compilação</u>
  - Não dinamicamente como no caso de métodos de instância que são resolvidos baseados no tipo do objeto em tempo de execução
- Não podem ser sobrescritos (herança)
- Dentro de métodos *static* só podem ser acessados atributos e métodos *static*!
  - 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;
}
```

https://wandbox.org/permlink/FaqH01bie7oZCdPk



## Classes
### Modularização

**ClasseTeste.hpp**

```
#ifndef CLASSETESTE_H
#define CLASSETESTE_H

#include <iostream>
#include <string>

using namespace std;

class ClasseTeste {

  public:
    int    _atributo1;
    string _atributo2;

    ClasseTeste();
    ClasseTeste(double, string);

    void metodoA();
    void metodoB(double);

};

#endif
```

Não é necessário colocar o nome do parâmetro, mas pode acabar causando confusão!



## Classes
### Modularização

**ClasseTeste.cpp**

```
#include "ClasseTeste.hpp"

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;
}
```

https://wandbox.org/permlink/NKcsGfhsAtnJp6Yk

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



# Exercício

- Como reimplementar o TAD Circunferência agora utilizando esses novos conceitos?
- Quais as possíveis classes envolvidas?
  - Ponto, Circunferencia
- Quais atributos e métodos?
  - Ponto → x, y
  - Circunferencia → pontoCentro, raio, calcularArea
- Por onde começar?
  - Definição dos contratos → .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 M_PI
#define M_PI  3.14159265358979323846
#endif

#ifndef CIRCUNFERENCIA_H
#define CIRCUNFERENCIA_H

#include <cmath>

#include "Ponto.hpp"

class Circunferencia {

  public:
    Ponto* _centro;
    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(Ponto* centro, double raio) {
  this->_centro = centro;
  this->_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(new Ponto(), 10);
  cout << c1->calcularArea() << endl;

  Ponto p(5.0, 5.0);
  Circunferencia* c2 = new Circunferencia(&p, 10);
  cout << c2->calcularArea() << endl;

  delete c1;
  delete c2;

  return 0;
}
```



# Exercício

```
==125== Memcheck, a memory error detector
==125== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==125== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==125== Command: ./main
==125==
==125== error calling PR_SET_PTRACER, vgdb might block
314.159
314.159
==125==
==125== HEAP SUMMARY:
==125==     in use at exit: 16 bytes in 1 blocks
==125==   total heap usage: 5 allocs, 4 frees, 73,264 bytes allocated
==125==
==125== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==125==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==125==    by 0x108BCF: main (main.cpp:10)
==125==
==125== LEAK SUMMARY:
==125==    definitely lost: 16 bytes in 1 blocks
==125==    indirectly lost: 0 bytes in 0 blocks
==125==      possibly lost: 0 bytes in 0 blocks
==125==    still reachable: 0 bytes in 0 blocks
==125==         suppressed: 0 bytes in 0 blocks
==125==
==125== For counts of detected and suppressed errors, rerun with: -v
==125== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

# Exercício

**Circunferencia.hpp**

```
#ifndef M_PI
#define M_PI  3.14159265358979323846
#endif

#ifndef CIRCUNFERENCIA_H
#define CIRCUNFERENCIA_H

#include <cmath>

#include "Ponto.hpp"

class Circunferencia {

  public:
    Ponto* _centro;
    double _raio;

    Circunferencia(Ponto* centro, double raio);
    ~Circunferencia();

    double calcularArea();

};

#endif
```

**Circunferencia.cpp**

```
#include "Circunferencia.hpp"

Circunferencia::Circunferencia(Ponto* centro, double raio) {
  this->_centro = centro;
  this->_raio = raio;
}

Circunferencia::~Circunferencia() {
  delete this->_centro;
}

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



# Exercício

```
314.159
314.159
double free or corruption (out)
Aborted (core dumped)
```

```
==223== Memcheck, a memory error detector
==223== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==223== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==223== Command: ./main
==223==
==223== error calling PR_SET_PTRACER, vgdb might block
314.159
314.159
==223== Invalid free() / delete / delete[] / realloc()
==223==    at 0x4C3123B: operator delete(void*) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==223==    by 0x108B32: Circunferencia::~Circunferencia() (Circunferencia.cpp:9)
==223==    by 0x108D03: main (main.cpp:19)
==223==  Address 0x1fff000290 is on thread 1's stack
==223==  in frame #2, created by main (main.cpp:8)
==223==
==223==
==223== HEAP SUMMARY:
==223==     in use at exit: 0 bytes in 0 blocks
==223==   total heap usage: 5 allocs, 6 frees, 73,264 bytes allocated
==223==
==223== All heap blocks were freed -- no leaks are possible
==223==
==223== For counts of detected and suppressed errors, rerun with: -v
==223== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```



# Exercício

**main.cpp**

```
#include <iostream>

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

using namespace std;

int main() {

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

  Ponto* p = new Ponto(5.0, 5.0);
  Circunferencia* c2 = new Circunferencia(p, 10);
  cout << c2->calcularArea() << endl;

  delete c1;
  delete c2;

  return 0;
}
```

Você pode testar outras soluções!

https://wandbox.org/permlink/9FO4Pwwf9NOyertm