UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Fundamentos de Programação

ET43I

Orientada a Objetos

Classes e instâncias. Construtores, destrutores e operador de atribuição de cópia. Objetos, atributos, operações: mensagens e métodos, estados. Encapsulamento, ocultamento. Lista de exercícios.

Aula 5
Prof. Daniel Cavalcanti Jeronymo

Universidade Tecnológica Federal do Paraná (UTFPR)
Engenharia Eletrônica – 3º Período
2016.1

Plano de Aula

- Classes
- Encapsulamento
- Construtores e destrutores
- Idiomas do C++

Classes

- Programação Orientada a Objetos (POO)
  - Objetos possuem:
    - Dados (**atributos**)
    - Comportamento (**métodos**)
  - Classes definem o **modelo** de um objeto

Classes

- Classes
  - São tipos definidos pelo usuário
    - Palavra reservada class
    - Corpo delimitado por chaves
    - Declaração terminada com ponto-e-vírgula

```
class nomeDaClasse
{
        ...
        ...
        ...
};
```

Classes

- *Class* vs *Struct*
  - Ambos são tipos definidos pelo usuário e são estruturas de dados
  - **Encapsulamento padrão:**
    - Classes – **private**
    - Structs – **public**

```
class nomeDaClasse
{
        ...
        ...
        ...
};
```

```
struct nomeDaStruct
{
        ...
        ...
        ...
};
```

Classes

- Classes
  - Modificadores de acesso especificam o nível de acesso (ou ocultamento) aos membros da classe

```
class nomeDaClasse
{
        public:
                int membroPublico;
        private:
                int membroPrivado;
        protected:
                int membroProtegido;
};
```

Lembrete de Prog. 1:
Modificadores de acesso são uma evolução da linguagem para permitir a construção de **tipos/ponteiros opacos** (e.g., FILE* em C)

Classes

- Classes
  - **Public**
    - Membros podem ser acessados externamente e internamente
  - **Private**
    - Membros podem ser acessados internamente **na classe**
  - **Protected**
    - Membros podem ser acessados internamente **na classe** e por **classes derivadas** (heranças)

Ao contrário de no objeto!

Classes

- Classes – Exemplo

```
#include <iostream>
using namespace std;
class minhaClasse
{
public:
   int membroPublico;
   int acessaPrivado() { return membroPrivado; }
   void escrevePrivado(int valor) { membroPrivado = valor; }
private:
   int membroPrivado;
};

int main()
{
   minhaClasse teste;
   teste.membroPublico = 10; // OK!
   cout << teste.membroPublico << endl; // OK!
   //teste.membroPrivado = 100; // ERRO!
   teste.escrevePrivado(100); // OK!
   cout << teste.acessaPrivado() << endl; // OK!

   return 0;
}
```

Classes

- Classes
  - Métodos podem ser definidos dentro da classe ou fora

```
class nomeDaClasse
{
   tipoRetorno nomeDaFuncao() { … }
};
```

```
class nomeDaClasse
{
 tipoRetorno nomeDaFuncao();
};

tipoRetorno nomeDaClasse::nomeDaFuncao()
{
   …
}
```

Lembrete de Prog. 1:
Declaração: introduz um identificador e seu tipo (necessidade do compilador)
Definição: implementa um identificador (necessidade do *linker*)

Classes

- Classes – Exemplo (definições pelo operador de escopo)

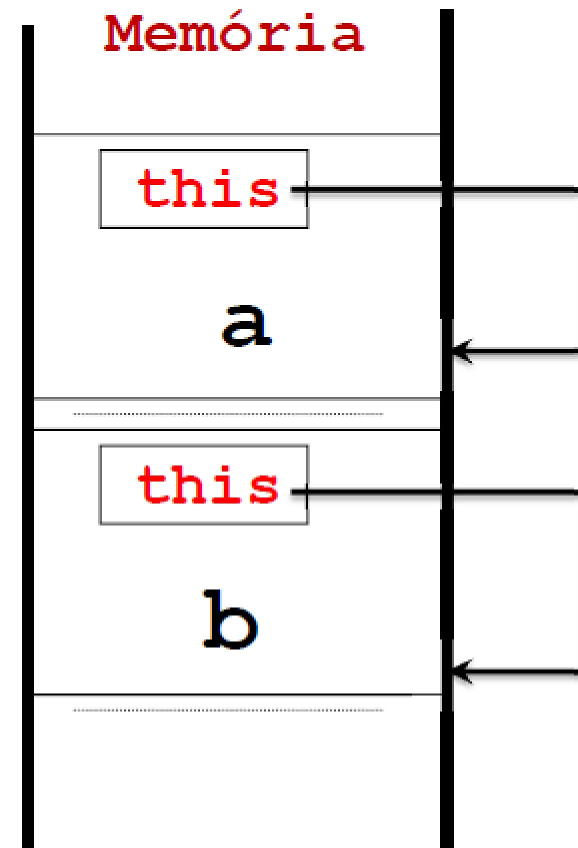
```
class minhaClasse
{
public:
   int membroPublico;

   int acessaPrivado();
   void escrevePrivado(int valor);
private:
   int membroPrivado;
};

int minhaClasse::acessaPrivado()
{
   return membroPrivado;
}

void minhaClasse::escrevePrivado(int valor)
{
   membroPrivado = valor;
}
```

Classes

- Classes – Ponteiro **this**
  - A instância é uma cópia única de uma classe e representa um objeto
  - Cada instância tem seu próprio endereço
  - Ponteiro **this**

Classes

- Classes – Ponteiro **this**
  - **This** é um ponteiro interno para o próprio objeto



  - O ponteiro é passado como um **argumento oculto** para todas as funções não estáticas de uma classe
  - Disponível como uma variável local dentro do corpo da função:

```
minhaClasse * const this;
```

Classes

- Classes – Exemplo ponteiro **this**

```
#include <iostream>
using namespace std;

class minhaClasse
{
private:
   int algo;
public:
   void mudaAlgo(int algo) { this->algo = algo; }
   minhaClasse * const meuponteiro() { return this; }
};

int main()
{
   minhaClasse mc;

   cout << "Endereco do objeto: " <<  &mc << endl;
   cout << "Endereco do ponteiro this: " << mc.meuponteiro() << endl;
   cout << "Tamanho do objeto: " << sizeof(mc) << endl;

   return 0;
}
```

Classes

- Classes – Exemplo ponteiro **this**
  - A chamada:

```
mc.mudaAlgo(10);
```

  - Na verdade é:

```
mudaAlgo(&mc, 10);
```

Classes

- Classes
  - Objetos são **criados** e **destruídos**

pilha

```
{
   ...
   minhaClasse obj;
   ...
}
```

heap

```
minhaClasse *obj = new minhaClasse;
...
delete obj;
```

Classes

- Classes
  - **Construtores** são funções que são chamadas na criação de um objeto
  - **Destrutores** são funções que são chamadas na destruição de um objeto

Classes

- Classes
  - **Construtores**
  - Nome da função é igual ao nome da classe, a função recebe parâmetros e não tem valor de retorno
  - Pode ser sobrecarregado
  - Usando a definição ao lado, é possível criar um objeto minhaClasse **sem parâmetros**?

```
class minhaClasse
{
public:
   int algo;
   minhaClasse(int valor)
   {
      algo = valor;
   }
};
```

Classes

- Classes
  - **Destrutores**
  - Nome da função é igual ao nome da classe com um til no início (~), a função não recebe parâmetros e não tem valor de retorno
  - Não pode ser sobrecarregado
  - Uso típico: **liberar recursos** (.e.g, desalocar memória de uso interno ao objeto)

```
class minhaClasse
{
public:
   int algo;
   minhaClasse(int valor)
   {
      algo = valor;
      cout << valor << endl;
   }
   ~minhaClasse()
   {
      cout << algo << endl;
   }
};
```

Classes

- Classes
  - O compilador cria implicitamente construtor e destrutor padrões caso estes sejam omitidos
  - Os padrões implícitos são equivalentes aos explícitos com corpo vazio

Classes

- Classes
  - Construtor de cópia
  - Construtor especial invocado ao criar um objeto a partir de outro do mesmo tipo
  - Caso omitido, é criado automaticamente pelo compilador e realiza uma **cópia membro a membro (*memberwise*)** do objeto

```
minhaClasse m1(mc);

minhaClasse m2 = mc;
```

Classes

- Classes
  - Construtor de cópia
  - *bitwise* vs *memberwise*
  - *shallow copy* vs *deep copy*

ISO/IEC 14882:1998 Programming languages — C++

The implicitly defined copy constructor for class X performs a memberwise copy of its subobjects. The order of copying is the same as the order of initialization of bases and members in a userdefined constructor (see 12.6.2).
Each subobject is copied in the manner appropriate to its type:
— if the subobject is of class type, the copy constructor for the class is used;
— if the subobject is an array, each element is copied, in the manner appropriate to the element type;
— if the subobject is of scalar type, the builtin assignment operator is used.

Classes

- Classes
  - Construtor de cópia
  - Chamado em 3 situações:
    - Quando fazemos uma cópia de um objeto
    - Quando passamos um objeto por valor para uma função
    - Quando retornamos um objeto por valor de uma função

Classes

- Classes
  - Sobrecarregando o construtor de cópia

```
class minhaClasse
{
public:
   int algo;
   minhaClasse(int valor)
   {
      algo = valor;
      cout << valor << endl;
   }
   minhaClasse(const minhaClasse & outro)
   {
      this->algo = outro.algo;
      cout << algo << endl;
   }
   ~minhaClasse()
   {
      cout << algo << endl;
   }
};
```

```
int main()
{
   minhaClasse mc(20);

   minhaClasse m1(mc);

    minhaClasse m2 = mc;

   cout << "fim" << endl;

   return 0;
}
```

Classes

- Classes
  - Construtor de cópia vs operador de cópia

```
minhaClasse first;                // inicialização pelo construtor padrão
minhaClasse second(first);   // inicialização pelo construtor de cópia
minhaClasse third = first;     // inicialização pelo construtor de cópia
second = third;                     // atribuição pelo operador de atribuição de cópia
```

OBS: pelo código apresentado para a classe minhaClasse o trecho acima contém um erro... como arrumar?

Classes

- Classes
  - Construtor de cópia
  - Utilizado quando um objeto **não existe** e é realizada alguma atribuição ou chamado explicitamente
  - Operador de cópia
  - Utilizado quando um objeto **já existe** e é realizada uma atribuição

```
class minhaClasse
{
public:
   int algo;
   minhaClasse(int valor)
   {
      algo = valor;
      cout << valor << endl;
   }
   minhaClasse(const minhaClasse & outro)
   {
      this->algo = outro.algo;
      cout << algo << endl;
   }
   minhaClasse & operator= (const minhaClasse & outro)
   {
      this->algo = outro.algo;
      cout << "copia!" << endl;
      return *this;
   }
   ~minhaClasse()
   {
      cout << algo << endl;
   }
};
```

Classes

- Classes – Exemplo “Ponto”

```
#include <iostream>
#include <cmath>
using namespace std;
class ponto
{
private:
   int x, y;
public:
   ponto(int x, int y)
   {
      this->x = x;
      this->y = y;
   }

   double distancia(const ponto & p2)
   {
      return sqrt((this->x - p2.x)*(this->x - p2.x) + (this->y - p2.y)*(this->y - p2.y));
   }
};
int main()
{
   ponto p1(1, 1), p2(0, 0);
   cout << p1.distancia(p2) << endl;
   return 0;
}
```

Classes

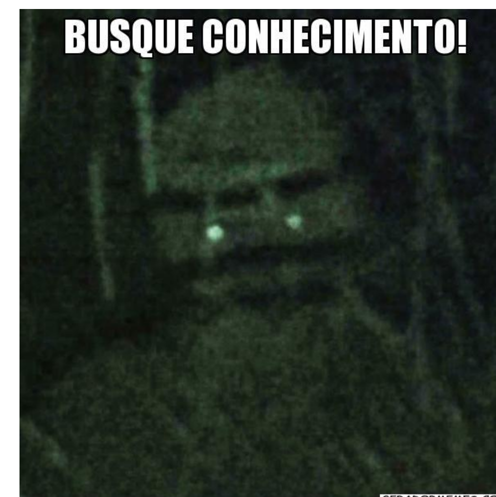
- Classes – Exemplo “Ponto”

- Uso de ocultamento
- Uso de construtor
- Uso do ponteiro this
- Uso de *const correctness*
- Uso de referência
- Acesso de membro privado em outro objeto

```
#include <iostream>
#include <cmath>
using namespace std;
class ponto
{
private:
   int x, y;
public:
   ponto(int x, int y)
   {
      this->x = x;
      this->y = y;
   }

   double distancia(const ponto & p2)
   {
      return sqrt((this->x - p2.x)*(this->x - p2.x) + (this->y - p2.y)*(this->y - p2.y));
   }
};
int main()
{
   ponto p1(1, 1), p2(0, 0);
   cout << p1.distancia(p2) << endl;
   return 0;
}
```

Classes

- Idiomas relevantes do C++ para esta aula
  - Regra dos três (*rule of three* – futuramente será *rule of five*)
  - Conte, não fale (*Tell, don't ask*)



Procedural code gets information then makes decisions.
Object-oriented code tells objects to do things.
— Alec Sharp

Classes

- Idiomas relevantes do C++ para esta aula
  - Getters / Setters – Leitura:

    http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html

Getters and setters
lead to the dark side...

  - RAII (Resource Acquisition Is initialization)