Conceitos de C++

INF1761 – Computação Gráfica

Waldemar Celes
celes@inf.puc-rio.br

Departamento de Informática, PUC-Rio

Classe

Classe representa um tipo que tem dados e métodos

- Dados são, em geral, *privados* (sem acesso externo direto)
- Métodos podem ser *privados*, *protegidos* ou *públicos*
  - Privados: acessados apenas por objetos da classe
  - Protegidos: acessados também por objetos de classe derivada
  - Públicos: acessados por códigos externos

Classe

Exemplo: classe `Circle`

- Arquivo "circle.h"

```
#ifndef CIRCLE_H  // to avoid multiple inclusions
#define CIRCLE_H

class Circle {
  float m_radius; // agregate data (advice: use prefix m_)
public:
  // methods
  Circle (float radius=1.0f);      // constructor
  ~Circle ();                      // destructor
  void SetRadius (float radius);   // modifiers
  float GetRadius ();              // accessors
  float ComputeArea();             // evaluators
};
#endif
```

Classe

Exemplo da classe `Circle`: implementação dos métodos

- Arquivo "circle.cpp"

```
#include "circle.h"

#define PI 3.14159265359f

Circle::Circle (float radius)
: m_radius(radius)     // preferred initialization
{
}
Circle::~Circle () {  // free resource, if any
}
void Circle::SetRadius (float radius) {
  m_radius = radius;
}
float Circle::GetRadius () {
  return m_radius;
}
float Circle::ComputeArea () {
  return PI * m_radius * m_radius;
}
```

Classe

Exemplo de cliente da classe

- Arquivo “main.cpp”

```
#include "circle.h"
#include <iostream>  //  builtin i/o ops (note: no .h)
int main ()
{
   Circle c(2.0f);                    // concrete object
   Circle* p = new Circle(3.0f);      // dynamic object
   std::cout << "Area: " << c.ComputeArea() << std::endl;
   std::cout << "Area: " << p->ComputeArea() << std::endl;
   p->SetRadius(1.0f);
   std::cout << "Area: " << p->ComputeArea() << std::endl;
   delete p;                          // free object
   return 0;
}
```

- Saída da execução

```
Area: 12.5664
Area: 28.2743
Area: 3.14159
```

Classe

Observando objetos deletados

- Arquivo “circle.cpp”

```
#include <iostream>
...
Circle::~Circle () {  // free resource, if any
  std::cout << "circle deleted" << std::endl;
}
```

Classe

Observando objetos deletados

- Arquivo “circle.cpp”

```
#include <iostream>
...
Circle::~Circle () {  // free resource, if any
  std::cout << "circle deleted" << std::endl;
}
```

- Saída da execução

```
Area: 12.5664
Area: 28.2743
Area: 3.14159
circle deleted
circle deleted
```

  - objeto dinâmico é deletado via `delete`
  - objeto concreto é deletado ao fim do seu escopo

Classe

Métodos `inline`

- Métodos podem ser declarados no .h
  - Indicado para métodos pequenos
  - Execução mais rápida: não tem o preço da chamada função
    - Método expandido *inline*
  - Compilação mais lenta e código gerado maior
  - Se todos os métodos forem `inline`, não há ".cpp"

Classe

Classe `Circle` com métodos `inline`

```
#ifndef CIRCLE_H
#define CIRCLE_H
class Circle {
  float m_radius;
public:
  Circle (float radius=1.0f) {
  }
  ~Circle () {
  }
  void SetRadius (float radius) {
    m_radius = radius;
  }
  float GetRadius () {
    return m_radius;
  }
  float ComputeArea () {
    const float PI = 3.14159265359f;
    return PI * m_radius * m_radius;
  }
};
#endif
```

Classe

Métodos `const`

- Boa prática: métodos que não modificam o objeto
  - Erro de compilação se objeto (seus dados) for modificado
  - Cliente ciente que objeto não será alterado
  - Candidatos: métodos de acesso e avaliação/computação

Classe

Métodos `const`

- Boa prática: métodos que não modificam o objeto
  - Erro de compilação se objeto (seus dados) for modificado
  - Cliente ciente que objeto não será alterado
  - Candidatos: métodos de acesso e avaliação/computação
- Arquivo "circle.h"

```
...
float GetRadius () const;          // accessors
float ComputeArea() const;         // evaluators
...
```

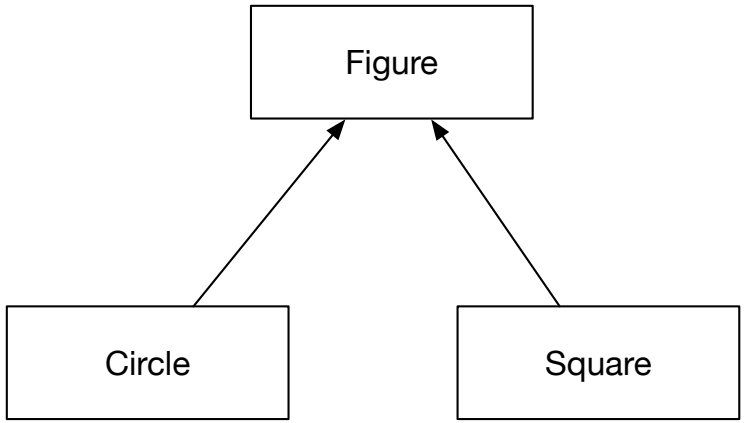
- Arquivo "circle.cpp"

```
float Circle::GetRadius () const {
  return m_radius;
}
float Circle::ComputeArea () const {
  return PI * m_radius * m_radius;
}
```

Herança e polimorfismo

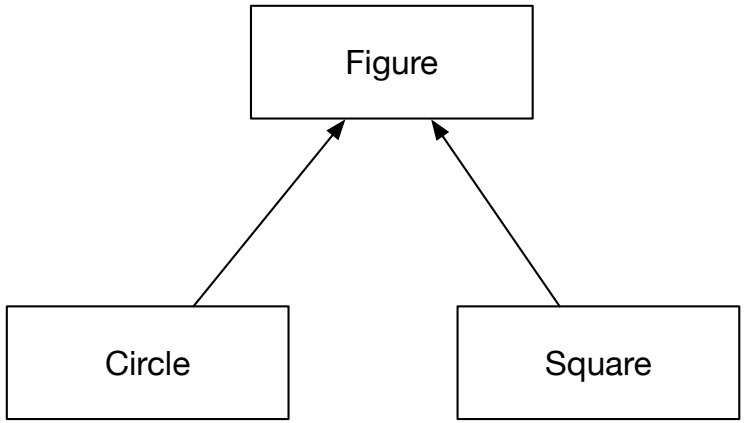
Exemplo: classes de figuras

- `Circle` e `Square` herdam de `Figure`



Herança e polimorfismo

Exemplo: classes de figuras



- `Circle` e `Square` herdam de `Figure`
- Arquivo "figure.h"

```
#ifndef FIGURE_H
#define FIGURE_H

class Figure {
public:
  Figure ();
  virtual ~Figure();
  virtual float ComputeArea () const = 0;
};
#endif
```

- Métodos `virtual`: classes *abstratas*
  - Podem ser redefinidos por classes derivadas
    - Se `= 0` **devem** ser definidos
    - E classe com `virtual = 0` não pode ser instanciada
  - Classes abstratas devem ter destrutor virtual

Herança e polimorfismo

Classse derivadas

- Arquivo “circle.h”

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "figure.h"

class Circle : public Figure
{
  float m_radius;
public:
  Circle (float radius);
  virtual ~Circle ();
  float ComputeArea () const;
};

#endif
```

- Arquivo “square.h”

```
#ifndef SQUARE_H
#define SQUARE_H

#include "figure.h"

class Square : public Figure
{
  float m_side;
public:
  Square (float side);
  virtual ~Square ();
  float ComputeArea () const;
};

#endif
```

Herança e polimorfismo

Cliente: “main.cpp”

```
#include "square.h"
#include "circle.h"

#include <iostream>

int main ()
{
  Figure* s = new Square(2.0f);
  Figure* c = new Circle(1.0f);
  float a_s = s->ComputeArea();
  float a_c = c->ComputeArea();
  std::cout << "square area: " << a_s << std::endl;
  std::cout << "circle area: " << a_c << std::endl;
  delete c;
  delete s;
  return 0;
}
```

Herança e polimorfismo

Cliente: "main.cpp"

```
#include "square.h"
#include "circle.h"

#include <iostream>

int main ()
{
   Figure* s = new Square(2.0f);
   Figure* c = new Circle(1.0f);
   float a_s = s->ComputeArea();
   float a_c = c->ComputeArea();
   std::cout << "square area: " << a_s << std::endl;
   std::cout << "circle area: " << a_c << std::endl;
   delete c;
   delete s;
   return 0;
}
```

- Saída:

```
square area: 4
circle area: 3.14159
```

Classe `std::string`

Implementa cadeia de caracteres dinâmica

- Cadeia de caracteres tratada como valor
- Implementada por um tipo concreto

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

int main ()
{
  std::string s1 = "PUC";
  std::string s2 = "Rio";
  std::string s3 = s1 + "-" + s2;
  std::cout << s3 << std::endl;
  return 0;
}
```

Classe `std::string`

Implementa cadeia de caracteres dinâmica

- Cadeia de caracteres tratada como valor
- Implementada por um tipo concreto

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

int main ()
{
  std::string s1 = "PUC";
  std::string s2 = "Rio";
  std::string s3 = s1 + "-" + s2;
  std::cout << s3 << std::endl;
  return 0;
}
```

- Saída:

```
PUC-Rio
```

  - Note que C++ permite sobrecarga de operadores

Ponteiro & Referência

Ponteiro

- Armazena o endereço de memória de uma variável
- Acesso indireto ao conteúdo

```
int a = 3;
int* p = &a;
*p = 5;
```

Ponteiro & Referência

Ponteiro

- Armazena o endereço de memória de uma variável
- Acesso indireto ao conteúdo

```
int a = 3;
int* p = &a;
*p = 5;
```

Referência

- Representa um sinônimo de uma variável
  - Implementada internamente via endereço de memória
- Acesso direto (como se tivéssemos acesso à própria variável)

```
int a = 3;
int& r = a;
r = 5;
```

Ponteiro & Referência

Onde usar ponteiro e/ou referência

- Ponteiro usado para tipos abstratos (dinâmicos)
- Referência usada para tipos concretos
  - Exemplos de interface de métodos:

```
void SetName (const std::string& s);

const std::string& GetName () const;
```

Herança e polimorfismo

Associando "cores" a figuras no exemplo

- Classe `Color`: Tem associado um nome da cor

```
#ifndef COLOR_H
#define COLOR_H
#include <string>
class Color
{
  std::string m_name;
public:
  Color (const std::string& name);
  ~Color ();
  const std::string& GetName ();
};
```

Herança e polimorfismo

Associando "cores" a figuras no exemplo

- Classe `Color`: Tem associado um nome da cor

```
#ifndef COLOR_H
#define COLOR_H
#include <string>
class Color
{
  std::string m_name;
public:
  Color (const std::string& name);
  ~Color ();
  const std::string& GetName ();
};
```

```
#include "color.h"
Color::Color (const std::string& name) : m_name(name) {
}
Color::~Color () {
}
const std::string& Color::GetName () const {
  return m_name;
}
```

Herança e polimorfismo

Herança ou agregação

- Herança: classe **é** uma outra classe
- Agregação: class **tem** uma outra classe

Herança e polimorfismo

Herança ou agregação

- Herança: classe **é** uma outra classe
- Agregação: class **tem** uma outra classe

Exemplo: relação entre `Figure` e `Color`

- Uma figura **é** uma cor ou uma figura **tem** uma cor?

Herança e polimorfismo

Herança ou agregação

- Herança: classe **é** uma outra classe
- Agregação: class **tem** uma outra classe

Exemplo: relação entre `Figure` e `Color`

- Uma figura **é** uma cor ou uma figura **tem** uma cor?
  - Figura **tem** uma cor: *agregação*

```
#ifndef FIGURE_H
#define FIGURE_H
class Color;
class Figure {
  Color* m_color;
public:
  Figure (Color* color);
  virtual ~Figure();
  Color* GetColor () const;
  virtual float ComputeArea () const = 0;
};
#endif
```

Herança e polimorfismo

Construtor de classes derivadas devem chamar construtor de classe base

- `Circle`

```
Circle (float radius, Color* color);
```

```
Circle::Circle (float radius, Color* color)
: Figure(color), m_radius(radius)
{
}
```

- `Square`

```
Square (float side, Color* color);
```

```
Square::Square (float side, Color* color)
: Figure(color), m_side(side)
{
}
```

Cliente

```
#include "square.h"
#include "circle.h"
#include <iostream>
int main ()
{
  Color* r = new Color("red");
  Figure* s = new Square(2.0f,r);
  Figure* c = new Circle(1.0f,r);
  float a_s = s->ComputeArea();
  float a_c = c->ComputeArea();
  std::cout << "square area (" << s->GetColor()->GetName() <
            << a_s << std::endl;
  std::cout << "circle area (" << s->GetColor()->GetName() <
            << a_c << std::endl;
  delete c;
  delete s;
  return 0;
}
```

- Saída

```
square area (red): 4
circle area (red): 3.14159
```