Programação Orientado a Objetos (POO)

Programação Avançada
Universidade Federal de Pernambuco
Adrien Durand-Petiteville
adrien.durandpetiteville@ufpe.br

Classes

- As classes são um conceito expandido de estruturas de dados:
  - Como estruturas de dados, elas podem conter membros de dados
  - Também podem conter funções como membros
- Um objeto é uma instanciação de uma classe.
- Em termos de variáveis, uma classe seria o tipo e um objeto seria a variável

Classes - 2

- As classes são definidas usando a palavra-chave `class`

```
class class_name {
  access_specifier_1:
  member1;
  access_specifier_2:
  member2;
  ...
} object_names;
```

- `class_name` é um identificador válido para a classe
- O corpo da declaração pode conter membros (`member`), que podem ser declarações de dados ou funções e, opcionalmente, especificadores de acesso (`access_specifier`)
- `object_names` é uma lista opcional de nomes para objetos desta classe

Classes - 3

- As classes têm o mesmo formato que as estruturas de dados simples, exceto que também **podem incluir funções** e ter essas coisas novas chamadas **especificadores de acesso**
- Um especificador de acesso é uma das três seguintes palavras-chave: `private`, `public` ou `protected`
- Esses especificadores modificam os direitos de acesso dos membros que os seguem:
  - Membros privados de uma classe são acessíveis apenas de dentro de outros membros da mesma classe (ou de seus "amigos")
  - Membros protegidos são acessíveis de outros membros da mesma classe (ou de seus "amigos"), mas também de membros de suas classes derivadas
  - Finalmente, os membros públicos são acessíveis de qualquer lugar onde o objeto esteja visível

Classes - 4

- Por padrão, todos os membros de uma classe declarada com a palavra-chave `class` têm acesso privado para todos os seus membros
- Portanto, qualquer membro declarado antes de qualquer outro especificador de acesso tem acesso privado automaticamente

```
class Rectangle {
  int width, height;
  public:
  void set_values (int,int);
  int area (void);
} rect;
```

- Declara uma classe (ou seja, um tipo) chamada Rectangle
- Esta classe contém quatro membros: dois membros de dados do tipo int (largura e altura do membro) com acesso privado e duas funções de membro com acesso público

- Após as declarações de `Rectangle` e `rect`, qualquer um dos membros públicos do objeto `rect` pode ser acessado como se fossem funções normais ou variáveis normais, bastando inserir um ponto (.) entre o nome do objeto e o nome do membro

```
rect.set_values (3,4);
myarea = rect.area();
```

- Os únicos membros de `rect` que não podem ser acessados de fora da classe são `width` e `height`, uma vez que eles têm acesso privado e só podem ser referidos por outros membros da mesma classe

Exemplo

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

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Definição de função

- O operador de escopo (::) é usado na definição da função `set_values` para definir um membro de uma classe fora da própria classe
- O operador de escopo (::) especifica que a função que está sendo definida é um membro da classe `Rectangle` e não uma função regular
- Ele especifica a classe à qual o membro que está sendo definido pertence, concedendo exatamente as mesmas propriedades de escopo como se essa definição de função fosse incluída diretamente na definição de classe

Exemplo

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

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area () {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Construtor

- Uma classe pode incluir uma função especial chamada construtor (`contructor`), que é automaticamente chamada sempre que um novo objeto desta classe é criado
- Um construtor permiti que a classe inicialize variáveis de membro ou aloque memória
- A função construtor é declarada como uma função de membro regular, mas com um nome que corresponde ao nome da classe e sem qualquer tipo de retorno; nem mesmo `void`
- Os construtores não podem ser chamados explicitamente como se fossem funções de membro regulares
- Eles são executados apenas uma vez, quando um novo objeto daquela classe é criado

Exemplo

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

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Sobrecarregando construtores

- Como qualquer outra função, um construtor também pode ser sobrecarregado com diferentes versões que usam parâmetros diferentes:
  - com um número diferente de parâmetros
  - e / ou parâmetros de tipos diferentes
- O compilador irá chamar automaticamente aquele cujos parâmetros correspondem aos argumentos

Exemplo

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

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}
```

Exemplo - 2

```
int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

- Este exemplo também apresenta um construtor de tipo especial: o construtor padrão
- O construtor padrão é o construtor que não aceita parâmetros e é especial porque é chamado quando um objeto é declarado, mas não é inicializado com nenhum argumento

Ponteiros para classes

- Os objetos também podem ser apontados por ponteiros
- Uma vez declarada, uma classe se torna um tipo válido, então ela pode ser usada como o tipo apontado por um ponteiro

```
Rectangle * prect;
```

- Da mesma forma que com estruturas de dados simples, os membros de um objeto podem ser acessados diretamente de um ponteiro usando o operador de seta $(->)$

Exemplo

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

class Rectangle {
  int width, height;
public:
  Rectangle(int x, int y) : width(x), height(y) {}
  int area(void) { return width * height; }
};
```

Exemplo - 2

```
int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  cout << "obj's area: " << obj.area() << '\n';
  cout << "*foo's area: " << foo->area() << '\n';
  cout << "*bar's area: " << bar->area() << '\n';
  cout << "baz[0]'s area:" << baz[0].area() << '\n';
  cout << "baz[1]'s area:" << baz[1].area() << '\n';
  delete bar;
  delete[] baz;
  return 0;
}
```

Sobrecarregando operadores

- As classes, essencialmente, definem novos tipos.
- Os tipos em C++ não interagem apenas com o código por meio de construções e atribuições
- Eles também interagem por meio de operadores
- C++ permite que a maioria dos operadores seja sobrecarregada para que seu comportamento possa ser definido para praticamente qualquer tipo, incluindo classes
- Os operadores são sobrecarregados por meio de funções de operador, que são funções regulares com nomes especiais: seu nome começa pela palavra-chave `operator` seguida pelo sinal do operador que está sobrecarregado

Exemplo

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

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}
```

Exemplo - 2

```
int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

This

- A palavra-chave `this` representa um ponteiro para o objeto cuja função de membro está sendo executada
- É usado na função de membro de uma classe para se referir ao próprio objeto

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

Template de classe

- Assim como podemos criar templates de função, também podemos criar templates de classe, permitindo que as classes tenham membros que usam parâmetros de modelo como tipos

```
template <class T>
class mypair {
  T values[2];
  public:
  mypair(T first, T second)
  { values[0]=first; values[1]=second; }
};
```

```
mypair<int> myobject (115, 36);
```

```
mypair<double> myfloats (3.0, 2.18);
```

Exemplo

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

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};
```

Exemplo - 2

```
template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

Especialização de template

- É possível definir uma implementação diferente para um modelo quando um tipo específico é passado como argumento do modelo
- Isso é chamado de especialização de modelo

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

// class template:
template <class T>
class mycontainer {
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};
```

Especialização de template - 2

```
// class template specialization:
template <>
class mycontainer <char> {
    char element;
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

Funções `friend`

- Em princípio, membros privados e protegidos de uma classe não podem ser acessados de fora da mesma classe em que foram declarados
- No entanto, esta regra não se aplica a "amigos"
- Amigos são funções ou classes declaradas com a palavra-chave `friend`
- Uma função não membro pode acessar os membros privados e protegidos de uma classe se for declarada amiga dessa classe
- Isso é feito incluindo uma declaração dessa função externa dentro da classe e precedendo-a com a palavra-chave `friend`

Exemplo

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

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};
```

Exemplo - 2

```
Rectangle duplicate (const Rectangle& param)
{
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}

int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

Classe `friend`

- Uma classe `friend` é uma classe cujos membros têm acesso aos membros privados ou protegidos de outra classe

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

class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};
```

Classe friend - 2

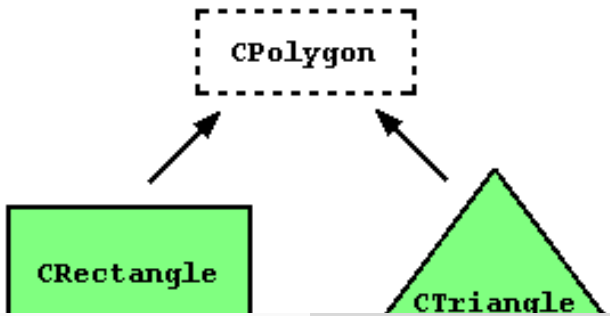
```
class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Herança

- As classes em C++ podem ser estendidas, criando novas classes que retêm características da classe base
- Esse processo, conhecido como **herança**, envolve uma **classe base** e uma **classe derivada**
- A classe derivada herda os membros da classe base, sobre a qual pode adicionar seus próprios membros

Exemplo

- Vamos imaginar uma série de classes para descrever dois tipos de polígonos: retângulos e triângulos
- Esses dois polígonos têm certas propriedades comuns, como os valores necessários para calcular suas áreas: ambos podem ser descritos simplesmente com uma altura e uma largura (ou base)
- Isso poderia ser representado no mundo das classes com uma classe Polygon da qual derivaríamos as outras duas: Retângulo e Triângulo
- A classe Polygon conteria membros comuns a ambos os tipos de polígono (largura e altura)
- Retângulo e Triângulo seriam suas classes derivadas, com características específicas que são diferentes de um tipo de polígono para outro



Herança - 2

- As classes derivadas de outras herdam todos os membros acessíveis da classe base
- Isso significa que se uma classe base incluir um membro A e derivarmos uma classe dela com outro membro chamado B, a classe derivada conterá o membro A e o membro B
- A relação de herança de duas classes **é declarada na classe derivada**
- Sintaxe:

```
class derived_class_name:  public base_class_name
{ /*...*/ };
```

  - `derived_class_name` é o nome da classe derivada
  - `base_class_name` é o nome da classe na qual se baseia

Acesso

- O especificador de acesso `public` pode ser substituído por qualquer um dos outros especificadores de acesso (`protected` ou `private`)
- Este especificador de acesso limita o nível mais acessível para os membros herdados da classe base:
  - Os membros com um nível mais acessível são herdados com este nível
  - Os membros com um nível de acesso igual ou mais restritivo mantêm seu nível restritivo na classe derivada
- Quando uma classe herda outra, os membros da classe derivada podem acessar os membros protegidos herdados da classe base, mas não seus membros privados

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no

Exemplo

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

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };
```

Exemplo - 2

```
class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

O que é herdado da classe base?

- Em princípio, uma classe derivada publicamente herda o acesso a todos os membros de uma classe base, exceto:
  - seus construtores e seu destruidor
  - seus membros de operador de atribuição (operator =)
  - seus amigos
  - seus membros privados
- Mesmo que o acesso aos construtores e destruidor da classe base não seja herdado como tal, eles são chamados automaticamente pelos construtores e destruidor da classe derivada
- Os construtores de uma classe derivada chamam o construtor padrão de suas classes base (ou seja, o construtor não aceita argumentos)
- Chamar um construtor diferente de uma classe base é possível, usando a mesma sintaxe usada para inicializar variáveis de membro na lista de inicialização:

```
derived_constructor_name (parameters) :  base_constructor_name (parameters) ...
```

Exemplo

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

class Mother {
  public:
    Mother ()
      { cout << "Mother: no parameters\n"; }
    Mother (int a)
      { cout << "Mother: int parameter\n"; }
};
```

Exemplo - 2

```
class Daughter : public Mother {
  public:
    Daughter (int a)
      { cout << "Daughter: int parameter\n\n"; }
};

class Son : public Mother {
  public:
    Son (int a) : Mother (a)
      { cout << "Son: int parameter\n\n"; }
};

int main () {
  Daughter kelly(0);
  Son bud(0);

  return 0;
}
```

Herança múltipla

- Uma classe pode herdar de mais de uma classe simplesmente especificando mais classes base, separadas por vírgulas, na lista de classes base de uma classe

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

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
};
```

Herança múltipla - 2

```
class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}
```

Herança múltipla - 3

```
class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```

Polimorfismo

- Um dos principais recursos da herança de classe é que um ponteiro para uma classe derivada é compatível com o tipo de um ponteiro para sua classe base
- O polimorfismo é a arte de tirar vantagem desse recurso simples, mas poderoso e versátil

Exemplo

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

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};
```

Exemplo - 2

```
class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Exemplo - 3

- Desreferenciar `ppoly1` e `ppoly2` (com `ppoly1->` e `ppoly2->`) é válido e nos permite acessar os membros de seus objetos apontados
- Mas, como o tipo de `ppoly1` e `ppoly2` é um ponteiro para o tipo `Polygon` (e não um ponteiro para `Rectangle` nem um ponteiro para `Triangle`):
  - Os membros herdados de `Polygon` **podem** ser acessados
  - Os membros das classes derivadas `Rectangle` e `Triangle` **não podem** ser acessados
- É por isso que o programa acima acessa os membros `area()` de ambos os objetos usando `rect` e `trgl` diretamente, em vez dos ponteiros; os ponteiros para a classe base não podem acessar os membros `area()`
- O membro `area()` poderia ter sido acessada com os ponteiros para `Polygon` se fosse um membro de `Polygon` em vez de um membro de suas classes derivadas, mas o problema é que `Rectangle` e o `Triangle` implementam versões diferentes da área

Membros virtuais

- Um membro virtual é uma função que pode ser redefinida em uma classe derivada, preservando suas propriedades de chamada por meio de referências
- A sintaxe para que uma função se torne virtual é preceder sua declaração com a palavra-chave `virtual`
- A palavra-chave `virtual` permite que um membro de uma classe derivada com o mesmo nome de um na classe base seja chamado apropriadamente de um ponteiro, e mais precisamente quando o tipo do ponteiro é um ponteiro para a classe base que está apontando para um objeto da classe derivada

Exemplo

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

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};
```

Exemplo - 2

```
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};
```

Exemplo - 3

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```