Herança em C++

Ana Paula Rocha (versão 2004/2005)

Luís Paulo Reis (versão 2006/2007)

Herança

- Herança
  - permite usar classes já definidas para derivar novas classes
  - nova classe herda propriedades (dados e métodos) da classe base

Exemplo:
`Gestor` é um `Empregado`

- Classe base `(Empregado)` também se denomina superclasse
- Classe derivada `(Gestor)` também se denomina subclasse

Visibilidade de membros

- Membro da classe derivada pode usar os membros públicos (`public`) e protegidos (`protected`) da sua classe base (como se fossem declarados na própria classe)
- Membro protegido `(protected)` funciona como:
  - membro público para as classes derivadas
  - membro privado para as restantes classes

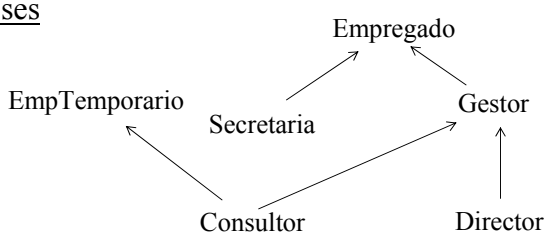
```
class classeDerivada: tipo_acesso classeBase
{
   ...
};
```

- Tipo de acesso
  - **`public`** : não altera a visibilidade dos membros da classe
  - **`private`** : herda os membros `public` e `protected` como privados
  - **`protected`** : herda os membros `public` e `protected` como protegidos

Hierarquia de classes

- Uma classe derivada pode também ser uma classe base

$\Rightarrow$ hierarquia de classes




```
class Empregado { ... };
class Gestor : public Empregado { ... };
class Director : public Gestor { ... };
class EmpTemporario { ... };
class Secretaria : public Empregado { ... };
class Consultor : public EmpTemporario, public Gestor { ... };
```

Sumário das Permissões de Acesso

Tipo de Acesso	public	protected	private
Membros da mesma classe	Sim	Sim	Sim
Membros de classes derivadas	Sim	Sim	não
não membros	Sim	não	não

- Protected semelhante a private. Diferença ocorre na herança:
  - Quando uma classe herda de outra, os membros da classe derivada podem aceder aos membros protegidos da classe base mas não aos privados

Classes e subclasses

```
class Empregado {
   string apelido;
   int departamento;
 public:
   Empregado(const string &n, int d);
   void imprime() const;
}
```

```
class Gestor: public Empregado {
   int nivel;
 public:
   Gestor(const string &n, int d, int nv);
   void imprime() const;
}
```

Classes e subclasses

```
void Empregado:: imprime() const
{
  cout << apelido << endl;
  cout << departamento << endl;
}
```

```
void Gestor:: imprime() const
{
  Empregado:: imprime();
  cout << nivel << endl;
}
```


Classes e subclasses

- Objecto de uma classe derivada pode ser tratado como objecto da classe base, quando manipulado através de apontadores e referências
  - Se classe **`Derivada`** possui uma classe base pública **`Base`**, então: **`Derivada*`** pode ser usada em variáveis do tipo **`Base*`** (o oposto não é verdade)

```
Gestor g1;
Empregado e1;
...
Vector<Empregado*> v;
v.push_back(&g1);
v.push_back(&e1);
```

Construtor e Destrutor

- Construtor
  - Se classe base possui construtor, este deve ser invocado. Construtor por omissão é invocado implicitamente
  - Em primeiro lugar, é invocado o construtor da classe base, e só depois o construtor da classe derivada

```
Empregado::Empregado(const string &n, int d):
  apelido(n), departamento(d) { }
```

```
Gestor::Gestor(const string &n, int d, int nv):
   Empregado(n,d) , nivel(nv) { }
```

- Destrutor
  - Em primeiro lugar, é invocado o destrutor da classe derivada, e só depois o destrutor da classe base

Funções virtuais

- Funções **virtuais** são declaradas na classe base, e redefinidas nas classes derivadas
  - Compilador garante que a função correcta é invocada para o objecto usado
- **Polimorfismo**
  - Obter o comportamento "certo" das funções independentemente do tipo exacto de objecto (classe base ou derivada) que é usado
  - Funções devem ser virtuais
  - Objectos devem ser manipulados por apontador

Funções virtuais

```
class Empregado {
   string apelido;
   int departamento;
public:
   Empregado(const string & n, int d);
   virtual void imprime() const;
}
```

```
void Empregado:: imprime() const
{ cout << apelido << " , dep: " << departamento << endl; }
```

```
class Gestor : public Empregado {
   int nivel;
public:
   Gestor(const string & n, int d, int nv);
   void imprime() const;
}
```

```
void Gestor:: imprime() const
{  Empregado:: imprime();
   cout << "    nivel: " << nivel << endl; }
```


Funções virtuais

```
void imprimeTodos(const vector<Empregado*> & v)
{
   for(vector<Empregado*>::iterator it=v.begin() ;
                it!=v.end(); ++it)
       (*it)->imprime();
}

int main()
{
   Empregado e1("Silva",1234);
   Gestor g1("Costa",1234,2);
   vector<Empregado*> pessoal;
   pessoal.push_front(&e1);
   pessoal.push_front(&g1);
   imprime_todos(pessoal);
   return 0;
}
```

Resultado:

Costa , dep: 1234
nivel: 2
Silva , dep: 1234

Classe abstracta

- Classe abstracta:
  - Representa conceitos abstractos, para as quais objectos não podem existir
  - Possui pelo menos um método abstracto
    - Método abstracto ou função virtual pura : é uma função virtual com inicialização = 0
- Se uma subclasse não implementa todos os métodos abstractos da classe base continua a ser abstracta
- Classes abstractas são usadas para especificar interfaces sem fornecer detalhes de implementação

Exemplo: Shape

```
// Shape class interface: abstract base class for shapes
class Shape
{
  protected:
     string name;
    float x,y;
    string color;
    string lineType;
  public:
     virtual ~Shape() { }
     virtual double area() const = 0;

     bool operator < (const Shape & rhs) const
     { return area() < rhs.area(); }

     friend ostream & operator << (ostream &out, const Shape &rhs);
};
```

método abstracto

```
ostream &operator << (ostream &out, const Shape & rhs) {
   out << rhs.name << " of area " << rhs.area() << endl;
   return out;
}
```

Exemplo: Shape

```
const double pi = 3.1415927;
class Circle: public Shape
{
   double radius;
  public:
    Circle(double r=0.0) : radius(r) { name="circle"; }
    double area() const { return pi*radius*radius; }
};
```

```
class Rectangle: public Shape
{
   double length, width;
  public:
    Rectangle(double l=0.0, double w=0.0) : length(l),
          width(w) { name="rectangle"; }
    double area() const { return length*width; }
};
```


Exemplo: Shape

```
class Square: public Rectangle
{
  public:
    Square(double s=0.0) : Rectangle(s,s) { name="square"; }
};
```

```
// struct pointer to Shape
struct PtrToShape
{
   Shape *ptr;
   bool operator < (const PTRToShape & rhs) const
   { return *ptr < *rhs.ptr; }
   const Shape & operator*() const
   { return *ptr; }
};
```

Exemplo: Shape

```
// read a pointer to a shape
istream & operator>>(istream
   &in, Shape *&s)
{
   char ch;
   double d1, d2;

   in >> ch;   // first character
   swith(ch)   // is shape
   {
     case 'c':
       in >> d1;
       s = new Circle(d1);
       break;
     case 'r':
       in >> d1 >> d2;
       s = new Rectangle(d1,d2);
       break;
     case 's':
       in >> d1;
       s = new Square(d1);
       break;
     default:
       cerr << "Needed one of c,
              r, or s" << endl;
       s = new Circle;
       break;
   }
   return in;
}
```


Exemplo: Shape

```
// insertionsort
template <class Comparable>
void insertionSort(vector<Comparable> &a)
{
   int n=a.size();
   for (int p=1; p<n; p++)
   {
     Comparable tmp = a[p];
     int j;
     for (j=p; j>0 && tmp<a[j-1]; j--)
       a[j] = a[j-1];
     a[j] = tmp;
}
```

Exemplo: `Shape`

```
int main()
{
   int numShapes;
   cout << "Enter number of shapes: ";
   cin >> numShapes;
   vector<ptrToShape> array(numShapes);
   //read the shapes
   for (int i=0; i<numShapes; i++)
   {
     cout << "Enter a shape: ";
     cin >> array[i].ptr;
   }
   insertionSort(array);
   cout << "Sorted by increasing size: "<< endl;
   for (int i=0; i<numShapes; i++)
     cout << *array[i] << endl;
}
```


Herança

- Herdados da Classe Base:
  - constructor e destructor
  - Membros operator=()
  - friends

- Embora os construtores e destrutores da classe base não sejam eles próprios herdados, o construtor por defeito (i.e., o seu construtor sem parâmetros) e o seu destrutor são sempre chamados quando um novo objecto de uma classe derivada é criado ou destruído.
- Se a classe base não têm default constructor ou é desejado que um construtor overloaded seja chamado quando se cria um objecto derivado, pode-se especificar isso na definição de construtores na classe derivada:

derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

Exemplo de Herança e Construtores

```
// construtores e classes derivadas
#include <iostream>
using namespace std;

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

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 cynthia (0);
  son daniel(0);
  return 0;
}
```

Resultado:

mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter

daughter (int a) // nada especificado: call default
son (int a) : mother (a) // construtor especificado: chamá-lo

Herança Múltipla

- Possível uma Classe herdar de mais do que uma outra classe:
- Exemplo:

```
class derived_class_name: public base_class_name1,
public base_class_name2, …{ /*...*/ };


class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;
```

Herança Múltipla (Exemplo)

```
// Herança Múltipla
#include <iostream>
using namespace std;

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

class COutput {
  public:
    void output (int i);
  };

void COutput::output (int i) {
  cout << i << endl;
  }
```


Herança Múltipla (Exemplo)

```
class CRectangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon, public COutput {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

Polimorfismo

- Um apontador para uma classe derivada é compatível com um apontador para a sua classe base
- Polimorfismo é a "arte" de utilizar esta característica simple mas extremamente poderosa

Polimorfismo (Exemplo)

```
// Apontadores para a classe Base
#include <iostream>
using namespace std;

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

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

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

Polimorfismo (Exemplo)

```
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Resultado:
20
10

- Criados 2 apontadores que apontam para objectos da class CPolygon (ppoly1 e ppoly2). Depois são-lhes atribuidas referencias para rect e trgl, válidas pois são ambos objectos de classes derivadas de CPolygon.
- Apontadores *ppoly1 e *ppoly2 são do tipo CPolygon* e só podem ser usados para referenciar membros que CRectangle e CTriangle herdam de CPolygon (não por exemplo a função area())
- Para usar area() com apontadores para a classe CPolygon, este membro deve também ser declarado em CPolygon (membro virtual)

Membros Virtuais (Exemplo)

- Membro de uma classe que pode ser redefinido nas suas classes derivadas (keyword virtual)

```
// virtual members
#include <iostream>
using namespace std;

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

class CRectangle: public CPolygon {
  public:
    int area () { return (width * height); }
  };

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

Membros Virtuais (Exemplo)

```
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Resultado:
20
10
0

- As 3 classes (CPolygon, CRectangle e CTriangle) têm os mesmos membros: width, height, set_values() e area(). area() declarado virtual na classe base porque vai ser redefenido em cada classe derivada
- Removendo a keyword virtual resulta em 0 para os 3 poligonos, em vez de (20, 10 e 0) pois a chamada é feita através de um apontador do tipo CPolygon*
- Uma classe que declara ou herda uma função virtual é designada por *classe polimórfica*

Classes Base Abstractas

- Classes Base Abstractas são similares à classe CPolygon anterior mas em que a função area() pode não ter mesmo implementação
- É feito adicionando =0 (igual a zero) à declaração da função:

```
// classe abstracta CPolygon
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area () =0;
};
```

- Como um dos membros não tem implementação, não podemos criar objectos (instâncias dele)

Classes Base Abstractas

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b) { width=a; height=b; }
    virtual int area (void) =0;
    void printarea (void) { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon {
  public:
    int area (void) { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void){ return (width * height / 2); }
  };
```


Classes Base Abstractas

- Exemplo com alocação dinâmica de memória:

```
int main () {
  CPolygon * ppoly1 = new CRectangle;
  CPolygon * ppoly2 = new CTriangle;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```