Herança em C++

Algoritmos e Estruturas de Dados

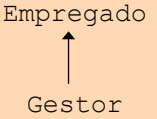
2020/2021

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

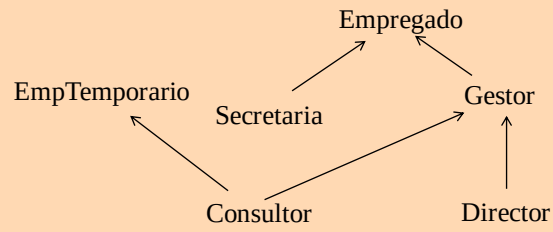
```
class classeDerivada: tipo_acesso classeBase
{
   ...
};
```

- 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

⇒ 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 { ... };
```

Classes e subclasses

```
class Empregado {
   string nome, 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;
}
```

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

Classes e subclasses

- Objeto de uma classe derivada pode ser tratado como objeto 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);
```

Funções virtuais

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

Funções virtuais

```
class Empregado {
    string nome, apelido;
    int departamento;
public:
    Empregado(const string & a, 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 & a, 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_back(&g1);
   pessoal.push_back (&e1);
   imprime_todos(pessoal);
   return 0;
}
```

Resultado:

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

Classe abstrata

- Classe abstrata:
  - Representa conceitos abstratos, para as quais objetos não podem existir
  - Possui pelo menos um método abstrato
    - Método abstrato ou função virtual pura : é uma função virtual com inicialização = 0
- Se uma subclasse não implementa todos os métodos abstratos da classe base continua a ser abstrata
- Classes abstratas 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;
  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 abstrato

```
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
    switch(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;
}
```