Aula 8: Polimorfismo

Paulo Cortez

2004/2005

Técnicas de Programação I
Licenciatura em Engenharia Electrónica e Computadores

`http://www.dsi.uminho.pt/disciplinas/EITPI`

Objectivos da Aula

- Utilizar polimorfismo em C++;

Polimorfismo

- No exemplo da aula 5, o apontador `vp` tanto poderia apontar para um `Vehicle` como para um `Truck`;
- A esta possibilidade, de uma variável poder alterar o seu comportamento conforme o seu tipo, chama-se de **polimorfismo**;
- Trata-se de uma característica importante na *Programação por Objectos*;
- No comportamento *static binding* (por defeito), na activação de uma função através de um apontador ou referência, é chamada a função associada ao tipo da variável;
- Por exemplo: para a variável `Vehicle *vp`, `vp->weight()` vai activar a função da classe `Vehicle`, mesmo que `vp` aponte para um `Truck`;

Funções virtuais

- O polimorfismo funciona através de um *late binding*, ou seja, só em tempo real de execução é que o compilador decide qual é a função que deve ser activada;
- Para se obter polimorfismo é necessário utilizar funções virtuais;
- Uma função é virtual se a sua declaração começar pela palavra `virtual`;
- Uma vez declarada como virtual numa classe base, a função mantém-se virtual EM TODAS AS SUAS CLASSES DERIVADAS, mesmo se a palavra `virtual` não é repetida na classe derivada;

Exemplo

```
// interface das classes
class Vehicle
    { public:
            virtual int weight() const;
            virtual void setWeight(int wt);
    };
class Truck: public Land
    { public:
            void setWeight(int engine_wt, int trailer_wt);
            int weight() const;
    };
```

Exemplo

```
int Vehicle::weight() const
    { return (weight); }
int Truck::weight() const
    { return (Auto::weight() + trailer_wt); }
// função main: apenas as funções membro da classe
// original é que podem ser chamadas, a não ser
// que se use a conversão dinâmica
int main()
    { Vehicle v(1200);
      Truck t(6000, 115, 15000);
      Vehicle *vp; // apontador genérico para veículos
      vp = &v;     // activa Vehicle::weight
      cout << vp->weight() << endl;
      vp = &t;     // activa Truck::weight
      cout << vp->weight() << endl;
      // Esta instrução dá erro pois speed não existe em Vehicle!
      // cout << vp->speed() << endl;
    }
```

Destrutores virtuais

- Quando o operador `delete` liberta a memória de um objecto alocado dinamicamente, o respectivo destrutor é chamado, para garantir que a memória alocada pelo objecto é também apagada;
- No entanto, considere o seguinte exemplo:

```
Vehicle *vp = new Land(1000, 120);
delete vp; // objecto destruído
```

- Aqui é chamado o destrutor de `Vehicle` e não o destrutor de `Land`!

Destrutores virtuais

- A solução reside em utilizar um destrutor virtual:

```
class Vehicle
    { public:
        virtual ~Vehicle(); // destrutor virtual
        virtual int weight() const;
    };
```

- Agora, a instrução `delete vp;` activa o destrutor da classe `Land`, como é esperado!

Regras para destrutores

- Um destrutor deve ser definido quando uma classe contém objectos que irão alocar memória;
- Um destrutor virtual deve ser definido se uma classe contém pelo menos uma função virtual;
- Muitas das vezes, um destrutor virtual não faz nada, devendo ser definido como vazio:

```
Vehicle::~Vehicle(){}
```

Classes abstractas

- Até agora, a classe `vehicle` continha implementações das funções virtuais `weight()` e `setWeight()`;
- No entanto, é possível definir funções virtuais (**puras**) numa classe, sem as implementar;
- Nesta abordagem, define-se apenas um protocolo, que tem de ser seguido nas classes derivadas;
- A classe base passa então a ser uma classe abstracta, definindo apenas um modelo que as classes derivadas tem obrigatoriamente de implementar;
- Uma **função virtual pura** é definida pela palavra `virtual` antes da função, e pela expressão `=0` no fim da declaração;

Classes abstractas

```
class Object
   { public:
        virtual int size() const = 0;
   };
```

- Agora, todas as classes derivadas de `Object` têm de implementar o método `size()`;

Classes abstractas

```
class Object2 : public Object
   { int d_size;
     public:
        // método obrigatório
        int size() const { return d_size};
   };
```

- Na maior parte dos casos, as funções virtuais puras são apenas acessores, ou seja, utilizam o termo `const` a seguir à declaração;
- Muitas vezes, as classes abstractas não têm atributos;
- Nota: os destrutores virtuais não devem ser puros, não faz sentindo que cada classe derivada redifina o destrutor;

Conversão dinâmica

- A conversão dinâmica (`dinamic_cast<>()`) permite converter um apontador de uma classe base para uma classe derivada, em tempo real;
- O pré-requisito é que tem de existir pelo menos um método virtual na classe base;

Conversão dinâmica

```
class Base
    { public: virtual ~Base(); };
class Derived: public Base
    { public:
            char const *toString()
            { return ("Derived object");}
    };
int main()
    { Base *bp; Derived *dp, d;
      bp = &d;
      dp = dynamic_cast<Derived *>(bp);
      if(dp)//verifica em tempo real se há sucesso!
           cout << dp->toString() << endl;
      else cout << "dynamic cast conversion failed\n";
    }
```