Aula 5: Herança Múltipla e Desenho de Software

Paulo Cortez

2005/2006

Técnicas de Programação I
Licenciatura em Engenharia Electrónica e Computadores

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

Objectivos da Aula

- Descrever os conceitos relevantes da herança múltipla, aplicando-os em C++;
- Desenhar diagramas de classes com vista a modelar objectos (e suas relações) em problemas reais;

Herança Múltipla

- O C++ suporta uma derivação múltipla (uma classe pode ser derivada a partir de diversas classes);
- A herança simples é mais utilizada;
- É preferível usar **composição** do que herança múltipla;

```
class Knife
{ int d_blade;
  public:
    Knife(int blade):d_blade(blade){};
    int blade() const{ return d_blade; }
    void setBlade(int blade){ blade=blade;}
};
class CorkScrew
{ int d_screw;
  public:
    CorkScrew(int screw):d_screw(screw){};
    int screw() const{ return d_screw; }
    void setScrew(int blade){ d_screw=screw;} };
```

Herança Múltipla

```
class SwissKnife: public Knife, public CorkScrew
{ public:
    SwissKnife(int blade, int screw)
       : Knife(blade),CorkScrew(screw) {}
}
```

- Neste caso, a classe `SwissKnife` combina as funcionalidades das classes ascendentes, não introduzindo funcionalidades novas;
- Uma classe complexa foi criada a partir de classes mais simples, trata-se de um procedimento comum no C++;
- A ordem de execução dos construtores não depende do código do construtor, mas sim da ordem pelo qual as classes estão definidas na interface;

Conflitos: operador de escopo

- E quando existirem métodos com o mesmo nome em ambas classes superiores? É necessário utilizar o operador de escopo `::`;
- Por exemplo, se na classes `Knife` e `CorkScrew` existisse um método chamado `increase` então:

```
int main()
{ SwissKnife k(5,3);
  k.Knife::increase(2); // aumenta a lamina
  k.CorkScrew::increase(2); // aumenta o saca rolhas
}
```

Conversão entre objectos de classes derivadas

- Uma classe derivada **é** do mesmo tipo que a sua classe base;

```
SwissKnife k(5,3); Knife f(10);
f=k; // possível, d_blade de f = 5
```

- Na atribuição `f=k`, tudo que `f` não tem é ignorado (`d_screw`);
- A conversão de `k=f` dá erro, pois `k` é mais do que `f`;
- O mesmo se sucede com argumentos de funções:

```
int lbs_weight(Vehicle const &v)// converte peso em libras
{ return v.weight()*4.86;}
int main()
{ Land l(1200, 130); Truck t(2600, 120, 6000);
  cout << lbs_weight(l) << "\n";
       << lbs_weight(t) << "\n";
}
```

Conversão entre apontadores de classes derivadas

```
Land l(1200, 130);
Truck t(2600, 120, 6000);
Vehicle *vp;
vp = &l; // conversão implícita para Vehicle
vp = &t; // conversão implícita para Vehicle
```

- Apenas a funcionalidade de `Vehicle` pode ser aplicada a `vp`;

Conversão entre apontadores de classes derivadas

```
Truck t(2600,120, 6000); Vehicle *vp;
vp = &t; // vp aponta para Truck
cout << pt->weight() << "\n"; // 2600
Truck *pt;
pt = reinterpret_cast<Truck *>(vp);
cout << pt->weight() << "\n"; // 8600
```

- Aqui a conversão (`reinterpret_cast<Truck*>`) transforma `vp` em `Truck`;
- Mas é preciso ter cuidado, só funciona se `vp` apontar para um `Truck`;

O processo do desenho de *Software*

- O processo do desenho de software é **iterativo**, envolvendo as etapas como:
  - *análise* - compreensão dos requisitos exigidos;
  - *desenho* - criação de um modelo (de classes);
  - *implementação* - codificação numa linguagem de programação (C++);
  - *teste* - verificar se o código desenvolvido funciona como deveria;
- À medida que se trabalha numa etapa, podem surgir novos desenvolvimentos que afectam as etapas anteriores, pelo que este processo não é cíclico, poderá avançar em frente, para atrás e mesmo em espiral;
- Uma *linguagem de modelação* serve para representar em papel um modelo (*desenho*);
- O objectivo é criar um nível de abstracção: algo preciso mas mais simples do que o mundo real;

Desenho de *Software*

- Nesta disciplina, a etapa de desenho consistirá em dois aspectos:
  - Descrição das classes;
  - Diagramas de classes em *UML*;
- A **descrição das classes** envolve a identificação dos objectos/classes principais, bem como uma descrição dos mesmos:

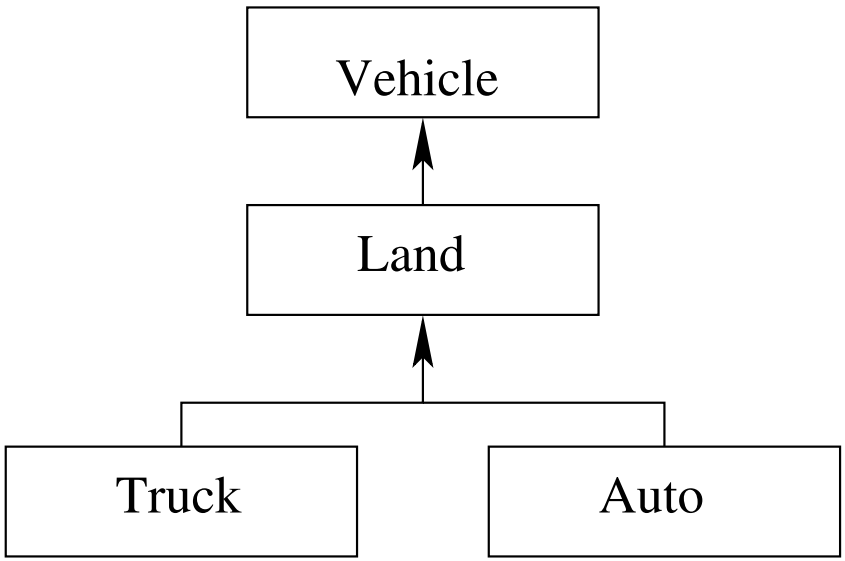
Classe	**Descrição**
Vehicle	Veículo móvel genérico
Land	Veículo terrestre
Truck	Camião com parte dianteira e atrelado
Auto	Autocarro de passageiros

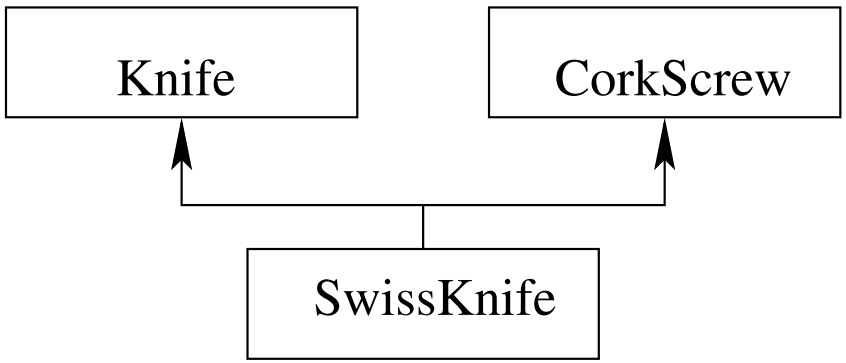
- Os **diagramas de classes** envolvem a definição das relações entre classes, bem como o conhecimento (**atributos**) e acções (**métodos**) de cada classe;

UML - Unified Modeling Language

- O **UML - Unified Modeling Language** é uma linguagem de modelação, sendo adoptada pela grande parte da indústria de software;
- No UML as relações entre classes podem ser definidas de acordo com:
  - as classes são desenhadas como rectângulos;
  - a herança (`é-um`) é definida por uma seta, que parte da classe descendente para a classe ascendente;

Exemplos de diagramas de classes simples





Diagramas de classes completos

- Os **diagramas de classes completos** servem para representar e explorar as relações entre classes, permitindo também representar o interface público;
- O rectângulo da classe é então dividido em três partes:
  - topo - onde surge o nome da classe;
  - meio - onde se representam os membros privados;
  - base - onde aparece tudo que é público;
- Porquê se deve utilizar **diagramas de classes**?
  - A correcta definição de classes é um problema complexo;
  - Antes de se construir uma casa, convém ter uma planta, com o projecto da mesma;
  - É mais fácil visualizar um problema no papel, num modelo simplificado, do que ter uma visão global através da implementação de código;

Exemplos de diagramas de classes completos