Aula 4: Classes e Herança

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 para a programação por classes, nomeadamente objectos, classes e herança;
- Aplicar o conceitos de objectos, classes e herança em C++;

Classes: Conceitos básicos

- O mundo real é composto por **objectos/classes**, que contém **atributos** (cor, forma, etc) e que interagem com o mundo através de um **interface** (som, movimento, etc);
- A **Programação Orientada aos Objectos** tenta simular esse mundo real, onde uma *classe* é composta por um conjunto de *atributos* (dados) e *métodos* (funções que manipulam os dados);
- A declaração (o *interface* da classe), é normalmente efectuada num ficheiro *header* (`.h`);
- Por sua vez, a definição das funções *membro* ou *métodos* (a implementação da classe), costuma ser efectuada num ficheiro `.cc` ou `.cpp`;

Visibilidade em classes

- A visibilidade de cada atributo/método é controlada por 3 instruções especiais:
  - `public` - de livre acesso, em qualquer situação;
  - `protected` - apenas acessível na própria classe e suas descendentes; e
  - `private`- apenas acessível na própria classe;
- Por defeito, nas estruturas (`struct`) tudo é `public`, enquanto que nas classes (`class`) tudo é `private`;

Exemplo: classe *Person*

```
#include <string>
class Person // Nome da classe
    {
      string d_name; // nome da pessoa
      int d_weight;  // peso em kg

      public:        // interface público
         void setName(string const &n);// alterar nome
         void setWeight(int weight);   // alterar peso
         string const &name() const; // aceder ao nome
          int weight() const;        // aceder ao peso
    };
```

Regras sobre um bom uso de classes

- Todos os atributos devem ser privados e devem estar definidos no inicio da interface;
- Os atributos devem começar com a sequência "`d_`", que sugere uma variável;
- Os métodos *manipuladores*, que modificam o estado interno de um objecto, devem começar por *set* (ou *altera*);
- Os métodos *acessores*, que devolvem informação, devem ter nomes simples (por exemplo, *weight*);

Uso de const e referências

- O termo `const`:
  - quando declarado após um método indica que se trata de um *acessor*;
  - quando declarado em argumento de entrada, indica que o argumento não será alterado;
- Uso de referências (`&`):
  - quando se quer devolver um tipo complexo (sem ser `int`, etc);
  - quando se quer alterar o argumento de entrada;
  - quando se quer aumentar a eficiência (é passado ao método apenas uma referência e não uma cópia do objecto);

Construtores

- Trata-se de um método especial, com o mesmo nome da classe, que é activado pelo compilador sempre que se cria um objecto;
- Não retorna nada, nem mesmo `void`;
- Se não existir, o compilador activa um construtor vazio;
- Quando um objecto é definido localmente, o construtor é chamado cada vez que a correspondente função é chamada;
- Quando é definido como estático (`static`) ou global, o construtor é chamado somente uma vez;
- O construtor deve inicializar todos os atributos que necessitam de valores iniciais (por exemplo, contador);

Exemplo (ficheiro: person.h) - declaração da classe

```
#include <string>
class Person // Nome da classe
{
  string d_name;  // nome da pessoa
  int d_weight;   // peso em kg

  public:         // interface público
  Person();       // construtor sem argumentos
  Person(string const &name, int weight=75);// c/ args
  void setName(string const &n); // alterar nome
  void setWeight(int weight);    // alterar peso
  string const &name() const;    // aceder ao nome
  int weight() const;            // aceder ao peso
};
```

Exemplo (ficheiro: person.cc) - definição dos métodos

```
#include "person.h"   // ficheiro anterior

Person::Person()
    {   d_name = "Rui"; d_weight = 75;}
Person::Person(string const &name, int weight)
    {   d_name = name; d_weight = weight;}
void Person::setName(string const &name)
    {   d_name = name; }
void Person::setWeight(int weight)
    {   d_weight = weight; }
string const &Person::name() const
    {   return d_name; }
int Person::weight() const
    {   return d_weight; }
```

Exemplo (ficheiro: organizer.cc) - uso da classe

```
#include <iostream>
#include "person.h"   // para ter acesso à classe
using namespace std; // definição do namespace std
int main()
{
 Person p1; // construtor vazio (Person p1(); da erro MinGW
 Person p2("Margarida",55); // construtor c/ args
 cout << "Nome de p1:" << p1.name() << "\n"; // Rui
 cout << "Peso de p2:" << p2.weight() << "\n";// 55
 return 0;
}
```

Objectos dentro de objectos: Composição

- Objectos ser atributos de outros objectos;
- Exemplo: a classe `Person` já contem um objecto do tipo `string`;
- Se possível, usar a inicialização automática de atributos (aumenta a eficiência):

```
Person::Person(string const &name, int weight)
 : d_name(name), d_weight(weight) {}
```

- Antes de executar a função ({}), são activados os construtores;
- Também pode ser usada em tipos simples (`int`, `double`, etc);
- A ordem de execução depende da ordem na qual os atributos são definidos na classe;

Regras para a organização de ficheiros `.h`

- Em C++, não se pode incluir o mesmo ficheiro duas vezes;
- Solução:
  - No início do ficheiro de interface (`.h`), inserir a linhas (`Classe` denota o nome da classe):

```
#ifndef _Classe_h_
#define _Classe_h_
```

- No fim do ficheiro de interface (`.h`), inserir a linha: `#endif`

Cópia de objectos

- Objectos com atributos simples (não apontadores) podem ser facilmente copiados com o operador =;

```
// b fica com cópia de a
Person a("Manuel", 75); Person b=a;
```

Herança: Conceitos básicos

- Em C é costume utilizar uma abordagem *top-down*: funções e acções que são definidas em termos de sub-funções, que por sua vez são definidas em sub-sub-funções, etc...
- Tal origina uma hierarquia de código: `main()` no topo, seguido das funções chamadas pela `main()`, etc...
- Em C++ as dependências são definidas em termos de classes, existindo duas formas principais:
  - **composição** (*contém um*)- já descrita anteriormente, onde um objecto inclui outro objecto;
  - **herança/derivação** (*é um*)- onde uma classe *descendente* (*sub-classe*) depende de outra classe *ascendente* (*super-classe*);

Herança: Conceitos básicos

- Uma classe *descendente* herda tudo da *super-classe* (atributos e métodos), podendo acrescentar atributos e/ou métodos novos;
- Uma *sub-classe* é uma especialização da classe *ascendente* e uma *super-classe* é uma generalização da classe *descendente*;

Exemplo: Veículos (vehicle.h)

```
class Vehicle // classe base, mais genérica
{ int d_weight;
  public:
     Vehicle() {}
     Vehicle(int weight){ d_weight=weight;}
     int weight() const { return d_weight;}
     void setWeight(int weight){ d_weight=weight;}
};
class Land: public Vehicle // classe descendente
{ int d_speed; // novo atributo
  public:
    Land(){}
    Land(int weight, int speed){setWeight(weight);d_speed=speed;}
    void setSpeed(int speed){ d_speed=speed;}
    int speed() const {return speed;}
};
```

- A classe `Land` contém todas as funcionalidades da classe ascendente `Vehicle`;
- A classe `Land` acrescenta um atributo novo (`d_speed`), bem como métodos para o manipular;
- A classe `Land` não pode aceder directamente ao atributo `d_weight`, pois está definido como privado;

```
int main()
{ Land v(1200, 145);
  cout <<"Peso: "<<v.weight()<<// método herdado
       <<" Vel.: "<<v.speed()<< "\n";
 return 0;
}
```

Derivação de subclasses

- Uma classe derivada pode derivar outras classes:

```
class Auto: public Land // classe descendente de Land
{ string d_name; // novo atributo
  public: ...
};
```

Construtor de uma classe derivada

- Na implementação anterior, ao activar o construtor de `Land`, o compilador tem de chamar o construtor da classe ascendente (que inicializa o peso) e depois chamar de novo o método `setWeight`;
- Uma solução mais eficiente consiste em chamar o construtor com o valor correcto de inicialização:

```
Land::Land(int weight, int speed) : Vehicle(weight)
    { d_speed=speed;}
```

Construtor de uma classe derivada

- Outra hipótese consiste em definir o atributo como protegido:

```
protected int d_weight;
```

definindo o construtor da forma:

```
Land::Land(int weight, int speed)
    { d_weight=weight; d_speed=speed;}
```

- Construtores são chamados segundo a ordem de uma pilha (*stack*): primeiro o construtor base, depois o construtor da classe derivada, etc...

Redefinição de métodos

```
class Truck: public Land // contém atrelado
{ int d_trailer_weight;
  public:
    Truck(){};
    Truck(int engine_wt, int speed, int trailer_wt);
    void setWeight(int engine_wt, int trailer_wt);
    int weight() const;
};
Truck::Truck(in engine_wt, int speed, int trailer_wt)
    : Land(engine_wt, speed)
{ d_trailer_weight = trailer_wt; }
```

- Apenas o método `setWeight` com dois argumentos pode ser aplicado a um objecto `Truck`, devido à redefinição do método;
- Contudo, o método `setWeight` do objecto ascendente (com um argumento)pode ainda ser usado, mas tem de ser de modo explícito: `Land::setWeight()`;

Redefinição de métodos

- Fora da classe, `setWeight` com 1 argumento pode ser acedido da seguinte forma:

```
int main()
{Truck t(3000,80,600);
 t.Land::setWeight(2900); // Land:: é obrigatório
 return 0;
}
```

Redefinição de métodos

- Uma alternativa elegante consiste em definir um novo método `setWeight` dentro da interface da classe `Truck`:

```
void setWeight(int engine_wt)
   { Auto::setWeight(engine_wt); }
```

- A função `weight` deverá ser também redefinida, pois altera o seu significado:

```
int Truck::weight() const
    { return Land::weight()+ d_trailer_weight;}
```