Corso di Laboratorio 2 – Programmazione C++

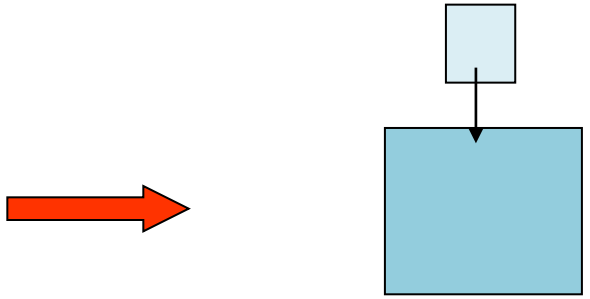
Silvia Arcelli

31 Ottobre 2014

Reimpiego di Codice

- Un punto fondamentale nei linguaggi di programmazione moderni è la possibilità di riutilizzo del codice. Nel caso specifico del C++, possiamo pensare di riutilizzare la realizzazione di una classe che descrive un certo tipo di oggetto per descrivere alcune proprietà di un altro tipo di oggetto, senza necessità di duplicare il codice.
- Il C++ offre questa possibilità secondo due approcci distinti:
  - Reimpiego per composizione
    - Contenimento diretto (istanze)
    - Contenimento indiretto (puntatori)

  - Reimpiego per Ereditarietà



Reimpiego per Composizione

- Come abbiamo visto, non c'e` alcun limite alla complessita` di un membro dato di un oggetto, che può essere sia di tipo nativo che tipo definito dall'utente; in particolare un attributo puo` a sua volta essere un oggetto.

```
class Veicolo
{ public:
Veicolo(const Motore & unmotore){/* .. */ };
private:
Motore motore;
 /* ... */
};
```

```
class Motore {
public:
Motore(/* Argomenti */);
 /* ... */
private:
 /* ... */
};
```

- Questo esempio modella una relazione di tipo **“Has-a”** (possesso), in cui una entita` è parte di un’altra esprimendo una proprietà della prima (una proprietà della macchina è il motore, qui intesa come “componente”).

Reimpiego per Composizione

La composizione puo` essere utilizzata anche per modellare una relazione di tipo "**Is-a**", in cui invece un oggetto di un certo tipo (Student) puo` essere vista anche come un oggetto di un tipo piu` generale(Person): Qui Person esprime più distintamente una proprietà in senso stretto e non un componente.

```
class Student {
public:
Student(const Person & p, const unsigned int code);
Student(const char *name, unsigned int age, const  int c
void PrintName();
private:
Person Self;
const int IdCode; // numero di matricola
};
Student::Student(const char* name,  int age
const  int code) :Self(name, age), IdCode(code) {}
Student::Student(const Person &p, const int code) :
 Self(p), IdCode(code) {}
void Student::PrintName() { Self.PrintName(); }
```

```
class Person {
public:
Person(const char* name, int
age): Name(name),Age(age){}
void PrintName();
private:
const char* Name;
int Age;
};
void Person::PrintName(){cout
<< "Nome:"<<Name <<endl;}
```

Reimpiego di Codice -Ereditarietà

Il meccanismo dell'ereditarieta` e` per molti aspetti simile a quello della composizione nel modellare una relazione di tipo "**Is-a**".

```
class Person {
public:
Person(const char* name, int age):
Name(name),Age(age){}
Person();
~Person();
void PrintName();
private:
char* Name;
unsigned int Age; };
```

```
class Student : public Person {
Student(const char *name,
unsigned int age, const  int code)
:Person(name, age), IdCode(code)
{}
Student();
~Student();
private:
const unsigned int IdCode; };
```

Nell'ereditarietà una nuova classe (la classe derivata) e` ottenuta da una preesistente (detta classe base) "importandone" il codice nella classe derivata, eventualmente sostituendone una parte (ridefinizione delle funzioni membro)e aggungendo degli attributi.

Reimpiego di Codice -Ereditarietà

•E’,come la composizione, un meccanismo per creazione di software riutilizzabile e per controllare la complessita’ del codice

• le classi derivate acquisiscono gli attributi e i comportamenti delle classi preesistenti (classi basi) ed aggiungono caratteristiche nuove o raffinano caratteristiche, specializzandone i comportamenti

•una classe derivata ha più dati e funzioni membro della classe base, e rappresenta tuttavia un gruppo più ristretto di oggetti, è più specializzata

Accesso ai Campi Ereditati

La classe derivata puo` accedere al suo interno ai membri protetti e pubblici della classe base come propri, e il codice ereditato continua a comportarsi nella classe derivata esattamente come si comportava nella classe base *(ereditarietà di implementazione):*

```
class Student : public Person {
public: Student();
~Student();
void DoNothing(); // Metodo proprio di Student
void Study(); // Metodo proprio di Student
private:
unsigned int IdCode; /* ... */
};
void Student::DoNothing() { Sleep();
//richiama Person::Sleep() }
void Student::Study() { /*-----*/; }
```

```
class Person {
public:
Person();
~Person();
void PrintName();
void Sleep();
private:
char* Name;
unsigned int Age;
/* ... */
};
```

Ereditarietà pubblica o privata

- Per default l'ereditarieta` è  privata, tutti i membri ereditati (pubblici o protected) diventano cioe` membri privati della classe derivata

- E` possibile richiedere esplicitamente un'ereditarieta` protetta o pubblica (ricordandosi che non si puo` comunque “allentare” il grado di protezione di un membro ereditato). In generale:

```
class < DerivedClassName > : [< Qualifier >] BaseClassName { /* ... */ };
```

- dove Qualifier e` opzionale e puo` essere o public, o protected o private; se omesso si assume private.

Ereditarietà pubblica o privata

I diversi "tipi" di derivazione (ereditarieta`) hanno conseguenze concettuali che vanno al di la` della semplice variazione del livello di protezione di un membro:

- Con l'ereditarieta` pubblica si modella effettivamente una relazione di tipo *Is-a* poiche` la classe derivata continua ad esportare l'interfaccia (cioè i metodi pubblici) della classe base (e` possibile utilizzare un oggetto *derived* come un oggetto *base*);

- con l'ereditarieta` privata questa relazione cessa, in un certo senso si può vedere l'ereditarieta` come una sorta di contenimento.

Reimpiego di Codice -Ereditarietà

Due differenze sostanziali tra l'ereditarieta` e la composizione:

- Con la composizione ciascuna istanza della classe contenitore contiene una istanza della classe componente, mentre con l'ereditarieta` le istanze della classe derivata non contengono nessuna istanza della classe base

- Un oggetto composto puo` accedere al suo interno solo ai membri pubblici della componente, l'ereditarieta` permette invece di accedere direttamente anche ai membri protetti della classe base (quelli privati rimangono inaccessibili alla classe derivata).

Ridefinizione dei Campi Ereditati

In molti casi e` desiderabile che una certa funzione membro, ereditata dalla classe base, si comporti diversamente nella classe derivata. In questo caso è possibile ridefinire il metodo ereditato (*overriding*);

```
void Student::PrintData() {
Person::PrintData();
cout << "Matricola: " << IdCode; }
```

In questo esempio per accedere direttamente ai dati privati della classe base si ridefinisce il metodo in modo da richiamare la versione ereditata attraverso l'operatore di risoluzione di ambito (che visualizza i campi privati della base) e quindi si aggiunge il codice specializzato alla classe derivata.

```
void Student::PrintData() { PrintData();
cout << "Matricola: " << IdCode; }
```

Notare che l'omissione di (Person::) avrebbe generato codice recursivo...

Ridefinizione dei Campi Ereditati

Non tutti i membri vengono effettivamente ereditati (sono da ridefinire):

- costruttori
- distruttore
- operatore di assegnamento

non vengono ereditati perche` il loro contenuto e` troppo legato alla effettiva struttura di una classe. Il codice di questi membri della classe base, se dichiarati public o protected, e` comunque disponibile all'interno della classe derivata (nel senso che possiamo sempre richiamarli tramite il risolutore di scope ::).

Costruttori per Classi derivate

Il costruttore per classi derivate è assolutamente “standard”. Ad esempio per il costruttore di Default :

```
Student::Student() { /* ... */ }
```

Tuttavia anche in presenza di ereditarietà pubblica, non si puo` accedere ai campi privati della classe base, e quindi come inizializzarli? Notare che codice del tipo:

```
Student::Student() { Person(/* ... */); /* ... */ }
```

Non funzionerebbe, perche` quando si giunge all'interno del corpo del costruttore, l'oggetto e` gia` stato costruito attraverso il costruttore di default. Ancora, non esiste la possibilita` di eseguire un assegnamento di un attributo della classe base, se questo è privato (a meno di non farlo attraverso un metodo pubblico ad hoc della base, ma a livello di costruzione è uno schema per nulla lineare..).

Costruttori per Classi derivate

Come nel caso dei membri const, la soluzione consiste nell'utilizzare la lista di inizializzazione:

```
Student::Student() : Person(/* ... */) { /* ... */ }
```

- Se nessun costruttore per la classe base viene menzionato, il compilatore richiama il costruttore di default (generando un errore se la classe base non ne possiede uno, meglio implementarlo sempre).
- Se il programmatore non specifica alcun costruttore per la classe derivata, il compilatore ne fornisce uno di default che richiama quello di default della classe base.
- Considerazioni analoghe valgono per il costruttore di copia fornito dal compilatore (richiama quello della classe base).

Ridefinizione di Campi Ereditati

- La classe derivata puo` anche definire nuovi metodi, compresa la possibilita` di eseguire l'overloading di una funzione ereditata (naturalmente la versione overloaded deve differire dalle precedenti per tipo e/o numero di parametri, non è una ridefinizione....)

- Infine non occorre ridefinire gli attributi (membri dato) della classe base (e non sarebbe neanche opportuno per la chiarezza del codice usare nomi uguali), ma tipicamente se ne aggiungono altri

ESERCIZIO-PARTE II

- Per l'esempio discusso a lezione la scorsa volta, si consideri ora questo caso. Il rivelatore di posizione è stato integrato con un sistema di misura di tempi di impatto (rispetto ad un tempo 0 coincidente con l'emissione delle particelle dalla sorgente).
- Modificare il software esistente in modo che si possano trattare sia i dati "vecchi" con solo la misura di posizione, sia i dati "nuovi" comprendenti anche l'informazione temporale, in modo più trasparente possibile per i programmi che utilizzano questi oggetti.

Ridefinizione di Campi Ereditati
la keyword virtual

- Nel caso di metodi definiti nella classe base ed eventualmente ridefiniti dalle classi derivate, è possibile utilizzare la keyword virtual:

```
virtual Type member_function(/* ... */) { /* ... */ };
```

- Questo fa sì che la funzione dichiarata come virtual nella classe base e ridefinita nelle derivate sia legata dinamicamente all'oggetto che la invoca. Ciò permette di scegliere correttamente al run-time il metodo invocato, in base al tipo dinamico dell'oggetto (polimorfismo al run-time).

Ridefinizione di Campi Ereditati
la keyword virtual

.

- Ad esempio, se puntiamo diversi oggetti derivati usando un puntatore di tipo classe base, l'invocazione del metodo attraverso il puntatore alla classe base avrà l'effetto di richiamare il metodo eventualmente ridefinito nella classe derivata.

- Questo non avviene se il metodo non è dichiarato virtual: verrà comunque chiamato il metodo della classe base. Notare che se non si usano puntatori ma istanze, comunque verrà chiamato il metodo della classe base.

ESERCIZIO-PARTE III

- Scrivere un programma che, avendo a disposizione un set di dati “vecchi”, e un set di dati “nuovi”, possa accedere all’informazione posizionale e all’informazione temporale dei dati in modo quanto più possibile trasparente (polimorfo ;-)) rispetto al tipo di dato (“vecchio” o “nuovo” che sia)

Regole di conversione fra classi in ereditarità

.

- Se la classe base dichiara ma non implementa il metodo virtual, ovvero contiene una *funzione virtuale pura:*

```
virtual Type member_function(/* ... */) =0;
```

si dice classe astratta. Tutte le figlie sono obbligate ad implementare il metodo. La classe astratta non si può istanziare, ma funge da pura interfaccia comune. In questo caso si parla di *ereditarietà di interfaccia.*

Ridefinizione di Campi Ereditati
la keyword virtual

- suggerimento: se una classe potrebbe in futuro essere derivata , e` buona prassi che i metodi dell'interfaccia (potenzialmente soggetti a ridefinizione) siano dichiarati virtual;

- Non e` possibile avere costruttori virtuali. Invece, per i distruttori, e` bene che una classe che possiede metodi virtuali abbia anche il distruttore virtual, in modo che distruggendo oggetti polimorfi venga sempre invocato il distruttore corretto.

Ridefinizione di Campi Ereditati
la keyword virtual

Esempio di uso di Virtual:

- classe base Forma
- classe derivata Punto
- classe derivata Cerchio

si vuole poter invocare una unica espressione per stampare la caratteristiche (nome, area,..) delle varie forme definite dalle classi derivate.

Ridefinizione di Campi Ereditati
la keyword virtual

```
//Definition of abstract  base class Shape
#ifndef SHAPE_H
#define SHAPE_H
class Shape
{
public:
virtual double area() const{return 0.0; }
virtual double volume() const{ return 0.0; }
virtual void printShapeName()const= 0;
virtual void print() const= 0;
};
#endif
```

File di intestazione
shape.h

Ridefinizione di Campi Ereditati
la keyword virtual

```
#ifndef POINT_H
#define POINT_H
#include <iostream>
#include "shape.h"
class Point: public Shape{
public:
Point(int= 0,int= 0 );
int getX()const{return x; }
int getY()const{return y; }
virtual void printShapeName()const{cout<< "Point: "; }
virtual void print()const;
private:
int x, y;
void setPoint(int,int);};
#endif
```

point.h

```
//Member function definitions for class Point
#include "point.h“
Point::Point(int a,int b ) {setPoint( a, b ); }
void Point::setPoint(int a,int b ){x = a; y = b;}
void Point::print()const{
cout<< '[' << x << ", " << y << ']';
 }
```

point.cpp

Ridefinizione di Campi Ereditati
la keyword virtual

circle.h

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"
class Circle: public Point{
public:
Circle( double r = 0.0, int x = 0, int y = 0 );
void setRadius ( double);
double getRadius() const;
virtual double area() const;
virtual void printShapeName() const{ cout << " Circle: "; }
virtual void print () const;
private:
double radius; // radius of Circle
};
#endif
```

Ridefinizione di Campi Ereditati
la keyword virtual

Member function definitions for class Circle

circle.cpp

```
#include <iostream>
using std::cout;
#include "circle.h”
Circle::Circle(double r, int a, int b ):Point( a, b ){ setRadius ( r ); }
void Circle::setRadius(double r ){ radius = r > 0 ? r : 0; }
double Circle::getRadius()const{return radius; }
double Circle::area()const{ return 3.14159 *radius*radius; }
void Circle::print()const{Point::print();cout<< ";Radius= " <<radius;}
```

Ridefinizione di Campi Ereditati
la keyword virtual

main.cpp

```
#include <iostream>
#include "shape.h"
#include "point.h"
#include "circle.h"
int main(){
Point point( 7, 11 ); // create aPoint
Circle circleA( 3.5, 22, 8 ), Circle circleB( 1.5, 10, 3 ); // create two Circles
Point.printShapeName(); //static binding
Shape*arrayOfShapes[ 3 ]; //array of base-class pointers
arrayOfShapes[ 0 ] = & point;arrayOfShapes[ 1] = & circleA;arrayOfShapes[ 2] =
    &circleB;
for (int i=0;i<3;i++){
arrayOfShapes[i]->printShapeName(); //dynamic binding
arrayOfShapes[i]->print();arrayOfShapes[i]->area();arrayOfShapes[i]->volume();
}}
```