Introduzione alle classi

Corso di Programmazione 3 - Ingegneria dell'Informazione e dell'Organizzazione
23 ottobre , 2001

Gino Perna

Esempi di semplici classi in C++

INTRODUZIONE AGLI OGGETTI

Fin dall'inizio abbiamo utilizzato la parola oggetto, indicando con tale termine un'entità che fornisse ad un dato una astrazione software avente lo scopo di rendere più semplice l'utilizzo di esso. Sappiamo che quando definiamo una variabile essa costituisce l'astrazione di una cella di memoria principale; quando compiamo delle operazioni in C++ su tale variabile, il compilatore pensa poi a tradurre tali operazioni astratte in operazioni in un linguaggio abbastanza semplice da potere essere compreso dalla macchina. Per gli oggetti il discorso è il medesimo: noi "*descriviamo*" un oggetto, il quale può essere modificato con delle operazioni ad alto livello di astrazione, ed il compilatore si occupa della sua implementazione a livello hardware. Tale procedimento è efficiente in quanto il compilatore riesce a tradurre molto bene i nostri programmi, i quali risultano d'altronde semplici per il programmatore.

Come facciamo però a "*descrivere*" un oggetto? Con le variabili di tipo primitivo non è possibile: se noi, ad esempio, definiamo con la variabile `int n` il voto di uno studente, è ovvio che n dovrà essere positivo per avere un significato; tuttavia il compilatore sa che tale variabile è di tipo intero e non si preoccuperà minimamente di controllare eventuali controsensi, perché troppo astratti per una macchina. Oppure, se sappiamo che con `double x` stiamo indicando lo stipendio di un impiegato, è per noi ovvio che non ha alcun significato calcolare il seno di `x`; tuttavia se proviamo ad eseguire tale operazione, il compilatore non si lamenterà affatto. Descrivere un oggetto corrisponde in C++ invece a creare una classe, cioè un nuovo tipo, descrivendo al suo interno i dati che l'oggetto deve contenere e dichiarando le operazioni che è possibile effettuare su tali dati. Si tratta di nascondere l'implementazione, cioè fare in modo che l'utilizzatore di una classe possa compiere solo determinare operazioni su oggetti di tale classe. Mentre calcolare il seno di un `double` è sempre possibile, e non possiamo agire in nessuna

maniera sulle operazioni permesse sui tipi primitivi, se creiamo un classe siamo noi a decidere quali operazioni sono permesse e quali non lo sono.

C'è anche un altro vantaggio nel nascondere la rappresentazione interna di un dato. Immaginiamo di costruire una classe la quale rappresenti un certo dato; siccome tale dato risulta nascosto ad un utilizzatore di tale classe, possiamo un domani modificare liberamente l'implementazione interna della classe (per aggiornarla o ottimizzarla o per qualunque altro motivo) senza che i programmi che la utilizzano ne risentano minimamente.

PROTEZIONE DEI MEMBRI DI UNA CLASSE

La sintassi di dichiarazione di una **class** è molto simile a quella di una struttura; la differenza principale risiede nel fatto che i campi membri di una classe hanno un grado di protezione, definito tramite le parole chiave **public**, **protected** e **private**. Tralasceremo la seconda di esse in quanto, per gli scopi del corso, i membri dichiarati **protected** si comportano esattamente come quelli dichiarati **private**. Vediamo innanzitutto un esempio di dichiarazione di una classe:

```
class Rettangolo {
 public:
   double area ();
 private:
   double altezza;
   double larghezza;
 };
```

I campi altezza e larghezza sono stati dichiarati **private**: essi non sono accessibili da nessuna funzione che non sia una contenuta nella classe Rettangolo. Al contrario, la funzione **area** è stata dichiarata **public** proprio perché studiata per essere acceduta dal mondo esterno. Dunque, la larghezza e l'altezza di un oggetto di tipo Rettangolo non possono essere modificate in maniera semplice: abbiamo bisogno di introdurre una funzione la quale ci consenta tale operazione, come vediamo nel seguente esempio:

Es. 1.ex1.cpp

```
class Rettangolo {
public:
  double area () {    // definizione
    return larghezza * altezza;
  }
  void modifica (double, double);  // dichiarazione
private:
  double altezza, larghezza;
};

void Rettangolo::modifica (double Altezza, double Larghezza) {
  altezza = Altezza;
  larghezza = Larghezza;
```

```
}

void main() {
  Rettangolo r;
  r.modifica (5, 7);
  r.area();            // = 35
  // r.altezza = 10;   // NO! altezza e` PRIVATO
}
```

Una prima importante osservazione: i campi di una classe possono essere variabili (primite o derivate), funzioni, *enum*, *strutture*, altre classi (sconsigliato); in ogni caso valgono le regole della protezione dei membri: si tratti di una funzione o una variabile, dall'esterno ci si può accedere solo se essa è public. Per quanto riguarda le funzione, esse possono essere definite all'interno della classe ex1.cpp , a pagina 1 oppure semplicemente dichiarate e definite altrove; per definire una funzione già dichiarata all'interno di una classe, dobbiamo utilizzare il nuovo operatore "*risolutore di visibilità*": '**::**' il quale accetta come operando sinistro il nome della classe che contiene la dichiarazione di funzione, e come operando destro il nome che la funzione stessa ha all'interno della classe.

Un'ultima nota: i membri di una classe che non sono preceduti da nessuna dichiarazione di protezione sono implicitamente **private**.

Vediamo un altro semplice esempio:

Es. 2.ex2.cpp

```
#include <iostream.h>

const int MAX = 100;

class Voti {
  double voti[MAX];  // campo PRIVATO
  int n;     // numero dei voti inseriti
public:
  // funzione di inserimento che torna `true'
  // in caso di successo, `false' in caso di fallimento
  bool inserisci (double v);
  // funzione che inizializza i campi dati privati
  // dal prossimo capitolo sara` sostituita
  // dalle funzioni costruttori
  void inizializza ();
  // funzioni che tornano la media dei voti,
  // il massimo e il minimo di essi
  double media ();
  double max ();
  double min ();
};

bool Voti::inserisci (double v) {
  if (n >= MAX || v < 0)
    return false;
  voti[n++] = v;
```

```
  return true;
}

void Voti::inizializza () {
  for (int i = 0; i < MAX; i++)
    voti[i] = 0;
  n = 0;
}

double Voti::media () {
  if (n == 0) return -1;   // errore: nessun dato!
  double somma = 0;
  for (int i = 0; i < n; i++)
    somma += voti[i];
  return somma / n;
}

double Voti::max () {
  if (n == 0) return -1;   // errore: nessun dato!
  double m = voti[0];
  for (int i = 1; i < n; i++) // attenzione alle graffe!
    if (voti[i] > m)
      m = voti[i];
  return m;
}

double Voti::min () {
  if (n == 0) return -1;   // errore: nessun dato!
  double m = voti[0];
  for (int i = 1; i < n; i++)
    if (voti[i] < m)
      m = voti[i];
  return m;
}

void main() {
  Voti v;
  v.inizializza ();
  cout << "scrivi un numero negativo per terminare\n";
  for (int i = 0; i < MAX; i++) {
    double voto;
    cout << "? "; cin >> voto;
    if (voto < 0)
      break;
    v.inserisci (voto);
  }
  cout << "media : " << v.media() << "\n";
  cout << "massimo : " << v.max() << "\n";
  cout << "minimo : " << v.min() << "\n";
}
```

esempio di output:

```
scrivi un numero negativo per terminare
```

```
? 7
? 7.5
? 6
? 6
? 7
? -1
media : 6.7
massimo : 7.5
minimo : 6
```

Come si vede, una volta costruita, la classeVoti è semplicissima da utilizzare, molto più di un semplice array, per il quale dovremmo di volta in volta controllare gli indici e passare il numero degli elementi ad ogni funzione. Non solo: nel prossimo capitolo vedremo che è possibile cambiare la struttura dati interna, che in questo caso è costituita da un array non dinamico, senza modificare il programma principale; in particolare, utilizzeremo un array dinamico.

Ovviamente, come vediamo nel seguente esempio, anche gli oggetti classe possono essere allocati dinamicamente.

Es. 3.ex3.cpp

```
#include <iostream.h>

const double RADICE2 = 1.4142135624;

class Quadrato {
public:
  void impostaDiag (double);
  void impostaLato (double);
  double lato () { return l; }
  double diag () { return d; }
  double area () { return l*l; }
private:
  double l;
  double d;
};

void Quadrato::impostaDiag (double x) {
  d = x;
  l = x / RADICE2;
}

void Quadrato::impostaLato (double x) {
  l = x;
  d = x * RADICE2;
}

void stampa (const Quadrato* q) {
  cout << "diagonale: " << q->diag() << "\n";
  cout << "lato: " << q->lato() << "\n";
  cout << "area: " << q->area() << "\n";
}
```

```
void main() {
  Quadrato* q = new Quadrato;
  double x;
  cout << "diagonale? "; cin >> x;
  q->impostaDiag (x);
  stampa (q);
  cout << "lato? "; cin >> x;
  q->impostaLato (x);
  stampa (q);
}
```

esempio di output:

```
diagonale? 6
diagonale: 6
lato: 4.24264
area: 18
lato? 4.5
diagonale: 6.36396
lato: 4.5
area: 20.25
```

Il vantaggio di avere a nostra disposizione una classe Quadrato consiste, ad esempio, nel fatto che possiamo inserire il lato oppure la diagonale, facendo in modo che il programmi imposti opportunamente la dimensione non data. Importante: è probabile che chi utilizza la classe Quadrato abbia intenzione di accedere le lunghezze di lato e diagonale; tuttavia non bisogna dichiarare tali informazioni public, bensì creare delle funzioni di accesso (nel nostro esempio: impostaDiag, impostaLato, lato e diag) le quali permettano all'utilizzatore della classe di avere accesso alla struttura dati, mantenendo quest'ultima protetta e coerente. Cosa succederebbe infatti se avessimo la seguente classe Quadrato2 in vece di Quadrato?

```
class Quadrato2 {
public:
  area ();
  double lato;
  double diag;
};
```

L'utilizzatore potrebbe impostare la lunghezza del lato (dal main ad esempio) ma non quella della diagonale (o viceversa) per cui ci troveremmo con un dato incoerente e, per questo, inutilizzabile. Uno dei vantaggi delle classi in C++ è proprio il seguente: esse “forzano” il programmatore ad utilizzare strutture dati solide eppure flessibili, dando spazio in futuro ad aggiornamenti, modifiche e nuovi utilizzi.

esercizio

Si scriva una classe Libro contentente i seguenti campi dati: nome del libro (array di caratteri non dinamico), costo in euro, numero di scaffale; si abbiano inoltre le seguenti funzioni membro:

```
void inizializza (const char*, double, int);
void stampa ();
void applicaSconto ();
```

le quali, rispettivamente, hanno i seguenti compiti:

- inizializzare i campi dati dell'oggetto classe;
- stampare tutti i dati dell'oggetto;
- diminuire del 20% il prezzo del libro in oggetto.