Utilizzo e scrittura di classi

Corso di Programmazione 3 - Ingegneria dell'Informazione e dell'Organizzazione
14 novembre , 2001

Gino Perna

Implementazione di classi in C++

UN ESEMPIO DI CLASSE: RISULTATI E PARTITE GIOCATE DA UN ATLETA

Vedremo un semplice esempio di classe per la gestione dei risultati delle partite di un atleta. Si vogliono memorizzare e gestire:

- Nome atleta;
- Partite giocate;
- Numero di punti effettuati;
- Calcolo della media.

Definizione della classe

```
class giocatori{
private:
     char nome[80];
     int partite;
     int punti;
     double media;
     double calcola_media(void);
public:
giocatori(); //costruttore default
giocatori(char *,int,int); // cotruttore 3 argomenti
giocatori(); // distruttore
void print(); // stampa dati
void update(); // aggiorna numero partite e punti
void setall(); // input dati totale

};
```

Implementazione classe

```
#include <iostream>
#include <giocatori.h>
#include <string.h>

giocatori::giocatori(char *nome, int partite, int punti){

// inizializzare nome, partite e punti

     media = calcola_media();
}

giocatori::giocatori(){

// azzerare tutte le variabili compresa la stringa
     media = calcola_media();

}

double calcola_media(void){
     return (double) punti/partite;
}

giocatori::giocatori(){
     cout << " distruggo "<<nome<<"\n";
}

giocatori::update(){

// da implementare

     media = calcola_media();
}

giocatori::print(){
     cout << "giocatore "<<nome<<"\n";
     cout << "partite "<<partite<<"\n";
     cout << "punti "<<punti<<"\n";
     cout << " Media:"<<media<<"\n\n";

}
```

Programma test

```
#include <iostream>
#include "giocatori.h>
#include <string.h>



int main(){
     giocatori A;
     giocatori B=giocatori("max",5,3);
```

```
    giocatori C("eli",8,3);

    C.print();
    B.print();
    B.update();
    B.print


return 0;

}
```

Esercizio

Si completi la classe nelle parti mancanti e si proceda alla verifica dell'implementazione.

Esercizio

Si implementi il lcostruttore di copia (vedi oltre) per la classe giocatore e si utilizzi un array di oggetti per simulare una squadra di 12 giocatori.

IL COSTRUTTORE DI COPIA

La filosofia di utilizzo dell'operatori di assegnamento è molto simile a quella di un altra tipica funzione membro di una classe: il costruttore di copia. Esso viene chiamato ogni volta che si ha la necessità di creare un oggetto a partire da uno preesistente, ovvero nelle inializzazioni sulla base di oggetti dello stesso tipo; il costruttore di copia (copy constructor) è indicato solitamente con X::X(X&). Le operazioni da eseguire all'interno di esso sono formalmente identiche a quelle dell'operatore di assegnamento, con l'unica importante differenza che non è necessario deallocare memoria: gli oggetti da inizializzare non esistono prima di tale operazione. Si faccia bene attenzione che, supponendo che x1 sia un oggetto di tipo X, lo statement:

```
X* x2 = new X(x1);
```

effettua una inizializzazione di x1 con i dati di x2; non ci si faccia trarre in inganno dal simbolo =: non si tratta di un assegnamento!

Vediamo un esempio con una classe ArrayS :

```
class ArrayS {
  /* ... */
public:
  /* ... */
  ArrayS (const ArrayS& a);
};
```

```
ArrayS::ArrayS (const ArrayS& a) {
  n = a.n;
  array = new double[n];
  for (int i = 0; i < n; i++)
    array[i] = a.array[i];
}
```

Il costruttore di copia, come ogni costruttore, viene chiamato automaticamente dal C++, non è possibile cioè invocarlo esplicitamente come l'operatore di assegnamento, o qualunque altra funzione membro. Se il costruttore di copia non è presente, esso viene automaticamente generato dal compilatore; in tal caso l'unica operazione compiuta è la copia membro a membro dei dati della classe; come l'operatore di assegnamento e il distruttore, il costruttore di copia è dunque necessario solo se la classe alloca memoria dinamica al suo interno. Tali speciali funzioni membro, qualora non fosse necessaria la loro esistenza, è bene dichiararle commentate, come abbiamo visto per il distruttore nella sezione precedente. Un esempio del costruttore di copia per ArrayS è il seguente:

```
#include "arrays2.h"
void main() {
  int n;
  cout << "quanti elementi? "; cin >> n;
  ArrayS x(n);
  for (int i = 0; i < n; i++) {
    cout << "? ";
    cin >> x[i];
  }
  // copiamo il vettore x in un vettore y dinamico
  // ed in un vettore z non dinamico
  ArrayS y(x);
  ArrayS* z = new ArrayS(x);
  // ordiniamo i due vettori y e z
  z->ordina();
  cout << "\ncopia del vettore:\n"; y.stampa();
  cout << "\ncopia ordinata:\n"; z->stampa();
}

esempio di output:
quanti elementi? 4
? 2.71
? 3.14
? -1.4
? 5e-6

copia del vettore:
2.71 3.14 -1.4 5e-06

copia ordinata:
3.14 2.71 5e-06 -1.4
```

Un'ultima osservazione: l'operazione di copia di una classe potrebbe essere priva di senso; si pensi ad esempio ad una classe la quale rappresenti il sistema sul quale viene

eseguito il programma: che senso avrebbe crearne una copia? In casi come questi, è possibile utilizzare uno stratagemma sottile: si dichiarino il costruttore di copia e l'operatore di assegnamento come funzioni membro private, senza definirle; sarà compito del compilatore inibire l'utilizzatore della nostra classe ad effettuare copie di un oggetto di tale classe.

LE LISTE LINKATE SEMPLICI

Introduzione

Una lista è costituita dal unità elementare, che noi chiameremo celle o elementi o unità, come abbiamo già fatto per l'array; i dati che le celle possono contenere sono del tutto arbitrari, ma devono essere omogenei: si possono avere liste di interi, di reali, di stringhe, di array, di oggetti strutture, ma non liste di cui alcuni elementi siano di un tipo e altri di uno diverso da esso. Inoltre le liste non sono indicizzate, come l'array; ovvero non è possibile accedere in maniera diretta un singolo elemento di una lista, ma bisogna scorrere necessariamente tutti gli elementi che lo precedono; una lista è dunque un dato ad accesso sequenziale.

Per capire le liste è molto importante avere bene fissato un modello mentale cui fare riferimento; per le liste semplici possiamo immaginare di avere delle celle, ognuna delle quali contiene un dato di un certo tipo, collegate l'un l'altra tramite una freccia la quale può portarci solo in un senso; il primo elemento ha una freccia che indica il nome della lista, la freccia dell'ultimo elemento punta 0:

Ciascun elemento è dunque costituito da due zone: una contenente l'informazione della cella, l'altra che permette di identificare l'elemento a essa successivo; un caso particolare è dato dall'ultimo elemento della lista, il quale non riferisce nessun ulteriore elemento. Definiamo infine come lista vuota quella che non ha alcun elemento.

Una lista dunque ha come scopo quello di rappresentare un insieme di dati, la cui dimensione varia fortemente durante l'esecuzione del programma; tale struttura dati è infatti intrinsecamente dinamica: ogni volta che un nuovo elemento viene aggiunto alla lista si alloca nuova memoria nello heap, facendo in modo che un puntatore a tale zona di memoria venga inserito all'interno della lista, nella maggior parte dei casi all'inizio o alla fine di essa. Tale flessibilità richiede tuttavia un prezzo da pagare: le liste non sono efficaci come gli array per la ricerca all'interno di esse, non consentendo esse modalità di accesso diretto; notiamo infatti che una lista è costituita da celle fondamentali, in ognuna delle quali è esclusivamente contenuto l'indirizzo della cella successiva: per accedere un elemento di una lista bisogna necessariamente accedere tutti i precedenti.

Liste semplici

In C++ non esiste un tipo ``lista", per cui dobbiamo costruire autonomamente una serie di funzioni le quali ci permettano di effetturare delle operazioni sulle nostre liste. In realtà ci sono numerose librerie, fornite a corredo di ogni distribuzione del C++, che forniscono una classe lista, ma ci sono due motivi per i quali non ne faremo uso: è un buon

esercizio ``reinventare la ruota'', cioè costruire da zero buona parte delle funzioni permesse sulle liste; inoltre, non abbiamo ancora le conoscenza necessarie per affrontare lo studio di una classe complessa, come quelle fornite con i compilatori C++. Ovviamente le liste che utilizzeremo saranno a scopo esclusivamente didattico, nel senso che ogni loro utilizzo professionale è impensabile per scarsezza di flessibilità ed efficienza.

Definizione classe

```
struct NODE {
    char nome[80]; // nome concorrente
    int voto; //voto
    NODE * next; //puntatore al prossimo elemento
};

class LISTA {
  private:
     NODE *head;
     NODE *tail;


  public:
     LISTA(); //cotruttore
     ~LISTA();//distruttore
     void insertstart(char *, int ); //inserisci all'inizio
     void insertend(char *, int ); //inserisci alla fine
     void print(); //stampa la lista
     void printTail(); // stampa la fine della lista
     void printHead(); // stampa l'inizio della lista
        void clear(); //cancella la lista
     NODE * search(char *); // ricerca l'elemento per nome
};
```

L'elemento principale è la struttura NODE che contiene le informazioni riguardanti i dati da memorizzare ed un puntatore al prossimo elemento;

La classe di per sè stessa non contiene nulla di nuovo eccetto che per la dichiarazione di due puntatori HEAD e TAIL che servono a tenere traccia del primo e dell'ultimo elemento della lista (normalmente nelle liste semplici si tiene traccia della sola testa).

Implementazione classe

```
#include <iostream.h>
#include <string.h>
#include "lli.h"


LISTA::LISTA(){
   head=NULL; // inizializzo a NULL sia la testa che la coda
   tail = NULL;
}

LISTA::~LISTA(){
   clear(); // distruttore: cancello la lista
}
```

```
void LISTA::clear(){

     NODE *walker = head; // Set a "walker node" to the front of
the list

     while(walker != NULL)
     {
          NODE *temp = walker; // Create a temp node

          walker = walker->next; // Move the walker to the next
node on the list

          delete temp; // Free the memory
     }

     head = tail = NULL; // Set everything to NULL

}
```

Inserzione in testa e in coda

L'operazione più semplice che si possa effettuare su di una lista è l'inserzione di un elemento in testa, ovvero come primo elemento; si noti che tale operazione in pratica provoca uno shift (trad. ``scorrimento") della lista verso destra: se inseriamo dei dati sempre in testa, otterremo una lista con i nostri dati inseriti in ordine inverso. Una semplice funzione che effettua l'inserzione in testa di un elemento è la seguente:

```
void LISTA::insertstart(char *nome, int voto){

    NODE *temp = new NODE;
    strcpy(temp->nome,nome);
    temp->voto = voto;
    temp->next = head;
    if(head == NULL) {
      tail=temp;
    }
    head = temp;
}
```

L'inserzione in testa presenta un notevole svantaggio: i dati risultano inseriti in ordine inverso rispetto a quello con il quale li si immette; conviene allora spesso eseguire delle inserzioni in coda, cioè aggiungere ogni nuovo elemento alla fine della lista. Tale procedura è un poco più articolata perché, mentre sappiamo bene dove inizia una lista, non possiamo immaginare dove essa termini se non la scorriamo completamente da sinistra a destra (immaginiamo sempre di avere la testa all'estrema sinistra e la coda all'estrema destra). Ricordiamo che la fine della lista è segnalata dal puntatore next messo a 0:

```
void LISTA::insertend(char *nome, int voto){

    NODE *temp = new NODE;
```

```
    strcpy(temp->nome,nome);
    temp->voto = voto;
    temp->next = NULL;
    if(head == NULL) head=temp;
    if (tail != NULL)tail->next = temp;
    tail = temp;
}


NODE * LISTA::search(char *nome){

   NODE *walker=head;
   while(walker != NULL){
      if(strcmp(nome,walker->nome) == 0) return walker;
      walker=walker->next;
   }
   return walker;

}

void LISTA::print(){
   NODE *walker=head;
   while(walker != NULL){
      cout << "Elemento:"<<walker->nome<<" Voto "<<walker-
>voto<<"\n";
      walker=walker->next;
   }
   cout << "Fine Lista\n";

}


void LISTA::printTail(){
      cout << "ElementoT:"<<tail->nome<<" Voto "<<tail-
>voto<<"\n";
}

void LISTA::printHead(){
      cout << "ElementoH:"<<head->nome<<" Voto "<<head-
>voto<<"\n";
}
```

Programma di test

```
#include "lli.h"
#include <string.h>
#include <iostream.h>

int main(){

  LISTA A;

  for(int i=0;i<5;i++){
```

```
    A.insertstart("pippoooo",i);
  }
  A.print();

  A.printTail();
  A.printHead();
  A.clear();
  A.insertend("clarabella",-1);
  A.print();
  A.insertstart("orazio",27);
  A.print();
  if(A.search("clarabella") != NULL) cout <<"Ti ho Trovato\n";




return 0;
}
```

Esercizi

Si completi la classe con i metodi per:

- Inserimento dopo un certo elemento;
- Cancellazione di un elemento (attenzione se è il primo o l'ultimo).