Programmazione 2 e Lab. di programmazione 2

Corso di Laurea in Informatica - Anno Accademico 2022-23

Docenti
Prof. Angelo Ciaramella
[angelo.ciaramella@uniparthenope.it]
Prof. Luigi Catuogno
[luigi.catuogno@uniparthenope.it]
Tutor
Dott. Antonio Vanzanella
[antonio.vanzanella@studenti.uniparthenope.it]

1

Descrizione del Corso

Libro di testo

[FdP]

H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

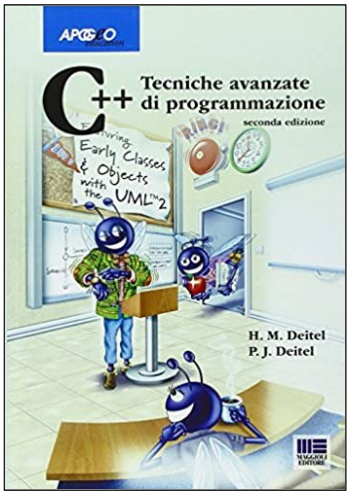
2

Descrizione del Corso

Libro di testo H. M. Deitel, P. J. Deitel

[TAP] C++ Tecniche avanzate di programmazione

II ed. (2011) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8572-6



3

Orari e modalità di ricevimento studenti

Docenti:

Prof. Angelo Ciaramella	Martedì dalle 14:00 alle 16:00 - telematico (codice Teams r3p3w0z)
Prof. Luigi Catuogno	Giovedì dalle 11:00 alle 13:00 - telematico (codice Teams)

Tutor:

Dott. Antonio Vanzanella	Martedì dalle 11:00 alle 13:00 - telematico (codice Teams 92dbag0)

4

Il Linguaggio C++

(per programmatori C)

Parte prima

5

Le **`class`** in C++

6

Esercizio: *dadi da gioco*

Scrivere la classe **dado** che abbia, (tra gli altri), il seguente attributo privato:

```
int facce;
```

La classe **dado** presenta la seguente interfaccia:

```
dado(int nf=6)
```

Costruttore di default. Il parametro **nf** è il numero di facce del dado . Se il valore fornito è minore di due, esso è posto a due.

7

Esercizio: *dadi da gioco*

La classe **dado** presenta la seguente interfaccia:

```
void agita(int seed)
```

Utilizza il parametro **seed** per riposizionare il PRNG

```
int lancio()
```

Restituisce un numero casuale compreso tra 1 e **facce**

8

Esercizio: *dadi da gioco*

```
 1  #include<iostream>
 2  #include<cstdlib>
 3  using namespace std;
 4
 5  class dado {
 6          private:
 7                  int facce;
 8          public:
 9                  dado (int nf=6) {
10                          facce = (nf<2?2:nf);
11                  };
12                  void agita(int seed) {
13                          srand(seed);
14                  };
15                  int lancio() {
16                          return rand()%facce+1;
17                  }
18  };
```

9

Esercizio: *bussolotto*

Scrivere la classe **bussolotto** che abbia, (tra gli altri), i seguenti attributi privati

```
dado *p;
int numdadi;
```

p è il puntatore a un array di **numdadi** oggetti di tipo **dado**.

10

Esercizio: *bussolotto*

La classe **bussolotto** presenta la seguente interfaccia:

```
public:
bussolotto(int num=2, int nf=6);
~bussolotto();
```

Il costruttore di default, alloca dinamicamente un array di **num** oggetti di tipo **dadi** da **nf** facce e il distruttore.

11

Esercizio: *bussolotto*

La classe **bussolotto** presenta la seguente interfaccia:

```
void agita(int seed)
```

Utilizza il parametro **seed** per riposizionare il PRNG

```
int lancio(int num=0)
```

Lancia **num** dadi e restituisce la somma dei numeri estratti. Se **num** è minore di 1 o maggiore di **numdadi**, allora li lancia tutti.

12

Esercizio: *bussolotto*

```
 5  class bussolotto {
 6          private:
 7                  dado *p;
 8                  int numdadi;
 9          public:
10                  bussolotto(int num=2, int nf=6){
11                          p=new dado[num](nf);
12                          numdadi=num;
13                  };
14                  ~bussolotto(){
15                          delete [] p;
16                  }
17          …
```

Questo non si può fare!! L'allocazione dinamica di un array di oggetti può invocare solo il costruttore di default (senza parametri)…

13

Esercizio: *bussolotto*

```
 5  class bussolotto {
 6          private:
 7                  dado *p;
 8                  int numdadi;
 9          public:
10                  bussolotto(int num=2, int nf=6){
11                          p=new dado[num];
12                          // ???
13                          numdadi=num;
14                  };
15                  ~bussolotto(){
16                          delete [] p;
17                  }
18          …
```

Dovremmo permettere al costruttore bussolotto() di modificare il numero di facce dei dadi dopo la loro creazione…

14

Classi (e funzioni) **`friend`**

Regole di visibilità dei membri di una classe *«servente»*:

public: l'accesso ai membri (sia attributi, sia funzioni) public è consentito a qualunque funzione e ai metodi di qualsiasi classe

private: l'accesso ai membri private è consentito:

- alle funzioni membro della stessa classe
- alle funzioni dichiarate **`friend`** dalla classe *«servente»*
- a tutti i metodi delle classi dichiarate **`friend`** dalla classe *«servente»*

15

Esercizio: *bussolotto*

```
class dado {
        friend class bussolotto;
        private:
                int facce;
        public:
        …
};
```

Modifichiamo la classe **dado** dichiarando la classe **bussolotto** come *friend*

```
 5  class bussolotto {
 6          private:
 7                  dado *p;
 8                  int numdadi;
 9          public:
10                  bussolotto(int num=2, int nf=6){
11                          p=new dado[num];
12                          for (int i=0;i<num;i++)
13                                  p[i].facce=nf;
14                          numdadi=num;
15                  };
16                  ~bussolotto(){
17                          delete [] p;
18                  }
 …          …
```

Ora il costruttore di **bussolotto** può modificare il membro **facce** degli oggetti **dado**, benchè privato.

16

Esercizio: *bussolotto*

```
class dado {
        friend class bussolotto;
        private:
                int facce;
        public:
        …
};
```

Modifichiamo la classe **dado** dichiarando la classe **bussolotto** come *friend*

```
 5  class bussolotto {
 6          private:
 7                  dado *p;
 8                  int numdadi;
 9          public:
10                  bussolotto(int num=2, int nf=6){
11                          p=new dado[num];
12                          for (int i=0;i<num;i++)
13                                  p[i].facce=nf;
14                          numdadi=num;
15                  };
16                  ~bussolotto(){
17                          delete [] p;
18                  }
```

Ora il costruttore di **bussolotto** può modificare il membro **facce** degli oggetti **dado**, benchè privato.

Il distruttore del bussolotto invoca prima il distruttore invoca il distruttore di tutti i dadi di p

17

Esercizio: *bussolotto*

```
50  int main()
51  {
52          bussolotto b1;
53          int seed;
54
55          cout << "Inserisci un seed: ";
56          cin >> seed;
57          b1.agita(seed);
58          cout << "Lancio i dadi: punteggio=" << b1.lancio() <<endl;
59  }
```

18

Assegnamento tra oggetti

19

Assegnamento tra oggetti

L'operatore di assegnamento **=** può essere utilizzato per assegnare un oggetto a un altro dello stesso tipo

Per default, questa operazione è effettuata copiando ogni membro di un oggetto in quello corrispondente dell'altro (la c.d. *copia membro a membro*)

20

Esempio: *assegnamento tra punti*

```
 5  class Punto {
 6          double x;
 7          double y;
 8  public:
 9          Punto() { x=y=0; };
10          Punto(double c1,double c2) {
11                  x=c1;   y=c2;
12          };
13          double getx() { return x; };
14          double gety() { return y; };
15  };
16  int main()
17  {
18          Punto p1(3,4), p2(5,6);
19          p1=p2;
20          cout << "p1.x="<<p1.getx()<<", p1.y="<<p1.gety()<<endl;
21  }
```

```
p1.x=5, p1.y=6
```

21

Assegnamento tra oggetti

L'operatore di assegnamento **=** può essere utilizzato per assegnare un oggetto a un altro dello stesso tipo

Per default, questa operazione è effettuata copiando ogni membro di un oggetto in quello corrispondente dell'altro (la c.d. *copia membro a membro*)

! Occorre tenere presente che questa operazione può dare dei problemi se gli oggetti dell'assegnamento, contengono membri allocati dinamicamente…

22

Esempio: *copia default membro-a-membro*

Facciamo un esperimento…

Utilizziamo la classe **`prova`** vista in precedenza, con una piccola variante

Effettuiamo un assegnamento tra due oggetti di questo tipo e poi valutiamo i risultati

23

Esempio: *copia default membro-a-membro*

```
 5  class prova {
 6          int *p;
 7          int size;
 8  public:
 9          int objId;
10          prova(int num = 10) {
11                  size=num;
12                  p=new int[size];
13                  objId = 0;
14          }
15          bool set(int pos, int data);
16          bool get(int pos, int &data);
17          void show();
18          int len();
18          int *pointer() {
19                  return p;
20          }
21  };
```

Ricordiamo che il metodo **`pointer()`** permetterà di ispezionare l'array allocato dalla classe indipendentemente dall'interfaccia della classe.

24

Esempio: *copia default membro-a-membro*

```
37  int main()
38  {
39          prova P, Q(50);
40          int *pp,*pq, data;
41          P.set(0,666); P.objId=6;
42          Q.set(0,777); Q.objId=7;
43          P.show();
44          Q.show();
45          pp=P.pointer(); pq=Q.pointer();
46          P=Q;
47          cout <<"Dopo l'assegn. P=Q, P.show()-> "     ;
48          P.show();
49          cout << "(!!) pp="<<pp<<", pp[0]="<<pp[0]<<endl;
50          Q.set(0,1997);
51          cout <<"Dopo Q.set(0,1997), P.show()-> "     ;
52          P.show();
53  }
```

25

Esempio: *copia default membro-a-membro*

Creiamo i due oggetti prova, alimentati con dati diversi e visualizziamo l'assetto «iniziale»

```
37  int main()
38  {
39          prova P, Q(50);
40          int *pp,*pq, data;
41          P.set(0,666); P.objId=6;
42          Q.set(0,777); Q.objId=7;
43          P.show();
44          Q.show();
45          pp=P.pointer(); pq=Q.pointer();
46  Id:6, len=10, pointer=0x7d1a60, p[0]=666
47  Id:7, len=50, pointer=0x7d5b50, p[0]=777
48  ...
49          Q.set(0,1997);
50          cout <<"Dopo Q.set(0,1997), P.show()-> "     ;
51          P.show();
52  }
53
```

26

Esempio: *copia default membro-a-membro*

```
37  int main()
38  {
39          prova P, Q(50);
40          int *pp,*pq, data;
41          P.set(0,666); P.objId=6;
42          Q.set(0,777); Q.objId=7;
43          P.show();
44          Q.show();
45          pp=P.pointer(); pq=Q.pointer();
46          P=Q;
47          cout <<"Dopo l'assegn. P=Q, P.show()-> "     ;
48          P.show();
49          cout << "(!!) pp="<<pp<<", pp[0]="<<pp[0]<<endl;
50          Q.set(0,1997);
51          cout <<"Dopo Q.set(0,1997), P.show()-> "     ;
52          P.show();
53  }
```

Conserviamo una copia dei puntatori agli array di ciascun oggetto e poi facciamo una copia di **Q** in **P**

27

Esempio: *copia default membro-a-membro*

```
37  int main()
38  {
39      ...
40      Dopo l'assegn. P=Q, P.show()-> Id:7, len=50,
41      pointer=0x7d5b50, p[0]=777
42      (!!) pp=0x7d1a60, pp[0]=666
43
44
45      ...
46          P=Q;
47          cout <<"Dopo l'assegn. P=Q, P.show()-> "     ;
48          P.show();
49          cout << "(!!) pp="<<pp<<", pp[0]="<<pp[0]<<endl;
50          Q.set(0,1997);
51          cout <<"Dopo Q.set(0,1997), P.show()-> "     ;
52          P.show();
53  }
```

L'array precedentemente allocato dal costruttore di **P** esiste ancora!

28

Esempio: *copia default membro-a-membro*

```
37  int main()
38  {
...
Dopo Q.set(0,1997),
P.show()-> Id:7, len=50, pointer=0x7d5b50, p[0]=1997
44          Q.show();
45          pp=P.pointer(); pq=Q.pointer();
46          P=Q;
47          cout <<"Dopo l'assegn. P=Q, P.sh
48          P.show();
49          cout << "(!!) pp="<<pp<<", pp[0]="<<
50          Q.set(0,1997);
51          cout <<"Dopo Q.set(0,1997), P.show()-> "
52          P.show();
53  }
```

Ora, qualsiasi accesso all'array di **Q** ha effetto anche su **P**!

29

Esempio: *copia default membro-a-membro*

Che cosa è successo?

30

Esempio: *copia default membro-a-membro*

Che cosa è successo?

Inizialmente, il membro **`p`** di ciascun oggetto conteneva il puntatore a un array di interi

Effettuando la copia membro a membro dall'oggetto **`Q`** all'oggetto **`P`**, il valore del puntatore proveniente da **`Q`** ha sovrascritto quello di **`P`**

Tuttavia, il vecchio array di **`P`**, non è stato deallocato e rimane in memoria (benchè privo di riferimenti)

31

Esempio: *copia default membro-a-membro*

Che cosa è successo?

Dopo l'assegnamento, gli oggetti **`P`** e **`Q`**, puntano allo stesso array (il campo **`p`** è uguale)

Qualsiasi modifica all'array di uno ha effetto anche sull'altro

32

Esercizio: *copia default membro-a-membro*

Scrivere la seguente funzione:

`copiaIn(prova sorgente, prova &destinazione)`

Che copia membro a membro dell'oggetto **`sorgente`** nell'oggetto **`destinazione`** in modo che il precedente array di **`destinazione`** sia deallocato e…

Che sia sostituito con un nuovo array delle stesse dimensioni di quello di **`sorgente`** (ma allocato indipendentemente);

Che sia popolato dalle copie degli elementi dell'array di **`sorgente`**

33

Esempio: *copia default membro-a-membro*

```
100 void copiaIn (prova sorgente, prova &destinazione)
101 {
102         int *tmp;
103
104         tmp=destinazione.p;
105         destinazione=sorgente;
106         delete [] tmp;
107
108         destinazione.p=new int[destinazione.size];
109         for(int i=0;i<destinazione.size;i++)
200                 destinazione.p[i]=sorgente.p[i];
201         return;
202 }
```

Naturalmente, la funzione **`copiaIn`** deve essere dichiarata *friend* dalla classe **`prova`**.

34

La classe *template* **`vector`**

35

La classe *template* **`vector`**

La classe **`vector`** come una alternativa evoluta agli array, per moltissime applicazioni

Come la classe **`string`**, **`vector`** non è un tipo di dato nativo, ma è una classe implementata in C++ e disponibile nella dotazione di librerie standard di questo linguaggio.

La classe vector è fornita dalla libreria *Standard Template Library*

36

La classe *template* **`vector`**

Gli oggetti **`vector`** superano alcune limitazioni degli array:

- Conservano la conoscenza della loro dimensione
- Verificano il rispetto delle dimensioni del vettore in ogni accesso.
- Possono essere confrontati tra loro (e.g. con **==** e **!=**)
- E' possibile effettuare assegnamenti tra vettori

37

La classe *template* **`vector`**

La classe **`vector`** utilizza i *template*

I *template* sono una costruzione del C++ che permette di «generalizzare» funzioni e classi *«parametrizzando»* la loro definizione rispetto al tipo di dati che utilizzano.

Nelle funzioni e nelle classi template, il codice è scritto in modo da funzionare nello stesso modo a prescindere dal tipo dato applicato come parametro.

38

La classe *template* **vector**

Dichiariamo un array di interi utilizzando i **vector**

```
vector<int> estrazione(6);
```

Si dichiara un oggetto della classe **vector** di nome **estrazione** composto da sei elementi di tipo **int**

```
vector<punto> ottagono(8);
```

Si dichiara un **vector** composto da otto elementi di classe **punto**

39

La classe *template* **vector**

Nonostante l'aspetto «esotico», si tratta di una normale dichiarazione di variabile…

TIPO: Classe template + <tipo base>

```
vector<punto> ottagono(8);
```

Si dichiara un **vector** composto da otto elementi di classe **punto**

TIPO: Classe template + <tipo base>

40

La classe *template* **vector**

Dichiariamo un array di interi utilizzando i **vector**

```
#include<vector>
using std::vector;
vector<int> estrazione1(6), estrazione2(6);
```

Lunghezza di un oggetto **vector**

```
cout << "Lunghezza: " << estrazione1.size();
```

41

La classe *template* **vector**

Input/ output dati:

```
for(i=0;i<estrazione1.size();i++)
      cin >> estrazione1[i];
```

Assegnamento tra vettori:

```
estrazione2=estrazione1
```

L'assegnamento può essere fatto tra due vettori di lunghezza diversa. In questo caso, la dimensione di quello ricevente viene adattata per contenere i dati necessari.

42

La classe *template* **vector**

Confronto:

```
if (estrazione1==estrazione2)
     cout << "sono uguali! " << endl;
```

Inizializzazione (con un altro vettore):

```
vector<int> estrazioni3(estrazioni1);
```

Distruzione:

```
delete estrazioni2;
```

43

Esercizio: *bussolotto #2*

Scrivere una versione della classe **bussolotto** del tutto analoga alla precedente ma:

Che utilizzi i **vector** invece degli array

44

Esercizio: *bussolotto #2*

```
 1  class bussolotto {
 2          private:
 3                  vector<dado> *p;
 4                  int numdadi;
 5          public:
 6                  bussolotto(int num=2, int nf=6){
 7                          p=new vector<dado>(num,nf);
 8                          numdadi=p->size();
 9                  };
10                  ~bussolotto(){
11                          delete p;
12                  }
13                  void agita(int seed);
 …  …
```

Alloca un vettore di **num** oggetti di tipo **dado**. Per istanziare ciascun oggetto utilizza il costruttore **dado(nf)**

45

Esercizio: *bussolotto #2*

```
 …  …
14                  int lancio(int num=0) {
15                          int nlanci, sum=0;
16                          if (num<1||num>numdadi)
17                                  nlanci=numdadi;
18                          else
19                                  nlanci=num;
20                          for(int i=0;i<nlanci;i++)
21                                  sum+=p->at(i).lancio();
22                  // oppure:  sum+=(*p)[i].lancio();
23                          return sum;
24                  }
    };
```

Il metodo **at()**, è equivalente a **[]** ma è più comodo se applicato a un puntatore

p è un puntatore a un vettore di **dado**. Occorre prima referenziarlo, per poi accedere ai singoli elementi.

46

La classi *container*

La classe **`vector`** rientra in una famiglia di classi dette *container*

Si tratta di classi il cui scopo è contenere *collezioni* di oggetti e una serie di strumenti per effettuare operazioni come: inserimento, cancellazione, ricerca di elementi, ordinamento…

Trattandosi di operazioni che non richiedono che l'uso di pochissimi metodi e operatori comuni a qualunque oggetto/tipo «collezionabile» (*e.g.* costruttori/distruttori, assegnamento, confronto) sono generalmente implementate come *template*

47

La classi *container*

La libreria STL fornisce molte utilissime classi *container* divise in categorie a seconda del modo in cui organizzano i dati

Contenitori sequenziali: tra cui **`vector`**, **`array`**, **`list`**;

Contenitori associativi: tra cui **`map`** e **`set`**;

48