Programmazione 2
e Laboratorio di Programmazione

Corso di Laurea in

Informatica

Università degli Studi di Napoli "Parthenope"

Anno Accademico 2023-2024

Prof. Luigi Catuogno

1

Informazioni sul corso

Docente	Luigi Catuogno **luigi.catuogno@uniparthenope.it**
Orario	Lun: 9:00-11:00 Mer: 11:00-13:00
Sede	Centro Direzionale Napoli **Aula Magna**
Ricevimento	Mer: 14:00-16:00 (previo appuntamento) Ufficio docente oppure Team: **cxxa3bo**

2

Libri di testo

Introduzione al linguaggio – costrutti e tecniche di base

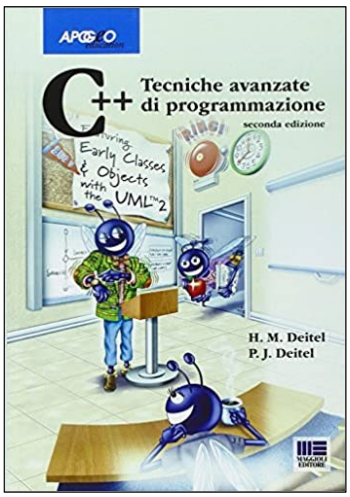
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

3

Libri di testo

Tecniche avanzate e strutture dati elementari

[TAP] H. M. Deitel, P. J. Deitel
C++ Tecniche avanzate di programmazione
II ed. (2011) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8572-6



4

Risorse on-line

Team del corso

Programmazione 2 AA 2023-24 - Prof. Catuogno
Comunicazioni, incontri e avvisi per il corso
Codice: **ftomzjx**

Piattaforma e-learning

Programmazione II e Laboratorio di Programmazione II - A.A. 2023-24
Materiale didattico, manualistica, esercitazioni.
URL: https://elearning.uniparthenope.it/course/view.php?id=2386

5

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

6

Esempio: *ancora sugli angoli*

file: angolo.hpp

```
1  #ifndef _ANGOLO_HPP_
2  #define _ANGOLO_HPP_
3
4  class Angolo {
5  private:
6          int gradi;
7          int primi;
8          int secondi;
9
10 public:
11         Angolo():       gradi(0), primi(0), secondi(0) {}
12
13 Angolo(int ,int, int);
14
15
```

Header file dedicato alla definizione della nostra nuova classe **angolo**. Non contiene la definizione dei metodi ma soltanto la dichiarazione dei loro prototipi.

7

Esempio: *ancora sugli angoli*

file: angolo.hpp

```
1  #ifndef _ANGOLO_HPP_
2  #define _ANGOLO_HPP_
3
4  class Angolo {
5  private:
6          int gradi;
7          int primi;
8          int secondi;
9
10 public:
11         Angolo():       gradi(0), primi(0), secondi(0) {}
12
13 Angolo(int ,int, int);
14
15
```

Queste sono direttive del *preprocessore* dell'ambiente C/C++. Non fanno parte del linguaggio e sono utilizzate per «personalizzare» alcuni aspetti della compilazione. Per esempio: per definire *MACRO* e selezionare il codice da/da non compilare;

8

Il preprocessore del C/C++

Il preprocessore è un interprete di comandi che ha lo scopo di «preparare» il codice C/C++ per la compilazione.

Inclusione dei file header

Gestione delle MACRO: vere e proprie *abbreviazioni* definite dall'utente per *personalizzare* la stesura del codice

Compilazione *condizionata*: possibilità di decidere, a seconda delle circostanze, quali parti del codice compilare e quali no.

9

Il preprocessore del C/C++

I comandi del preprocessore sono generalmente detti *direttive,* sono posti sempre all'inizio sempre e iniziano col carattere **# .**

Nell'ambiente di sviluppo C/C++ il preprocessore elabora i file sorgenti *prima* del compilatore vero e proprio:

Modifica il file sorgente attuando le direttive in esso contenute

Quando ha finito, trasmette al compilatore il file sorgente così modificato.

10

Alcune direttive del preprocessore

#include <*headerfile*>
#include "*localhfile.hpp*"

Aggiunge il contenuto del file indicato nel punto in cui compare la direttiva. Nel primo caso, il file tra **<>** si assume sia contenuto nelle directory standard del compilatore;

nel secondo caso, il nome del file (riportato per intero tra **"**) si riferisce a un file contenuto nella directory in cui avviene la compilazione

E' possibile modificare i path di ricerca degli header file mediante opportune opzioni del compilatore.

11

Alcune direttive del preprocessore

#define NOMEMACRO *VALOREMACRO*
#define MACRO2
#undef NOMEMACRO

La prima direttiva definisce la corrispondenza tra la stringa **NOMEMACRO** e il valore ad esso assegnato. Nel testo del file sorgente, il preprocessore sostituirà *ogni occorrenza* di **NOMEMACRO** con **VALOREMACRO** *;*

La seconda direttiva assegna alla macro **MACRO2** una *stringa vuota.* La macro risulta comunque definita, ma la sua sostituzione non produce risultati;

L'effetto della direttiva **#undef** è quello di rimuovere qualsiasi corrispondenza alla macro **NOMEMACRO**. Di qui in poi, il precompilatore copierà eventuali occorrenze **NOMEMACRO** nel file sorgente, senza effettuare la sostituzione, come se fosse *«testo normale».*

12

Alcune direttive del preprocessore

```
#ifdef NOMEMACRO
      …porzione di codice…
#endif
```

Se la macro **NOMEMACRO** risulta definita (nel file corrente o in uno di quelli inclusi sinora) il preprocessore trasmette al compilatore la porzione di codice riportata fino alla direttiva **#endif**

Nel caso in cui la macro non risulti essere definita (o se sia stata rimossa con una direttiva

#undef allora il compilatore non riceverà la porzione di codice compresa.

13

Alcune direttive del preprocessore

```
#ifndef NOMEMACRO
      …porzione di codice…
#endif
```

Funziona come **#ifdef** ma in questo caso, il codice riportato fino alla **#endif** sarà trasmesso al compilatore solo nel caso in cui **NOMEMACRO** non dovesse risultare definita

14

Alcune direttive del preprocessore

```
#ifdef NOMEMACRO
     …porzione di codice da includere se vero…
#else
     …codice da includere altrimenti…
#endif
```

15

Alcune direttive del preprocessore

```
#ifndef NOMEMACRO
#define NOMEMACRO
     …porzione di codice…
#endif
```

Questa sequenza di direttive ha lo scopo di impedire che la porzione di codice compresa sia inclusa più volte nei sorgenti da passare al compilatore.

16

Esempio: *ancora sugli angoli*

file: angolo.hpp

```
 1  #ifndef _ANGOLO_HPP_
 2  #define _ANGOLO_HPP_
 3
 4  class Angolo {
 5  private:
 6          int gradi;
 7          int primi;
 8          int secondi;
 9
10  public:
11          Angolo(): gradi(0), primi(0), secondi(0) {}
12
13          Angolo(int ,int, int);
14
15
```

Costruttore di default (inizializza tutti i campi a zero)

Prototipo di un costruttore secondario

17

Esempio: *ancora sugli angoli*

file: angolo.hpp

```
16          int getG();
17          int getP();
18          int getS();
19
20          void setG(int);
21          void setP(int);
22          void setS(int);
23
24          void set(int,int,int);
25
26          Angolo somma(Angolo);
27  };
28
29  #endif
30
```

Prototipi degli altri metodi della classe.

18

Esempio: *ancora sugli angoli*

File in cui sono contenuti i metodi della classe angolo

file: angolo.cpp

```
1  #include "angolo.hpp"
2
3  Angolo::Angolo(int g, int p, int s)
4  {
5          set(g,p,s);
6  }
7
8  int Angolo::getG()
9  {
10         return gradi; // getP() e getS() sono analoghi…
11 }
12 …
```

Includo il file che contiene la definizione della classe

Definisco un metodo appartenente alla classe Angolo *all'esterno* della definizione della classe.

19

Esempio: *ancora sugli angoli*

file: angolo.cpp

```
28 void Angolo::setG(int g)
29 {
30         g=(g>0?g:-g);
31         gradi=g%360; // setP() e setS() sono analoghi (ma %60)…
32 }
…  …
44 void Angolo::set(int g, int p, int s)
45 {
46         setG(g);
47         setP(p);
48         setS(s);
49 }
```

20

Esempio: *ancora sugli angoli*

file: angolo.cpp

```
51  Angolo Angolo::somma(Angolo b)
52  {
53          Angolo c;
54          int g,p,s;
55          g=gradi+b.gradi;
56          p=primi+b.primi;
57          s=secondi+b.secondi;
58          p+=s/60;
59          s=s%60;
60          g+=p/60;
61          p=p%60;
62          g=g%360;
63          c.set(g,p,s);
64          return c;
65  }
```

21

Esempio: *ancora sugli angoli*

file: angolo_main.cpp

```
1   #include<iostream>
2   #include"angolo.hpp"
3
4   using namespace std;
5
6   int main()
7   {
8           int g,p,s;
9           Angolo a,b,c;
10
11          cout<<"Inserisci l'angolo A: ";
12          cin >> g >> p >> s;
13          a.set(g,p,s);
14          cout<<"Inserisci l'angolo B: ";
15          cin >> g >> p >> s;
16          b.set(g,p,s);
```

22

Esempio: *ancora sugli angoli*

file: angolo_main.cpp

```
17          c=a.somma(b);
18
19          cout <<"Angolo C=A+B: ";
20          cout << c.getG() << " gradi, " << c.getP();
21          cout << " primi e " << c.getS()<< " secondi." <<endl;
22  }
23
```

```
$ g++ angolo.cpp main.cpp -o angolo
$ ./angolo
```

23

Argomenti di *default*

24

Argomenti di *default*

In C++, le funzioni e i metodi delle classi possono avere argomenti *di default*

- Sono dichiarati nel prototipo (o nell'intestazione) con l'indicazione del valore *«preimpostato»*
- Se omessi nella chiamata, assumono (nella funzione) automaticamente il valore indicato

25

Argomenti di *default*

In C++, le funzioni e i metodi delle classi possono avere argomenti *di default*

```
int foo(int x, int y, int c=15) {
        …
        cout << "c=" << c << endl;
        …
}
```

```
        …
        retval1=foo(3,4);
        retval2=foo(11,45,22);
        …
```

26

Argomenti di *default*

In C++, le funzioni e i metodi delle classi possono avere argomenti *di default*

```
int foo(int x, int y, int c=15) {
        …
        cout << "c=" << c << endl;
        …
}
```

```
        …
        retval1=foo(3,4);
        retval2=foo(11,45,22);
        …
```

```
c=15
c=22
```

27

Argomenti di *default*

Alcune regole:

- Possono esserci più argomenti di default
- Gli argomenti di default devono trovarsi agli ultimi posti della lista dei parametri
- Se se ne omette uno che non è l'ultimo, dovranno essere necessariamente omessi tutti i successivi

28

Argomenti di *default*

Gli argomenti di default devono trovarsi agli ultimi posti della lista dei parametri

SI

```
int foo(int a, int b = 10, int c = 15, int d = 20) {
        …
        }
```

NO

```
int bar(int uno, int bis = 10, int ter) {
        …
        }
```

29

Argomenti di *default*

Se se ne omette uno che non è l'ultimo, dovranno essere necessariamente omessi tutti i successivi

```
int zoo(int a, float b = 0.1, string s = "hello") {
        … }
```

SI

```
…      x=zoo(34,11.0) // omesso s;
```

NO

```
…      x=zoo(34, "ciao")    // omesso b, ma non s;
```

Gli argomenti di default sono identificati dalla loro posizione nella lista dei parametri. Qui, la stringa "ciao" viene identificata erroneamente con l'argomento b…

30

Esempio: *serviamo il numero…*

```
 5  class distributoreNumeri {
 6  private:
 7          int num;
 8  public:
 9          distributoreNumeri() {
10                  num=1;
11          };
12          void reset(int nr=1) {
13                  num=nr;
14          };
15          int prossimo() {
16                  return num;
17          }
18          int servizio() {
19                  return num++;
20          }
21  };
```

Il metodo **reset** prende un solo parametro per il quale è indicato un valore di *default.*

Il metodo può essere invocato indicando esplicitamente il valore del parametro oppure, omettendolo. In questo caso, il valore passato alla funzione è quello di default.

31

Metodi: argomenti di *default*

Alcune regole:

Per i metodi delle classi, valgono le stesse regole.

Anche un costruttore può avere argomenti di default

Se un costruttore possiede *solo* argomenti di default, allora diventa il *costruttore di default* per la classe

32

Esempio: *costruttore di default*

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

Costruttore di default per la classe **prova**

La dichiarazione di **P** e **Q** è effettuata utilizzando lo stesso costruttore.

```
int main()
{
        prova P, Q(15);
        cout << "P.len="<<P.len()<<endl;
        cout << "Q.len="<<Q.len()<<endl;
}
```

33

I distruttori

34

Il *distruttore*

Il *distruttore* è un metodo speciale (pubblico) di una classe.

Un distruttore è il metodo *duale* del costruttore, in quanto viene invocato *implicitamente* quando un oggetto della sua classe è distrutto.

Questo avviene in diverse situazioni, incluse la terminazione dell'ambito di visibilità dell'oggetto e alcune modalità di terminazione dell'intero programma.

35

I *distruttori*

Il distruttore della classe **`myClass`** ha lo stesso nome della classe, preceduto da **`~`** (quindi **`~myClass()`**)

Ogni classe può avere un solo distruttore

Non prende argomenti, non restituisce valori (neppure **`void`**) e non può essere ridefinito.

Può contenere il codice che deve essere eseguito contestualmente alla distruzione di un oggetto

36

Esempio: *costruttori & distruttori…*

```
 5  class Punto {
 6  public:
 7          double x;
 8          double y;
 9          Punto() {
10                  x=y=0;
11                  cout << "Creo punto ("<<x<<","<<y<<")"<<endl;
12          };
13          Punto(double c1,double c2) {
14                  x=c1;
15                  y=c2;
16                  cout << "Creo punto ("<<x<<","<<y<<")"<<endl;
17          };
18          ~Punto(){
19                  cout << "Distruggo punto ("<<x<<","<<y<<")"<<endl;
20          }
21  };
```

37

Esempio: *costruttori & distruttori…*

```
22  double distanza (Punto p1, Punto p2) {
23          cout << "Funzione distanza"<<endl;
24          Punto p3(9,9);
25          return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
26  }
27  int main()
28  {
29          Punto p1(3,4), p4(11,22), *p2;
30          cout << "Allocazione p2" << endl;
31          p2= new Punto();
32          cout << "Distanza p1-p2= " << distanza(p1,*p2) << endl;
33          cout << "Distruzione p2" <<endl;
34          delete p2;
35          cout << "Distanza p1-p4= " << distanza(p1,p4) << endl;
36          cout <<"Fine"<<endl;
37  }
```

38

Esempio: *costruttori & distruttori…*

```
22  double distanza (Punto p1, Punto p2) {
23          cout << "Funzione distanza"<<endl;
24          Punto p3(9,9);
25          return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
26  }
27  int main()
28  {
29          Punto p1(3,4), p4(11,22), *p2;
30          cout << "Allocazione p2" << endl;
31          p2= new Punto();
32          cout << "Distanza p1-p2= " << distanza(p1,*p2) << endl;
33          cout << "Distruzione p2" <<endl;
34          dele  2;
35          co    "Distanza p1-p4= " << distanza
36                Fine"<<endl;
37  }
```

La dichiarazione dei punti **p1** e **p4** è effettuata con il secondo costruttore. La dichiarazione del puntatore *p2 non ha effetto…

…fino a che l'operatore **new** non invoca implicitamente il costruttore di default (con **x=y=0**)

```
Creo punto (3,4)
Creo punto (11,22)
Allocazione p2
Creo punto (0,0)
…
```

39

Esempio: *costruttori & distr*

La dichiarazione di p3, locale alla funzione distanza, produce una chiamata al costruttore. Quando la funzione termina, le variabili locali **p1**, **p2** e **p3** sono distrutte (da distruttore)

```
22  double distanza (Punto p1, Punto p2)
23          cout << "Funzione di    za"<<endl;
24          Punto p3(9,9);
25          return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
26  }
27  int main()
28  {
29          Punto p1(3,4), p4(11,22), *p2;
30          cout << "Allocazione p2" << endl;
31          p2= new Punto();
32          cout << "Distanza p1-p2= " << distanza(p1,*p2) << endl;
33          cout << "Distruzione p2" <<endl;
34          delete p2;
35          cout << "Distanza p1-p4= " << distanza(p1,
36          cout <<"Fine"<<endl;
37  }
```

```
Funzione distanza
Creo punto (9,9)
Distruggo punto (9,9)
Distanza p1-p2= 5
Distruggo punto (3,4)
Distruggo punto (0,0)
```

40

Esempio: *costruttori & distruttori…*

```
22  double distanza (Punto p1, Punto p2) {
23          cout << "Funzione distanza"<<endl;
24          Punto p3(9,9);
25          return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
26  }
27  int main()
28  {
29          Punto p1(3,4), p4(11,22), *p2;
30          cout << "Allocazione p2" << endl;
31          p2= new Punto();
32          cout << "Distanza p1-p2= " << distanza(p1,*p2) << endl;
33          cout << "Distruzione p2" <<endl;
34          delete p2;
35          cout << "Distanza p1-p4= " << distanza(p1,p4) << endl;
36          cout <<"Fine"<<endl;
37  }
```

```
Distruzione p2
Distruggo punto (0,0)
```

La chiamata all'operatore **delete** sul puntatore **p2** produce una chiamata al distruttore di default

Esempio: *costruttori & distr*

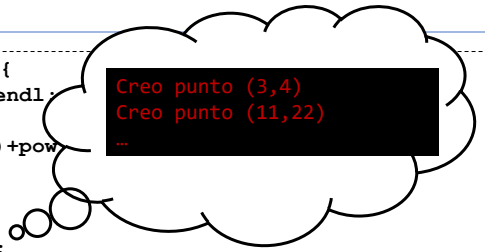
La nuova invocazione della funzione produce la creazione di nuove istanze delle variabili locali. Al termine, le variabili locali sono nuovamente distrutte.

```
22  double distanza (Punto p1, Punto p2)
23          cout << "Funzione di      nza"<<endl;
24          Punto p3(9,9);
25          return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
26  }
27  int main()
28  {
29          Punto p1(3,4), p4(11,22), *p2;
30          cout << "Allocazione p2" << endl;
31          p2= new Punto();
32          cout << "Distanza p1-p2= " << distanza(p
33          cout << "Distruzione p2" <<endl;
34          delete p2;
35          cout << "Distanza p1-p4= " << distanza(p1,p4) << endl;
36          cout <<"Fine"<<endl;
37  }
```

```
Funzione distanza
Creo punto (9,9)
Distruggo punto (9,9)
Distanza p1-p4= 19.6977
Distruggo punto (3,4)
Distruggo punto (11,22)
```

Esempio: *costruttori & distruttori…*

```
22  double distanza (Punto p1, Punto p2) {
23          cout << "Funzione distanza"<<endl;
24          Punto p3(9,9);
25          return sqrt(pow((p2.x-p1.x),2)+pow
26  }
27  int main()
28  {
29          Punto p1(3,4), p4(11,22), *p2;
                                    ione p2" << endl;
                                a p1-p2= " << distanza(
                                    ione p2" <<endl;
35          cout <<                 stanza p1-p4= " << distanza(p1,p4) << endl;
36          cout <<"Fine"<<endl;
37  }
```



Al termine del programma, le variabili locali alla funzione **main**: **p1** e **p4** sono distrutte nell'ordine inverso col quale erano state create.

```
Fine
Distruggo punto (11,22)
Distruggo punto (3,4)
```

43

Il *distruttore*

Può contenere il codice che deve essere eseguito contestualmente alla distruzione di un oggetto

Questo è utile quando l'oggetto contiene dati e altri oggetti allocati dinamicamente a *run time*.

In questo caso, la distruzione dell'oggetto dovrebbe essere preceduta dalla liberazione della memoria che ha allocato dinamicamente. Il distruttore è il posto adatto per il codice con questo scopo.

44

Esempio: *costruttori & distruttori #2*

Facciamo un esperimento…

Utilizziamo la classe **`prova`** vista in precedenza.

La classe utilizza un array di interi allocato dinamicamente dal costruttore

Analizziamo l'uso del distruttore in questo caso.

45

Esempio: *costruttori & distruttori #2*

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

Aggiungiamo *(solo per gli scopi dell'esperimento)* il metodo **`pointer()`** che permetterà di ispezionare l'array allocato dalla classe indipendentemente dall'interfaccia della classe.

46

Esempio: *costruttori & distruttori #2*

```
 5  int main()
 6  {
 7          prova P, *R;
 8          int *p;
 9          cout << "P.len="<<P.len()<<endl;
10          R=new prova(20);
11          cout << "R->len="<<R->len()<<endl;
12          if(!R->set(0,777))
13                  cout <<"Errore!"<<endl;
14
15          p=R->pointer();
16          cout << "p="<<p[0]<<endl;
17          delete R;
18          cout << "p="<<p[0]<<endl;
19          delete p;
20  }
```

Questo codice ha lo scopo di verificare cosa accade nell'array dopo che la l'oggetto che lo contiene è distrutto

47

Esempio: *costruttori & distruttori #2*

```
 5  int main()
 6  {
 7          prova P, *R;
 8          int *p;
 9          cout << "P.len="<<P.len()<<endl;
10          R=new prova(20);
11          cout << "R->len="<<R->len()<<endl;
12          if(!R->set(0,777))
13                  cout <<"Errore!"<<endl;
14
15          p=R->pointer();
16          cout << "p="<<p[0]<<endl;
17          delete R;
18          cout << "p="<<p[0]<<endl;
19  }
```

```
P.len=10
R->len=20
p=777
p=777
```

Creiamo dinamicamente un oggetto di classe **prova** inseriamo un dato nell'array tramite l'interfaccia della classe. Visualizziamo il dato inserito tramite il puntatore all'array

48

Esempio: *costruttori & distruttori #2*

```
 5  int main()
 6  {
 7          prova P, *R;
 8          int *p;
 9          cout << "P.len="<<P.len()<<endl;
10          R=new prova(20);
11          cout << "R->len="<<R->len()<<endl;
12          if(!R->set(0,777))
13                  cout <<"Errore!"<<endl;
14
15          p=R->pointer();
16          cout << "p="<<p[0]<<endl;
17          delete R;
18          cout << "p="<<p[0]<<endl;
19  }
```

```
P.len=10
R->len=20
p=777
p=777
```

Osserviamo che nonostante l'oggetto sia stato distrutto (col distruttore di default), l'array è ancora là…

Esempio: *costruttori & distruttori #3*

```
 5  class prova {
 6          int *p;
 7          int size;
 8  public:
 9          prova(int num = 10) {
10                  size=num;
11                  p=new int[size];
12          }
13          ~prova() {
14                  delete [] p;
15          }
16          bool set(int pos, int data);
17          bool get(int pos, int &data);
18          int len();
19          int *pointer() {
20                  return p;
21          }
22  };
```

Dichiariamo esplicitamente il distruttore della classe in cui inseriamo il codice per deallocare l'array.

Esempio: *costruttori & distruttori #3*

```
 5  int main()
 6  {
 7          prova P, *R;
 8          int *p;
 9          cout << "P.len="<<P.len()<<endl;
10          R=new prova(20);
11          cout << "R->len="<<R->len()<<endl;
12          if(!R->set(0,777))
13                  cout <<"Errore!"<<endl;
14
15          p=R->pointer();
16          cout << "p="<<p[0]<<endl;
17          delete R;
18          cout << "p="<<p[0]<<endl;
19  }
```

```
P.len=10
p=777
p=8520016
```

Ora, dopo l'esecuzione del distruttore, l'area a cui puntava p, non contiene più l'array (che non esiste più)

51

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.

52

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**

53

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  };
```

54

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**.

55

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.

56

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.

57

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)…

58

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…

59

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»*

60

Esercizio: *bussolotto*

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

```
 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                  }
 …          …
```

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

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

61

Esercizio: *bussolotto*

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

```
 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                  }
```

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

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

62

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  }
```

63

Assegnamento tra oggetti

64

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*)

65

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
```

66

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…

67

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

68

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.

69

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  }
```

70

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
47
48
49
50          Q.set(0,1997);
51          cout <<"Dopo Q.set(0,1997), P.show()-> "     ;
52          P.show();
53  }
```

```
Id:6, len=10, pointer=0x7d1a60, p[0]=666
Id:7, len=50, pointer=0x7d5b50, p[0]=777
...
```

71

Esempio: *copia default membro-a-membro*

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

```
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  }
```

72

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          P=Q;
45          cout <<"Dopo l'assegn. P=Q, P.show()-> "     ;
46          P.show();
47          cout << "(!!) pp="<<pp<<", pp[0]="<<pp[0]<<endl;
48          Q.set(0,1997);
49          cout <<"Dopo Q.set(0,1997), P.show()-> "     ;
50          P.show();
51  }
52
53
```

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

73

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]="<<pp[0]<<endl;
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**!

74

Esempio: *copia default membro-a-membro*

Che cosa è successo?

75

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)

76

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

77

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`**

78

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`**.

79