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

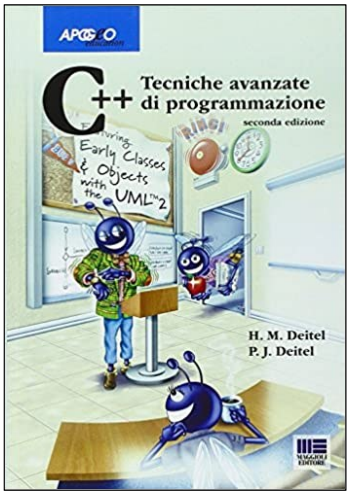
2

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: *tic-tac-toe #1*

Scrivere una classe **`tttPlayGround`** che abbia, tra gli altri, i seguenti attributi privati:

```
int playground[3][3];
int prossimoG;
```

7

Esercizio: *tic-tac-toe #1*

La classe **`tttPlayGround`** presenta la seguente interfaccia:

```
tttPlayGround()
```

il costruttore di default. Azzera il contenuto di **`playground`** e imposta **`prossimoG=1`**;

```
void show()
```

visualizza il contenuto di **`playground`** in modo che, le celle contenenti 0 appaiano vuote, quelle a 1 riportino il carattere **`O`** e quelle con 2 il carattere **`X`**

8

Esercizio: *tic-tac-toe #1*

`char prossimo()`

restituisce il il carattere **`O`** se **`prossimoG`** vale 1, oppure **`X`** se **`prossimoG`** vale 2;

`bool muovi(int riga, int col)`

assegna il valore corrente di **`prossimoG`** alla cella indicata da **`riga`** e **`col`** . Quindi aggiorna il valore di **`prossimoG`** per dare il turno all'altro giocatore (se è 1 passa a 2, se è 2 passa a 1). restituisce false se non è possibile fare la mossa (true se è ok).

`void reset()`

9

Esercizio: *tic-tac-toe #1*

```
 5  class tttPlayGround {
 6  private:
 7                  int playground[3][3];
 8                  int prossimoG;

 9                  int vincitore;
10                  int counter;
11                  const char icons[3]={' ','O','X'};
12
13                  bool check();
```

10

Esercizio: *tic-tac-toe #1*

```
14  public:
15                  tttPlayGround()
16                  {
17                          reset();
18                  }
19                  void reset()
20                  {
21                          for(int i=0;i<3;i++)
22                                  for(int j=0;j<3;j++)
23                                          playground[i][j]=0;
24                          prossimoG=1;
25                          vincitore=0;
26                          counter=0;
27                  };
28
```

11

Esercizio: *tic-tac-toe #1*

```
29                  bool finita()
30                  {
31                          return (check() || counter==9);
32                  };
33
34                  char prossimo()
35                  {
36                                  return icons[prossimoG];
37                  };
38                  char vince()
39                  {
40                          return icons[vincitore];
41                  }
42                  bool muovi(int i,int j);
43                  void show();
44  };
45
```

12

Esercizio: *tic-tac-toe #1*

```
46  bool tttPlayGround::check()
47  {
48          int pr,pc,pd1=0,pd2=0;
49          int v=0;
50
51          for (int i=0;i<3;i++){
52                  pr=pc=0;
53                  for (int j=0;j<3;j++){
54                          if (playground[i][j]==1)
55                                  pr+=1;
56                          if (playground[i][j]==2)
57                                  pr+=10;
58                          if (playground[j][i]==1)
59                                  pc+=1;
60                          if (playground[j][i]==2)
61                                  pc+=10;
62                  } // fine for j
```

13

Esercizio: *tic-tac-toe #1*

```
64                  if (playground[i][i]==1)
65                          pd1+=1;
66                  if (playground[i][2-i]==1)
67                          pd2+=1;
68                  if (playground[i][i]==2)
69                          pd1+=10;
70                  if (playground[i][2-i]==2)
71                          pd2+=10;
72
73                  if ((pr==3)||(pc==3)||(pd1==3)||(pd2==3))
74                          v=1;
75                  if ((pr==30)||(pc==30)||(pd1==30)||(pd2==30))
76                          v=2;
```

14

Esercizio: *tic-tac-toe #1*

```
78
79                  if (v>0)
80                          break;
81          } // fine for i
82
83          vincitore=v;
84          return v!=0; // restituisce true se la partita è stata vinta.
85  }
```

15

Esercizio: *tic-tac-toe #1*

```
86  void tttPlayGround::show()
87  {
88          int iconNumber=0;
89          cout << "+---+"<<endl;
90          for (int i=0;i<3;i++){
91                  cout<<"|";
92                  for(int j=0;j<3;j++){
93                          iconNumber=playground[i][j];
94                          cout <<icons[iconNumber];
95                  }
96                  cout <<"|"<<endl;
97          }
98          cout << "+---+"<<endl;
99  }
```

16

Esercizio: *tic-tac-toe #1*

```
101 bool tttPlayGround::muovi(int i, int j)
102 {
103         if (playground[i][j]!=0||counter==9)
104                 return false;
105         playground[i][j]=prossimoG;
106         counter++;
107         prossimoG=3-prossimoG;
108         check();
109         return true;
110 }
```

17

Esercizio: *tic-tac-toe #1*

```
112 int main()
113 {
114         tttPlayGround match;
115         int riga,col;
116         bool mossaOk;
117         do {
118                 match.show();
119                 cout << "Giocatore " << match.prossimo() <<": ";
120                 cin >> riga >> col;
121                 mossaOk=match.muovi(riga,col);
122                 if (!mossaOk)
123                         cout << "Mossa sbagliata!"<<endl;
124         } while(!match.finita());
125         match.show();
126         cout <<"Vince:"<<match.vince()<<endl;
127 }
```

18

Argomenti di *default*

19

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

20

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);
        …
```

21

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
```

22

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

23

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) {
        …
        }
```

24

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…

25

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.

26

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

27

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;
}
```

28

I distruttori

29

I *distruttori*

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.

30

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

31

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  };
```

32

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  }
```


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)
…
```

Esempio: *costruttori & distr*

```
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  }
```

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)

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

35

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 p...
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

36

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...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(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;
            ...one p2" << endl;
            ... p1-p2= " << distanza(
            ...one p2" <<endl;
35          cout <<...stanza p1-p4= " << distanza(p1,p4) << endl;
36          cout <<"Fine"<<endl;
37  }
```

```
Creo punto (3,4)
Creo punto (11,22)
…
```

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)
```

I *distruttori*

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 *runtime*.

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.

39

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.

40

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.

41

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

42

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

43

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…

44

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)