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

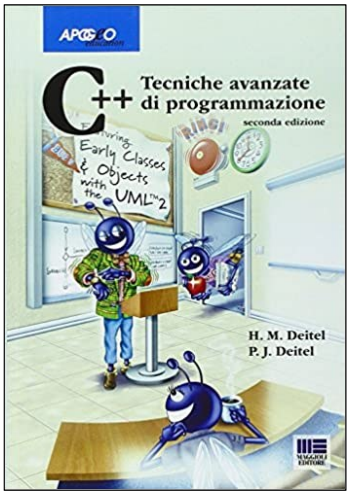
2

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

Il qualificatore **const**

7

Il qualificatore **const**

Nella dichiarazione di variabili, il qualificatore **const** informa il compilatore che il valore della variabile cui è applicato non può essere modificato *(read-only)*

```
const double g=9.81;
```

Qualsiasi espressione di assegnamento che veda **g** *a sinistra dell'uguale* produrrà un errore in fase di compilazione.

```
g=9.80665; // NO!
```

8

Il qualificatore **const**

Le variabili dichiarate **const** non possono essere modificato neppure tramite un puntatore o un riferimento :

```
const double g=9.81;
double *gPtr;
```

Le seguenti espressioni producono un errore durante la compilazione:

```
gPtr=&g;
double &gRef=g;
```

9

Il qualificatore **const**

Tuttavia, mediante l'uso combinato di **const** con di riferimenti e puntatori, è possibile imporre un vero e proprio *controllo di accesso* alle variabili.

10

Il qualificatore **`const`**

Costanti e non…

```
const double g=9.81;
double h=2.37, i=0.047;
```

Le variabili **`const`** sono variabili *read-only (RO).*

Queste variabili, invece, sono variabili *read/write (R/W).*

11

Il qualificatore **`const`**

Costanti e non…

```
const double g=9.81;
double h=2.37, i=0.047;
```

Le variabili **`const`** sono variabili *read-only (RO).*

Queste variabili, invece, sono variabili *read/write (R/W).*

12

Il qualificatore **const**

Costanti e non…

```
const double g=9.81;
double h=2.37, i=0.047;
```

Le variabili *r/w* ammettono anche riferimenti *a costanti*

```
const double &hRef=h;
```

Qui, è possibile assegnare un nuovo valore a **h**, ma non è possibile farlo tramite **hRef**

13

Il qualificatore **const**

La variabile **h** è R/W

Le variabili *r/w* ammettono anche riferimenti *a costanti:*

```
double h=2.37;
const double &hRef=h;

h=4.74;
cout << "hRef=" << hRef << endl;
```

Il riferimento **hRef** ad **h** è *a costanti.*

Qui, è possibile assegnare un nuovo valore a **h**, ma non è possibile farlo tramite **hRef**. La seguente riga produce un errore.

```
hRef=7.11; // NO!
```
hRef è un'accesso a sola lettura per **h**

14

Il qualificatore **const**

La variabile **h** è R/W

Le variabili *r/w* ammettono anche riferimenti *a costanti:*

```
double h=2.37;
const double &hRef=h;

h=4.74;
cout << "hRef=" << hRef << e
```

Il riferimento **hRef** ad **h** è *a costanti.*

*«come se **hRef** pensasse di riferirsi a una variabile **const** e quindi non lasciasse passare gli assegnamenti»*

Qui, è possibile assegnare un nuov o
tramite **hRef**. La seguente riga produ

```
hRef=7.11; // NO!  hRef è un'accesso a sola lettura per h
```

15

Esempio: *riferimenti a costanti*

```
1  #include<iostream>
2  using namespace std;
3
4  void client (const int &ro_data){
5          cout <<"Dati read-only: "<<ro_data<<endl;
6  }
7
8  int main(){
9          int rw_data=20;
10         client(rw_data);
11         rw_data=40;
12         cout <<"Dati read/write: "<<rw_data<<endl;
13 }
```

Nell'ambito della funzione **client()**, la variabile della **main()** è esposta mediante un riferimento **const** che non permette modifiche.

Nell'ambito **main()** la variabile **rw_data** è r/w.

Questo «protegge» i dati della funzione chiamante da eventuali effetti collaterali indesiderati/imprevisti del codice delle funzioni che vi accedono

16

Il qualificatore **const**

Puntatore a variabile **const**:

```
const double g=9.81;
const double *gPtr=&g;
```

Qui, non è possibile modificare **g**, neppure attraverso **gPtr**, poichè questo è *definito per essere* un puntatore a una variabile *read-only*.

Questa caratteristica è propria di un puntatore così definito, infatti…

17

Il qualificatore **const**

Puntatore *a costanti* per una variabile R/W:

```
double h=10;                // variabile RW
const double *hPtr;         // puntatore a costanti
…
hPtr=&h;
```

Sebbene **h** sia una variabile r/w, questa non può essere modificata tramite **hPtr**. La seguente riga produce un errore:

```
*hPtr=101;
```

18

Esempio: *puntatori a costanti*

```
 1  void showBuffer(const int *ptr2prot, const int &prot_size)
 2  {
 3          for(int i=0;i<prot_size;i++)
 4                  cout << "Elemento "<<i<<" = "<< *ptr2prot++ <<endl;
 5  //      *ptr2prot=10; // NO!
 6  }
 7
 8  int main()
 9  {
10          int rw_array_size=5, rw_array[5]={0,1,2,3,4};
11          cout << "dati RW, puntatore a CONST:"<<endl;
12          cout << "Buffer =" << endl;
13          showBuffer(rw_array,rw_array_size);
14  }
```

19

Esempio: *puntatori a costanti*

```
 1  void showBuffer(const int *ptr2prot, const int &prot_size)
 2  {
 3          for(int i=0;i<prot_size;i++)
 4                  cout << "Elemento "<<i<<" = "<< *ptr2prot++ <<endl;
 5  //      *ptr2prot=10; // NO!
 6  }
 7
 8  int main()
 9  {
10          int rw_array_size=5, rw_array[5]={0,1,2,3,4};
11          cout << "dati RW, puntatore a CONST:"<<endl;
12          cout << "Buffer =" << endl;
13          showBuffer(rw_array,rw_array_size);
14  }
```

Nell'ambito della funzione **showBuffer()**, non è consentito fare assegnamenti nell'area di memoria di **main()** puntata da **ptr2prot**.

Nell'ambito **main() rw_array** è r/w.

Questo «protegge» i dati della funzione chiamante da eventuali effetti collaterali indesiderati/imprevisti del codice delle funzioni che vi accedono

20

Oggetti e funzioni membro costanti

21

Esempio: *la classe punto (ancora…)*

```
 5  class Punto {
 6  private:
 7          double x;
 8          double y;
 9  public:
10          Punto(double c1, double c2) {
11                  set(c1,c2);
12          };
13          void set(double c1,double c2) {
14                  x=c1;
15                  y=c2;
16          };
17          double getX() {return x;}
18          double getY() {return y;}
19  };
```

22

Dichiarare oggetti **const**

Dichiariamo dei *punti*

```
Punto arrivo(42,13);
const Punto origine(0,0);
```

Due oggetti della stessa classe, danno luogo a due comportamenti diversi…

23

Dichiarare oggetti **const**

Dichiariamo dei *punti*

```
Punto arrivo(42,13);
const Punto origine(0,0);
```

del primo sappiamo tutto:

```
cout <<   "arrivoX=" << arrivo.getX() << endl;
cout << ", arrivoY=" << arrivo.getY() << endl;
arrivo.set(24,31);
```

24

Dichiarare oggetti **const**

Dichiariamo dei *punti*

```
Punto arrivo(42,13);
const Punto origine(0,0);
```

ma il secondo, ci riserva qualche sorpresa…

25

Dichiarare oggetti **const**

Dichiariamo dei *punti*

```
const Punto origine(0,0);
```

Così come è definita, la classe punto non permette l'invocazione di alcun metodo sugli oggetti **const**. Neppure di quelli che non apportano modifiche ai membri della classe.

La seguente riga produce un errore in fase di compilazione

```
cout <<   «origineX=" << origine.getX() << endl;
```

26

Dichiarare oggetti **`const`**

La regola generale è che di un oggetto **`const`**, possono essere invocati solo i metodi dichiarati **`const`**

Dichiarazione del prototipo di un metodo:

```
double getX() const;
```

Definizione:

```
double Punto::getX() const {
      return x;
}
```

27

Esempio: *la classe punto (ancora…)*

```
 5  class Punto {
 6  private:
 7          double x;
 8          double y;
 9  public:
10          Punto(double c1, double c2) {
11                  set(c1,c2);
12          };
13          void set(double c1,double c2) {
14                  x=c1;
15                  y=c2;
16          };
17          double getX() const { return x;  }
18          double getY() const { return y;  }
19  };
```

Questa dichiarazione non modifica il comportamento dei metodi se invocati da un oggetto non costante, ma permette la loro invocazione su un oggetto costante.

28

Esempio: *la classe punto (ancora…)*

```
 5  class Punto {
 6  private:
 7          double x;
 8          double y;
 9  public:
10          Punto(double c1, double c2) {
11                  set(c1,c2);
12          };
13          void set(double c1,double c2) const {
14                  x=c1;
15                  y=c2;
16          };
17          double getX() const { return x;  }
18          double getY() const { return y;  }
19  };
```

ATTENZIONE. Dichiarare un metodo **`const`** non è sufficiente.

Nei metodi così definiti non è consentita alcuna modifica dei membri della classe. Queste righe producono un errore in fase di compilazione

Questa dichiarazione non modifica il comportamento dei metodi se invocati da un oggetto non costante, ma permette la loro invocazione su un oggetto costante.

29

Esempio: *la classe punto (ancora…)*

```
 5  class Punto {
 6  private:
 7          double x;
 8          double y;
 9  public:
10          Punto(double c1, double c2) {
11                  set(c1,c2);
12          };
13          void set(double c1,double c2) {
14                  x=c1;
15                  y=c2;
16          };
17          double getX() const { return x;  }
18          double getY() const { return y;  }
19  };
```

Il costruttore non può essere qualificato **`const`**

…e può invocare metodi per l'inizializzazione dei membri.

Costruttori e distruttori fanno eccezione. In realtà, l'effetto del qualificatore const, si estende dal *completamento* della creazione di un oggetto fino alla *invocazione* del distruttore

30

Maneggiare i membri **`const`**

E' frequente che una classe dichiari uno o più membri qualificati **`const`**

Si assume che tali membri ricevano un valore (che non sarà più modificato) in fase di inizializzazione.

Modifichiamo la classe **`punto`**, introducendo il membro privato costante

```
const string label;
```

31

Esempio: *la classe punto (ancora…)*

```
 5  class Punto {
 6  private:
 7          double x;
 8          double y;
 9          const string label;
10  public:
11          Punto(double c1, double c2, string l) {
12                  label=l;
13                  set(c1,c2);
14          };
15          void set(double c1,double c2); // prototipo
16          double getX() const { return x;  }
17          double getY() const { return y;  }
18          string getLabel() const { return label; }
```

Costante non inizializzata?

Assegnamento a una costante?

Questo codice produce diversi errori in fase di compilazione. C'è un'altra strada…

32

Inizializzazione dei dati membro

Data una classe con i seguenti membri:

```
class prova {
       double membroD;
       const int membroCI;
       int &ref;
```

Il costruttore ne inizializza i membri utilizzando la seguente sintassi:

```
public:
       prova(double m1, int m2, int &r) : membroD(m1),
              membroCI(m2), ref(r) {
              … // altro codice del costruttore
       }
```

33

Inizializzazione dei dati membro

Il costruttore utilizza una *lista di inizializzatori*:

```
public:
       prova(double m1, int m2) : membroD(m1), membroCI(m2) {
              … // altro codice del costruttore
       }
```

La lista è separata dal prototipo dal carattere **:** e precede la **{** che da l'inizio del codice del costruttore.

Gli inizializzatori sono separati da virgola, hanno ciascuno il nome del membro che devono inizializzare e recano, quale unico argomento, il valore da assegnarvi.

34

Inizializzazione dei dati membro

La *lista di inizializzatori* segue alcune regole…

In presenza di membri non costanti (né riferimenti), l'uso degli inizializzatori è facoltativo (i membri si possono inizializzare nel codice *as usual*).

E' obbligatorio l'uso degli inizializzatori per i membri qualificati **`const`** e per i *riferimenti*;

35

Esempio: *lista di inizializzatori*

```
 5  class prova {
 6          double membroD;
 7          const int membroCI;
 8          int &ref;
 9
10  public:
11          prova(double m1,int m2, int &k)
12                  : membroD(m1), membroCI (m2), ref(k)
13          { }
14          double getD() { return membroD; }
15          int getRef() {return  ref; }
16          int getCI() const {return membroCI;}
17          void incRef() {ref+=1;}
18
19  };
```

Costante e riferimento non inizializzati

Inizializzatori obbligatori

Per accedere a un membro **`const`**, ci vuole un metodo **`const`**

36

Esempio: *lista di inizializzatori*

```
20  int main()
21  {
22
23          int k=12;
24          prova x(0.1,10,k);
25          cout << "D="<<x.getD()<<endl;
26          cout << "CI="<<x.getCI()<<endl;
27          cout << "ref=" << x.getRef()<<endl;
28          x.incRef();
29          cout << "k="<<k<<endl;
30
31  }
```

```
D=0.1
CI=10
ref=12
k=13
```

37