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

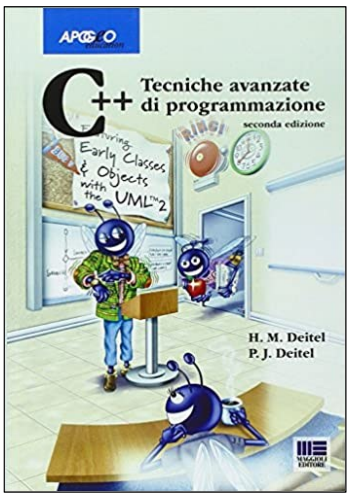
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

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

Dal C al C++

6

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

7

Il qualificatore **`const`**

8

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!
```

9

Il qualificatore **const**

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

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

Le seguenti espressioni producono un errore durante la compilazione:

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

10

Il qualificatore **const**

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

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

Il modo corretto per dichiarare riferimenti e puntatori a variabili const:

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

11

Il qualificatore **const**

Combinando opportunamente l'uso di **const** con riferimenti e puntatori, è possibile descrivere un vero e proprio *controllo di accesso* alle variabili.

12

Il qualificatore **`const`** e i riferimenti

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).*

13

Il qualificatore **`const`** e i riferimenti

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`**

14

Il qualificatore **const** e i riferimenti

La variabile **h** è R/W

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

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

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

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

15

Il qualificatore **const** e i riferimenti

La variabile **h** è R/W

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

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

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

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

L'assegnamento è lecito, la variabile **h** non è dichiarata **const**.

Si può accedere al nuovo valore di h attraverso il suo riferimento.

```
Ora è hRef=4.74
```

16

Il qualificatore **const** e i riferimenti

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 << "Ora è hRef=" << hRef << endl;
```

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

NO! **hRef** è un accesso *a sola lettura* per **h**

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

```
hRef=7.11;
```

17

Il qualificatore **const** e i riferimenti

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 << "Ora è hRef=" << hRe
```

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

NO! **hRef** è un accesso *a sola lettura* per **h**

Tuttavia, non è possibile assegnar
La seguente riga produce un errore.

```
hRef=7.11;
```

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

18

Il qualificatore **`const`** e i riferimenti

L'utilizzo più frequente di riferimenti **`const`** a variabili *che non lo sono* è nel passaggio di parametri alle funzioni *«per riferimento»*

```
int rw_var=20, risultato;
risultato=funzione(rw_var);
```

Nel **`main()`** la variabile **`rw_var`** è utilizzata come parametro a **`funzione()`**

19

Il qualificatore **`const`** e i riferimenti

L'utilizzo più frequente di riferimenti **`const`** a variabili *che non lo sono* è nel passaggio di parametri alle funzioni *«per riferimento»*

main:

```
int rw_var=20, risultato;
risultato=funzione(rw_var);
```

Nel **`main()`** la variabile **`rw_var`** è utilizzata come parametro a **`funzione()`**

funzione:

```
int funzione(const int &arg){
        …
}
```

Quando **`funzione()`** è invocata, si effettua il *binding* tra **`rw_var`** e il riferimento a costanti intere **`arg`**. Per tutta l'esecuzione di **`funzione()`** **`arg`** è una costante intera.

20

Il qualificatore **const** e i riferimenti

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

Lo scopo di tutto questo è:

1) Utilizzare un passaggio di parametri efficiente (*e.g.* in presenza di parametri molto «lunghi» da copiare);
2) Proteggere i dati della funzione chiamante da eventuali effetti collaterali indesiderati/imprevisti del codice delle funzioni che vi accedono;

21

Il qualificatore **const** e i puntatori

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*.

22

Il qualificatore **`const`** e i puntatori

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;
```

23

Il qualificatore **`const`** e i puntatori

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, [illegible] tramite **`hPtr`**. La seguente riga produ[illegible]

```
*hPtr=101;
```

NO! **`hPtr`** è un accesso *a sola lettura* per **`h`**

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

24

Il qualificatore **const** e i puntatori

```
 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  }
```


Il qualificatore **const** e i puntatori

```
 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

Il qualificatore **`const`** e i puntatori

L'uso combinato dei puntatori e di **`const`** copre diverse casistiche:

1. Puntatori liberi a dati liberi
   È possibile modificare i dati attraverso il puntatore ed è possibile modificare il puntatore stesso:

27

Il qualificatore **`const`** e i puntatori

L'uso combinato dei puntatori e di **`const`** copre diverse casistiche:

1. Puntatori liberi a dati liberi
   È possibile modificare i dati attraverso il puntatore ed è possibile modificare il puntatore stesso:

```
char x[11]="0123456789";
char *freePtr;

for(freePtr=x;*freePtr!='\0';freePtr++)
        *freePtr+=('9'-*freePtr);
```

Queste istruzioni **modificano** il valore della variabile puntatore **`freePtr`** (*i.e.* l'indirizzo a cui punta)

Questa istruzione **modifica** il contenuto della memoria puntata da **`freePtr`** (*i.e.* l'array **`x[]`**)

28

Il qualificatore **`const`** e i puntatori

L'uso combinato dei puntatori e di **`const`** copre diverse casistiche:

1. Puntatori liberi a dati liberi
   È possibile modificare i dati attraverso il puntatore ed è possibile modificare il puntatore stesso:
2. Puntatori liberi a dati costanti
   Non è possibile modificare i dati attraverso il puntatore ma è ancora possibile modificare il puntatore stesso:

29

Il qualificatore **`const`** e i puntatori

L'uso combinato dei puntatori e di **`const`** copre diverse casistiche:

2. Puntatori liberi a dati costanti
   Non è possibile modificare i dati attraverso il puntatore ma è ancora possibile modificare il puntatore stesso:

```
const char y[11]="abcdefghij";
const char *free2const;

for(free2const=y;*free2const!='\0';free2const++)
          cout<< *free2const + ('a'-*free2const);
```

Queste istruzioni **modificano** il valore della variabile puntatore **`free2const`** (*i.e.* l'indirizzo a cui punta)

Questa istruzione **non modifica** il contenuto della memoria puntata da **`free2const`** (*i.e.* l'array **`y[]`**)

30

Il qualificatore **const** e i puntatori

L'uso combinato dei puntatori e di **const** copre diverse casistiche:

1. Puntatori liberi a dati liberi
   È possibile modificare i dati attraverso il puntatore ed è possibile modificare il puntatore stesso:
2. Puntatori liberi a dati costanti
   Non è possibile modificare i dati attraverso il puntatore ma è ancora possibile modificare il puntatore stesso:
3. Puntatori costanti a dati liberi
   E'possibile modificare i dati attraverso il puntatore ma non è possibile modificare il puntatore stesso (è il caso degli array nativi in C/C++)

31

Il qualificatore **const** e i puntatori

L'uso combinato dei puntatori e di **const** copre diverse casistiche:

3. Puntatori costanti a dati liberi
   E'possibile modificare i dati attraverso il puntatore ma non è possibile modificare il puntatore stesso (è il caso degli array nativi in C/C++)

```
char x[11]="0123456789";
char * const const2free;

const2free=x;
```

NO! Ora, **const2free** non può più apparire a sinistra di un assegnamento, né incrementato…

32

Il qualificatore **`const`** e i puntatori

L'uso combinato dei puntatori e di **`const`** copre diverse casistiche:

3. Puntatori costanti a dati liberi
   E'possibile modificare i dati attraverso il puntatore ma non è possibile modificare il puntatore stesso (è il caso degli array nativi in C/C++)

Il puntatore *costante*, **`const2free`** deve essere inizializzato obbligatoriamente.

```
char x[11]="0123456789";
char * const const2free=x;

for(char *p=const2free;*p!='\0';p++)
        *p+=('9'-*p);
```

Non è concesso modificare il valore di **`const2free`** per iterare nell'array, usiamo il puntatore libero **`p`**

Questa istruzione **modifica** il contenuto della memoria puntata da **`const2free`** (*i.e.* l'array **`x[]`**)

33

Il qualificatore **`const`** e i puntatori

L'uso combinato dei puntatori e di **`const`** copre diverse casistiche:

1. Puntatori liberi a dati liberi
   È possibile modificare i dati attraverso il puntatore ed è possibile modificare il puntatore stesso:
2. Puntatori liberi a dati costanti
   Non è possibile modificare i dati attraverso il puntatore ma è ancora possibile modificare il puntatore stesso:
3. Puntatori costanti a dati liberi
   E'possibile modificare i dati attraverso il puntatore ma non è possibile modificare il puntatore stesso (è il caso degli array nativi in C/C++)
4. Puntatori costanti a dati costanti
   Non è possibile modificare né i dati attraverso il puntatore nè il puntatore

34

Il qualificatore **const** e i puntatori

L'uso combinato dei puntatori e di **const** copre diverse casistiche:

4. Puntatori costanti a dati costanti

   Non è possibile modificare né i dati attraverso il puntatore nè il puntatore

```
const char y[11]="abcdefghij";
const char * const const2const=y;
```

Il puntatore *costante*, **const2const** deve essere inizializzato obbligatoriamente.

35

Il qualificatore **const** e i puntatori

L'uso combinato dei puntatori e di **const** copre diverse casistiche:

4. Puntatori costanti a dati costanti

   Non è possibile modificare né i dati attraverso il puntatore nè il puntatore

```
const char y[11]="abcdefghij";
const char * const const2const=y;

for(char *p=const2const;*p!='\0';p++)
          cout<< *p + ('a'-*p);
```

Il puntatore *costante*, **const2const** deve essere inizializzato obbligatoriamente.

Non è concesso modificare il valore di **const2const**, usiamo il puntatore libero **p**

Questa istruzione **non modifica** il contenuto della memoria puntata da **const2const** (*i.e.* l'array **y[]**)

36

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

37

Oggetti e funzioni membro costanti

38

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  };
```

39

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…

40

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);
```

41

Dichiarare oggetti **const**

Dichiariamo dei *punti*

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

ma il secondo, ci riserva qualche sorpresa…

42

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;
```

43

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;
}
```

44

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.

45

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.

46

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

47

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;
```

48

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…

49

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
       }
```

50

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.

51

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*;

52

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**

53

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
```

54