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

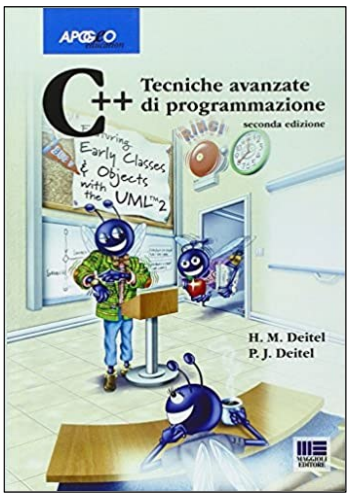
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

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

Ridefinizione degli operatori

6

Ridefinizione degli operataori

In C++ si possono *ridefinire* gli operatori del linguaggio (*e.g.* +, *, -> …) per utilizzarli con gli oggetti definiti dall'utente.

Si possono quindi comporre *espressioni*, in maniera analoga a quanto avviene per i tipi nativi.

- Comodità di scrittura del codice
- Coerenza semantica con le espressioni che coinvolgono i tipi nativi
- Includere i nuovi tipi nelle espressioni che coinvolgono quelli nativi

7

Ridefinizione degli operatori

Un esempio: nella dotazione standard del C++ è compreso il tipo di dati **`complex`** per effettuare calcoli con i numeri complessi. Per utilizzarla si include l'header **`complex`**

- Si tratta ovviamente di una classe (come **`string`**) che:
- fornisce le principali funzioni matematiche sui complessi;
- ridefinisce gli operatori aritmetici **`+`**, **`-`**, **`*`** e **`/`** per usarli in espressioni che contengono **`complex`**, **`double`** e **`float`** in ogni combinazione.

8

Esempio: *espressioni con numeri complessi*

```
 1  #include <iostream>
 2  #include <complex>
 3  int main ()
 4  {
 5    std::complex<double> mycomplex;
 6
 7    mycomplex = 10.0;
 8    mycomplex += 2.0;
 9
10    mycomplex = std::complex<double>(10.0,1.0);  // 10+i
11    mycomplex = mycomplex + 10.0 ;               // 20+i
12
13    if (mycomplex == std::complex<double>(20.0,1.0))
14      std::cout << "mycomplex is " << mycomplex << '\n';
15
16    return 0;
17  }
```

https://cplusplus.com/reference/complex/

9

Ridefinizione degli operatori

Si realizza definendo una funzione o un metodo non statico (gli operatori si riferiscono agli oggetti e non alle classi)

Sintassi:

```
TipoRest operatorXXX(tipo1 arg1)
TipoRest operatorXXX(tipo1 arg1, tipo2 arg2)
```

Dove ***XXX*** è sostituito dall'operatore (il simbolo che si vuole ridefinire)

10

Esempio: *ridefinizioni degli operatori*

```
 5  template <class ItemType>
 6  class couple
 7  {
 8          private:
 9                  ItemType data[2];
10          public:
11                  couple() {}
12                  couple(ItemType sx, ItemType dx) {
13                          set(sx,dx);
14                  }
15                  void set(ItemType sx, ItemType dx) {
16                          data[0]=sx;
17                          data[1]=dx;
18                  }
19                  void get(ItemType &sx, ItemType &dx) {
20                          sx=data[0];
21                          dx=data[1];
22                  }
```

11

Esempio: *ridefinizioni degli operato*

```
23                  couple operator-()
24                  {
25                          couple x;
26                          x.set(this->data[1],this->data[0]);
27                          return x;
28                  }
29                  bool operator==(couple &b) {
30                          return (this->data[0]==b.data[0])&&
31                                                  (this->data[1]==b.data[1]);
32                  }
33  };
34  template <class ItemType>
35  void show(couple<ItemType> a) {
36          ItemType sx, dx;
37          a.get(sx,dx);
38          cout << "sx="<<sx<<", dx="<<dx<<"; ";
39  }
```

Ridefinizione dell'operatore – unario. Restituisce una coppia con gli elementi invertiti.

L'operatore non ha argomenti, poiché si assume invocato sull'oggetto corrente (puntato da **this**).

12

Esempio: *ridefinizioni degli operatori*

```
23                  couple operator-()
24                  {
25                          couple x;
26                          x.set(this->data[1],this->data[0]);
27                          return x;
28                  }
29                  bool operator==(couple &b) {
30                          return (this->data[0]==b.data[0])&&
31                                                  (this->data[1]==b.data[1]);
32                  }
33  };
34  template <class ItemType>
35  void show(couple<ItemType> a) {
36          ItemType sx, dx;
37          a.get(sx,dx);
38          cout << "sx="<<sx<<", dx="<<dx<<"; ";
39  }
```

Ridefinizione dell'operatore **==** binario.

L'operatore ha un solo argomento (quello di destra), poiché si assume che quello di sinistra sia quello «corrente» (puntato da **this**).

13

Esempio: *ridefinizioni degli operatori*

```
40  int main()
41  {
42          couple<int> a(1,2), b(2,1), c;
43
44          cout<<"-a:";
45          show(-a);
46          cout << endl << "-a==b:" << (-a==b) <<endl;
47          c=-b;
48          cout << "c: ";
49          show(c);
50          cout <<endl;
51  }
```

La ridefinizione non cambia le precedenze: **-** viene prima di **<<** che viene prima di **==**.

14

Ridefinizione degli operatori

Regole:

1) Non si possono definire nuovi operatori (*e.g.* non si può definire l'operatore `@`), ma solo ridefinire quelli esistenti, con alcune eccezioni:

`.` (punto) `.*` `::` (scope)
`?:` (ternario) `sizeof()`

Gli operatori `=` (assegnamento) `,` (virgola)
`&` (referenziazione) `;`

funzionano per qualsiasi tipo. Possono essere ridefiniti ma è necessaria molta cautela.

15

Ridefinizione degli operatori

Regole:

2) La ridefinizione non modifica la precedenza degli operatori, sulla associatività (valutazione da sx→dx o da sx→dx)

3) Il numero degli operandi (la *arità*) degli operatori non può essere modificata. Gli operatori unari restano unari, quelli binari restano binari.

Gli operatori che sono sia unari, sia binari (*e.g.* `*`, `&`) devono essere ridefiniti separatamente.

16

Ridefinizione degli operatori

Regole:

4) Non è possibile modificare il comportamento degli operatori sui tipi nativi, *e.g.* non è possibile ridefinire il significato dell'operatore **+** tra operandi di tipo **`int`**;

5) La ridefinizione di operatori «componibili» (*e.g.* **+**, **&**) e dell'operatore di assegnamento **=**, non ridefinisce automaticamente gli operatori composti **+=** e **&=**

17

Ridefinizione degli operatori

Regole:

7) La ridefinizione di un operatore può essere implementata mediante il metodo di una classe oppure mediante una funzione globale.

Nella maggior parte dei casi, la scelta è del programmatore ed è determinata da considerazioni di ordine pratico

In alcune circostanze, la scelta è obbligata, in un senso o nell'altro

18

Ridefinizione degli operatori

Regole:

7) La ridefinizione di un operatore può essere implementata mediante il metodo di una classe oppure mediante una funzione globale.

Metodo della classe	Funzione globale
obbligatorio: `()`, `[]` e `->` il primo argomento (sx) è sempre l'oggetto su cui sono invocati	necessariamente: quando il tipo/ classe del primo argomento può variare
generalmente: gli operatori unari (*e.g.* `-`, `++` e `&` …) sono invocati sul oggetto che ne è l'unico argomento	generalmente: quando gli argomenti sono di tipo/classe diverso e l'operatore deve essere commutativo.
Quando l'operatore utilizza attributi e metodi private del primo/unico argomento	Quando utilizza attributi e metodi private degli argomenti, può essere dichiarato `friend` delle rispettive classi

19

Ridefinizione degli operatori

Esempio: ridefiniamo `&&` perché restituisca `true` quando: l'item `i` è compreso nella `couple c` con gli argomenti che possono essere passati indifferentemente nei due ordini: `i&&c` oppure `c&&i`

Regole:

7) La ridefinizione di un operatore pu
mediante il metodo di una classe
funzione globale.

Metodo della classe	Funzi bale
obbligatorio: `()`, `[]` e `->` il primo argomento (sx) è sempre l'oggetto su cui sono invocati	necessariamente: qu il tipo/ classe del primo argomento può riare
generalmente: gli operatori unari (*e.g.* `-`, `++` e `&` …) sono invocati sull'oggetto che ne è l'unico argomento	generalmente: quando gli argomenti sono di tipo/classe diverso e l'operatore deve essere commutativo.
Quando l'operatore utilizza attributi e metodi private del primo/unico argomento	Quando utilizza attributi e metodi private degli argomenti, può essere dichiarato `friend` delle rispettive classi

20

Esempio: *ridefinizioni degli operatori*

```
 5  template <class ItemType>
 6  class couple
 7  {
 8          private:
 9                  ItemType data[2];
10          public:
11                  couple() {}
 …  …
23                  bool operator&&(ItemType x) {
24                          return x==this->data[0]||x==this->data[1];
25                  }
26  };
27
28  template<class ItemType>
29  bool operator&&(ItemType fst, couple<ItemType> sec){
30          return sec && fst;
31  }
```

Questo ridefinisce l'operatore **&&** con un metodo della classe **couple** e implementa **c&&i**

Questo ridefinisce ulteriormente l'operatore **&&** con una funzione globale. E' invocato nei casi in cui l'oggetto di classe **couple** non è l'operando di sinistra. Lo usiamo per implementare **i&&c**

21

Esempio: *ridefinizioni degli operatori*

```
40  int main()
41  {
42          couple<int> a(1,2), b(2,1), c;
 …          …
47          c=-b;
 …          …
51          cout << "1 && c: "<< (1&&c) << ", c && 0: "<< (c&&0)<<endl;
52          show(c);
53  }
```

Questo invoca il metodo

Questo invoca la funzione

22

Esercizio: *ridefinizioni degli operatori*

Nell programma precedente, si ridefinisca l'operatore **>>** in modo che: date la **couple c** e l'item **i**:

c>>i, assegni il secondo elemento di **c** a **i** e:

i>>c assegni **i** al primo elemento di **c**

In entrambi i casi, l'espressione restituisce un *riferimento* a **c**

23

Esercizio: *ridefinizioni degli operatori*

```
 5  template <class ItemType>
 6  class couple
 7  {
 8          private:
 9                  ItemType data[2];
10          public:
11                  couple() {}
…   …
28          couple &operator>>(ItemType &i) {
29                          i=this->data[1];
30                          return *this;
31                  }
32  };
```

Restituisce il secondo valore della coppia mediante il riferimento alla variabile di «destinazione» **i**. Qui implementiamo **>>** come metodo poiché intercetta le chiamate in cui l'argomento di sinistra è l'oggetto di classe **couple** su cui è invocato.

Restituisce il riferimento all'oggetto su cui è stato invocato (l'operando di sinistra, quando questo è di tipo **couple**)

24

Esercizio: *ridefinizioni degli operatori*

```
40  template<class ItemType>
41  couple<ItemType> &operator>>(ItemType fst, couple<ItemType> &sec){
42          int sx,dx;
…           sec.get(sx,dx);
47          sec.set(fst,dx);
…           return sec;
51  }
…
```

Restituisce il riferimento all'oggetto su cui è stato invocato (l'operando di sinistra, quando questo è di tipo **couple**)

Assegna il valore di **fst** al primo valore della coppia. La modifica è effettuata sull'oggetto **sec** passato per riferimento come secondo parametro dell'operatore.
Qui implementiamo **>>** come funzione globale poiché intercetta le chiamate in cui l'argomento di sinistra è di tipo diverso dalla classe **couple**.

25

Esercizio: *ridefinizioni degli operatori #2*

Nell programma precedente, si ridefinisca l'operatore **<<** in modo che: date la **couple c** e l'item **i**:

c<<i, assegni **i** al secondo elemento di **c** e:

i<<c assegni a **i** il primo elemento di **c**

In entrambi i casi, l'espressione restituisce un *riferimento* a **c**

26

Ridefinizione degli operatori

Metodo della classe	Funzione globale
Operatori unari *prefissi* **–**, **+**, **&**, **++** … **`return_type operatorXXX() {...}`** Non ci sono argomenti, **`this`** punta all'operando.	Operatori unari *prefissi* **–**, **+**, **&**, … **`return_type operatorXXX(arg_type op1) {...}`** C'è un solo argomento: **`op1`** che è l'operando.
Operatori binari **+**, **<=**, **/**, … **`return_type operatorXXX(arg_type op2) {...}`** **`this`** punta all'operando di sinistra. La funzione ha un solo argomento che è l'operando di destra.	Operatori binari **+**, **<=**, **/**, … **`return_type operatorXXX(arg_type op1, arg_type op2) {...}`** Due parametri: **`op1`** è l'operando di sinistra, **`op2`** è l'operando di destra.
Operatori unari *postfissi* **––**, **++** **`return_type operatorXXX(int foo) {...}`** L'argomento **`foo`** è inserito per distinguere il prototipo dall'operatore prefisso e non viene utilizzato (il compilatore lo pone a **0**). **`this`** punta all'operando.	Operatori unari *postfissi* **––**, **++** **`return_type operatorXXX(arg_type op1, int foo) {...}`** L'argomento **`foo`** è inserito per distinguere il prototipo dall'operatore prefisso e non viene utilizzato (il compilatore lo pone a **0**). **`op1`** è l'operando.

27

Esercizio: *ridefinizioni degli operatori #3*

Nell programma precedente, si ridefinisca l' operatore **++** in modo che: date la **`couple c`**:

`c++`, incrementi secondo elemento di **`c`** e:

`++c` incrementi il primo elemento di **`c`**

In entrambi i casi, l'espressione restituisce un *riferimento* a **`c`**

28

Esercizio: *ridefinizioni degli operatori #4*

Nella classe angolo proposta in un precedente esempio, si ridefiniscano gli operatori **+** e **–** in modo che: dati gli **angoli a**, **b** e **c**:

c=a+b, calcoli la somma **c** dei due angoli **a** e **b** avendo cura di mantenere la coerenza della notazione (*no angoli di più di 360°*) …

c=a-b, calcoli la differenza **c** tra i due angoli **a** e **b** avendo cura di mantenere la coerenza della notazione (*no angoli minori di zero: -45°=360°-45°= 315°*) …

29

Esempio: *angoli in gradi e radianti*

file: angolo.hpp

```
 5  class Angolo {
 6  private:
 7          int gradi;
 8          int primi;
 9          int secondi;
10  public:
11          Angolo(): gradi(0), primi(0), secondi(0) {}
12          Angolo(int ,int, int);
13          Angolo(double);
14          int getG();
 …          …
20          void setS(int);
21          void set(int,int,int);
22          void set(double);
23          double get();
24          Angolo somma(Angolo);
25  };
```

30

Operatori bitwise: una classe **`bitarray`**

31

Esempio: la classe **`bitarray`**

Utilizzando gli operatori bitwise, si implementi la classe **`bitarray`** che realizzi un array di bit di lunghezza arbitraria a cui è possibile accedere singolarmente sia in lettura, sia in scrittura.

32

Esempio: la classe **bitarray**

file: bitarray.hpp

```
 5  class bitarray {
 6          private:
 7                  unsigned char *b;
 8                  int numbytes;
 9                  int len;
10
11                  bool bitpos(int, int&, int&);
12
13          public:
14                  bitarray(int);
 …                  ~bitarray();
20                  int size();
21                  bool get(int);
22                  bool set(int,bool);
23                  void zero();
24                  void show();
25  };
```

Per immagazzinare i bit, utilizziamo un array dinamico di bytes **b**, lungo **numbytes**. Questo ha una capienza massima **numbites*8** bit ma noi lasciamo all'utente la facoltà di scegliere una lunghezza **len** (numero di bit) arbitraria.

33

Esempio: la classe **bitarray**

file: bitarray.cpp

```
 5  #include<iostream>
 6  #include "bitarray.hpp"
 7  using namespace std;
 8
 9  bitarray::bitarray(int l=8) {
10          if (l<8)
11                  len=8;
12          else
13                  len=l;
14          numbytes=l/8+1;
 …
20          b=new unsigned char[numbytes];
21          zero();
22  }
23  bitarray::~bitarray() {
24          delete [] b;
25  }
```

L'utente invoca il costruttore indicando quanti bit deve contenere il suo array (*min. 8 per semplicità*). Il costruttore calcola il minimo numero di byte necessari per soddisfare la richiesta, quindi alloca un array di tale dimensione.

34

Esempio: la classe **bitarray**

file: bitarray.cpp

```
…
bool bitarray::bitpos(int i, int& bytenum, int& bitmask)
{
        if(i<0||i>len-1)
                return 0;

        bytenum=i/8;
        bitmask=1<<(i%8);
        return 1;
}
```

Funzione di servizio che «localizza» il bit *i*-esimo dell' bitarray, identificando il numero del byte e la maschera che distingue il bit all'interno del byte

byte2	byte1	byte0
00011000	00101000	00111100
0001000		

Il bit i=20 è
Nel byte2, la sua maschera è 00010000

35

Esempio: la classe **bitarray**

file: bitarray.cpp

```
int bitarray::size()
{
        return len;
}

bool bitarray::get(int i)
{
        int bytepos, bitmask;
        if(!bitpos(i,bytepos,bitmask))
                return 0;
        else
                return b[numbytes-bytepos-1]&bitmask;
}
…
```

Se l'and bitwise tra la maschera e il byte selezionato dà un valore diverso da zero, significa che il bit *i*-esimo ha valore 1. Diversamente, il risultato è 0.

36

Esempio: la classe **bitarray**

file: bitarray.cpp

```
48  bool bitarray::set(int i,bool val)
49  {
50          bool rv;
51          int bytepos,bitmask;
52          if(!bitpos(i,bytepos,bitmask))
53                  return 0;
54
55          rv=b[numbytes-bytepos-1]&bitmask;
56          if(val)
57                  b[numbytes-bytepos-1]|=bitmask;
58          else
59                  b[numbytes-bytepos-1]&=(~bitmask);
60
61          return rv;
62  }
…   …
```

Setta il bit *i*-esimo al valore **val** e restituisce il suo valore precedente.

37

Esempio: la classe **bitarray**

file: bitarray.cpp

```
63  void bitarray::zero()
64  {
65          for(int i=0;i<numbytes;i++)
66                  b[i]=0;
67  }
68
69  void bitarray::show()
70  {
71          cout<<"nb="<<numbytes<<", len="<<len<<endl<<"bytes:";
72          for(int i=0;i<numbytes;i++)
73                  cout << " "<<numbytes-i-1<<":"<<(int)b[i]<<",  ";
74          cout <<endl;
75  }
```

38

Esempio: Esempio: la classe **bitarray**

file: bittest.cpp

```
 5  int main()
 6  {
 7          int len, bit, scelta=0;
 8          bool val;
 9          bitarray x(30);
10
11          cout << "*** Classe bitarray ***"<<endl;
12          do{
13                  cout<<"     1 - get"<<endl;
14                  cout<<"     2 - set"<<endl;
…                   cout<<"     3 - zero"<<endl;
20                  cout<<"     4 - show"<<endl;
21                  cout<<"     0 - exit"<<endl;
22                  cout<<" Scelta: ";
23                  cin>>scelta;
…   …
```

39

Esempio: Esempio: la classe **bitarray**

file: bittest.cpp

```
24                  switch(scelta) {
25                          case(0):
26                                  break;
27                          case(1):
28                                  cout<<" -> bit? ";
29                                  cin>>bit;
30                                  cout << "x["<<bit<<"]="<<x.get(bit)<<endl;
31                                  break;
32                          case(2):
33                                  cout<<" -> bit? ";
34                                  cin>>bit;
35                                  cout<<" -> val? ";
36                                  cin>>val;
37                                  cout << "prev. x["<<bit<<"]="<<x.set(bit,val);
38                                  cout<<endl;
39                                  break;
40
```

40

Esempio: Esempio: la classe **bitarray**

file: bittest.cpp

```
41  case(4):
42                                      x.show();
43                  }
44          } while(scelta!=0);
45
46  }
```

41