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

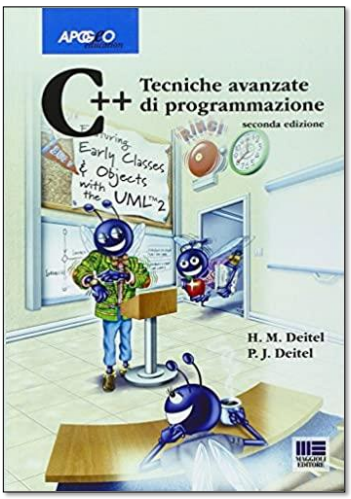
Tecniche avanzate e strutture dati elementari

[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

Esercizio: *ridefinizioni degli operatori #5*

Nella classe **`sequenza`** proposta in un precedente esempio, si ridefiniscano **`[]`** in modo che, dati l'indice **`i`**, gli item **`a`** e **`b`** e la sequenza **`seq`**:

`a=seq[i]` assegni all'item **`a`** l'item contenuto nella i-esima posizione di **`seq`**

`seq[i]=b` assegni l'item **`b`** aell'i-esima posizione di **`seq`**

7

Esercizio: *ridefinizioni degli operatori #5*

file: selectionsort.hpp

```
…  …
4  template <class TipoBase>
5  class sequenza {
6          private:
7                  TipoBase *b;
8                  int num;
9                  sequenza *swap(TipoBase&, TipoBase&);
10         public:
11                 sequenza(int);
12                 sequenza *mischia(int);
…  …
23                 TipoBase &operator[](int);
24                 TipoBase operator[](int) const;
25 };
…  …
```

Indicizzazione per oggetti non costanti (può comparire anche al lato sinistro di un assegnamento)

Indicizzazione per oggetti costanti

8

Esercizio: *ridefinizioni degli operatori #5*

file: selectionsort.hpp

```
 …  …
144 template <class TipoBase>
145 TipoBase &sequenza<TipoBase>::operator[](int i)
146 {
147         return this->b[i];
148 }
149
150 template <class TipoBase>
151 TipoBase sequenza<TipoBase>::operator[](int i) const
152 {
153         return this->b[i];
154 }
 …  …
```

9

Esempio: la classe **`bitarray`**

Ridefiniamo alcuni operatori della classe **`bitarray`**:

- confronto: **`==`** e **`!=`**
- accesso: **`[]`**
- assegnamento tra array: **`ba2=ba1`**
- assegnamenti di elementi dell'array: …

10

Esempio: la classe **bitarray**

file: bitarray.hpp

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

11

Esempio: la classe **bitarray**

file: bitarray.hpp

```
 …   …
24            bool operator==(const bitarray &) const;
25            bitarray& operator=(const bitarray&);
26            bool operator[](int) const;
 …   …
```

Restituisce il valore (**bool**) del bit memorizzato nella posizione indicata tra **[]**

Copia il contenuto dell'operando di destra (classe bitarray) nell'operando di sinistra. Se l'array dell'*lvalore* è di dimensione diversa, procede a distruggerlo e a riallocarlo della stessa dimensione dell'*rvalore*.

12

Esempio: la classe **bitarray**

file: bitarray.hpp

```
…
24          bool operator==(const bitarray &) const;
25          bitarray& operator=(const bitarray&);
26          bool operator[](int) const;
…
```

Restituisce il valore (**bool**) del bit memorizzato nella posizione indicata tra **[]**

13

Esempio: la classe **bitarray**

file: bitarray.cpp

```
…
77  bool bitarray::operator==(const bitarray &rx) const
78  {
79          if(len!=rx.len)
80                  return false;
81          for(int i=0;i<numbytes;i++)
82                  if(b[i]!=rx.b[i])
83                          return false;
84          return true;
85  }
…
```

Operatore di confronto tra **bitarray**. Ridefinito come metodo **const** della classe. Confronta prima la lunghezza (in bit) dei due operandi, e poi il loro contenuto.

14

Esempio: la classe **bitarray**

file: bitarray.cpp

```
…
87  bitarray &bitarray::operator=(const bitarray &rx)
88  {
89          if (&rx==this)
90                  return *this;
91
92          if (len!=rx.len) {
93                  delete[] b;
94                  numbytes=rx.numbytes;
95                  len=rx.len;
96                  b=new unsigned char[numbytes];
97          }
98          for (int i=0;i<numbytes;i++)
99                  b[i]=rx.b[i];
100
101         return *this;
102 }
…
```

No autoassegnamento

Se l'array dell'*lvalore* è di dimensione diversa, procede a distruggerlo e a riallocarlo della stessa dimensione dell'*rvalore*.

Copia il contenuto dell'operando di destra (classe bitarray) nell'operando di sinistra.

Esempio: la classe **bitarray**

file: bitarray.cpp

```
…
104 bool bitarray::operator[] (int index) const
105 {
106         return this->get(index);
107 }
…
```

Restituisce un valore di tipo **bool** che è un *rvalore* (può essere assegnato a una variabile booleana)

Si ricordi che in una espressione di assegnamento, la quantità posta a sinistra dell'operatore è detta *lvalore*, quella a destra *rvalore*

lvalue = rvalue

Il primo identifica un *«recipiente»* (una variabile, un riferimento o un puntatore) che indica dove (in memoria) deve essere immagazzinato il secondo. D'altro canto, l'*rvalore* è un dato *immutabile*, prodotto dalla valutazione di una espressione, che può essere impiegato in ulteriori calcoli o assegnato.

Esempio: la classe **bitarray**

Ridefiniamo alcuni operatori della classe **bitarray**:

- confronto: `==` e `!=`
- accesso: `[]`
- assegnamento tra array: `ba2=ba1`
- assegnamenti di elementi dell'array: …

```
bool x=true;
bitarray ba(10);
…
ba[9]=x;
```

Per fare qualcosa di simile a questo:

17

Esempio: la classe **bitarray**

Analizziamo questa espressione:

```
bool x=true;
bitarray ba(10);
…
ba[9] = x;
```

Stiamo assumendo che questo sia un riferimento a **bool** ed è a sua volta il risultato di una espressione (l'applicazione di `[]` al **bitarray ba**)

Questo è l' *rvalore* di tipo **bool**

18

Esempio: la classe **bitarray**

Si ricordi inoltre che in questa espressione, ci sono due operatori:

```
ba[9] = x;
```

L'operatore di accesso `[]` che ha operandi **ba** e **9**;

L'operatore di assegnamento che viene valutato *dopo* e che ha come operandi il risultato del *primo* e **x**;

Il modo in cui memorizzare il valore di **x** nel **bitarray** dipende dal valore di **x**. Ma questo viene valutato *prima* dell'operazione di assegnamento (la ridefinizione non cambia le precedenze…)…

19

Esempio: la classe **bitarray**

Si ricordi inoltre che in questa espressione, ci sono due operatori:

```
ba[9] = x;
```

L'operatore di accesso `[]` che ha operandi **ba** e **9**;

L'operatore di assegnamento che viene valutato *dopo* e che ha come operandi il risultato del *primo* e **x**;

Il modo in cui memorizzare il valore di **x** nel **bitarray** dipende dal valore di **x**. Ma questo viene valutato *prima* dell'operazione di assegnamento (la ridefinizione non cambia le precedenze…)…

A meno di riprogettare radicalmente la classe **bitarray**, questa strada non è percorribile…

20

Esempio: la classe **bitarray**

Potremmo ridefinire, due operatori di assegnamento: (***=** e **/=**) uno per assegnare il bit **1** a una data posizione dell'array, e un altro per assegnare il bit **0**:

```
bool x=true;
bitarray ba(10);
…
ba*=9;
ba/=3;
```

assegna **1** alla posizone **9** di **ba**;

assegna **0** alla posizone **3** di **ba**;

21

Esempio: la classe **bitarray**

file: bitarray.hpp

```
…
                bitarray& operator*=(int);
                bitarray& operator/=(int);
…
```

file: bitarray.cpp

```
…
bitarray &bitarray::operator*=(int index)
{
        this->set(index,1);
        return *this;
}

bitarray &bitarray::operator/=(int index)
{
        this->set(index,0);
        return *this;
}
…
```

22

Esempio: la classe **bitarray**

Ridefiniamo l'operatore **<<** per inviare un bitarray direttamente su uno stream di output (**cout**):

```
bool x=true;
bitarray ba(10);
…
ba*=9;
ba/=3;
cout << ba <<endl;
```

La sequenza di bit contenuta nel bitarray **ba** è visualizzata sulla console;

Esempio: la classe **bitarray**

file: bitarray.hpp

```
1  #ifndef _BITARRAY_HPP_
2  #define _BITARRAY_HPP_
3  #include<iostream>
4  using std::ostream;
5  class bitarray {
6          friend ostream &operator<<(ostream &, const bitarray&);
7          private:
8                  unsigned char *b;
9                  int numbytes;
10                 int len;
…  …
```

L'operatore **<<** dovrà essere ridefinito con una funzione globale poiché il suo operando di sinistra è un oggetto **ostream**.

Generalmente, queste funzioni sono dichiarate friend delle classi cui appartengono gli operandi di destra, nel caso in cui sia necessario accedere ai loro membri privati.

Esempio: la classe **bitarray**

file: bitarray.cpp

```
 …  …
 68  void bitarray::show() const
 69  {
 70          int d;
 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 << *this <<endl;
 75  }
 …  …
132  ostream &operator<<(ostream &output, const bitarray &btarr)
133  {
134          for (int bit=btarr.size()-1;bit>=0;bit--)
135                  output << (char)('0'+btarr[bit]);
136
137          return output;
138  }
```

25

Il qualificatore **static**

26

Il qualificatore **`static`**

In una classe: un membro qualificato come statico è considerato *comune a tutte le istanze* della classe.

Un *attributo statico (attributo «di classe»)* è una variabile il cui valore è condiviso tra tutti gli oggetti di quella classe;

Un *metodo statico (metodo «di classe»)* può essere invocato anche se non sono presenti istanze della classe…

27

Il qualificatore **`static`**

In una classe: un membro qualificato come statico è considerato *comune a tutte le istanze* della classe.

Un *attributo statico (attributo «di classe»)* è una variabile il cui valore è condiviso tra tutti gli oggetti di quella classe;

Ciascuna istanza può modificarne il valore. La modifica ha immediatamente effetto in tutte le altre istanze della classe.

Un *metodo statico (metodo «di classe»)* può essere invocato anche se non sono presenti istanze della classe…

E' un po' come un «servizio offerto» dalla classe, ma non necessariamente vincolato a essa…

28

Attributi **static**

Un attributo statico (attributo «di classe») è una variabile il cui valore è condiviso tra tutti gli oggetti di quella classe;

Si dichiara con il *qualificatore di persistenza* **static**

Ogni oggetto creato da quella classe avrà accesso allo stesso blocco di memoria per una variabile **static**.

Le modifiche all'attributo **static** saranno visibili a tutti gli altri oggetti.

29

Attributi **static**

```
 5  class ClasseX {
 6          static int counter;
 7          int sx, dx;
 8          public:
 9                  ClasseX(): sx(0), dx(0) {
10                          counter++;
11                  }
12                  ClasseX(int x, int y) {
13                          sx=x;
14                          dx=y;
15                          counter++;
16                  }
17                  ~ClasseX() {
18                          counter--;
19                  }
20          int get_counter() {
21                  return counter;
22          }
23  };
24  int ClasseX::counter=0;
```

Un'attributo **static** è definito all'interno della definizione della classe. Può essere **public**, **private** o **protected**.

I metodi della classe possono accedervi normalmente, come per qualsiasi altro attributo non **static** della stessa classe

Un'attributo **static** deve OBBLIGATORIAMENTE essere inizializzato fuori dal corpo della classe, una sola volta.

30

Attributi **static**

```
25  int main()
26  {
27          ClasseX a(0,0),b(0,0), *c;
28
29          cout << "ClasseX a.get_couter()="<<a.get_counter()<<", "<<endl;
30          c=new ClasseX(2,2);
31          cout << "ClasseX c->get_couter()="<<c->get_counter()<<", "<<endl;
32          delete c;
33          cout << "ClasseX b.get_couter()="<<b.get_counter()<<", "<<endl;
34          c=new ClasseX[10];
35          cout << "ClasseX a.get_couter()="<<a.get_counter()<<", "<<endl;
36
37  }
```

```
ClasseX a.get_couter()=2,
ClasseX c->get_couter()=3,
ClasseX b.get_couter()=2,
ClasseX a.get_couter()=12,
```

31

Attributi **static**

Attributo di istanza (non static)	**Attributo di classe (static)**
Dichiarato e inizializzato nel corpo della definizione della classe.	Dichiarato nel corpo della definizione della classe, ma inizializzato all'esterno di esso.
Ogni istanza ha la sua copia individuale dell'attributo. Le modifiche al suo valore hanno effetto solo per l'istanza medesima	Tutte le istanze della stessa classe, condividono la stessa copia dell'attributo. Eventuali modifiche apportate attraverso una istanza, hanno effetto su tutte le altre.
Esiste e può essere referenziato solo dopo la creazione dell'istanza.	Esiste e può essere referenziato anche se nessuna istanza della sua classe è stata ancora creata.

L'operatore di accesso è il `.` (*punto*):
`class prova { public: int b; };`
`prova x;`
`x.b=0;`
	L'operatore di accesso è `::` (*scope*)
	`class prova { public: static int b; };`
	`prova::b=5;`
	`prova x, y;`
	`x.b; // vale 5`
	`y.b; // vale 5`

32

Metodi **`static`**

Una funzione membro **`static`** rappresenta una funzionalità (o servizio) *indipendente* dallo stato delle istanze di quella classe.

Si dichiara con il *qualificatore di persistenza* **`static`** (sintassi analoga a quella degli attributi)

Una funzione membro **`static`** può manipolare solo membri **`static`**:

- Invocare altre funzioni membro **`static`**
- Manipolare attributi **`static`**

33

Metodi **`static`**

```
 5  class ClasseX {
 6          static int counter;
 7          int sx, dx;
 8          public:
 9                  ClasseX(): sx(0), dx(0) {
10                          counter++;
11                  }
12                  ClasseX(int x, int y) {
13                          sx=x;
14                          dx=y;
15                          counter++;
16                  }
17                  ~ClasseX() {
18                          counter--;
19                  }
20          static int get_counter() {
21                  return counter;
22          }
23  };
24  int ClasseX::counter=0;
```

Un metodo static può manipolare solo attributi static

34

Metodi **static**

```
25  int main()
26  {
27          ClasseX a(0,0),b(0,0), *c;
28
29          cout << "ClasseX a.get_couter()="<<a.get_counter()<<", "<<endl;
30          c=new ClasseX(2,2);
31          cout << "ClasseX c->get_couter()="<<c->get_counter()<<", "<<endl;
32          delete c;
33          cout << "ClasseX b.get_couter()="<<b.get_counter()<<", "<<endl;
34          c=new ClasseX[10];
35          cout << "ClasseX get_couter()="<<ClasseX::get_counter()<<", "<<endl;
36
37  }
```

```
ClasseX a.get_couter()=2,
ClasseX c->get_couter()=3,
ClasseX b.get_couter()=2,
ClasseX get_couter()=12,
```

35

Il *gioco del 15*

36

Il *gioco del 15*

37

Il *gioco del 15*

```
 5  class schema15 {
 6          private:
 7                  int pad[16], num_mosse, vX, vY;
 8                  const int maxiter=1000;
 9          public:
10                  schema15();
11                  schema15(int);
12                  bool inizializza();
13                  bool inizializza(int *);
14                  bool genera(int);
15                  void mischia(int);
16                  bool valido();
17                  bool vinto();
18                  bool alto();
19                  bool basso();
20                  bool destra();
21                  bool sinistra();
22                  int mosse();
23                  void mostra();
24  };
```

38

Il *gioco del 15*

```
10  schema15::schema15()
11  {
12          inizializza();
13  }
14
15  schema15::schema15(int seed)
16  {
17          if(!genera(seed))
18                  inizializza();
19  }
20
21  bool schema15::inizializza()
22  {
23          for (int i=1;i<=16;i++)
24                  pad[i-1]=i;
25          num_mosse=0;
26          vX=vY=4;
27          return true;
28  }
```

39

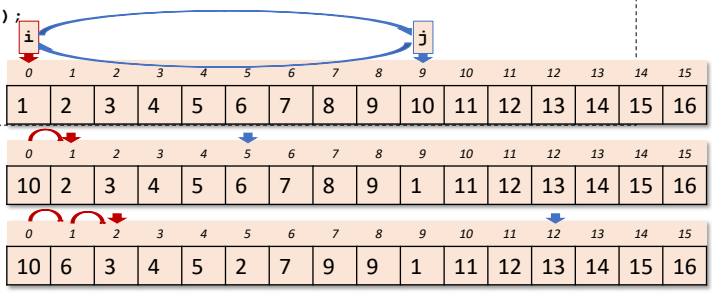
Il *gioco del 15*

```
29  bool schema15::genera(int seed)
30  {
31          bool v=false;
32          int tentativi=0;
33          for (int i=1;i<=16;i++)
34                  pad[i-1]=i;
35          num_mosse=0;
36          vX=vY=3;
37          do {
38                  mischia(seed);
39                  v=valido();
30                  tentativi++;
31          } while (!v&&tentativi<maxiter);
32          return v;
33  }
```

40

Il *gioco del 15*

```
34  void schema15::mischia(int seed)
35  {
36          int i,j,vpos=0;;
37          srand(seed);
38          for(i=0;i<16;i++) {
39                  j=rand()%16;
40                  if(pad[i]==16)
41                          vpos=j;
42                  if(pad[j]==16)
43                          vpos=i;
44                  swap(pad[i],pad[j]);
45          }
46          vX=vpos/4;
47          vY=vpos%4;
48  }
```



41

Il *gioco del 15*

Non tutti gli schemi sono risolvibili. Affinché una certa configurazione lo sia, è necessario che la «*somma del 15*» dia *risultato pari.*

sia una configurazione $\{b_1, \dots, b_{15}\}$ delle caselle:

per ogni i, sia nb_i il numero di caselle successive a b_i che abbiano valore minore di b_i

$$S_{15} = nb_1 + \cdots + nb_{15}$$

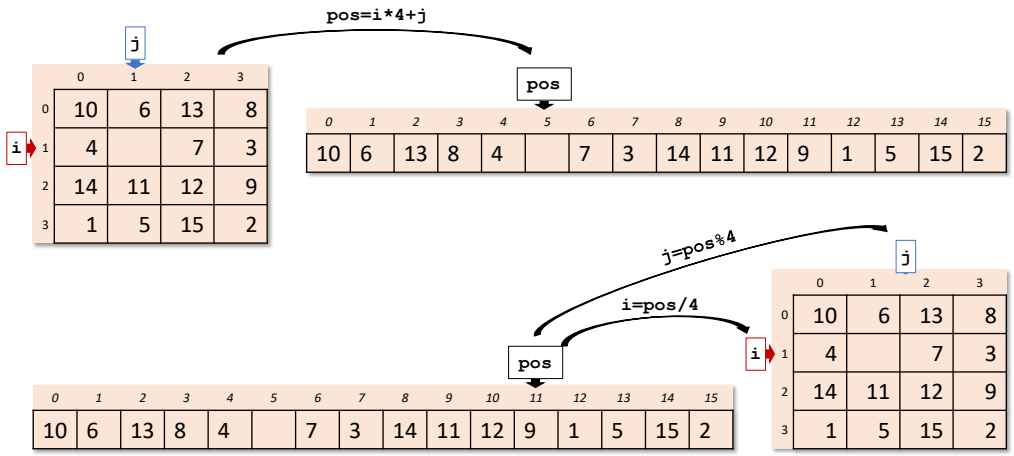
42

Il *gioco del 15*

```
49  bool schema15::valido()
50  {
51          int nb[16]={0};
52          int i,j,s15=0;
53
54          for(i=0;i<15;i++){
55                  for(j=i+1;j<16;j++)
56                          if(pad[j]<pad[i])
57                                  nb[i]++;
58                  s15+=nb[i];
59          }
60          return !(s15%2);
62  }
```

43

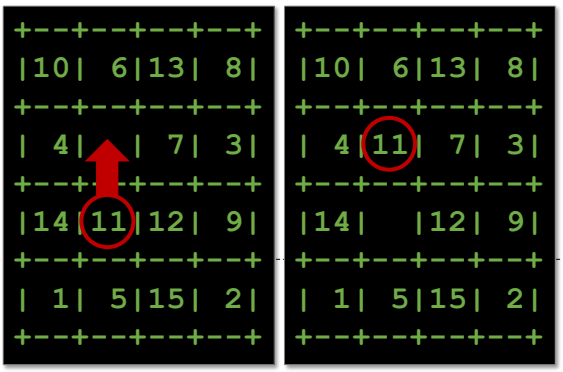
Il *gioco del 15*



44

Il *gioco del 15*

```
63  bool schema15::alto()
64  {
65          int i,j;
66          if(vX==3)
67                  return false;
68          i=vX+1;
69          j=vY;
70          swap(pad[i*4+j],pad[vX*4+vY]);
71          vX++;
72          num_mosse++;
73          return true;
74  }
```



45