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

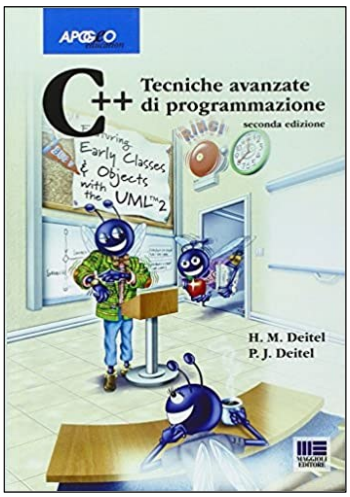
[FdP] H. M. Deitel, P. J. Deitel
C++ Fondamenti di programmazione
II ed. (2014) Maggioli Editore (Apogeo Education)
ISBN: 978-88-387-8571-9

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

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

6

Sovraccarico delle funzioni

7

Sovraccarico delle funzioni

In C++ è possibile definire funzioni diverse con lo stesso nome, purché abbiano una *firma* distinguibile:

elenco dei parametri diverso (nel numero, nel tipo o nell'ordine)

Questa caratteristica prende il nome di *ridefinizione* o *sovraccarico* della funzione.

Permette che funzioni che *«fanno la stessa cosa»* su dati diversi, possano essere chiamate *«nello stesso modo»*.

8

Sovraccarico delle funzioni

Un esempio: nella libreria matematica, lo standard del C++ richiede che ciascuna funzione sia sovraccaricata con i tipi **`float`**, **`double`** e **`long double`**

Ad esempio, nei sorgenti della libreria, saranno definite le seguenti funzioni:

```
float sin (float);
double sin (double);
long double sin (long double);
```

9

Sovraccarico delle funzioni

In fase di compilazione della libreria, ciascuna versione della funzione sovraccaricata viene rinominata in maniera univoca, a seconda dell'insieme degli argomenti che prende

Nel testo del programma che la utilizza, ciascun riferimento alla funzione sovraccaricata viene risolto con la versione giusta, scelta sempre in base all'insieme dei parametri reali della chiamata.

10

Sovraccarico delle funzioni

Ad esempio, nella libreria matematica, le tre versioni della funzione sovraccaricata **`sin()`** diventano *(qualcosa di simile a)*:

```
float       @sin$qf  (float);
double      @sin$qd  (double);
long double @sin$qld (long double);
```

Rimuovendo qualsiasi ambiguità in fase di *linking.*

11

Sovraccarico delle funzioni

Pertanto, il seguente codice..

```
float x=M_PI,y;
double w=0,z;
y=sin(x);
z=sin(w);
```

È «come se fosse»

```
float x=M_PI,y;
double w=0,z;
y=@sin$qf(x);
z=@sin$qd(w);
```

12

Sovraccarico delle funzioni

Pertanto, il seguente codice..

```
float x=M_PI,y;
double w=0,z;
y=sin(x);
z=sin(w);
```

Qui, **sin()** è risolta nella versione a parametro **float**, perché **x** è **float**…

È «come se fosse»

```
float x=M_PI,y;
double w=0,z;
y=@sin$qf(x);
z=@sin$qd(w);
```

13

Sovraccarico delle funzioni

Pertanto, il seguente codice..

```
float x=M_PI,y;
double w=0,z;
y=sin(x);
z=sin(w);
```

Qui, invece è risolta nella versione a parametro **double**, perché **w** è **double**

È «come se fosse»

```
float x=M_PI,y;
double w=0,z;
y=@sin$qf(x);
z=@sin$qd(w);
```

14

Sovraccarico delle funzioni (e dei metodi)

Anche le funzioni membro (o metodi) di una stessa classe possono essere sovraccaricati in maniera del tutto analoga alle funzioni.

In realtà – come si vedrà in seguito - per le classi, il C++ dà la possibilità di sovraccaricare persino gli *operatori* (aritmetici, logici, relazionali…).

15

Esempio: *angoli in gradi e radianti*

Modifichiamo la classe Angolo vista in precedenza per aggiungere la possibilità di impostare e leggere le ampiezze espresse in *radianti.*

1. Aggiungiamo un terzo costruttore: **`Angolo(double)`** che prende l'ampiezza del nuovo angolo da istanziare in radianti. Gli attributi **`gradi`**, **`primi`** e **`secondi`** saranno inizializzati utilizzando un apposita formula;
2. Aggiungiamo un nuovo metodo set(double) che prende il nuovo valore in radianti e ricava i corrispondenti valori per gli gli attributi **`gradi`**, **`primi`** e **`secondi`** dell'oggetto;
3. Aggiungiamo un nuovo metodo **`get()`** che restituisce il valore in radianti dell'angolo rappresentato dall'oggetto.

16

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  };
```

17

Esempio: *angoli in gradi e radianti*

file: angolo.cpp

```
48  void Angolo::set(int g, int p, int s)
49  {
50          setG(g);
51          setP(p);
52          setS(s);
53  }
54
55  void Angolo::set(double rad)
56  {
57          double g,p,s;
58          g=rad*180/M_PI;
59          gradi=floor(g);
60          rad=g-gradi;
61          p=rad*60;
62          primi=floor(p);
63          rad=p-primi;
64          secondi=floor(rad*60);
65          return;
66  }
```

Invocando (apparentemente) lo stesso metodo, si può modificare il valore dell'angolo fornendone il valore in radianti (double) o in gradi gradi, primi e secondi d'arco (int, int, int). Si tratta invece di due funzioni distinte che vengono identificate mediante la loro *firma*

Il metodo **set(double),** per impostare un nuovo valore all'angolo, ridefinisce (o sovraccarica) il metodo **set (int, int, int)**

18

Esempio: *angoli in gradi e radianti*

file: angolo.cpp

```
68  double Angolo::get()
69  {
70          double r=0;
71          r+=gradi*M_PI/180;
72          r+=(primi/60.0)*M_PI/180;
73          r+=(secondi/3600.0)*M_PI/180;
74          return r;
76  }
…   …
```

Il metodo **get()**, calcola l'ampiezza dell'angolo in radianti, partendo dalla sua misura in gradi:primi:secondi.

Per maggiori dettagli sulla misura degli angoli in radianti e sulla loro conversione da/in gradi, si veda:

- Misura angoli in radianti, "youmath.it"
- Voce "Radiante", Wikipedia.

19

Esempio: *angoli in gradi e radianti*

file: angolo_test.cpp

```
5   int main()
6   {
7           int g,p,s;
8           Angolo a,b,c,d;
9           double rad;
10          cout<<"Inserisci l'angolo A: ";
11          cin >> g >> p >> s;
12          a.set(g,p,s);
13          cout<<"Inserisci l'angolo B in radianti: ";
14          cin >> rad;
15          b.set(rad);
16          cout <<"Angolo B: ";
17          cout <<b.getG()<<"gradi, "<<b.getP()<<" primi e ";
18          cout <<b.getS()<<" secondi."<<endl;
19          cout <<"Angolo A:  ";
20          cout <<a.get()<<" radianti. "<<endl;
21  }
```

L'ampiezza di A è fornita in gradi e…

…visualizzata in radianti.

20

Esempio: *angoli in gradi e radianti*

file: angolo_test.cpp

```
5  int main()
6  {
7          int g,p,s;
8          Angolo a,b,c,d;
9          double rad;
10         cout<<"Inserisci l'angolo A: ";
11         cin >> g >> p >> s;
12         a.set(g,p,s);
13         cout<<"Inserisci l'angolo B in radianti: ";
14         cin >> rad;
15         b.set(rad);
16         cout <<"Angolo B: ";
17         cout <<b.getG()<<"gradi, "<<b.getP()<<" primi e ";
18         cout <<b.getS()<<" secondi."<<endl;
19         cout <<"Angolo A:  ";
20         cout <<a.get()<<" radianti. "<<endl;
21 }
```

L'ampiezza di B è fornita in radianti e…

…visualizzata in gradi.

21

Esercizio: *larde, piedi e pollici*

Sulla falsariga dell'esempio precedente, si implementi la classe *lunghezza,* nella quale; le misure sono espresse nel sistema anglosassone costituito da: iarde, piedi e pollici ma i valori possono essere assegnati e letti anche nel sistema metrico decimale.

1. Oltre al costruttore di default **`lunghezza()`**, si forniscano anche i costruttori **`lunghezza(double y, double f, double i)`** per le iarde, e **`lunghezza (double m)`** per i metri.
2. Si forniscano i metodi **`set(double m)`** e **`set (double y, double f, double i)`** per impostare i valori nei due sistemi di misura;
3. Si forniscano i metodi **`get/set`** degli attributi **`yard`**, **`feet`** e **`inches`** (tutti **`double`**)

22

Esercizio: *Iarde, piedi e pollici*

Sulla falsariga dell'esempio precedente, si implementi la classe *lunghezza,* nella quale; le misure sono espresse nel sistema anglosassone costituito da: yarde, piedi e pollici ma i valori possono essere assegnati e letti anche nel sistema metrico decimale.

Per maggiori dettagli sistema di misurazione delle distanze anglosassone (o *«imperiale»*) si veda:

- Voce "Iarda", Wikipedia.

scano anche i
uble i) per

2. Si forniscano i metodi **set(double m)** e **set (double y, double f, double i)** per impostare i valori nei due sistemi di misura;
3. Si forniscano i metodi **get/set** degli attributi **yard**, **feet** e **inches** (tutti **double**)

23

Template di funzioni

24

Template di funzioni

Con il sovraccarico, lo sviluppatore definisce diverse funzioni che effettuano operazioni «concettualmente» simili ma con implementazione differente e su tipi di dati diversi.

I *template di funzione (o funzioni generiche)* sono un meccanismo più compatto nel caso in cui occorra definire funzioni che effettuano le stesse operazioni su tipi diversi

Un template è uno schema *generale* di funzione in cui alcuni tipi sono «parametrizzati». Partendo dallo schema, il compilatore genera tutte le funzioni richieste nel codice a seconda dei tipi richiesti.

25

Template di funzioni

La definizione e il prototipo della funzione sono preceduti dalla dichiarazione

```
template <class T> // o <typename T>
```

Che indica il tipo-parametro **`T`** segue la definizione vera e propria

```
T somma (T p1, T p2) { … }
```

Il resto è del tutto analogo a una normale definizione di funzione.

I template di funzioni non ammettono *argomenti di default*

26

Template di funzioni

La definizione e il prototipo della f
dichiarazione

```
template <class T> // o <typename T>
```

Che indica il tipo-parametro **T** segue la definizione vera e propria

```
T somma (T p1, T p2) { … }
```

Il resto è del tutto analogo a una normale definizione di funzione.

I template di funzioni non ammettono *argomenti di default*

Parola chiave **template< >** dichiara una funzione template. Tra le parentesi angolate figurano i tipi-parametro, *i.e.* i tipi che possono variare… Qui, c'è il solo tipo **T**

27

Template di funzioni

La definizione e il prototipo della fun
dichiarazione

```
template <class T> // o <typename T>
```

Che indica il tipo-parametro **T** segue la definizione vera e propria

```
T somma (T p1, T p2) { … }
```

Il resto è del tutto analogo a una normale

I template di funzioni non ammettono *argo*

Parola chiave **template< >** dichiara una funzione template. Tra le parentesi angolate figurano i tipi-parametro, *i.e.* i tipi che possono variare… Qui, c'è il solo tipo **T**

Dati due parametri di un certo tipo **T**, questa funzione restituisce un valore dello stesso tipo. Il codice della funzione è scritto *«indipendentemente da **T**»*

28

Esempio: *template di funzione*

```
 5  template <typename T>
 6  T somma (T a, T b)
 7  {
 8          return a+b;
 9  }
10
11  int main()
12  {
13          int i1=10, i2=20;
14          string s1("Ciccio "), s2("Formaggio");
15
16          cout << "i1+i2="<< somma(i1,i2)<<endl<<endl;
17          cout << "s1+s2="<< somma(s1,s2)<<endl<<endl;
18  }
```

La funzione restituisce il risultato dell'espressione **a+b** «qualsiasi» sia il tipo **T** dei due operandi, e assumendo che il risultato sia dello stesso tipo.

29

Esempio: *template di funzione*

```
 5  template <typename T>
 6  T somma (T a, T b)
 7  {
 8          return a+b;
 9  }
10
11  int main()
12  {
13          int i1=10, i2=20;
14          string s1("Ciccio "), s2("Formaggio");
15
16          cout << "i1+i2="<< somma(i1,i2)<<endl<<endl;
17          cout << "s1+s2="<< somma(s1,s2)<<endl<<endl;
18  }
```

Qui il tipo **T** diventa **int**

Qui il tipo **T** diventa **string**

30

Esempio: *template di funzione #2*

```
5  template <typename T>
6  void swap (T &a, T &b)
7  {
8          T tmp;
9          tmp=a;
10         a=b;
11         b=tmp;
12 }
```

La funzione prende due riferimenti a variabili di un generico tipo **T** e non restituisce alcun valore.

Nel corpo della funzione possono essere definite variabili del tipo **T**

A prescindere da come sia istanziato **T** le espressione tra le variabili dichiarate di quel tipo devono essere *definite* e *coerenti*

31

Template di funzioni

A prescindere da come sia istanziato **T** le espressione tra le variabili dichiarate di quel tipo devono essere: *definite* e *coerenti*

definite: tutti gli operatori, i metodi e le funzioni utilizzati devono essere definiti per qualsiasi tipo **T** istanziato

coerenti: indipendentemente da **T** la valutazione delle espressioni devono produrre un risultato del tipo atteso

32

Esempio: *template di funzione #3*

```
 5  class intcounter {
 6          int cnt;
 7  public:
 8          intcounter() { cnt=0; };
 9          void inc(){    cnt++; };
10          int val() {    return cnt; };
11  };
12
13  class stringcounter {
14          string s;
15  public:
16          stringcounter(){ s=""; };
17          void inc() { s=s+"1";  };
18          int val() { return s.size(); };
19  };
```

33

Esempio: *template di funzione #3*

```
20  template <class T>
21  void printval(T x){
22          int r;
23          r=x.val()
24          cout << "Val="<< r <<endl;
25  }
26
27  int main(){
28          intcounter ic;
29          stringcounter sc;
30          ic.inc();
31          ic.inc();
32          printval(ic);
33          sc.inc();
34          printval(sc);
35  }
```

E' legittimo invocare la funzione **printval()** solo se la classe dell'oggetto **x** (quale che sia) fornisce il metodo **val()**

Il metodo **val()** deve restituire sempre un intero, indipendentemente da **T**

34

Esercizio: *scalari e vettori*

In una applicazione che tratta dati vettoriali, scriviamo delle funzioni che gestiscano l'I/O di vettori (**`vector`**) di valori numerici e una che implementi il prodotto di uno *scalare* per un vettore, entrambi di tipo numerico arbitrario…

```
void input (string messaggio, vettore &v)
void show  (string messaggio, vettore &v)
scalare scaXvett(scalare s, vettore &v)
```

35

Esercizio: *scalari e vettori*

Lo scalare e i componenti del vettore potrebbero essere di due tipi diversi, per esempio: **`int`** e **`int`**, **`int`** e **`float`**, **`double`** e **`float`** …

```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  template <class T, class U>
6  void scaXvett (T scalare, vector<U> &vettore)
7  {
8          T rv=0;
9          for (int i=0;i<vettore.size();i++)
10                 vettore[i]=vettore[i]*scalare;
11 }
```

36

Esercizio: *scalari e vettori*

L'argomento **messaggio** è semplicemente una stringa arbitraria da visualizzare per rendere più chiaro l'output.

```
12  template <class U>
13  void show (string messaggio, vector<U> vettore)
14  {
15          cout <<messaggio<<"["<<vettore.size()<<"]={";
16          for(int i=0;i<vettore.size();i++)
17                  cout <<vettore[i]<<" ";
18          cout << "}"<<endl;
19  }
20
21  template <class U>
22  void input (string messaggio, vector<U> &vettore)
23  {
24          cout << messaggio<<"["<<vettore.size()<<"]? ";
25          for(int i=0;i<vettore.size();i++)
26                  cin >>vettore[i];
27  }
```

Qui usiamo un *riferimento* al **vector vettore** perché intendiamo modificarne il contenuto

38

Esercizio: *scalari e vettori*

Il tipo degli scalari e dei componenti dei vettori, varia arbitrariamente a seconda della necessità dello sviluppatore,

```
28  int main()
29  {
30          int sca1=2;  double sca2=0.33;
31          vector<double> v1(3);  vector<int> v2(4);
32
33
34          input("immetti v1 (double)",v1);
35          input("immetti v2 (int)",v2); cout<<endl;
36
37          scaXvett(sca1,v1);
38          show("v1",v1);
39          scaXvett(sca1,v2);
40          show("v2",v2);
41          scaXvett(sca2,v2);
42          show("v2",v2);
43  }
```

Messaggi arbitrari (poteva anche esserci scritto «Pippo Baudo»)

39

Esercizio: *scalari e vettori*

```
28  int main()
29  {
30          int sca1=2;  double sca2=0.33;
31          vector<double> v1(3);  vector<int> v2(4);
32
33
34          input("immetti v1 (double)",v1);
35          input("immetti v2 (int)",v2); cout<<endl;
36
37          scaXvett(sca1,v1);
38          show("v1",v1);
39          scaXvett(sca1,v2);
40          show("v2",v2);
41          scaXvett(sca2,v2);
42          show("v2",v2);
43  }
```

Questo codice è scritto in maniera indipendente dai tipi che di volta in volta si scelgono per gli scalari e i vettori. Potenza dei **template**!

40

Template di funzioni

Durante la compilazione, la funzione viene *istanziata* in base all'analisi del tipo dei parametri (e del valore restituito).

```
template <class T> // o <typename T>
T foo (T p1, T p2) { … }
```

In alcuni casi, può essere necessario indicare esplicitamente il/i tipo/i richiesto/i:

```
rv=foo<int> (v1, v2);
```

41

Template di funzioni

Durante la compilazione, la funzione viene *istanziata* in base all'analisi del tipo dei parametri (e del valore restituito).

```
template <class T, class U>
T bar (U arg1, int arg2) { … }
```

In alcuni casi, può essere necessario indicare esplicitamente il/i tipo/i richiesto/i:

```
rv=bar<int,double> (v1, v2);
```

42

Esercizio: *scalari e vettori #2*

Scrivere la funzione che effettui il prodotto scalare di due vettori di tipo numerico arbitrario. Il tipo del valore restituito deve essere uguale a quello del primo vettore.

```
tipov1 PSVett(tipov1 v, tipov2 w)
```

Si ricordi che il prodotto scalare tra due vettori *v* e *w* di *n* elementi è dato dalla formula:

$$pv = \sum_{i=0}^{n-1} (v_i w_i)$$

43

Esercizio: *scalari e vettori #2*

```
5  template <…, …>
6  … PSVett (… v, … w)
7  {
8          … rv=0;
9          for (int i=0;i<v.size();i++)
10                 rv+=v[i]*w[i];
11         return rv;
12 }
```

Il codice *generale* per il calcolo del prodotto scalare dei due vettori (si assume che abbiano lunghezza uguale)

44

Esercizio: *scalari e vettori #2*

```
5  template <class T, class U>
6  T PSVett (vector<T> v, vector<U> w)
7  {
8          T rv=0;
9          for (int i=0;i<v.size();i++)
10                 rv+=v[i]*w[i];
11         return rv;
12 }
```

45

Overloading vs. Template

Funzioni «sovraccaricate»	**Template di funzioni**
Diverse funzioni con lo stesso nome (*firme diverse*)	*Una sola funzione* generica la cui *unica firma* segue uno schema fisso (ancorché parametrizzato)
Lo sviluppatore scrive il codice di *tutte* le funzioni di cui potrebbe aver bisogno	Lo sviluppatore scrive *una sola volta* il codice della funzione, purché «vada bene» per *qualsiasi* scelta legittima dei tipi parametrizzati
Tutte le funzioni (*anche quelle non utilizzate*) «fanno parte» del programma compilato.	Il compilatore , in base al template, genera automaticamente *tutte e sole* le implementazione del template necessarie ad eseguire le chiamate nel codice
Se la funzione viene invocata con una firma che *non corrisponde a nessuna delle sue implementazioni*, la compilazione cessa con un errore	Se la funzione viene invocata *senza rispettare lo schema*, la compilazione cessa con un errore

46

La classe *template* **`vector`**

47

La classe *template* **`vector`**

La classe **`vector`** come una alternativa evoluta agli array, per moltissime applicazioni

Come la classe **`string`**, **`vector`** non è un tipo di dato nativo, ma è una classe implementata in C++ e disponibile nella dotazione di librerie standard di questo linguaggio.

La classe **`vector`** è fornita, insieme a moltissime altre, dalla libreria *Standard Template Library (STL)*

48

La classe *template* **`vector`**

Gli oggetti **`vector`** superano alcune limitazioni degli array:

- Conservano la conoscenza della loro dimensione
- Verificano il rispetto delle dimensioni del vettore in ogni accesso.
- Possono essere confrontati tra loro (e.g. con **==** e **!=**)
- E' possibile effettuare assegnamenti tra vettori

49

La classe *template* **vector**

La classe **vector** utilizza i *template*

Dichiariamo un array di interi utilizzando i **vector**

```
vector<int> estrazione(6);
```

Si dichiara un oggetto della classe **vector** di nome **estrazione** composto da sei elementi di tipo **int**

```
vector<punto> ottagono(8);
```

Si dichiara un **vector** composto da otto elementi di classe **punto**

50

La classe *template* **vector**

Nonostante l'aspetto «esotico», si tratta di una normale dichiarazione di variabile…

TIPO: Classe template + <tipo base>

```
vector<punto> ottagono(8);
```

Si dichiara un **vector** composto da otto elementi di classe **punto**

TIPO: Classe template + <tipo base>

51

La classe *template* **vector**

Dichiariamo un array di interi utilizzando i **vector**

```
#include<vector>
using std::vector;
vector<int> estrazione1(6), estrazione2(6);
```

Lunghezza di un oggetto **vector**

```
cout << "Lunghezza: " << estrazione1.size();
```

52

La classe *template* **vector**

Input/ output dati:

```
for(i=0;i<estrazione1.size();i++)
      cin >> estrazione1[i];
```

Assegnamento tra vettori:

```
estrazione2=estrazione1
```

L'assegnamento può essere fatto tra due vettori di lunghezza diversa. In questo caso, la dimensione di quello ricevente viene adattata per contenere i dati necessari.

53

La classe *template* **vector**

Confronto:

```
if (estrazione1==estrazione2)
     cout << "sono uguali! " << endl;
```

Inizializzazione (con un altro vettore):

```
vector<int> estrazioni3(estrazioni1);
```

Distruzione:

```
delete estrazioni2;
```

54

Esercizio: *bussolotto #2*

Scrivere una versione della classe **bussolotto** del tutto analoga alla precedente ma:

Che utilizzi i **vector** invece degli array

55

Esercizio: *bussolotto #2*

```
 1  class bussolotto {
 2          private:
 3                  vector<dado> *p;
 4                  int numdadi;
 5          public:
 6                  bussolotto(int num=2, int nf=6){
 7                          p=new vector<dado>(num,nf);
 8                          numdadi=p->size();
 9                  };
10                  ~bussolotto(){
11                          delete p;
12                  }
13                  void agita(int seed);
 …  …
```

Alloca un vettore di **num** oggetti di tipo **dado**. Per istanziare ciascun oggetto utilizza il costruttore **dado(nf)**

56

Esercizio: *bussolotto #2*

```
 …  …
14                  int lancio(int num=0) {
15                          int nlanci, sum=0;
16                          if (num<1||num>numdadi)
17                                  nlanci=numdadi;
18                          else
19                                  nlanci=num;
20                          for(int i=0;i<nlanci;i++)
21                                  sum+=p->at(i).lancio();
22                  // oppure:  sum+=(*p)[i].lancio();
23                          return sum;
24                  }
    };
```

Il metodo **at()**, è equivalente a **[]** ma è più comodo se applicato a un puntatore

p è un puntatore a un vettore di **dado**. Occorre prima referenziarlo, per poi accedere ai singoli elementi.

57

La classi *container*

La classe **`vector`** rientra in una famiglia di classi dette *container*

Si tratta di classi il cui scopo è contenere *collezioni* di oggetti e una serie di strumenti per effettuare operazioni come: inserimento, cancellazione, ricerca di elementi, ordinamento…

Trattandosi di operazioni che non richiedono che l'uso di pochissimi metodi e operatori comuni a qualunque oggetto/tipo «collezionabile» (*e.g.* costruttori/distruttori, assegnamento, confronto) sono generalmente implementate come *template*

58

La classi *container*

La libreria STL fornisce molte utilissime classi *container* divise in categorie a seconda del modo in cui organizzano i dati

Contenitori sequenziali: tra cui **`vector`**, **`array`**, **`list`**;

Contenitori associativi: tra cui **`map`** e **`set`**;

59