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

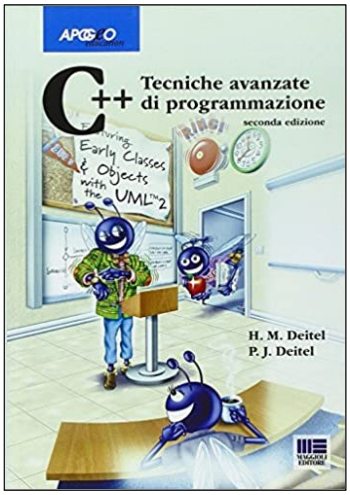
2

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

Sovraccarico delle funzioni

7

Sovraccarico delle funzioni

In C++ è possibile definire funzioni diverse con lo stesso nome, purchè 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

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 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 (e demoni)*

```
 5  class gradi {
 6          private:
 7                  int gr;
 8                  int pri;
 9                  double sec;
10          public:
11                  gradi(int g=0, int p=0, double s=0) {
12                          set(g,p,s);
13                  };
14                  int     getG() { return gr;  };
15                  int     getP() { return pri; };
16                  double  getS() { return sec; };
17                  void    set(int g, int p, double s){
18                          gr=g; pri=p; sec=s;
19                  };
20  };
```

La classe **gradi** rappresenta valori espressi nella notazione sessagesimale (gradi angolari)

16

Esempio: *angoli (e demoni)*

```
21  class angolo {
22          private:
23                  double rad;
24          public:
25                  angolo(double r=0){
26                          rad=r;
27                  };
28                  double get();
29                  void set(double val);
30                  void set(gradi g);
31                  void set(int g, int p, double s);
32
33  };
34
35  double angolo::get() {
36          return rad;
37  }
```

La classe **angolo** rappresenta gli angoli in un piano espressi in *radianti*.

Il metodo **set()**, per impostare un nuovo valore all'angolo, è sovraccaricato.

17

Esempio: *angoli (e demoni)*

```
38  void angolo::set(double val) {
39          if ((val<0)||(val>(2.0*M_PI)))
40                  rad=0;
41          else
42                  rad=val;
43  }
44  void angolo::set(gradi g) {
45          set (g.getG(),g.getP(),g.getS());
46  }
47  void angolo::set(int g, int p, double s){
48          double r=0;
49          r+=g*M_PI/180;
50          r+=(p/60)*M_PI/180;
51          r+=(s*3600)*M_PI/180;
52          rad=r;
53  }
```

Invocando (apparentemente) lo stesso metodo, si può modificare il valore dell'angolo fornendone il valore in radianti (**double**), in **gradi** oppure indicandone l'ampiezza con i tre ordini di grandezza dei gradi angolari: gradi, primi e secondi d'arco (**int, int, double**)…

Le tre funzioni condividono lo stesso nome ma hanno tre *firme* diverse.

18

Esempio: *angoli (e demoni)*

```
55 void show(angolo x) {
56                 double ris;
57                 ris=x.get();
58                 cout<<"l'angolo misura: "<<ris<<" (circa "<<ris/M_PI<<
59 "pi)"<<endl;
60 }
61
62 void show(gradi x){
63         cout<<"misura: "<<x.getG()<<":"<<x.getP()<<":"<<x.getS() <<endl;
64 }
```

Un esempio di funzione sovraccaricata. La funzione **show(),** è ridefinita per visualizzare il valore espresso dagli oggetti di classe **angolo** o di classe **gradi**

19

Esempio: *angoli (e demoni)*

```
65 int main(){
66         angolo a,b,c;
67         gradi m(57,0,0.0);
68         int g,p;
69         double s,ris;
70         a.set(6.28);
71         show(a);
72         cout<<"Inserisci l'angolo b in gradi, primi
73         cin>>g>>p>>s;
74         b.set(g,p,s);
75         ris=b.get();
76         cout << "b in radianti:" << ris <<" (circa "<<ris/M_PI<< "pi)"<<endl;
77         c.set(m);
78         show(m);
79         show(c);
80 }
```

Il metodo **set()**, della classe **angolo** è invocato di volta in volta con argomenti di tipo diverso. Il codice da eseguire è scelto in base a numero e tipo dei parametri reali.

20

Esempio: *angoli (e demoni)*

```
65  int main(){
66          angolo a,b,c;
67          gradi m(57,0,0.0);
68          int g,p;
69          double s,ris;
70          a.set(6.28);
71          show(a);
72          cout<<"Inserisci l'angolo b in gradi, primi e secondi:";
73          cin>>g>>p>>s;
74          b.set(g,p,s);
75          ris=b.get();
76          cout << "b in radianti:" << ris <<" (circa "<<ris/M_PI<< "pi)"<<endl;
77          c.set(m);
78          show(m);
79          show(c);
80  }
```

21

Template di funzioni

22

Template di funzioni

Con il sovraccarico, lo sviluppatore definisce diverse funzioni che effettuano operazioni molto simili ma con logiche differenti 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.

23

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*

24

Template di funzioni

La definizione e il prototipo della f
dichiarazione

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

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`**

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*

25

Template di funzioni

La definizione e il prototipo della fun
dichiarazione

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

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`**

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

```
T somma (T p1, T p2) { … }
```

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`**»*

Il resto è del tutto analogo a una normale

I template di funzioni non ammettono *argo*

26

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.

27

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**

28

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*

29

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 il tipo **T** istanziato

coerenti: indipendentemente da **T** le espressioni devono avere valore del tipo atteso

30

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  };
```

31

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  }
```

Gli oggetti di tipo **T** (quale che sia) devono avere un metodo **val()**

Il metodo **val()** restituisce sempre un intero

32

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)
```

33

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 }
```

34

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

36

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»)

37

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**!

38

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);
```

39

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);
```

40

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:

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

41

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)

42

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 }
```

43