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

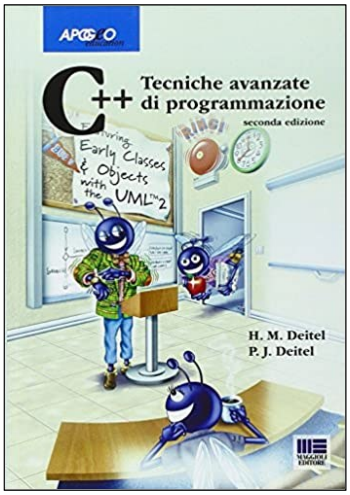
2

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)

5

Il Linguaggio C++

(per programmatori C)

Parte prima

6

Tipi di dati aggregati

7

Tipi di dati aggregati

- Definizione di nuovi tipi di dato «combinando» i tipi già esistenti
- Il C/C++ fornisce tre metodi di aggregazione:
  - **`struct:`**
    - Raggruppano in un'unica struttura (record) diverse variabili (campi);
    - I campi sono accessibili singolarmente e indipendentemente
    - La memoria occupata da una struttura è pari alla somma della memoria occupata dai suoi campi
  - **`union:`**
  - **`class:`**

8

Tipi di dati aggregati

- Definizione di nuovi tipi di dato «combinando» i tipi già esistenti
- Il C/C++ fornisce tre metodi di aggregazione:
  - **`struct:`**
  - **`union:`**
    - Permettono di utilizzare una stessa area di memoria con diverse modalità
    - I «campi» di una union «insistono» sulla stessa area di memoria pertanto, modificando uno di essi, si producono modifiche anche negli altri
    - La quantità di memoria utilizzata da una union è pari alla dimensione del più grande dei suoi campi
  - **`class:`**

9

Tipi di dati aggregati

- Definizione di nuovi tipi di dato «combinando» i tipi già esistenti
- Il C/C++ fornisce tre metodi di aggregazione:
  - **`struct:`**
  - **`union:`**
  - **`class:`**
    - Evoluzione della struct finalizzata alla *programmazione orientata agli oggetti (OOP)*
    - Definizione di operatori sui nuovi tipi (con accesso «esclusivo» ai campi)
    - Ridefinizione degli operatori nativi (`'+'`, `'--'`, `'%'`) sui record

10

Le **`struct`** in C/C++

11

Esempio: un tipo per i punti sul piano

- Un punto p sul piano cartesiano è rappresentato da due numeri reali che ne costituiscono le coordinate x e y
- La distanza d tra due punti $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ è:

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- Realizziamo un programma C++ che implementi il tipo *«punto»* e definisca su due punti la funzione *«distanza»* a valori reali;
  - Per la radice quadrata e l'elevamento a potenza utilizziamo funzioni della libreria matematica standard

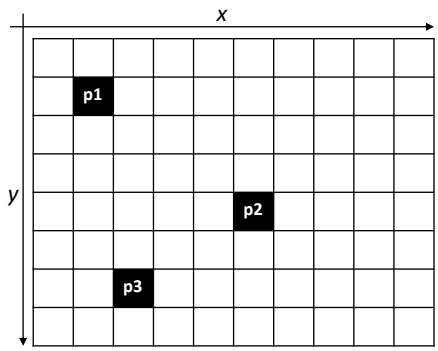
12

Esempio: un tipo per i punti sul piano

Si *definisce* la struttura punto fornendone una descrizione, i.e. l'elenco dei *campi* che contiene.

```
struct punto {
      double x;
      double y;
};
```

La *definizione* della **`struct`** non riserva alcun spazio in memoria (allocazione) per l'aggregato **`punto`**, né per i due double di cui è composto.



13

Esempio: un tipo per i punti sul piano

Si *definisce* la struttura punto fornendone una descrizione, i.e. l'elenco dei *campi* che contiene.

```
struct punto {
      double x;
      double y;
};
```

La *definizione* della struct non riserva alcun spazio in memoria (allocazione) per l'aggregato punto, né per i due double di cui è composto.

14

Esempio: un tipo per i punti sul piano

In seguito alla definizione, si introduce un nuovo *tipo di dato* nel quale è possibile dichiarare nuove variabili.

```
struct punto p1, p2, p3 = { 0.5, 5.7 };
```

Nella dichiarazione con inizializzazione, si forniscono i valori di tutti i campi della **`struct`**, rispettandone il tipo e l'ordine.

Le variabili **`p1`**, **`p2`** e **`p3`** rappresentano coppie di double in memoria aggregate secondo lo schema di definizione della **`struct punto`**

15

Esempio: un tipo per i punti sul piano

Per estrarre il valore dei singoli campi di una **`struct`**, o per assegnarvi nuovi valori, si utilizza l'operatore di accesso **`.`** (punto);

Identif_var.*nome_campo*;

```
p1.x = 2;
p1.y = 2;

p2.x=p1.x;
p2.y = 2*p1.y;
```

16

Esempio: *distanza tra due punti*

```
#include<iostream>
#include<cmath>

using namespace std;

struct punto {
       double x;
       double y;
};

double distanza(struct punto, struct punto);
```

Include le dichiarazioni e definizioni delle funzioni matematiche «standard» del C++

Definiamo la il tipo «punto» aggregando due reali «double» che rappresentano le coordinate x e y

Dichiarazione della funzione «distanza» di due variabili «punto» a reali in doppia precisione

17

Esempio: *distanza tra due punti*

```
int main()
{
        struct punto p1,p2;

        cout << "Inserire le coordinate di p1: "<<endl;
        cout << "p1.x: ";
        cin >> p1.x;
        cout << "p1.Y: ";
        cin >> p1.y;

        cout << "Inserire le coordinate di p2: "<<endl;
        cout << "p2.x: ";
        cin >> p2.x;
        cout << "p2.y: ";
        cin >> p2.y;

        cout << "La distanza tra i due punti e': " << distanza(p1,p2) << endl;
}
```

Dichiarazioni di variabili «struct punto»: **p1** e **p2** sono gli identificatori delle variabili e sono del tutto equivalenti a quelli delle variabili di tipo «nativo»

Per accedere a ciascun campo di una variabile «struct» si usa l'operatore '**.**'

p2.y è l'identificatore del campo **y** della variabile **p2**

18

Esempio: *distanza tra due punti*

```
double distanza (struct punto p1, struct punto p2)
{
                return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
}
```

```
Inserire le coordinate di p1:
p1.x: 1
p1.Y: 1
Inserire le coordinate di p2:
p2.x: 6
p2.y: 13
La distanza tra i due punti e': 13
```

19

Esercizio: *confronto tra due punti*

Con riferimento alla **`struct punto`** appena definita, si scriva un programma C++ che:
1) definisca la seguente funzione:

```
bool piu_a_destra (struct punto p1, struct punto p2)
```

*(Che risulti vera se **`p1.x > p2.x`** o falsa altrimenti)*

2) Chieda all'utente di immettere le coordinata di due punti **a** e **b** e dica se **a** è più a destra di **b**;

20

Esercizio: *confronto tra due punti*

```
 1  #include<iostream>
 2  using namespace std;
 3
 4  struct punto {
 5        double x;
 6        double y;
 7  };
 8
 9  bool piu_a_destra(struct punto p1, struct punto p2)
10  {
11        return p1.x > p2.x;
12  }
```

21

Esercizio: *confronto tra due punti*

```
13  int main(){
14          struct punto a,b;
15
16          cout << "Inserisci le coordinate di a: ";
17          cin >> a.x >> a.y;
18          cout << "Inserisci le coordinate di b: ";
19          cin >> b.x >> b.y;
20
21          if (piu_a_destra(a,b))
22                  cout << "il punto a è più a destra di b"<<endl;
23          else
24                  cout << "forse il punt a è alla sinistra di b"<<endl;
25  }
```

22

Definizione di nuovi tipi: **`typedef`**

- La keyword **`typedef`** permette la definizione di nuovi tipi di dato
  - In realtà il nuovo tipo è un «sinonimo» di un tipo già esistente

```
typedef <tipo-esistente> <newtyp>;
```

- Alcuni esempi:

Definizione	Dichiarazione variabili	Impiego
`typedef int intero;`	`intero a=0,b=100;` `int c;`	`c=a+b;`

`typedef char parola10[10];`	`parola10 nome, cognome;`
	`char c='z';`
		`cin >> nome;`
		`cin >> cognome;`
		`cout << nome << cognome <<endl;`
		`nome[1]=c;`

23

Definizione di nuovi tipi: **typedef**

- Nel nostro programma, potremmo definire il sinonimo PUNTO per la struttura punto, in questo modo:

```
typedef struct punto {
      double x;
      double y;
} PUNTO;
```

- Le dichiarazioni coinvolte diverrebbero:

```
double distanza(PUNTO, PUNTO);
…
PUNTO p1,p2;
…
double distanza(PUNTO p1, PUNTO p2){…}
```

24

Definizione di nuovi tipi: **typedef**

- Nella prassi, la definizione di **struct** e il **typedef** sono spesso utilizzate insieme, in tal caso, si procede nel modo seguente:

```
typedef struct {
     double x;
     double y;
} PUNTO;
```

- In questo caso, la struttura si dice «anonima»
- La dichiarazione di variabili di questo tipo può avvenire solo utilizzando l'identificatore di tipo «PUNTO»;

25

Esercizio: *e ora… il perimetro!*

- Utilizzando il codice dell'esempio precedente, si scriva un programma C++ che:
  - chieda in input le coordinate dei tre vertici v1, v2 e v3 di un triangolo
  - fornisca in output il suo perimetro;
    - Utilizzando la funzione distanza, si definisca la funzione perimetro che prende i tre vertici di un triangolo e ne calcola il perimetro distanza :

```
double perimetro(PUNTO v1, PUNTO v2 , PUNTO v3)
…
```

26

Esercizio: *e ora… il perimetro!*

```
1  #include<iostream>
2  #include<cmath>
3  using namespace std;
4
5  typedef struct {
6        double x;
7        double y;
8  } PUNTO;
9
```

27

Esercizio: *e ora… il perimetro!*

```
10  double distanza (PUNTO p1, PUNTO p2)
11  {
12          return sqrt(pow((p2.x-p1.x),2)+pow((p2.y-p1.y),2));
13  }
14
15  double perimetro (PUNTO a, PUNTO b, PUNTO c)
16  {
17          return distanza(a,b)+distanza(b,c)+distanza(c,a);
18  }
```

28

Esercizio: *e ora… il perimetro!*

```
19  int main()
20  {
21          PUNTO a,b,c;
22          double p;
23
24          cout << "Inserire le coordinate di a: "<<endl;
25          cout << "a.x: ";
26          cin >> a.x;
27          cout << "a.y: ";
28          cin >> a.y;
..  …
45
46          p=perimetro (a,b,c);
47          cout << "il perimetro di del triangolo abc e': " << p << endl;
48
49  }
```

29

Esercizio: *e ora… il perimetro!*

```
Inserire le coordinate di a:
a.x: 0
a.y: 0
Inserire le coordinate di b:
b.x: 0
b.y: 2
Inserire le coordinate di c:
c.x: 2
c.y: 0
il perimetro di del triangolo abc e':
6.82843
```

30

Esempio: *un mazzo di carte napoletane*

La realizzazione di una applicazione complessa, passa necessariamente per la progettazione delle strutture dati che essa dovrà trattare.

Supponiamo di sviluppare una app per giocare a Tressette. Ci occorre rappresentare:

Le singole carte, aggregando due informazione: il *valore* e il *palo* (il *seme*);

Il mazzo di carte, come un array di 40 carte;

31

Esempio: *un mazzo di carte napoletane*

```
1  #include<iostream>
2  #include<cstdlib>
3  #include<iomanip>
4  using namespace std;
5
6  const int BASTONI=0, COPPE=1, DENARI=2, SPADE=3;
7
8  typedef struct {
9         int palo;
10        int valore;
11 } CARTA;
12
13 Typedef CARTA[40] MAZZO40;
```

Una possibilità tra le tante: rappresentiamo il *palo* delle carte con quattro costanti intere.

Definiamo il tipo MAZZO40 quale «sinonimo» di una array di quaranta elementi CARTA.
Si ricordi che malgrado il cambio di nome, una variabile di tipo MAZZO40 è in tutto e per tutto un array di 40 elementi.

32

Esercizio: *Carte napoletane*

In relazione alla rappresentazione delle carte napoletane vista in precedenza, scrivere una funzione C++ che visualizzi il contenuto di una carta.

Per esempio: data le carte X e Y

```
CARTA x = {SPADE,5}, Y={BASTONI,10};
```

La funzione **`void mostra_carta (CARTA)`** scriverà a schermo:

```
[ 5   Spade]
[10 Bastoni]
```

33

Esercizio: *Carte napoletane*

```
14  void mostra_carta(CARTA x){
15          cout << "[" << setw(2)<< right << x.valore << setw(8);
16          switch(x.palo)
17          {
18                  case BASTONI:
19                          cout<<"Bastoni";
20                          break;
21                  case COPPE:
22                          cout<<"Coppe";
23                          break;
24                  case DENARI:
25                          cout<<"Denari";
26                          break;
27                  case SPADE:
28                          cout<<"Spade";
29          }
30          cout << "]" << endl;
31  }
```

34

Esercizio: *Carte napoletane*

Data il mazzo di carte m

```
MAZZO40 m;
```

Si scriva la funzione **`void mischia_mazzo(MAZZO40)`** che:

Inizializzi ciascuna carta **`m[i]`** (per **`i`** che va da 0 a 39), impostando correttamente i campi **`m[i].palo`** e **`m[i].valore`**

35

Esercizio: *Carte napoletane*

Data il mazzo di carte m

```
MAZZO40 m;
```

Si scriva la funzione **`void mischia_mazzo(MAZZO40)`** che:

Inizializzi ciascuna carta **`m[i]`** (per **`i`** che va da 0 a 39), impostando correttamente i campi **`m[i].palo`** e **`m[i].valore`**

Mischi il mazzo di carte, in modo che siano disposte nell'array in ordine causale

36

Esercizio: *Carte napoletane*

Suggerimenti:

1) Com'è noto, il mazzo deve contenere 10 carte (numerate da 1 a 10) per ciascun seme (numerato da 0 a 3).
2) Una volta inizializzato l'array, occorre mescolarne le carte. Una buona idea procedura potrebbe essere…

37

Esercizio: *Carte napoletane*

Suggerimenti:

1) Com'è noto, il mazzo deve contenere 10 carte (numerate da 1 a 10) per ciascun seme (numerato da 0 a 3).
2) Una volta inizializzato l'array, occorre mescolarne le carte. Una buona idea procedura potrebbe essere:
   - *Per ogni* ***i*** *da 0 a 39:*
     scegliere casualmente un ***j*** *(compreso tra 0 e 39);*
     effettuare lo scambio tra ***m[i]*** *e* ***m[j];***
     passare al prossimo ***i;***

38

Esercizio: *Carte napoletane*

```
32  void mischia_mazzo(MAZZO40 m)
33  {
34          int segnalino=0,altracarta;
35          CARTA temp;
36
37          for(int i=BASTONI;i<=SPADE;i++)
38                  for(int j=0;j<10;j++){
39                          m[segnalino].palo=i;
40                          m[segnalino].valore=j+1;
41                          segnalino++;
42                  }
43
```

39

Esercizio: *Carte napoletane*

```
44          for(int i=0;i<40;i++){
45                  altracarta=rand()%40;
46                  temp=m[i];
47                  m[i]=m[altracarta];
48                  m[altracarta]=temp;
49          }
50  }
```

40

Esercizio: *Carte napoletane*

```
51  int main()
52  {
53          MAZZO40 mazzo;
54          unsigned int seed;
55
56          cout << "Inserisci il seed: ";
57          cin >> seed;
58          srand(seed);
59
60          mischia_mazzo(mazzo);
61          cout << "Prima carta:" ;
62          mostra_carta(mazzo[0]);
63  }
```

41

Le **union** in C/C++

42

Le **union** in C/C++

Per certi aspetti, la **union** è molto simile alla **struct**.

```
union angolo {
      double radianti;
      int gradi[3];
};
```

Tuttavia, a differenza della **struct**, la **union** i custodisce *un solo campo alla volta*. Pertanto, se del codice utilizza uno campo, non può utilizzare gli altri

43

Esempio: *impiego delle* **union**

```
1  #include<iostream>
2  using namespace std;
3
4  union angolo {
5         double rad;
6         int gradi[3];
7  };
```

44

Esempio: *impiego delle* **union**

```
8  int main()
9  {
10        union angolo x, y;
11        cout << "inserisci x in radianti: ";
12        cin >> x.rad;
13
14        cout << "inserisci y in gradi, primi e se
15        cin >> y.gradi[0]>>y.gradi[1]>>y.gradi[2];
16
17        cout << "l'angolo y è di:" << y.rad <<endl;
18 }
```

Se abbiamo inserito i valori nel campo gradi di y, l'accesso al campo rad per la lettura, darà risultati incoerenti. Se abbiamo scelto di usare un campo…

45

Esempio: *impiego delle* **union**

```
 8  int main()
 9  {
10          union angolo x, y;
11          cout << "inserisci x in radianti: ";
12          cin >> x.rad;
13
14          cout << "inserisci y in gradi, primi e se
15          cin >> y.gradi[0]>>y.gradi[1]>>y.gradi[2];
16
17          cout << "l'angolo y è di:" << y.rad <<endl;
18  }
```

Se abbiamo inserito i valori nel campo gradi di y, l'accesso al campo rad per la lettura, darà risultati incoerenti. Se abbiamo scelto di usare un campo…

```
inserisci x in radianti: 6.28
inserisci y in gradi, primi e secondi: 13 4 5
l'angolo y è di:8.48798e-314
```


Le **union** in C/C++

Permettono di utilizzare una stessa area di memoria con diverse modalità

I «campi» di una union «insistono» sulla stessa area di memoria pertanto, modificando uno di essi, si producono modifiche anche negli altri

La quantità di memoria utilizzata da una union è pari alla dimensione del più grande dei suoi campi

Le **`union`** in C/C++

Le **`union`**, sono utilizzate piuttosto raramente.

Generalmente sono impiegate nell'implementazione delle strutture dati interne dei sistemi operativi, nell'implementazione dei protocolli di rete e dei device driver.

48