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

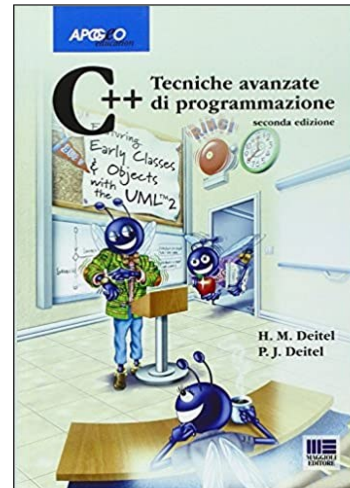
Descrizione del Corso

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

Array e puntatori

7

Array e puntatori

Array e puntatori sono strettamente correlati in C++ e possono essere utilizzati in maniera *quasi* equivalente;

- Il nome di un array è in realtà un puntatore *costante*
- I puntatori possono essere utilizzati in tutte le operazioni che coinvolgano gli indici di un array *(anche con la stessa sintassi)*

8

Array e puntatori

Il nome di un array è in realtà un puntatore *costante* che punta al suo primo elemento

```
float farr[100]={0.1} // il resto tutto a 0
float *fPtr;

fPtr=farr;
cout << *fPtr << endl;
```

9

Esempio: *array e puntatori #1*

```
1  #include<iostream>
2  using namespace std;
3
4  int main()
5  {
6          const int MAX=3;
7          int arr[MAX]={1,2,3}, *arrPtr;
8
9          arrPtr = arr;
10
11         cout << "arr[0]=" << arr[0]<<endl;
12         cout << "*arrPtr=" << *arrPtr << endl;
13         cout << "*arr=" << *arr <<endl;
14
15 }
```

```
arr[0]=1
*arrPtr=1
*arr=1
```

10

Array e puntatori

Mediante l'*aritmetica dei puntatori*, è possibile ispezionare una struttra dati (*e.g.* un array)

```
float farr[100]={0.1,4,1.6} //il resto
float *fPtr;

fPtr=farr;
cout << "farr[1]=" <<  *(fPtr+1) << endl;
cout << "farr[2]=" <<  *(fPtr+2) << endl;
```

Per raggiungere il secondo elemento (**farr[i]**) sommiamo **i** a **fPtr**. La quantità **i** è detta *offset*

```
4
1.6
```

11

Esempio: *array e puntatori #2*

```
1   #include<iostream>
2   using namespace std;
3
4   int main()
5   {
6           const int MAX=3;
7           int main_array[MAX]={1,2,3}, *arrayPtr;
8           arrayPtr=main_array;
9
10          for (int i=0;i<MAX;i++) {
11                  cout << "arrayPtr+" << i << "=" << *arrayPtr << endl;
12                  arrayPtr++;
13          }
14          cout << endl;
15  }
```

Qui, ogni volta, **arrayPtr** è incrementato di un offset pari a 1

12

Esempio: *array e puntatori #3*

```
1   #include<iostream>
2   using namespace std;
3
4   void somma10(int *fun_array, int size){
5           for(int i=0;i<size;i++){
6                   *fun_array+=10;
7                   fun_array++;
8           }
9   }
10  int main()
11  {
12          const int MAX=3;
13          int main_array[MAX]={1,2,3};
14          somma10(main_array,MAX)
15          for (int i=0;i<MAX;i++)
16                  cout << "main_array["<<i<<"]="<<main_array[i]<<endl;
17  }
```

Chiamo la funzione **somma10**, passandole l'indirizzo base dell'array e la sua lunghezza.

13

Esempio: *array e puntatori #3*

```
1  #include<iostream>
2  using namespace std;
3
4  void somma10(int *fun_array, int size){
5          for(int i=0;i<size;i++){
6                  *fun_array+=10;
7                  fun_array++;
8          }
9  }
10 int main()
11 {
12         const int MAX=3;
13         int main_array[MAX]={1,2,3};
14         somma10(main_array,MAX)
15         for (int i=0;i<MAX;i++)
16                 cout << "main_array["<<i<<"]="<<main_array[i]<<endl;
17 }
```

Inizialmente **fun_array**, punta alla *base* dell'array (al primo elemento);

Ad ogni iterazione sommiamo 10 al contenuto dell'elemento dell'array puntato da **fun_array;**

«spostiamo» **fun_array** sull'elemento successivo...

Aritmetica dei puntatori

Operazioni consentite con gli operatori

```
float *fPtr;
```

Incremento e decremento (*offset = 1*) prefissi o postfissi;

```
++fPtr; --fPtr;
fPtr++; fPtr--;
```

Occorre fare attenzione a che l'indirizzo puntato non oltrepassi limiti inferiore e superiore della struttura dati «spazzata» da fPtr; (*e.g.* per gli array, fPtr non deve essere minore dell'indirizzo base, né superiore all'indirizzo dell'ultimo elemento;

Aritmetica dei puntatori

Operazioni consentite con gli operatori

```
int i=1;
float *fPtr;
```

Somma di costanti o espressioni a valori interi;

```
fPtr=fPtr-1;
fPtr+=2*i;
```

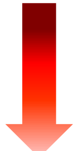
Occorre fare attenzione a che l'indirizzo puntato non oltrepassi limiti inferiore e superiore della struttura dati «spazzata» da fPtr; (*e.g.* per gli array, fPtr non deve essere minore dell'indirizzo base, né superiore all'indirizzo dell'ultimo elemento;

16

Precedenza tra gli operatori

Priorità

maggiore



minore

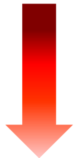
Operatore	Descrizione
`()`, `[]`, `.`*(punto)*, `->`	funzione, accesso array, struct, union e classi
`++`, `--`, …	unari incremento postfissi
`++`, `--`, `&`, `*`, `+`, `-`, `!`	unari incremento *(de)*referencing, prefissi, segno algebrico, not logico
`*`, `/`, `%`	binari moltiplicativi
`+`, `-`	binari additivi
`<<`, `>>`	inserzione/estrazione
`<` `<=`, `>`, `>=`	relazionali
`==`, `!=`	uguaglianza
…	…

17

Precedenza tra gli operatori

Priorità

maggiore



minore

Operatore	Descrizione
`&&`	and logico
`||`	or logico
`?:`	condizionale ternario
`=, +=, -=, …`	assegnamento e composti
`,`	virgola

18

Aritmetica dei puntatori

Per esempio nelle espressioni

```
x = *fPtr++;
```

Prima c'è l'incremento del puntatore e poi il dereferenziamento, **x** è una variabile di tipo **float**

```
*fPtr+=1; *fPtr=*f2Ptr;
```

Il deferenziamento viene prima degli assegnamenti. Qui si incrementa di 1 il float «puntato», non il puntatore. L'assegnamento è tra i due float puntati.

19

Aritmetica dei puntatori

Per esempio nelle espressioni

```
(*fPtr)++;
```

Incrementa la variabile puntata da **`fPtr;`**

```
*fPtr+=1; *fPtr=*f2Ptr;
```

Il deferenziamento viene prima degli assegnamenti. Qui si incrementa di 1 il float «puntato», non il puntatore. L'assegnamento è tra i due float puntati.

20

Aritmetica dei puntatori

Supponiamo di avere un insieme di interi memorizzati in un'area contigua di memoria (*e.g.* un array), sommare (sottrarre) a un puntatore un certo offset *i*, significa: «far sì che esso punti un intero *i* posizioni più in avanti (indietro).

Tuttavia, questa operazione si traduce in un incremento (decremento) dell'indirizzo contenuto dal puntatore di *i volte* la taglia del tipo base, nel nostro caso **`int`**.

21

L’operatore **sizeof()**

L’operatore **sizeof()** restituisce il numero di byte necessari per memorizzare la variabile o il tipo indicati tra parentesi:

Tipo/variabile	Espressione	Ris.
`char`	`sizeof(char)`	1
`int`	`sizeof(int)`	4
`double`	`sizeof(double)`	8
`char x[8]`	`sizeof(char[8]) o sizeof(x)`	8
`Struct {double y;double z;} k;`	`sizeof(k)`	16
`int*`	`Sizeof(int*)`	8

Esempio: *aritmetica degli operatori*

```
4   int main()
5   {
6           const int MAX=5;
7           int inarr[MAX]={1,2,3,4,5},*inarrPtr=inarr;
8           double dbarr[MAX]={0.1,0.2,0.3,0.4,0.5}, *dbarrPtr=dbarr;
9
10          cout << "inarrPtr="<< inarrPtr << ", *inarrPTR="<<(*inarrPtr) << endl;
11          cout << "dbarrPtr="<< dbarrPtr << ", *dbarrPTR="<<(*dbarrPtr) << endl;
12
13          inarrPtr++;
14          dbarrPtr++;
15
16          cout << "inarrPtr="<< inarrPtr << ", *inarrPTR="<<(*inarrPtr) << endl;
17          cout << "dbarrPtr="<< dbarrPtr << ", *dbarrPTR="<<(*dbarrPtr) << endl;
18  }
```

Visualizzo l’indirizzo contenuto dal puntatore e il dato che vi è contenuto;

Esempio: *aritmetica degli operatori*

```
inarrPtr=0x6ffde0, *inarrPTR=1
dbarrPtr=0x6ffdb0, *dbarrPTR=0.1
inarrPtr=0x6ffde4, *inarrPTR=2
dbarrPtr=0x6ffdb8, *dbarrPTR=0.2
```

```
 4 int main()
 5 {
 6         const in   =5
 7         int inar    ]=
 8         double dbarr[MA
 9
10         cout << "inarrP
11         cout << "dbarrP
12
13         inarrPtr++;
14         dbarrPtr++;
15
16         cout << "inarrPtr="<< inarrPtr << ", *inarrPTR="<<(*inarrPtr) << endl;
17         cout << "dbarrPtr="<< dbarrPtr << ", *dbarrPTR="<<(*dbarrPtr) << endl;
18 }
```

Gli indirizzi dei puntatori a intero, aumentano di 4 byte per ogni unità di offset, poiché sizeof(int)=4;

24

Esempio: *aritmetica degli operatori*


```
inarrPtr=0x6ffde0, *inarrPTR=1
dbarrPtr=0x6ffdb0, *dbarrPTR=0.1
inarrPtr=0x6ffde4, *inarrPTR=2
dbarrPtr=0x6ffdb8, *dbarrPTR=0.2
```

```
 4 int main()
 5 {
 6         const int MAX=5
 7         int inarr[MAX]=
 8         double db
 9
10         cout << "inarrP
11         cout << "dbarrP
12
13         inarrPtr++;
14         dbarrPtr++;
15
16         cout << "inarrPtr="<< inarrPtr << ", *inarrPTR="<<(*inarrPtr) << endl;
17         cout << "dbarrPtr="<< dbarrPtr << ", *dbarrPTR="<<(*dbarrPtr) << endl;
18 }
```

Gli indirizzi dei puntatori a double, aumentano di 8 byte per ogni unità di offset, poiché sizeof(double)=8;

25

Esercizio: *sequenze palindrome coi puntatori*

Scrivere un programma C++ che chieda all'utente di riempire un array di 5 interi e verifichi se la sequenza inserita è palindroma. Nello svolgimento:

La verifica deve essere effettuata da una funzione che prende il puntatore all'array e la sua lunghezza e restituisce un bool.

Nella funzione, lo scorrimento dell'array deve essere effettuato con dei puntatori.

26

Esercizio: *sequenze palindrome coi puntatori*

```
23  int main()
24  {
25          const int MAX=3;
26          int main_array[MAX]={1,2,1};
27
28          cout << "isPalindrome=" << isPalindrome(main_array,MAX) << endl;
29  }
```

27

Esercizio: *sequenze palindrome coi puntatori*

```
 4  bool isPalindrome(int *seq, int size){
 5      int *i, *j;
 6
 7      i=seq;
 8      j=seq+(size-1);
 9
10      while(i<j){
11          if (*i!=*j)
12              return false;
13          i++;
14          j--;
15      }
16      return true;
17  }
```


28

Esercizio: *numeri uguali in posti uguali*

Scrivere un programma C++ che chieda all'utente di riempire due array di 5 interi calcoli quante volte, nei due array compare lo stesso numero nella stessa posizione.

Per esempio se `arr1={1,12,54,79,39}` e `arr2={1,79,47,81,39}` il risultato è 2 (1 e39)

Il conteggio deve essere effettuata da una funzione che prende il puntatore ai due array e la loro lunghezza e restituisca un intero.

Nella funzione, lo scorrimento dell'array deve essere effettuato utilizzando l'aritmetica dei puntatori

29

Esercizio: *numeri uguali in posti uguali*

```
 4  int contauguali(int *p1, int *p2, int len)
 5  {
 6          int cnt=0;
 7          for (int i=0;i<len;i++){
 8                  if (*p1==*p2)
 9                          cnt++;
10                  p1++;
11                  p2++;
12          }
13          return cnt;
14  }
```

Le variabili puntatore sono variabili locali (il dato puntato non lo è). Queste espressioni hanno effetto soltanto all'interno della funzione.

30

Esercizio: *al contrario*

Scrivere un programma C++ che chieda all'utente di riempire un array di 5 interi. Al termine dell'input, gli elementi dell'array devono essere messi al contrario.

Per esempio l'utente inserisce **`arr1={1,12,54,79,39}`** al termine del programma, deve risultare che **`arr1={39,79,54,12,1}`**

Serve una funzione **`swap`** che scambi due elementi (passaggio dei parametri per indirizzo)

Nel programma, lo scorrimento dell'array deve essere effettuato utilizzando l'aritmetica dei puntatori

31

Gestione dinamica della memoria (prologo)

32

Gestione dinamica della memoria (prologo)

Il C++ consente ai programmatori di allocare e deallocare memoria a *runtime* per tutti i tipi di dato. Questa caratteristica è nota come *Gestione dinamica della memoria.*

Questa capacità del C++, permette di riservare spazio in memoria per nuovi dati che si sono resi necessari durante l'esecuzione e la cui dimensione non era nota al momento della scrittura del programma.

Allo scopo il C++ rende disponibili gli operatori **`new`** e **`delete`**

33

Gestione dinamica della memoria (prologo)

L'operatore new, crea un oggetto del tipo e della misura indicata. Possibilmente fornendo anche un valore di inizializzazione

```
int array[20], *dynArray, dsize
float *fp1;
…
fp1=new float (3.14);
cin >> dsize;
…
dynArray = new int[dsize];
…
```

L'operatore new restituisce il puntatore all'oggetto creato

34

Gestione dinamica della memoria (prologo)

Per l'allocazione di memoria ai nuovi oggetti, la new attinge a un area di memoria a disposizione del programma proprio per questo scopo: lo *heap.*

Lo *heap* non è infinito. E' buona norma liberare la memoria allocata quando non è più necessaria. Per questo c'è l'operatore delete

```
…
delete fp1;
…
delete [] dynArray;
…
```

35

Esempio: *quanti punti vuoi?*

```
 9  int main()
10  {
11          PUNTO *p1;
12          int quantipunti=0;
13
14          cout << "Quanti punti servono: ";
15          cin >> quantipunti;
16          p1=new PUNTO[quantipunti];
17
18          for(int i=0;i<quantipunti;i++){
19                  cout << "punto " << i << ": ";
20                  cin >> (p1+i)->x >> (*(p1+i)).y ;
21          }
…           …
27          delete [] p1;
28  }
```

Crea un array *dinamico* della taglia richiesta. Ora **p1** punta al primo elemento dell'array.

Esempio: *quanti punti vuoi?*

```
 9  int main()
10  {
11          PUNTO *p1;
12          int quantipunti=0;
13
14          cout << "Quanti punti servono: ";
15          cin >> quantipunti;
16          p1=new PUNTO[quantipunti];
17
18          for(int i=0;i<quantipunti;i++){
19                  cout << "punto " << i << ": ";
20                  cin >> (p1+i)->x >> (*(p1+i)).y ;
21          }
…           …
27          delete [] p1;
28  }
```

L'accesso ai campi (membri) di **struct** e **class** tramite puntatori alle stesse è molto frequente.

Per semplicità, l'***operatore freccia***, sostituisce l'operatore *punto* quando alla sua sinistra c'è un puntatore. Dati:

```
PUNTO x, *xPtr;  xPtr= &x;
```

le espressioni:

```
xPtr->x e (*xPtr).x
```

Sono del tutto equivalenti.

Qui i due modi di referenziare il campo del punto puntato da p1 sono equivalenti. L'operatore freccia rende più semplice la sintassi