# Utilizzo di puntatori

*Corso di Programmazione 3 - Ingegneria dell'Informazione e dell'Organizzazione*
*17 ottobre , 2001*

**Gino Perna**

# Esempi di utilizzo di puntatori in C++

## PUNTATORI (PARTE PRIMA)

Scopo principe di qualunque linguaggio di programmazione è quello di fornire dei gradi di astrazione, i quali rendono la vita al programmatore molto meno grave di quanto sarebbe se egli dialogasse nella stessa lingua della macchina. Le variabili, ad esempio, sono la astrazione software delle celle di memoria principale: quando creiamo una variabile il programma segna una certa zona di memoria come occupata; se le assegnamo un valore il programma scrive tale valore (opportunamente codificato nel sistema binario5.1) nella zona precedentemente riservaa; ogni volta che utilizziamo la variabile nel nostro programma, la zona di memoria che astrae la variabile viene acceduta e il valore in esso contenuta viene letto e utilizzato. Cosa succede ad esempio nella seguente istruzione?

```
int a = b;
```

Avviene che una certa zona di memoria (o cella) viene riservata alla variabile a, si accede la zona di memoria della variabile b, si legge il valore in essa contenuto e lo si copia nella cella della variabile a. Tale modello non è del tutto conforme alle operazioni realmente eseguite a un programma, ma è sufficiente per schematizzare qualitativamente il funzionamento di esso, riguardo le variabili.

Dopo la precedente breve introduzione, vediamo cosa è un puntatore: così come la variabile è l'astrazione software della cella di memoria, un puntatore è l'astrazione software dell'indirizzo di tale cella. Si tratta dunque di uno strumento che ci fornisce un nuovo accesso ad una variabile: non tramite valore, ma tramite indirizzo. I puntatori sono delle variabili come tutte le altre (aventi a loro volta degli indirizzi di memoria ...), dichiarate (o definite) grazie al primo derivatore di tipi che incontriamo: il segno di moltiplicazione *, applicato al tipo della variabile al momento della creazione di essa.

```
int *a // puntatore a variabile intera

double *a // Puntatore a variabile reale

char *a // puntatore a variabile carattere
```

Le due sintassi (* a destra del tipo o a sinistra della variabile) sono equivalenti; in generale si consiglia di adottare la prima delle due, che meglio evidenzia il fatto che * sia un costruttore di tipo. Si faccia attenzione però che l'operatore * si applica solo alla prima variabile dichiarata, cioè:

```
void main() {
  char* a;
  double *b;
  int* c, d;     // ATTENZIONE
}
```

nello statement int* c, d la variabile c è un puntatore ad interi, mentre la variabile d è una variabile intera! Per evitare simili errori, è conveniente dichiarare sempre una sola variabile per statement; negli esempi presentati in questo testo non adotteremo tala regola per motivi di publishing.

Se abbiamo una variabile intera, possiamo assegnarle un qualunque valore intero, perché essa è l'astrazione di una cella di memoria contenente numeri interi; cosa possiamo invece assegnare ad un puntatore? Essendo esso l'astrazione dell'indirizzo di una variabile, possiamo assegnargli solo indirizzi di variabili, le quali abbiano lo stesso tipo del puntatore: se abbiamo dichiarato un puntatore con int* a, possiamo assegnare ad a solo indirizzi di variabili intere, altrimenti il compilatore tornerà un messaggio di errore. Nasce però un problema: come facciamo a sapere gli indirizzi delle variabili che abbiamo creato? Non c'è nessun modo, se non quello di utilizzare il nuovo operatore indirizzo: & (si legge at, che in inglese vuol dire ``presso''), il quale ha come unico compito quello di tornare l'indirizzo della variabile cui è applicato. Vediamo un esempio di indirizzi di variabili:

```
#include <iostream.h>
void main() {
  int n = 19;
  double x = 2.71;
  cout << &n << "\n";
  cout << &x << "\n";
}
```

esempio di output:

```
0xbffff844
0xbffff83c
```

Il programma ci ha stampato due indirizzi; qualche osservazione:

- gli indirizzi vengono stampati tramite cout nel sistema esadecimale, di cui non abbiamo trattato; si può immaginare comunque di cosa si tratti: un sistema di numerazione posizionale, il quale ha 16 cifre (0-9,A-F), mentre il sistema decimale ne ha 10 (0-9)e quello binario solo 2 (0,1); per convenzione i numeri in formato esadecimale hanno il prefisso ``0x";
- il numero di cifre che costituiscono l'indirizzo di una variabile varia a seconda del compilatore e, soprattutto, della macchina sulla quale esso lavora;
- come è ovvio, gli indirizzi per qualunque tipo di variabile (meglio, per qualunque tipo di oggetto) hanno sempre la medesima forma: non si può distinguere a priori, ad esempio, un int da un double conoscendo solo i loro indirizzi;
- in questo esempio e nei successivi riguardanti i puntatori, non si trattano volutamente variabili char, in quanto ci sarebbero problemi in fase di stampa; puntatori a char verranno comunque presi in considerazione separatamente, quando si tratteranno le stringhe di caratteri.

# PUNTATORI (PARTE SECONDA)

Abbiamo visto come si dichiarano i puntatori e come si può ottenere l'indirizzo di una variabile; combiniamo le due cose e otterremo la sintassi per l'assegnamento (o per la definizione) dei puntatori. Vediamo il seguente esempio:

```
void main() {
  int n = 19;
  double x = 2.71;
  int* p = &n;      // inizializzazione
  double* q;        // dichiarazione
  q = &x;           // assegnamento
  // q = &n;        // ERRORE: n e' una variabile intera,
                    // mentre q punta a variabili reali
}
```

Abbiamo creato dunque due variabili, le quali contengono certi valori, e due puntatori, i quali anch'essi contentono certi valori, solo che tali valori sono gli indirizzi delle due variabili precedenti. Ovviamente, come già notato, non possiamo assegnare ad un puntatore di un certo tipo, l'indirizzo di una variabile di un tipo diverso da esso. Così come le variabili non inizializzate, prima che fosse loro assegnato un valore, contengono dati del tutto privi di significato, anche per i puntatori vige la stessa regola; tuttavia un puntatore non inizializzato è potenzialmente più pericoloso di una variabile non inizializzata: siccome il programma interpreta il valore dei puntatori come indirizzi di memoria, se effettuiamo una operazione di scrittura, ad esempio, su di un puntatore non inizializzato e non assegnato potremmo causare danni al sistema sul quale il programma viene eseguito. La spiegazione risiede nel fatto che, malauguratamente, il contenuto arbitrario

del puntatore potrebbe corrispondere proprio ad una zona di memoria utilizzata da un altro programma, o dal sistema operativo stesso. Si faccia bene attenzione dunque a creare un puntatore immediatamente prima del suo utilizzo in maniera tale da inizializzarlo con l'indirizzo della variabile da puntare; nel caso ciò non fosse possibile, o comunque fosse sconveniente, si inizializzi il puntatore con il valore , il quale indica ``nessun indirizzo".

Ora che sappiamo come dichiarare, inizializzare, assegnare puntatori, non ci resta che capire il loro utilizzo: i puntatori se preceduti dall'operatore *, costituiscono degli alias5.2 delle variabili puntate. Questo significa che se abbiamo

```
int n = 5;
int* p = &n;
```

possiamo indifferentemente utilizzare al medesimo scopo n e *p, il che è spesso utile, ed in certi casi indispensabile. Vediamo il seguente esempio:

```
#include <iostream.h>
void main() {
  int a = 5;
  int b = 17;
  cout << "a = " << a << "\tb = " << b << "\n";
  int* p_a = &a;
  int* p_b = &b;
  cout << "p_a = " << p_a << "\np_b = " << p_b << "\n";
  a += 2;
  *p_b += 2;     // *p_b e b sono sinonimi
  // p_b += 2;   // NO: in questa maniera si cambia
                 // l'indirizzo cui punta p_b
  cout << "a = " << a << "\tb = " << b << "\n";
  cout << "p_a = " << p_a << "\np_b = " << p_b << "\n";
}
```

esempio di output:

```
a = 5 b = 17
p_a = 0xbffff844
p_b = 0xbffff840
a = 7 b = 19
p_a = 0xbffff844
p_b = 0xbffff840
```

Come si vede, modificando *p_b il puntatore rimane invariato, infatti continua a puntare sempre lo stesso indirizzo di variabile, mentre cambia il valore contenuto in b. Abbiamo appena visto un modo di cambiare il valore di una variabile senza utilizzare la variabile stessa; vedremo tra non molto come questa possibilità di operare apre nuove strade della programmazione in C++.

Vediamo ora come modificare il valore di un puntatore, semplicemente cambiando l'indirizzo in esso contenuto:

```
#include <iostream.h>
```

```
void main() {
  double x = 3.14, y = 2.71;
  double* p = 0;
  cout << "p = " << p << "\n";
  // cout << "p = " << p << "\t*p = " << *p << "\n";
  // NO: *p NON e' l'alias di nessuna variabile
  // per cui NON ha alcun valore
  p = &x;  // *p e' alias di x
  cout << "p = " << p << "\t*p = " << *p << "\n";
  p = &y;  // *p e' alias di y
  cout << "p = " << p << "\t*p = " << *p << "\n";
}
```

esempio di output:

```
p = (nil)


p = 0xbffff840
*p = 3.14

p = 0xbffff838
*p = 2.71
```

In fase di stampa ci viene ricordato che p inizialmente non punta alcuna variabile, per cui il suo valore è nil (dal latino nihil = ``niente"); ovviamente non possiamo stampare inizialmente il valore di *p in quanto non vi corrisponde nessuna variabile.

Per ogni variabile, possiamo avere un numero arbitrario di puntatori che fungono da alias:

```
#include <iostream.h>
void main() {
  int n = 3;
  int* p = &n;
  int* q = p;    // quindi anche q = &n
  cout << "n = " << n << "\t*p = " <<
    *p << "\t*q = " << *q << "\n";
  cout << "p = " << p << "\n";
  cout << "q = " << q << "\n";
  // int m = n**p**q;
  // Il precedente statement, anche se non si
  // direbbe, e' equivalente al successivo, il
  // quale e' tuttavia di gran lunga piu' chiaro.
  // Si tratta di un esempio di cattiveria pura
  // nei confronti di un possibile utilizzatore
  // del nostro codice (o dell'insegnante che
  // ci corregge l'esercizio)
```

```
  int m = n * (*p) * (*q);
  cout << "m = " << m << "\n";
}
```

esempio di output:

```
n = 3
*p = 3
*q = 3
p = 0xbffff844
q = 0xbffff844
m = 27
```

Si sarà notato che il simbolo * ha significati diversi nel linguaggio: operatore (binario) di moltiplicazione tra interi o reali, operatore (binario) di derivazione tipo per dichiarare puntatori, operatore (unario) che applicato a sinistra di un puntatore lo trasforma nella variabile puntata. Nel caso si debbano utilizzare in una sola espressione delle diverse accezioni del simbolo *, come nell'esempio precedente, si faccia uso dell'operatore parentesi tonde '()' al fine di evitare errori o confusioni in un eventuale utilizzatore del nostro codice.

# PUNTATORI E FUNZIONI

Un primo utilizzo dei puntatori consiste nel loro utilizzo come argomenti di funzioni; in tal caso si dice che gli argomenti sono passati alla funzione per indirizzo piuttosto che, come abbiamo già visto, per valore. La differenza tra queste due diverse modalità di scambio dei dati è fondamentale: se una funzione riceve solo il valore di certe variabili contenute nella funzione chiamante (negli esempi fino ad ora mostrati la funzione chiamante è sempre stata void main()), le variabili di essa non vengono mai modificate; al contrario scambiando gli indirizzi, tramite l'operatore * abbiamo a disposizione nella funzione chiamata degli alias di variabili della funzione chiamante. Il discorso non è semplice. Vediamo un esempio:

```
#include <iostream.h>

void scambia (int* a, int* b) {
  int c = *a;     // alias della variabile della
                  // funzione chiamante,
                  // che in questo caso e' void main()
  *a = *b;
  *b = c;
}

void main() {
  int a = 5, b = 7;
  cout << "a = " << a << "\tb = " << b << "\n";
  scambia (&a, &b);
```

```
  cout << "a = " << a << "\tb = " << b << "\n";
}
```

# VETTORI E FUNZIONI

Le funzioni possono passare argomenti: per valore, per indirizzo, per riferimento. Il primo di essi inizializza la variabile locale alla funzione chiamata, con il valore della variabile della funzione chiamante; si tratta in definitiva di una copia. Un array potrebbe anche essere molto grande, in termini di memoria occupata, e farne una copia ogni volta che una funzione viene chiamata potrebbe essere un'operazione del tutto inefficiente. Per questo motivo, il C++ consente per il passaggio di array un solo metodo: per indirizzo. La sintassi è molto semplice, in quanto rispecchia quella del passaggio per valore delle variabili; vediamo subito un semplice esempio:

## *Calcolo della media*

```
// Vettori.cpp
#include <iostream.h>

double * crea_vettore(int nelem)
{
    int n = 1;
    double *array;

    if(nelem >= 1) { n=nelem; }

    array = new double[n];


  for (int i = 0; i < n; i++) { // azzeriamo il vettore
    array[i] = 0;
  }

  return array;
}


double media_vettore(double *array, int nelem)
{

  double somma = 0;
  for (int i = 0; i < nelem; i++){
    somma += array[i];
  }
  return somma / (double) nelem;
}
```

```
void stampa_vettore(double *array, int n, int nColonne)
{
  int nc = 0;
  if (nColonne == 0) nc = n+1;
  else nc = nColonne;
  for (int i = 0; i < n; i++)
    if (i % nc == nc - 1){
      cout << array[i] << "\n";
    } else {
      cout << array[i] << "\t";
    }

    cout << "\n";
}


void input_vettore(double *array, int n)
{
  for( int i=0; i<n;i++){
     cout << "Elemento "<< i << " :";
     cin >> array[i];
  }
}



int main()
{

      int nesami;
     double *voti;


     cout << "Dammi il numero di termini:";
     cin >> nesami;

     voti = crea_vettore(nesami);
     stampa_vettore(voti,nesami,2);
     input_vettore(voti, nesami);
     stampa_vettore(voti,nesami,2);

     cout << "La media e': " << media_vettore(voti,nesami) <<
"\n";


  return 0;

}
```

Il programma contiene esempi di passaggio per indirizzo di vettori e l'instruzione per creare un vettore dinamicamente:

```
double *array;
array = new double [n];
```

in questo modo si dichiara un puntatore ad un tipo double e successivamente si riserva lo spazio per un vettore di 'n' elementi.

Il resto del programma è banale.

## *Vettori di stringhe*

Trattare stringhe nello stesso modo come abbiamo trattoro un vettore di double non è possibile! Infatti la stringa è di per sè stessa un vettore. Ecco che per generare un vettore di stringhe è necessario procedere alla allocazione di un vettore di puntatori a stringa, e successivamente per ciascun puntatore allocare la memoria sufficiente ad immagazinare la stringa.

```
// stringhe Vettori.cpp
#include <iostream.h>

char ** crea_vettoreS(int nelem)
{
    int n = 1;
    char  **array;

    if(nelem >= 1) { n=nelem; }

    array = new char * [n];

  return array;
}


void stampa_vettoreS(char  *array[], int n)
{
  for (int i = 0; i < n; i++)
      cout << array[i] << "\n";
}


void input_vettoreS(char  *array[], int n)
{
  char riga[80];
  for( int i=0; i<n;i++){
     cout << "Elemento "<< i << " :";
     cin >> riga;

     int len = strlen(riga);

     array[i] = new char [len+1];
```

```
      strcpy(array[i],riga);
  }
}


int main()
{

      int nragazze;
      char ** nomi;


      cout << “Dammi il numero di ragazze:”;
      cin >> nragazze;

      nomi = crea_vettoreS(nragazze);

      input_vettoreS(nomi, nragazze);

      stampa_vettoreS(nomi, nragazze);



  return 0;

}
```

In particolare si noi la differenza col programma precedente:

- Il vettore è dichiarato ** anzichè *;
- conseguentemente la prima allocazione è diversa;
- per ciascuna riga si alloca la quantità di caratteri necessaria ad immagazzinare la stringa.

**Provare a tracciare lo schema del vettore di stringhe come visto dal programma.**

# ESERCIZIO FINALE

Visti gli esempi di allocazione per vettori e stringhe si supponga di dover implementare il seguente programma:

1. Il programma leggerà in input una serie di punti geometrici costituiti dalla coordinata X (reale), Y (reale), stringa (caratteri) di etichettatura del punto;
2. Il numero di punti non è noto a priori;

3. La lunghezza delle etichette non è nota a priori;
4. Il programma dovrà fornire la distanza progressiva della spezzata che collega tutti i punti vicino alla etichetta del punto.

Esempio:

```
Nro punti 3
punto 1 x = 0 y = 0 et= INIZIO;
punto 2 x = 1 y = 1 et= INMEZZO;
punto 3 x = 3 y = 3 et= HOFINITO;
```

Stampo:

```
Distanza a punto 2 INMEZZO 1.4142
Distanza a punto 3 HOFINITO 3
```

## *Suggerimenti*

Le funzioni di base sono tutte contenute nei due esempi precedenti.

Ad esempio si possono chiamare i vettori x,y,nomi e procedere con istruzioni del tipo:

```
y = crea_vettore(npunti);
nomi = crea_vettoreS(npunti);
```

Si crei una funzione input_punto che chieda e momorizzi le due coordinate ed il nome per ciascun punto:

```
input_punto(x,y,nomi,npunti);
```

Attenzione! La funzione dovrà allocare lo spazio per il nome!

```
cout << " Nome: ";
cin >> riga;
int len = strlen(riga);
nome[i] = new char [len+1]; // Lo spazio per il carattere \0 !!!!!
strcpy(nome[i],riga);
```

La formula della distanza:

```
dist = sqrt((x[i]-x[i-1])*(x[i]-x[i-1]) + (y[i]-y[i-1])*(y[i]-
y[i-1]));
```