



UNIVERSITÀ DEGLI STUDI DI PARMA

Corso di Laurea in
"Ingegneria dei Sistemi Informativi"

Programmazione di Applicazioni Software

Puntatori in C++

Andrea Prati

Riferimenti a variabili

- Nel C++ è stato introdotto un **modificatore di tipo** che consente la creazione di ***riferimenti* (reference)** a variabili: <tipo>& <identificatore>
- Un riferimento si comporta come un alias o sinonimo della variabile. Quindi ogni operazione effettuata sul riferimento in realtà è effettuata sulla variabile.
- Quando un riferimento viene dichiarato **deve essere anche inizializzato**:

int x = 0;

int& y = x; ***// riferimento***

y=1; ***// x vale 1***

int& z; ***// errore di compilazione***

- E' possibile avere riferimenti a puntatori

Riferimenti a variabili

- L'operatore & usato per il riferimento **non va confuso** con l'operatore & che ritorna l'indirizzo di una variabile. Il primo è usato al momento della dichiarazione di una variabile, mentre il secondo è usato nelle espressioni in cui si valuta una variabile:

int a;

int& rA = a; // riferimento

int pA = &a; // operatore indirizzo-di*

- In generale, come vedremo, i puntatori sono più potenti dei riferimenti perché consentono più operazioni (ad esempio, i riferimenti non possono essere assegnati e non hanno aritmetica), però sono più «pericolosi» e complessi da gestire
- In generale, è buona norma **usare i riferimenti quando possibile e usare i puntatori solo quando necessario.**

Puntatori

- una variabile viene memorizzata a partire da un certo indirizzo di memoria e occupa un certo numero di byte in memoria
- i puntatori contengono indirizzi di memoria
 - mentre una variabile contiene direttamente un valore
 - un puntatore lo contiene indirettamente: **indirection**
- se **v** è una variabile, **&v** è la locazione o indirizzo-di memoria dove è memorizzato il valore di **v**
- l'operatore ***** (indirection operator) ritorna un sinonimo (alias) dell'oggetto a cui il suo operando (un puntatore) punta
- i puntatori sono **tipizzati** (in fase di dichiarazione è necessario specificare il tipo dell'oggetto puntato)
- es: ***int *pi;*** ***// pi è un puntatore a un int***

Puntatori * e &

```
#include <iostream>
```

```
using std::cout;
```

```
using std::endl;
```

```
int main() {
```

```
    int a;
```

// a is an integer

```
    int *aPtr;
```

// aPtr is a pointer to an integer

```
    a = 7;
```

```
    aPtr = &a;
```

// aPtr set to address of a

```
    cout << "The address of a is " << &a << "\nThe value of aPtr  
is " << aPtr;
```

```
    cout << "\n\nThe value of a is " << a << "\nThe value of  
*aPtr is " << *aPtr;
```

```
    cout << "\n\nShowing that * and & are inverses of " << "each  
other.\n&*aPtr = "
```

```
        << &*aPtr << "\n*&aPtr = " << *&aPtr << endl;
```

```
    return 0;
```

```
}
```

```
The address of a is 0x6dfeec
The value of aPtr is 0x6dfeec

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0x6dfeec
*&aPtr = 0x6dfeec
```

Allocazione dinamica della memoria

- l'operatore ***new*** crea una variabile dinamica e restituisce un puntatore a questa
- ***Operatore new***
 - Esempio:
int *p;
p = new int;
- l'operatore ***delete*** distrugge la variabile e dealloca la memoria
 - Esempio:
delete p;
- la gestione della memoria è demandata al programmatore

Aritmetica dei puntatori

- $*(p + 3)$ rappresenta l'indirizzo che si ottiene sommando all'indirizzo base del puntatore p l'equivalente di 3 locazioni **del tipo a cui punta p**
- analogo discorso vale per gli array (un array è un puntatore al primo elemento dell'array)
- $a[3]$ rappresenta l'indirizzo che si ottiene sommando all'indirizzo base dell'array a l'equivalente di 3 locazioni del tipo degli elementi di a
- Se il sistema assegna all'indirizzo base dell'array di `int` a l'indirizzo 300, l'elemento $a[0]$ è all'indirizzo 300, $a[1]$ a 304, $a[2]$ a 308, ecc. (supponendo che la macchina usata riservi 4 byte per un `int`)
- L'istruzione `p=a` (notare che non uso `&`) è equivalente a `p=&a[0]` e assegna il valore 300 a p . L'istruzione `p=a+1` (o `p=&a[1]`) assegna a p il valore 304
- gli array sono puntatori costanti e non possono essere modificati
 - **`a++` //errore**

Uso dei puntatori e array

- i puntatori hanno alcuni usi fondamentali:
 - argomento funzioni che devono modificare le variabili passate
 - allocazione dinamica della memoria
 - strutture dati complesse (sfruttano l'allocazione dinamica)

```
#include <iostream>
using namespace std;
int main() {
    int v[10];
    int *pv;
    for (int i=0; i<10; i++)
        v[i] = i*10;
    pv = v;
    for (int i=0;i<10;i++) {
        cout << "indirizzo= " << pv << " valore= " << *pv << endl;
        pv = pv +1;
    }
    return 0;
}
```

```
indirizzo= 0x6dfecc valore= 0
indirizzo= 0x6dfed0 valore= 10
indirizzo= 0x6dfed4 valore= 20
indirizzo= 0x6dfed8 valore= 30
indirizzo= 0x6dfedc valore= 40
indirizzo= 0x6dfee0 valore= 50
indirizzo= 0x6dfee4 valore= 60
indirizzo= 0x6dfee8 valore= 70
indirizzo= 0x6dfeec valore= 80
indirizzo= 0x6dfef0 valore= 90
```


Array e puntatori

```
#include <iostream>
using namespace std;
int main() {
    int v[10];
    int *pv;
    pv = v;
    for (int i=0; i<10; i++) {
        *pv = i*10;
        pv++;
    }
    for (int i=0; i<10; i++) {
        cout << "indirizzo (&v[" << i << "])= " << &v[i]
        << " valore (v[" << i << "])= " << v[i] << endl;
    }
    return 0;
}
```

```
indirizzo (&v[0])= 0x6dfecb valore (v[0])= 0
indirizzo (&v[1])= 0x6dfec8 valore (v[1])= 10
indirizzo (&v[2])= 0x6dfec4 valore (v[2])= 20
indirizzo (&v[3])= 0x6dfec8 valore (v[3])= 30
indirizzo (&v[4])= 0x6dfec8 valore (v[4])= 40
indirizzo (&v[5])= 0x6dfed0 valore (v[5])= 50
indirizzo (&v[6])= 0x6dfed4 valore (v[6])= 60
indirizzo (&v[7])= 0x6dfed8 valore (v[7])= 70
indirizzo (&v[8])= 0x6dfedc valore (v[8])= 80
indirizzo (&v[9])= 0x6dfef0 valore (v[9])= 90
```

Problemi di gestione della memoria

- buffer overrun
 - superamento dei limiti di un buffer di memoria
 - problema analogo con gli indici degli array
- memory leak
 - consumo non voluto di memoria dovuto alla mancata deallocazione di variabili/dati non più utilizzati da parte dei processi
- lingering pointer
 - puntatore che fa riferimento a un'area di memoria non più allocata

Buffer overrun

- buffer overrun (buffer overflow) errore runtime
- in un buffer di una certa dimensione vengono scritti dati di dimensioni maggiori
- viene sovrascritta parte della zona di memoria immediatamente adiacente al buffer in questione
- in alcune situazioni provoca vulnerabilità di sicurezza
- linguaggi managed (Java, .Net) cercano di prevenire queste situazioni

Memory leak

```
#include <iostream>
#include <new>
using namespace std;
/* funzione chiamata se l'allocazione di memoria di new fallisce*/
void fine_memoria() {
    cout << " *** memoria terminata *** ";
    exit(0);
}
int main() {
    int *v;  long i,n;
    set_new_handler(fine_memoria);
    cout << "verranno allocati n array di n elementi -> n = "; cin >> n;
    for ( i = 0; i < n; i++) {
        v = new int[n];          // allocazione n interi
    }
    delete [] v;                // deallocazione
    return 0;
}
```

```
verranno allocati n array di n elementi -> n = 100000
*** memoria terminata ***
-----
(program exited with code: 0)
```

Gestione corretta

```
#include <iostream>
#include <new>
using namespace std;
/* funzione chiamata se l'allocazione di memoria di new fallisce*/
void fine_memoria() {
    cout << " *** memoria terminata *** ";
    exit(0);
}
int main() {
    int *v;  long i,n;
    set_new_handler(fine_memoria);
    cout << "verranno allocati n array di n elementi -> n = "; cin >> n;
    for ( i = 0; i < n; i++) {
        v = new int[n];           // allocazione n interi
        delete [] v;              // deallocazione
    }
    return 0;
}
```

```
verranno allocati n array di n elementi -> n = 100000
-----
(program exited with code: 0)
```

Lingering pointer

```
#include <iostream>

int *inputVal() {
    int tmp[1000];
    for(int i=0;i<1000;i++)
        tmp[i] = i*10;
    return tmp;
}

int main() {
    int *p;
    p = inputVal();
    std::cout << p[2];
}
```

Array statici vs array dinamici

- allocazione **statica**
 - a **compile time** è conosciuta la dimensione dell'array
- allocazione **dinamica**
 - a **runtime** viene definita la dimensione dell'array

Allocazione statica

```
#define LEN01 10
```

```
...
```

```
double stArr01[LEN01];           // lunghezza costante #define ...  
cout << "size of stArr01: " << sizeof stArr01 << " bytes " << endl;  
cout << "memory address of stArr01: " << stArr01 << endl;
```

```
const int len02 = 10;
```

```
double stArr02[len02];           // lunghezza costante const int ...  
cout << "size of stArr02: " << sizeof stArr02 << " bytes " << endl;  
cout << "memory address of stArr02: " << stArr02 << endl;
```

```
size of stArr01: 80 bytes  
memory address of stArr01: 0x6dfea8  
size of stArr02: 80 bytes  
memory address of stArr02: 0x6dfe58
```


Allocazione statica? non standard

```
int len03 = 10;
double stArr03[len03];           // lunghezza int ... = ...
cout << "size of stArr03: " << sizeof stArr03 << " bytes " << endl;
cout << "memory address of stArr03: " << stArr03 << endl;

/* if compiler implements a non-standard extension called VLA
 * (Variable Length Arrays)*/
int len04;
cout << "strArr04 lenght: ";
cin >> len04;
double stArr04[len04];          // lunghezza int ...
cout << "size of stArr04: " << sizeof stArr04 << " bytes " << endl;
```

Allocazione dinamica

```
int len05;
cout << "strArr05 lenght: ";
cin >> len05;
double *dyArr05;
dyArr05 = new double[len05];           // allocazione dinamica
cout << "size of dyArr05: " << sizeof dyArr05 << " bytes " << endl;
cout << "size of *dyArr05: " << sizeof *dyArr05 << " bytes " << endl;

cout << "memory address of dyArr05: " << dyArr05 << endl;
delete []dyArr05;
```

```
strArr05 lenght: 44
size of dyArr05: 4 bytes
size of *dyArr05: 8 bytes
memory address of dyArr05: 0x1fa748
```