



UNIVERSITÀ
DI PARMA

C++ puntatori

Alberto Ferrari

- 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 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`

```
#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
```

- l'operatore **new** crea una variabile dinamica e restituisce un puntatore a questa
- `#include <new>` `// ::operator 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

- $*(p + 3)$ 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
- gli array sono puntatori costanti e non possono essere modificati
`a++` //errore

- 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
```

```
#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])= 0x6dfebc valore (v[0])= 0
indirizzo (&v[1])= 0x6dfec0 valore (v[1])= 10
indirizzo (&v[2])= 0x6dfec4 valore (v[2])= 20
indirizzo (&v[3])= 0x6dfec8 valore (v[3])= 30
indirizzo (&v[4])= 0x6dfec4 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])= 0x6dfee0 valore (v[9])= 90
```


- 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 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

```
#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)
```

```
#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)
```

```
#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];
}
```