RIPASSO PUNTATORI

Un **puntatore** è utilizzato per memorizzati indirizzi di *array* allocati in memoria dinamica ma possono essere utilizzati per riferire oggetti di ogni tipi.

Allo stesso modo degli *array*, si possono allocare e deallocare oggetti dinamici di ogni tipo mediante gli operatori new e delete.

La sintassi nome_tipo * può essere usata per dichiarare un puntatore ad un oggetto singolo o ad un array. Con le parentesi quadre si fa riferimento ad un array, senza invece a uno scalare (int, float…).

```
const int *p   // puntatore ad oggetto di tipo int,
               // non modificabile tramite p
int * const p  // puntatore costante ad oggetto di
               // tipo int
int * p[10]      // array di 10 puntatori ad int
int (*p)[10]     // puntatore ad array di 10 interi
```

L'operatore unario e prefisso **di indirizzo &** restituisce l'indirizzo di memoria dell'oggetto a cui viene applicato assegnabile quindi a un puntatore.

Per accedere all'oggetto riferito da un puntatore si usa **l'operatore** unario e prefisso **di dereferenziazione *.** L'operatore * applicato ad un puntatore ritorna un riferimento all'oggetto puntato.

Problemi tipici dei puntatori:

Dangling reference (puntatore pendente): puntatore che punta ad un area di memoria già liberata tramite l'operatore free.

Memory leak: la memoria allocata non viene liberata.

Il valore di un puntatore può essere stampato mandandolo sullo stream di uscita mediante l'operatore << (viene rappresentato in base 16).

Un puntatore può puntare a un altro puntatore.

```
main() {
   int i, *p;
   int **q; // puntatore a puntatore a int

   q = &p;  // q = indirizzo di p
   p = &i;  // p = indirizzo di i
   **q = 3;// equivale a i = 3
   }
```

Dato un puntatore s *p a un oggetto di tipo struttura s ci sono due modi per riferire ad un campo a della struttura:

(*p).a

p->a

Nota: l'operatore . ha una precedenza maggiore dell'operatore * ⇒ necessarie le parentesi.

```
main() {
struct s {int a, b;} s1;
s *p2;    // punt. ad un oggetto di tipo s

p2 = &s1;

(*p2).a = 3;    // equivalente a s1.a = 3

p2->a = 3; // equivalente all'istruzione
           // precedente
}
```

Oltre ai puntatori il C++ supporta il concetto di riferimento dichiarato mediante l'operatore &.

A livello di utilizzo, un riferimento ad una variabile è un ulteriore nome per essa, in pratica un alias.

```
void main()
{ int n=75;
  int & rif=n;
  cout<<"n="<<n<<", rif="<<rif<<", ";
  rif = 30;
  cout<<"n="<<n<<", rif="<<rif<<", ";
}
```

Stampa: n=75, rif=75, n=30, rif=30

A livello di implementazione, un riferimento contiene l'indirizzo di un oggetto puntato, come un puntatore.

I riferimenti sono meno flessibili e quindi meno pericolosi in quanto viene realizzato un puntatore costante nascosto che ha per valore l'indirizzo dell'oggetto riferito.

```
int & rif = n;
```
Corrisponde a:
```
int * __ptr_rif = &n; // puntatore nascosto
```

Ogni volta che si coinvolge l'oggetto riferito si ha quindi una **deferenziazione** sul puntatore nascosto (rif viene sostituito dal puntatore).

Differenze riferimenti puntatori:

I riferimenti non possono avere valore nullo → necessaria inizializzazione
I riferimenti non possono poi essere riassegnati.

I riferimenti sono usati per la dichiarazione dei parametri:

Passaggio per valore: impedisce i cambiamenti e spreca memoria per le copie
Passaggio per riferimento: consente la modifica del parametro attuale attraverso la modifica al corrispondente parametro formale (che quindi è un indirizzo al parametro passato)

I puntatori possono essere usati in espressioni aritmetiche

Aritemtica degli indrizzi: insieme di regole che governano le operazioni effettuabili sugli indirizzi.

Sia *p* un puntatore contenente l'indirizzo di un oggetto di tipo *T.* Allora l'espressione *p+i* restituisce come valore l'indirizzo di un oggetto di tipo *T* che si trova in memoria dopo *i* oggetti consecutivi di tipo *T*.

L'incremento (p++) o il decremento (p–) applicato a un puntatore T *p = x restituisce come valore l'indrizzo di un oggetto di tipo T del puntatore che segue o precede x.

PUNTATORE A FUNZIONE

I puntatori in C Possono puntare anche alle funzioni.

Sintassi

```
int (* puntafun) (double, int) ;
```

I puntatori possono essere assegnati ad una funzione compatibile precedentemente dichiarata. (compatibile: ha gli stessi tipi di parametri di input e output).

```
int (* puntafun) (double, int) ;
int f1(double , int);
int f2(double , int);
if ( ......... ) puntafun = f1 ;
else puntafun = f2 ;
```

Solo nome senza argomenti

L'**assegnamento** corretto dovrebbe avvenire con l'operatore &. Ma il compilatore è flessibile, funziona anche senza.

```
puntafun = &f1 ;
```

Chiamata a funzione: per chiamare la funzione occorre dereferenziare il puntatore. Il compilatore accetta anche la versione senza *.

```
(*puntafun)(45.76, 5) ;
```

Si possono avere **array di puntatori a funzione.**

```
int f1 (double, int) ;
int f2 (double, int) ;
int (* puntafun[2])(double, int) = {f1,f2}
```

Una volta dichiarato un array di puntatori a funzione può essere utile per l'implementazione di un menu nel quale la funzione da eseguire dipende da un indice i, senza usare costrutti *if* e *switch*.

Una funzione può dichiarare tra i suoi argomenti un elemento di tipo puntatore a funzione.

```
void sel_fun(int(*pfun)(double))
  {int n ;
  ...
       n = pfun(64.7) ;
  ... }
```

Nell'invocazione della funzione è necessario indicare come parametro il nome di una funzione precedentemente indicata.

```
   int f1(double);
   int f2(double);
...
sel_fun(f1) ;
   sel_fun(f2) ;
```

```
sel_fun(f2(67.89)) ; //invocazione errata!!
```

Se volessi passare altri parametri alla funzione invece

```
void sel_fun(int(*pfun)(double), double r) ;
```