

# Parte 1 - Liste

## Puntatori



[S. Dalí – The temptation of St. Anthony, 1946]

# Puntatori

**Approfondimento** rispetto alla trattazione vista nel corso precedente

Finora come avete utilizzato i puntatori?

Principalmente per memorizzare indirizzi di array allocati in memoria dinamica

```
int *p=new int[N];
```

Ma i puntatori possono essere utilizzati per  
***referire oggetti di ogni tipo***

# Allocazione/deallocazione

- 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

```
int *p;
```

- Se non si tratta di array ma di un tipo scalare (int, float, double,...), non si utilizzano le parentesi quadre né si indica il numero di elementi quando si utilizza l'operatore **new**

```
p = new int;
```

**Perché è importante specificare nome\_tipo?**

# Esempi (1)

```
int *p ;    // puntatore ad un oggetto di  
            // tipo int
```

```
p = new int ; // allocazione di un oggetto  
              // dinamico di tipo int:  
              // NON è un array!
```

```
delete p ;   // deallocazione di un oggetto  
            // puntato da p
```

## Esempi (2)

```
struct s {int a, b ;} ;
```

```
s *p2 ; // punt. ad un oggetto di tipo s
```

```
p2 = new s ; // allocazione di un oggetto  
            // dinamico di tipo s:
```

```
delete p2 ; // deallocazione oggetto  
            // puntato da p2
```

# Ripasso

Cosa rappresentano i seguenti oggetti?

```
const int *p
```

```
int * const p
```

```
int * p[10]
```

```
int (*p)[10]
```

# Ripasso

Cosa rappresentano le seguenti dichiarazioni?

```
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 statico di 10  
                interi
```

# Operatore di indirizzo

*L'operatore di indirizzo &* restituisce l'indirizzo di memoria dell'oggetto a cui viene applicato

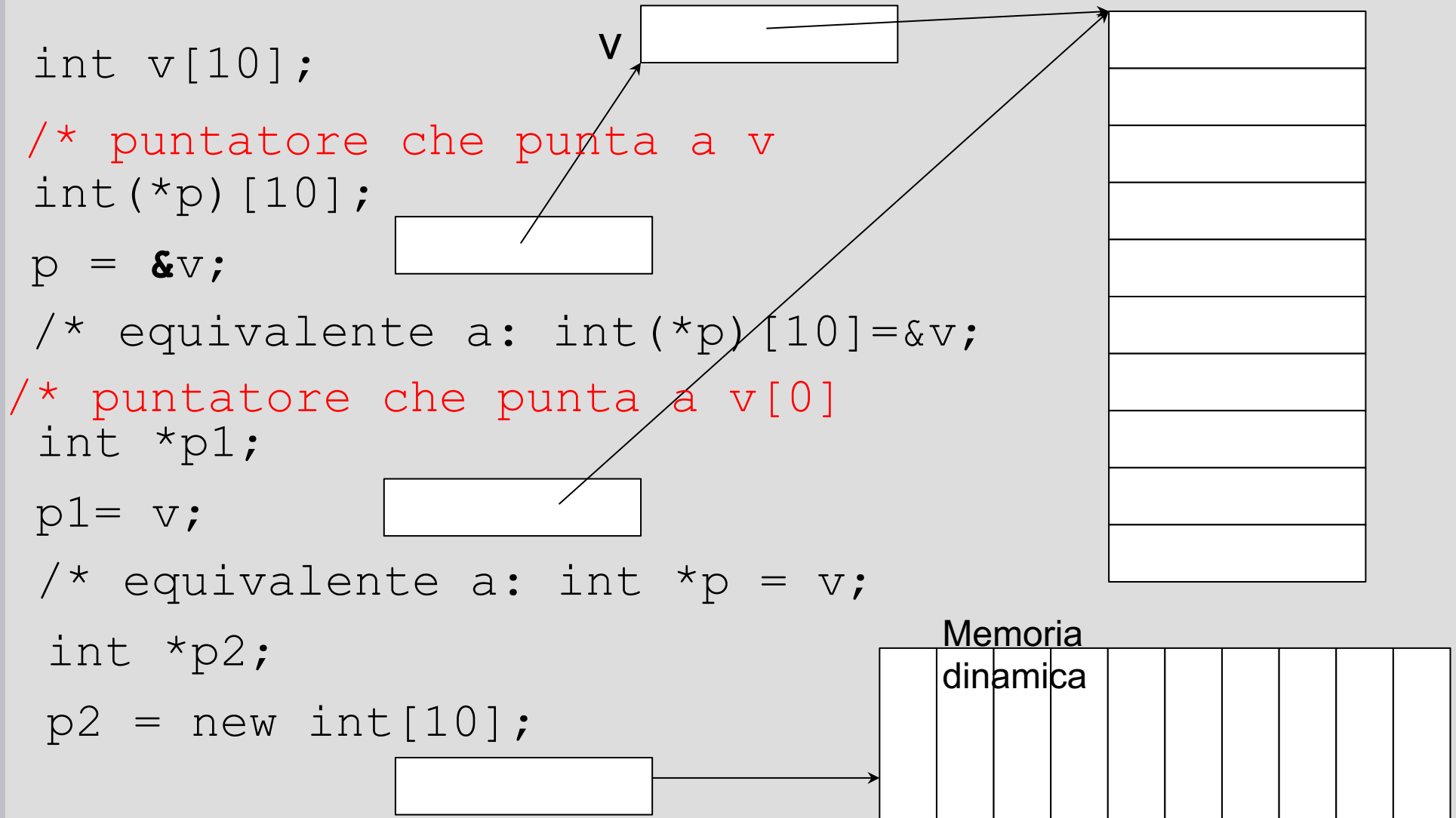
## Operatore unario e prefisso

Il risultato restituito dall'operatore di indirizzo può essere **assegnato ad un puntatore ad un oggetto dello stesso tipo**

- **&x** può essere tradotto come “l'indirizzo di **x**”



# Puntatori a vettori



# Esempi

```
main() {  
    int i, j, k;  
    // p = indirizzo di i  
    int *p = &i;  
    // p2 costante: p = indirizzo di j  
    int * const p2 = &j;  
  
    // cambio il valore di p: p = indirizzo di j  
    p = p2;  
  
    // cambio il valore di p2: p2 = indirizzo di k  
    p2 = &k;    // genera un errore a tempo di  
                // compilazione - p2 costante  
}
```

# Operatore di dereferenziazione

Per accedere all'oggetto riferito da un puntatore si usa l'**operatore di dereferenziazione** \*

## Operatore unario e prefisso

Si dice che il puntatore viene dereferenziato

L'operatore \* applicato ad un puntatore **ritorna un riferimento all'oggetto puntato**

- **\*p** puo' essere tradotto come “l'oggetto puntato da **p**”

# Esempi

```
main() {  
    int i, j;  
    int * p = &i;  
    int * const p2 = &j;  
  
    *p = 3;    // equivale a i = 3  
    *p2 = 4; // equivale a j = 4  
  
    int x = *p; // equivale a x = i  
  
    i = *p2; // equivale a i = j  
}
```

# Osservazione

L'uso dei puntatori **è una delle aree più inclini ad errori** della programmazione moderna

Alcuni linguaggi come Java, C# e Visual Basic **non forniscono alcun tipo di dato puntatore**

Problemi tipici:

- ***Dangling reference (puntatore pendente)***: il puntatore punta ad un'area di memoria già liberata (tramite l'operatore free)
- ***Memory leak***: la memoria allocata non viene liberata (tramite l'operatore free)

# Stampa di puntatori

Il valore di un puntatore, così come del risultato dell'operatore di indirizzo **&**, può essere stampato mandandolo sullo stream di uscita mediante l'operatore **<<**

Di norma l'operatore **<<** mette sullo stream di uscita la sequenza di caratteri che rappresenta il valore di un puntatore in **base 16**

**Vedi programma** *indirizzo\_punt.cc*

# Puntatori a puntatori

Un puntatore può puntare a (contenere l'indirizzo di) un altro puntatore

## ESEMPIO

```
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  
}
```

# Selettori di campo

## ESEMPIO

```
struct s {int a, b ;};
```

```
s *p;
```

Oggetto di tipo struttura indirizzato da un puntatore **p**

*Due modi* per riferire ad un campo della struttura indirizzata da p (ad esempio **a**)

1. `(*p) . a`

2. `p->a`

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



# Esempi

```
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  
}
```

# Ripasso dei Riferimenti

Oltre ai puntatori, il C++ supporta anche il concetto di *riferimento* (non esiste in C)

## ESEMPIO

```
int & rif = n;
```

I riferimenti sono **dichiarati** usando l'operatore **&** (anziché **\***)

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

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

# Riferimenti come alias

```
int & rif = n;
```

Definisce una variabile `rif` di tipo “riferimento a `int`” e la inizializza al valore `n`

**`rif` è un sinonimo di `n`**

## ESEMPIO

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

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

# Puntatori e riferimenti

- I riferimenti sono *meno flessibili, ma meno pericolosi* dei puntatori
- **Riferimento**: realizzato mediante un *puntatore costante nascosto* (non visibile al programmatore) che ha per valore l'indirizzo dell'oggetto riferito

```
int & rif = n;
```

Corrisponde a:

```
int * __ptr_rif = &n; // puntatore nascosto
```

- Ogni operazione che coinvolge l'oggetto riferito è realizzata da una **dereferenziazione sul puntatore nascosto**

... ogni volta che viene usato **rif** viene sostituito da (**\*\_\_ptr\_rif**)

## Differenze sostanziali tra riferimenti e puntatori:

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

# Riferimenti e parametri

I riferimenti sono usati soprattutto 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

Come qualsiasi altro tipo di oggetto, anche un *oggetto di tipo puntatore* può essere *passato attraverso un riferimento*

# Esempio riassuntivo (1)

```
void f(int i, int *p, int &ri, int *&rp) {  
    int *q = new int ;  
    i = 10 ;      // nuovo valore dell'argomento i  
    p = q ;       // nuovo valore dell'argomento p  
    *p = 10 ;     // nuovo valore dell'oggetto  
                  // puntato da p  
    ri = 10 ;     // nuovo valore dell'oggetto di  
                  // nome (sinonimo) ri  
    rp = q ;     // nuovo valore dell'oggetto di  
                  // nome (sinonimo) rp  
    *rp = 30;     // nuovo valore dell'oggetto  
                  // puntato dal puntatore  
                  // di nome (sinonimo) rp  
}
```

# Esempio riassuntivo (2)

```
int main()
{
    int a = 20, b=15, c=12, d=8;
    int *punt;
    punt = &b;
    int *prp;
    prp = &d;
    f(a, punt, c, prp);
    cout << a << " " << *punt << " " << c << " "
    << *prp << " " << endl;
}
```

Qual è la differenza tra `punt` che viene passato per valore e `prp` che viene passato per riferimento?

Cosa stampa? [Vedi programma `punt\_rif.cc`](#)

# Puntatori ed array

- Il nome di un array corrisponde ad un puntatore al primo elemento dell'array stesso

**SE**  $x[N]$  è un array di  $N$  elementi di tipo  $T$

**ALLORA**  $x$  equivale a  $\&x[0]$  (*riferimento*)

- Tale puntatore è (ovviamente) *costante*
- Questo spiega perché l'assegnamento tra due array dà luogo ad un errore a tempo di compilazione
- Se dichiaro un puntatore  $p$  di tipo  $T$  allora posso assegnare a  $p$  l'indirizzo del primo elemento di  $x$  nel seguente modo:

$T^* p = x;$



# Aritmetica degli indirizzi

- I puntatori possono essere usati in espressioni aritmetiche
- **ARITMETICA DEGLI INDIRIZZI (O DEI PUNTATORI)**
- Insieme di regole che governano le operazioni effettuabili sugli indirizzi
- Deriva dalla stretta correlazione tra puntatori e vettori

## ***Somma di un intero***

Sia  $\mathbf{p}$  un puntatore contenente l'indirizzo di un oggetto di tipo  $\mathbf{T}$

→ l'espressione  $\mathbf{p} + \mathbf{i}$  restituisce come valore l'indirizzo di un oggetto di tipo  $\mathbf{T}$  che si trova in memoria dopo  $\mathbf{i}$  oggetti consecutivi di tipo  $\mathbf{T}$  (o prima se  $\mathbf{i}$  è negativo)

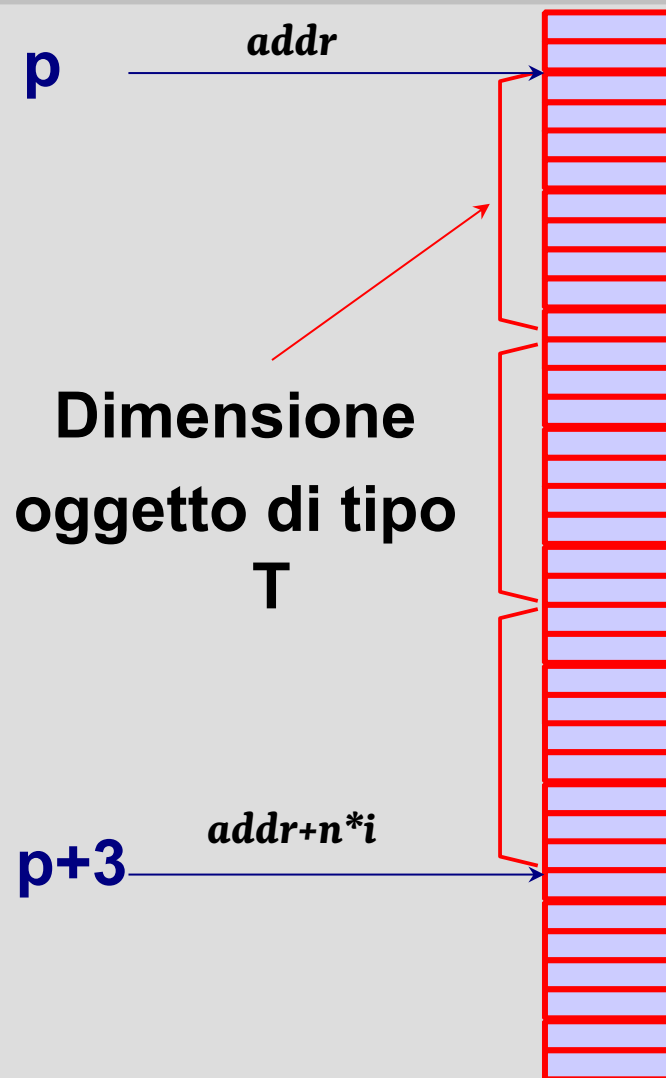
# Somma di un intero

**SE**

**p** ha come valore numerico  
l'indirizzo **addr**, e **T** occupa **n**  
locazioni di memoria

**ALLORA**

l'espressione **p+i** ha come valore  
numerico l'indirizzo **addr+n\*i**



# Altre operazioni con puntatori

## *Incremento (++)* e *Decremento (--)*

Sia **p** un puntatore contenente l'indirizzo di un oggetto **x** di tipo **T**

→ l'espressione **p++** (o **p--**) restituisce come valore l'indirizzo di un oggetto di tipo **T** che segue (o precede) immediatamente **x** in memoria

## *Differenza* tra due indirizzi di oggetti di tipo **T**

Restituisce il numero di elementi di tipo **T** contenuti nella zona di memoria compresa tra i due indirizzi

# Esempio 1

```
main() {  
    const int N = 10 ;  
    int v[N] ;
```

```
    int *p = v ;    // è legale ? Cosa fa?
```

*E' legale: assegna a p l'indirizzo  
(del primo elemento) di v*

```
    *(p + 2) = 7 ;    // è legale ? Cosa fa?
```

*E' legale: equivale a v[2]=7*

```
}
```

# Esempio 2

```
main() {  
    const int N = 10 ;  
    int v[N], z[N] ;  
    *v = *z ;        // è legale ? cosa fa?  
}
```

*è legale... equivale a  $v[0] = z[0]$*

# Esercizio

Scrivere un programma che stampi il contenuto di un vettore di interi *fino al valore -1 escluso* attraverso due funzioni distinte:

Entrambe le funzioni non devono utilizzare l'operazione di selezione con indice (ovvero [])

La seconda funzione, inoltre, non deve utilizzare nemmeno una variabile locale

Per la traccia, **vedi programma** *stampa\_array\_pun.cc*

Per la soluzione, **vedi programma** *stampa\_array\_pun\_sol.cc*