Parte 3

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

Ma i puntatori possono essere utilizzati per ***riferire 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`**
- Se non si tratta di array, non si utilizzano le parentesi quadre nè si indica il numero di elementi quando si utilizza l'operatore **new**
- La sintassi **nome_tipo *** può essere usata per dichiarare un puntatore ad un oggetto singolo o ad un array (con parentesi quadre)
- **Perchè è importante specificare nome_tipo?**

Esempi (1)

```
main() {
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)

```
main() {
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:
             // NON è un array!

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]

int *p = new int[10]
```

Ripasso

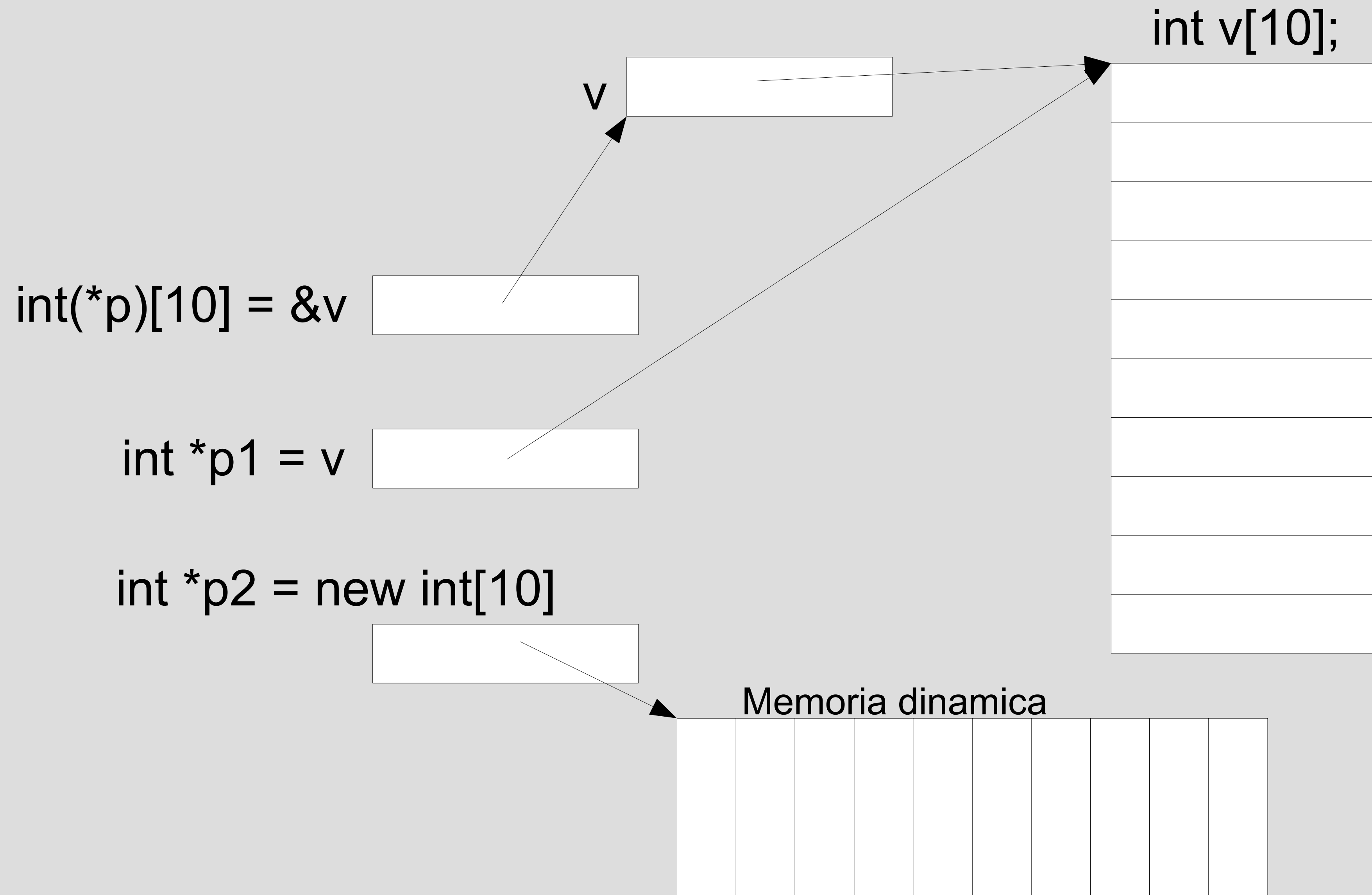
Cosa rappresentano i seguenti oggetti?

```
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

int *p = new int[10] // puntatore (ad int) al I
                     // elemento di un array di
                     // 10 int allocato in
                     // memoria dinamica
```

Puntatori a vettori



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`** puo' essere tradotto come “l'indirizzo di **`x`**”

Esempi

```
main() {
int i, j;
int *p = &i;
int * const p2 = &j;  //quando possibile
                     //è bene usare questa
                     //definizione
p = p2;  // equivale a p = indirizzo di j
int k;
p2 = &k;  // genera un errore a tempo di
          // compilazione - p2 costante
}
```

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)
  - ***Memory leak*** – memoria irraggiungibile causa perdita del puntatore

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
}
```

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***
- 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

- Oggetto di tipo struttura indirizzato da un puntatore **p**
- ***Due modi*** per riferire un campo **m** della struttura indirizzata da p

1) **p->m**
2) **(*p).m**

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
}
```

Riferimenti

- Oltre ai puntatori, il C++ supporta anche il concetto di ***riferimento*** **(non esiste in C)**
- **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
- I riferimenti sono **dichiarati** usando l'operatore **&** (anziché *)

Esempi di riferimenti

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

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

ES:

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

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

Puntatori e riferimenti (1)

- ***Differenze sostanziali***:
  - I riferimenti non possono avere valore nullo → necessaria inizializzazione
  - I riferimenti non possono poi essere riassegnati
- 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

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

Implementazione riferimenti

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

Corrisponde a:

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

e ogni volta che viene usato `rif` viene sostituito da

(`*__ptr_rif`)

Puntatori e riferimenti (2)

- I riferimenti sono usati soprattutto per la ***dichiarazione dei parametri***
- **Passaggio per valore**: impedisce i cambiamenti e spreca memoria per le copie
- **Passaggio attraverso puntatori**: introduce il rischio di usi scorretti (puntatori nulli, tentativi di modifiche al puntatore)

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

Esempio riassuntivo

```
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;
}
```

Esempio riassuntivo

```
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
}
```

Cosa stampa? programma *punt_rif.cc*

Aritmetica degli indirizzi

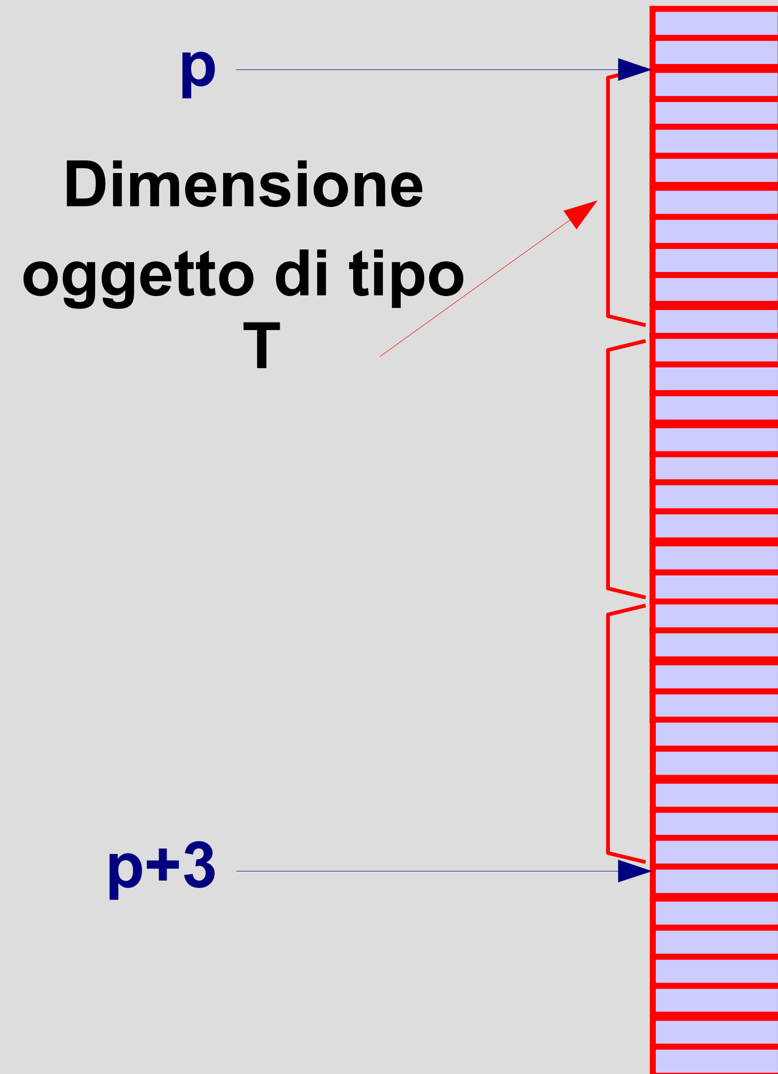
- Insieme di regole che governano le operazioni effettuabili sugli indirizzi
- Detta anche ***aritmetica dei puntatori***
- Esempio: somma di un intero
- Sia **p** un puntatore contenente l'indirizzo di un oggetto di tipo ***T***

  → 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*** (o prima se **i** è negativo)

Somma di un intero

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

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



Altre operazioni possibili

- ***Incremento*** e ***decremento*** di un puntatore ad un oggetto **x** di tipo ***T***
  - assegnano a **p** l'indirizzo dell'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

Puntatori ed array

- Il nome di un array corrisponde ad un puntatore al primo elemento dell'array stesso
  - Tale puntatore è (ovviamente) ***costante***
- Quindi se **x[N]** è un array di **N** elementi
  - **x** equivale a **&x[0]** ***(riferimento)***
- Questo spiega perché l'assegnamento tra due array dà luogo ad un errore a tempo di compilazione
- In funzione dell'aritmetica dei puntatori, si ha:
  - ***(x + i)** equivale a **x[i]**

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]

Programma

- *stampa_array_pun.cc*
- Programma che stampa il contenuto di un vettore di interi attraverso due funzioni distinte
- Entrambe le funzioni non devono utilizzare l'operazione di selezione con indice
- La seconda funzione, inoltre, non deve utilizzare nemmeno una variabile locale