Il linguaggio C

Puntatori e dintorni

Puntatori : idea di base

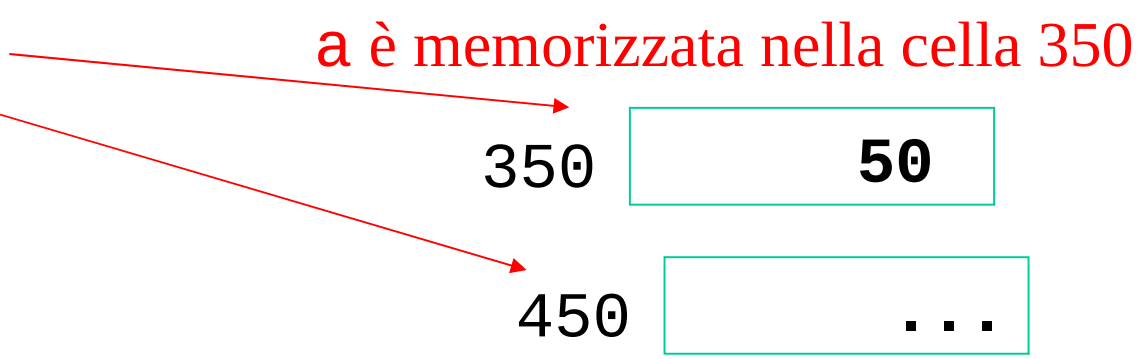
- In C è possibile *conoscere* e *denotare* l'indirizzo della cella di memoria in cui è memorizzata una variabile (il *puntatore*)
  - es :

```
int a = 50; /* una var intera */
int *   b; /* una var puntatore a interi
 */
  ...
b = &a;
/* assegna a b l'indirizzo della cella in
 cui è memorizzata a */
```

Puntatori : idea di base (2)

- In C è possibile *conoscere* e *denotare* l'indirizzo della cella di memoria in cui è memorizzata una variabile (il *puntatore*)
  - es :

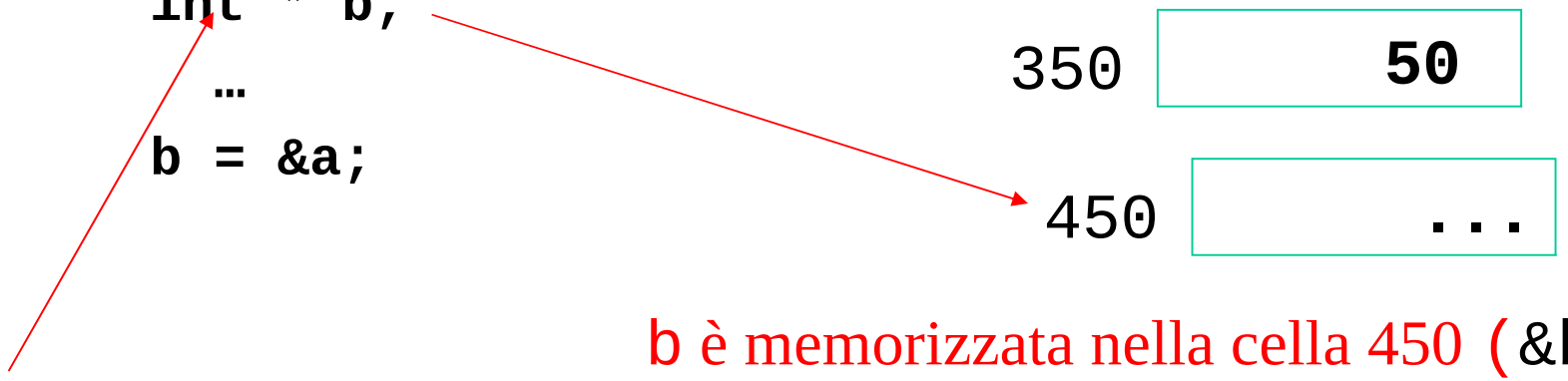
```
int a = 50;
int *b;
  …
b = &a;
```



Puntatori : idea di base (3)

- **`nometype *`**
  - è il tipo degli indirizzi delle variabili di tipo `nometype`
    - es :

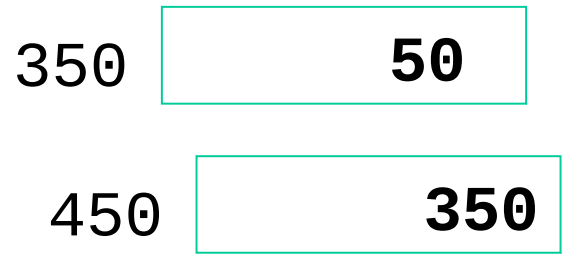
```
int a = 50;
int * b;
  …
b = &a;
```



Puntatori : idea di base (4)

- Operatore &
  - denota l'indirizzo della cella di memoria in cui è memorizzata una variabile (il *puntatore*)
    - es :

```
int a = 50;
int *b;
  …
b = &a;
```

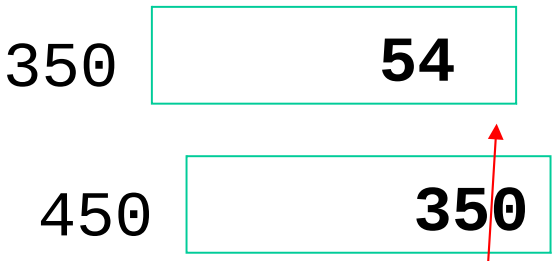


Dopo questo assegnamento in b è memorizzato l'indirizzo di a

Puntatori : idea di base (5)

- Operatore di *dereferenziazione* ‘ * ’
  - è possibile *conoscere e/o modificare* il contenuto di una variabile manipolando direttamente il suo puntatore
    - es :

```
int a = 50;
int *b = &a;
…
*b = *b + 4;
```



Denota la variabile a indirizzo b

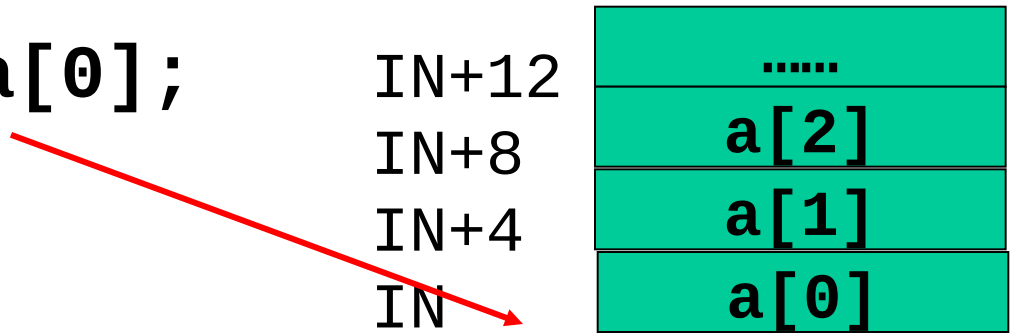
Dopo questo assegnamento in a è memorizzato il valore 50 + 4

Puntatori : idea di base (6)

- **`NULL`**
  - costante predefinita (in **`stdio.h`**) che denota il puntatore nullo
- È possibile definire puntatori per tutti itipi base e le strutture con (*)
  - `double  *a, *b;  /* ripetere '*' */`
  - `int *a, b, c[4], **d;`
  - `struct studente * t1;`
- Segnaposto (**`%p`**)
  - stampa il valore dell'indirizzo in notazione esadecimale

Aritmetica dei puntatori

- È possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)
  - **`int a[3],*p=&a[0];`**

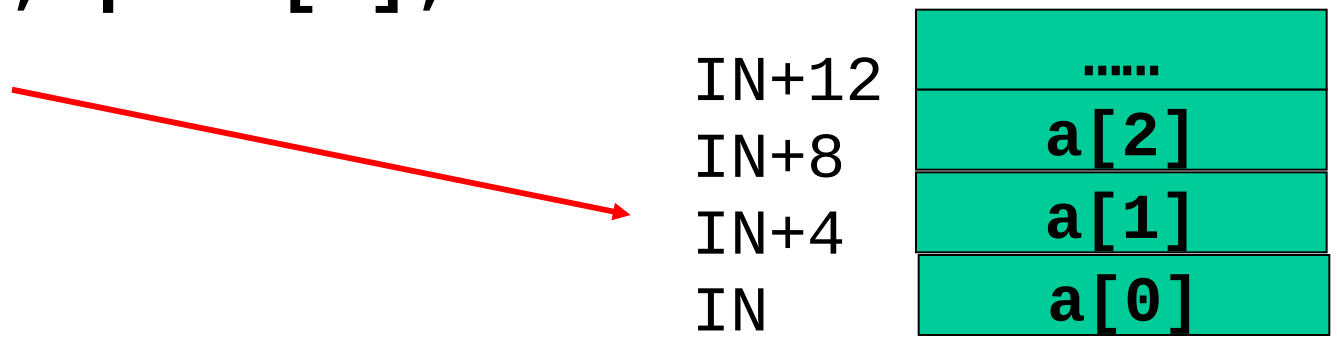


p contiene l'indirizzo IN (es. IN=200)

Aritmetica dei puntatori (2)

- È possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici

```
int a[3],*p=&a[0];
p = p+1;
```

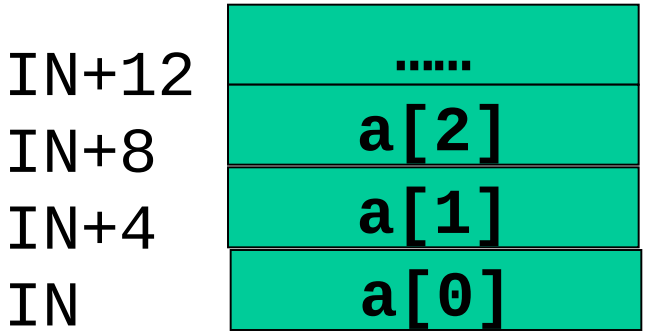


p contiene l'indirizzo IN + 4

Aritmetica dei puntatori (3)

- È possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici

```
int a[3],*p=&a[0];
p = p+1;
p--;
```

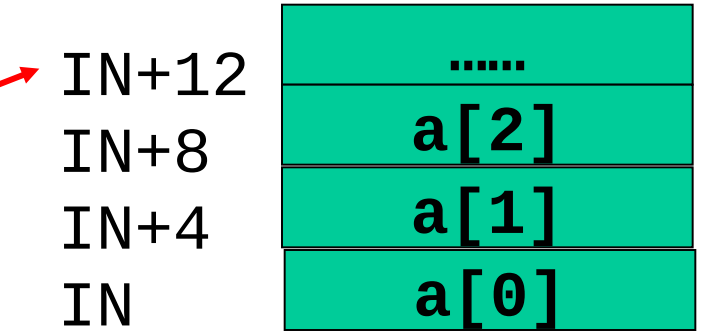


p contiene l'indirizzo IN

Aritmetica dei puntatori (4)

- È possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, + +)

```
int a[3],*p=&a[0];
p = p+1;
p--;
p+=3;
```

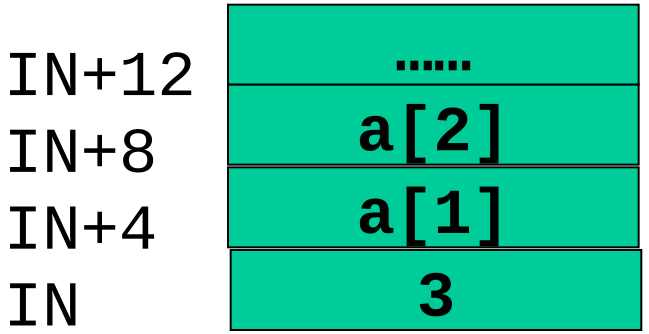


p contiene l'indirizzo IN + 12
(sizeof(int)==4…..)

Aritmetica dei puntatori (5)

- È possibile scrivere espressioni puntatore usando alcuni degli usuali operatori aritmetici (+, -, --, ++)

```
int a[3],*p=&a[0],*q;
p = p+1;
p--;
q = p;
p+=3;
a[0] = p-q;
```

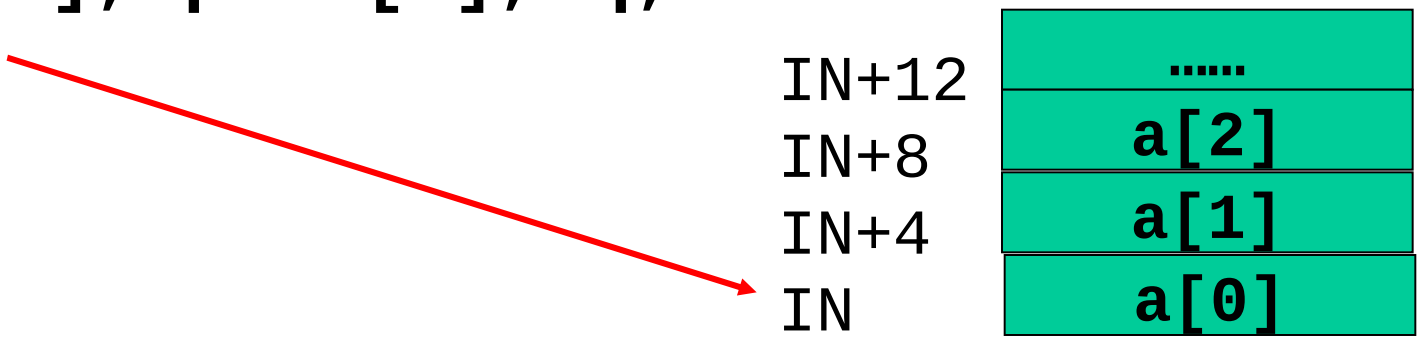


a[0] contiene 3, numero di int memorizzabili fra p e q

Puntatori e array….

- Il nome di un array, è il puntatore (costante) al primo elemento dell'array

```
int a[3],*p=&a[0],*q;
q = a;
```



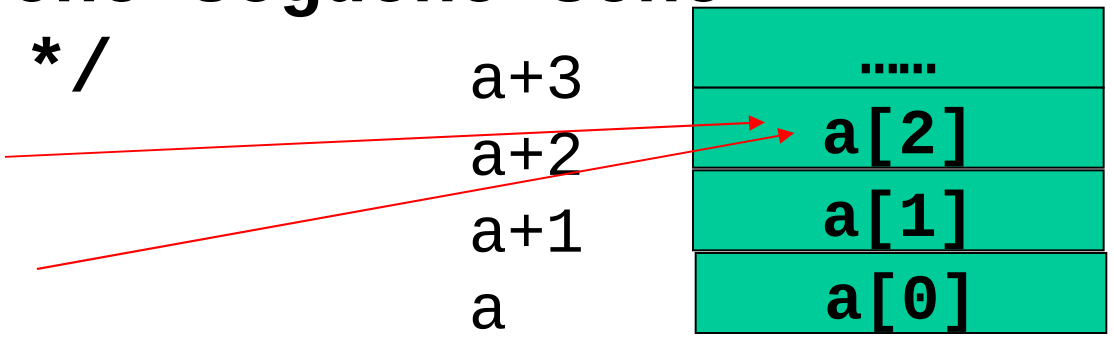
q contiene l'indirizzo IN

a == IN

Puntatori e array…. (2)

- L'operatore [-] è una abbreviazione ….

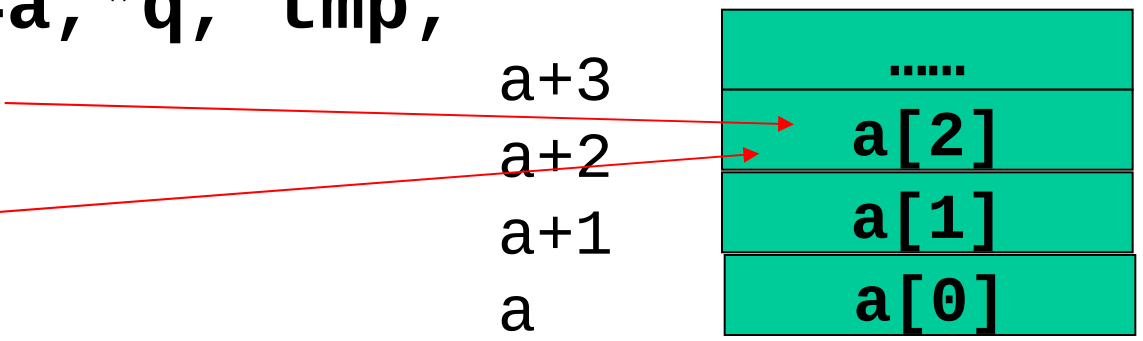
```
int a[3],*p=&a[0],*q, tmp;
/* i due stm che seguono sono
  equivalenti */
tmp = a[2];
tmp = *(a+2);
```



Puntatori e array…. (3)

- L'operatore [–] è una abbreviazione …. *e può essere usato con una qualsiasi variabile puntatore*

```
int a[3],*p=a,*q, tmp;
tmp = a[2];
tmp = p[2];
```



Puntatori e array…. (4)

- I seguenti frammenti di codice sono equivalenti :

```
int a[N],*p=a,*q, i;
int sum = 0;
/* versione 1 */
for(i=0;i<N;i++)
   sum+= a[i];
/* versione 2 */
for(i=0;i<N;i++)
   sum+= *(a+i);
```

Puntatori e array…. (5)

- I seguenti frammenti di codice sono equivalenti (segue) :

```
int a[N],*p=&a[0],*q, i;
int sum = 0;
/* versione 3 */
for(i=0;i<N;i++)
    sum+= p[i];
/* versione 4 */
for(p=a;p<(a+N);p++)
    sum+= *p;
```

Stringhe

- Una riflessione sulle stringhe in C
  - le stringhe sono array di caratteri terminate dal carattere nullo **'\0',** ad esempio

    char a[5]="ciao";

    dichiara una stringa notate che la lunghezza dell'array deve comprendere anche l'invisibile terminatore.
  - La libreria **string.h** mette a disposizione varie funzioni per lavorare sulle stringhe
  - La funzione **strlen()** da la lunghezza di una stringa in caratteri, escludendo il terminatore

    strlen(a)=strlen("ciao")=4

Stringhe (2)

- Altre funzioni interessanti:
  - **`strcpy(char* s,char*p)`** che copia la stringa **`p`** nella stringa **`s`**
  - **`strcmp(char* s,char*p)`**che confronta lessicograficamente **`p`** ed **`s`**
  - **`strcat(char* s,char*p)`**che concatena **`p`** ed **`s`** (modifica **`s)`**
  - **`strstr(char* s, char*p)`** che cerca la prima occorrenza della stringa **`p`** in **`s`** e restituisce il puntatore a tale occorrenza

Stringhe (3)

- Se non gestite bene le stringhe sono pericolose e generano errori difficili da rilevare e catastrofici
  - Le funzioni di libreria si aspettano sempre di lavorare con stringhe correttamente terminate dal carattere nullo **`'\0',`**
  - Es: implementazione di **`strcpy`** da K&R

```
void strcpy (char*s, char*t){
  while ( ( *s++ = *t++ ) != '\0');
}
```

Stringhe (4)

```
void strcpy (char*s, char*t){
  while ( ( *s++ = *t++ ) != '\0');
}
```

- Se la stringa non è terminata (o se s non ha abbastanza spazio) si continuano ad incrementare i puntatori andando avanti a leggere (e scrivere!) valori in memoria *(buffer overrun)*
- Si può sovrascrivere e danneggiare lo spazio di memoria di altre variabili, i frame sullo stack o la tabella di allocazione dello heap
- Si può raggiungere memoria non allocata ricevendo segnali di violazione di Segmento con conseguente terminazione del processo in esecuzione

Stringhe (5)

- **Morale :**
  - Assicuratevi sempre che le stringhe siano terminate e che ci sia abbastanza spazio nei buffer
  - Se non siete sicuri usate funzioni che non permettano l'overrun, perchè è possibile dire quanto è grande il buffer
  - Es: **`strncpy(char* s,char*p, size_t n)`** in cui il terzo parametro serve per dire quanto è lungo il buffer
    - In questo caso dopo la copia bisogna controllare che il risultato contenga effettivamente il terminatore perchè la fine del buffer può essere stata raggiunta prima
  - Usate **`valgrind`** se avete dubbi sul comportamento del vostro programma (segnala scritture e letture fuori dai buffer – sullo heap)