Il linguaggio C

Puntatori e dintorni

Puntatori : idea di base

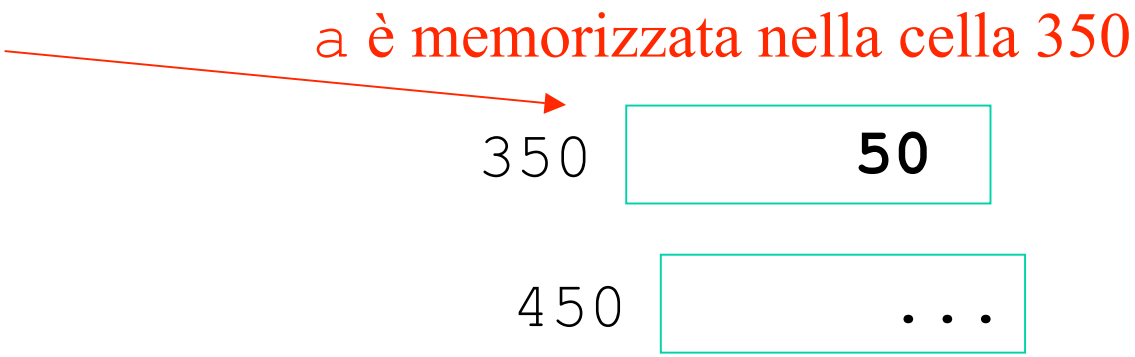
- In C è possibile *conoscere* e *denotare* l'indirizzo della cella di memoria in cui è memorizzata una variabile (il *puntatore*)
  - es :

```
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 :

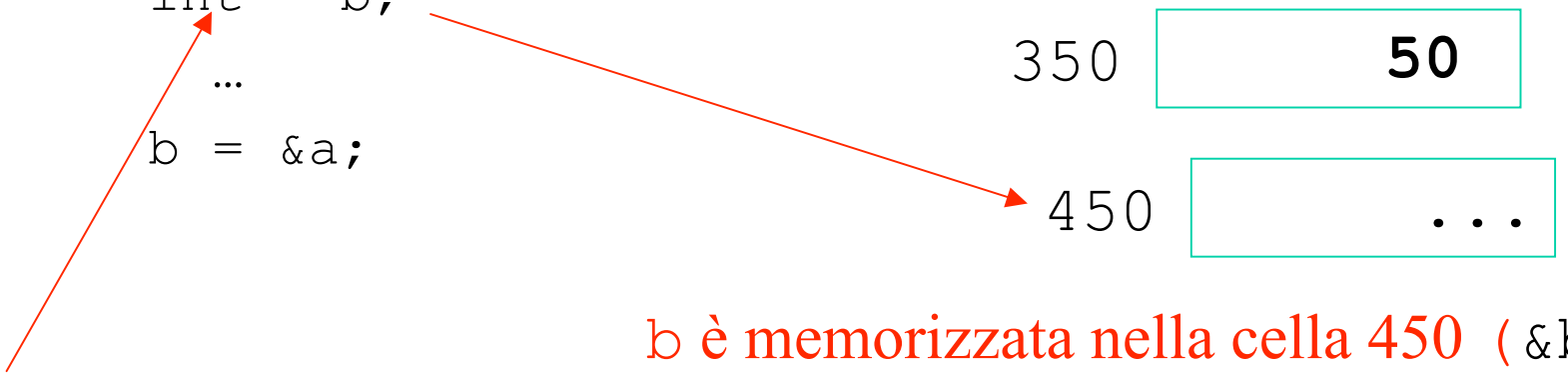
```
int a = 50;
int * b;
  …
b = &a;
```



Puntatori : idea di base (3)

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

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



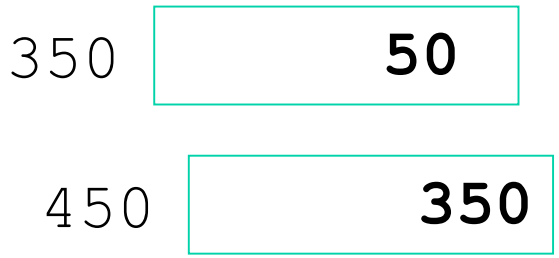
`b` è memorizzata nella cella 450 (`&b`)

tipo dei puntatori a intero

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;
```



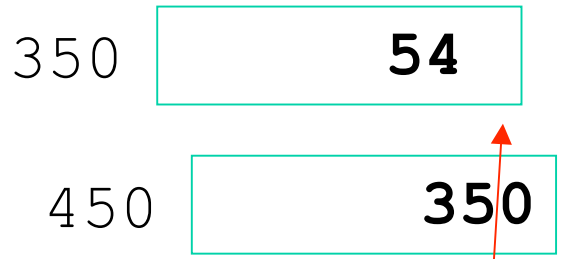
Dopo questo assegnamento in `b` è memorizzato l'indirizzo di `a`

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`



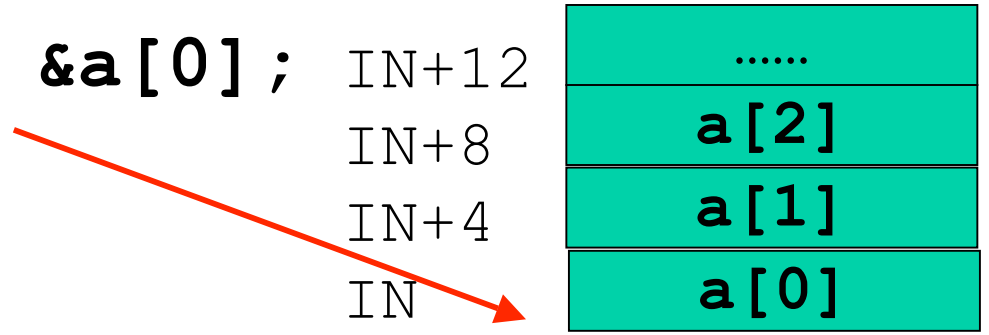
Dopo questo assegnamento in `a` è memorizzato il valore 50 + 4

Puntatori : idea di base (6)

- **`NULL`**
  - costante predefinita (in `stdio.h`) che denota il puntatore nullo
- È possibile definire puntatori per tutti i tipi 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];`

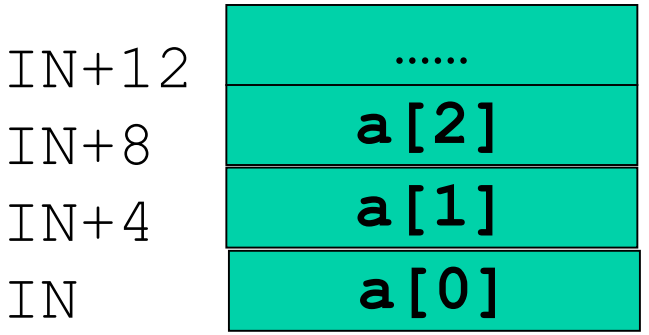


p contiene l'indirizzo IN

Aritmetica dei puntatori (2)

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

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

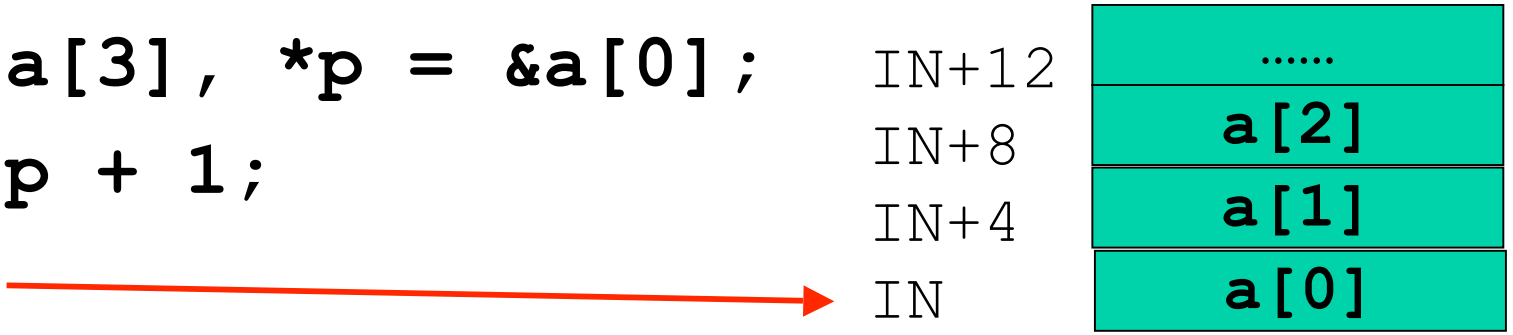


p contiene l'indirizzo IN + 4

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--;
```

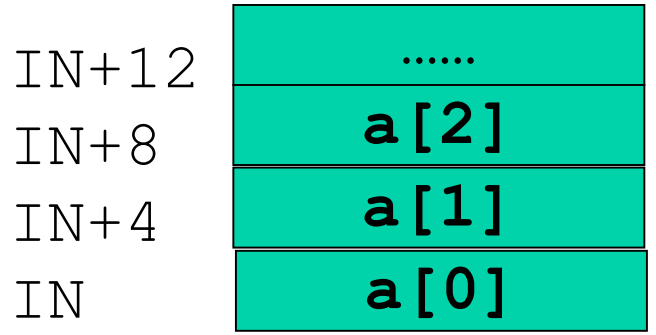


p contiene l'indirizzo IN

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;
```

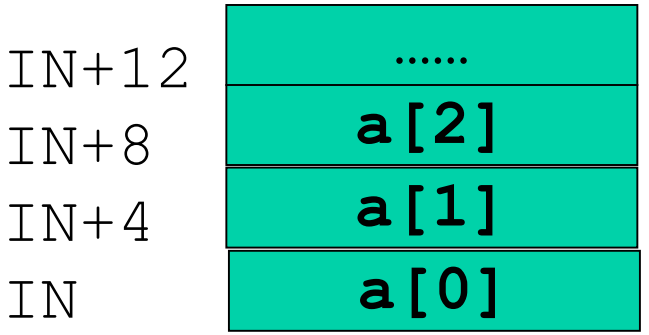


p contiene l'indirizzo IN + 12
(sizeof(int) == 4…..)

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;
```

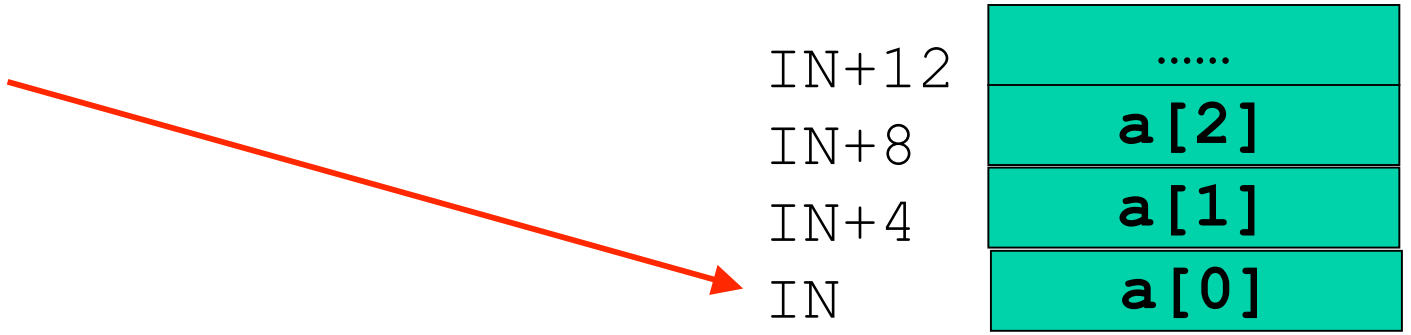


a[0] contiene 3, numero di int memorizzabili fra p e q

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;
```

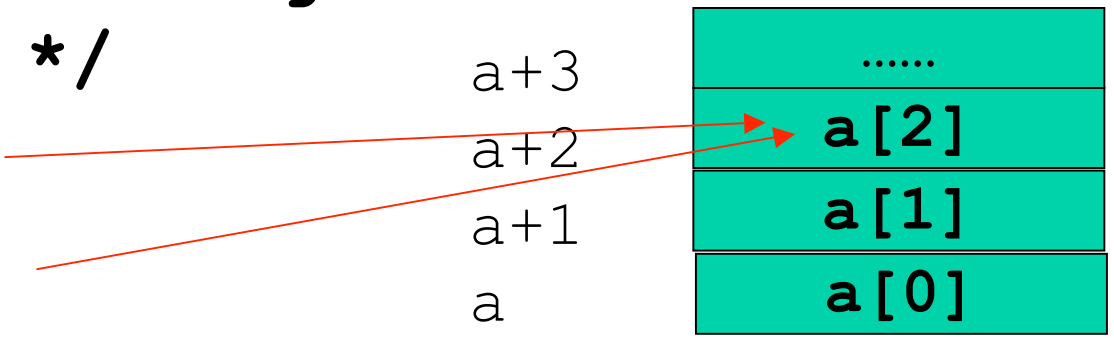


q contiene l'indirizzo IN
a == IN

Puntatori e array… (2)

- L'operatore [-] è una abbreviazione …

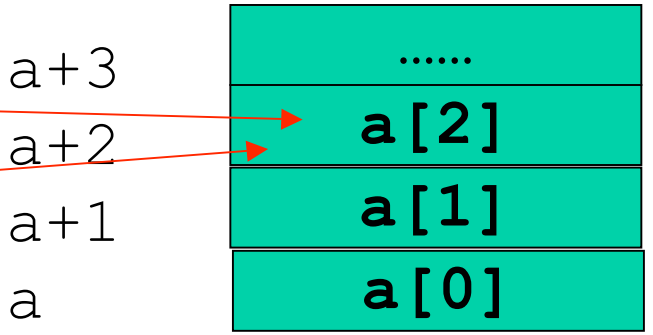
```
int a[3], *p = &a[0], *q, tmp;
/* i due stm che seguono sono
   equivalenti */
tmp = a[2];
tmp = *(a+2);
```



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, tmp;
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, tmp;
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;
```

Puntatori e array… (6)

- Una riflessione sulle stringhe
  - le stringhe sono array di caratteri

```
char a[7] = "ciao";
```

  - le stringhe costanti possono essere definite anche come

```
const char * pp = "ciao";
```

```
char * pp = "ciao";
```

    - attenzione! Se a questo punto cercate di modificare un elemento di pp (es. `pp[2] = 'f';`) avete un errore a run time
    - mentre `a[2] = 'f';` è completamente corretto

Passaggio di parametri per riferimento

- Tutti i parametri delle funzioni C sono passati per valore
  - il loro valore viene copiato sullo stack
  - ogni modifica al parametro nel corpo della funzione non modifica l'originale
- È possibile realizzare passaggi per riferimento utilizzando i puntatori
  - i passaggi per riferimento permettono di modificare il valore di una variabile nell'ambiente del chiamante

Passaggio di parametri per riferimento (2)

- Esempio : la funzione che scambia fra loro i valori di due variabili
  - si potrebbe pensare di programmarla come ...

```
void scambia (int x, int y){
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```

  - e poi chiamare **`scambia(a,b)`**

Passaggio di parametri per riferimento (3)

- Esempio : la funzione che scambia fra loro i valori di due variabili
  - si potrebbe pensare di programmarla come ...

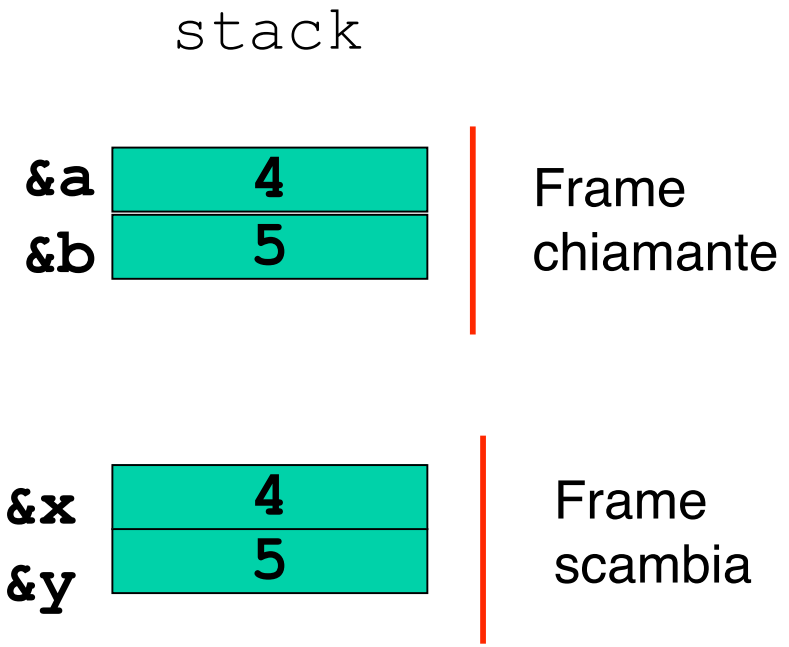
```
void scambia (int x, int y){
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```

  - **non funziona! Perché lo scambio viene fatto sulle copie**

Passaggio di parametri per riferimento (3.1)

- Esempio : la funzione che scambia fra loro i valori di due variabili
  - esempio di chiamata

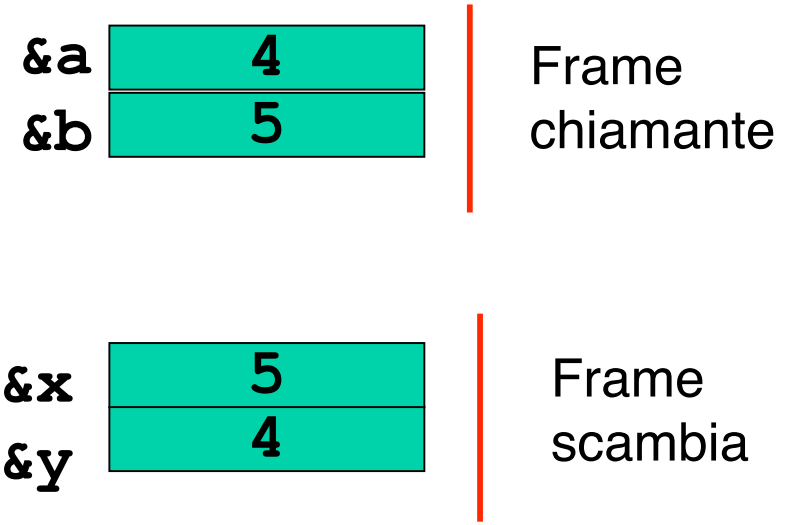
```
int a = 4, b = 5;
scambia(a,b);
```



Passaggio di parametri per riferimento (3.2)

- Esempio : la funzione che scambia fra loro i valori di due variabili
  - alla fine dell'esecuzione di scambia (prima di ritornare al chiamante)

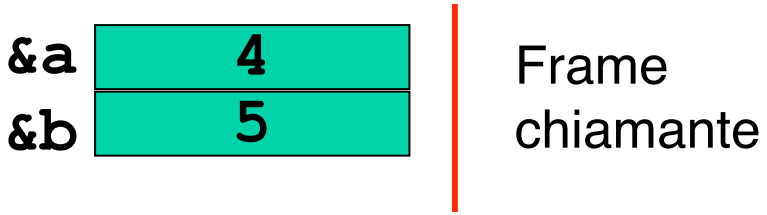
```
int a = 4, b = 5;
scambia(a,b);
```



Passaggio di parametri per riferimento (3.3)

- Esempio : la funzione che scambia fra loro i valori di due variabili
  - al momento di eseguire la printf (il frame di scambia non è più significativo)

```
int a = 4, b = 5;
scambia(a,b);
printf("%d,%d",a,b);
```



Passaggio di parametri per riferimento (4)

- Esempio : la funzione che scambia fra loro i valori di due variabili
  - la versione corretta è ...

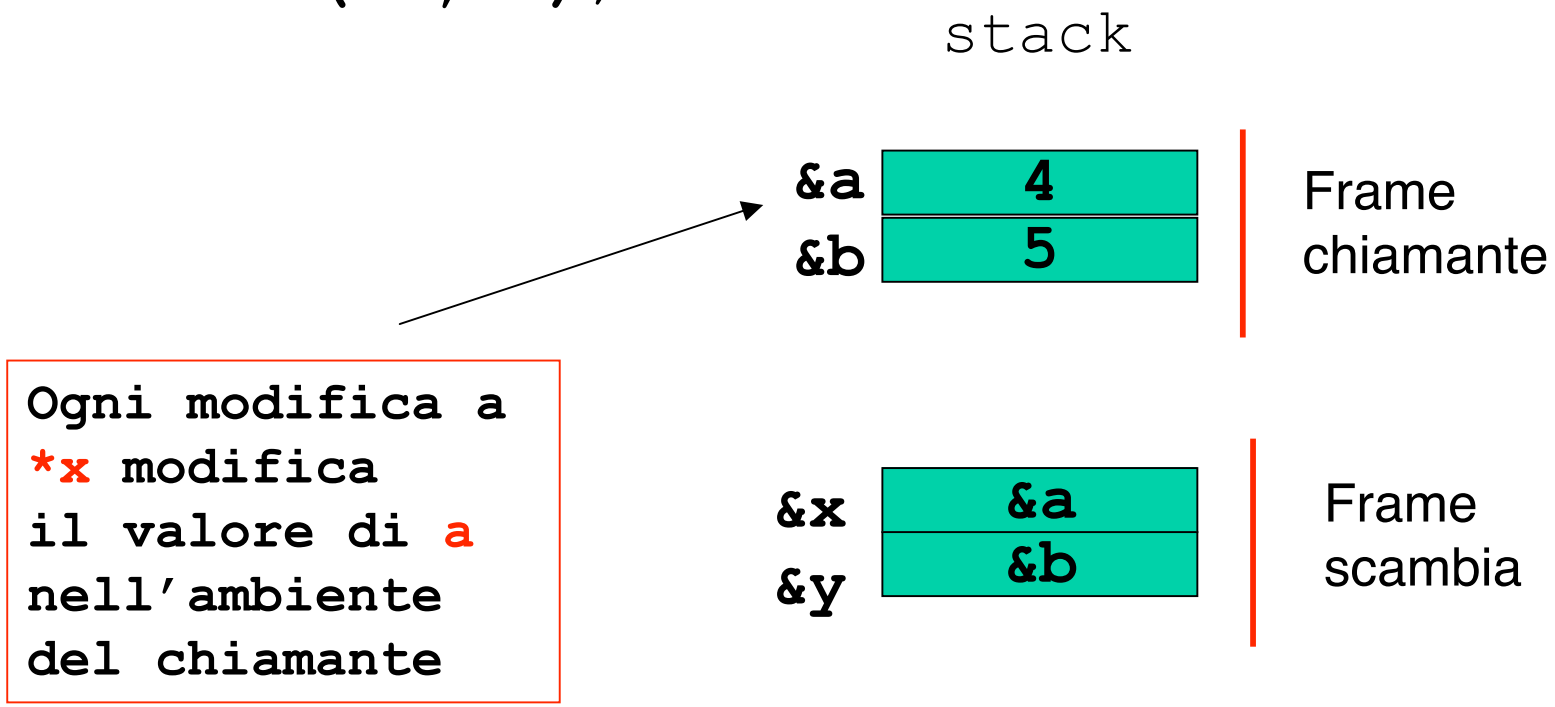
```
void scambia (int *x, int *y){
   int tmp;
   tmp = *x;
   *x = *y;
   *y = tmp;
}
```

  - con chiamata **`scambia(&a,&b)`**

Passaggio di parametri per riferimento (4.1)

- Versione corretta di scambia ...
  - esempio di chiamata

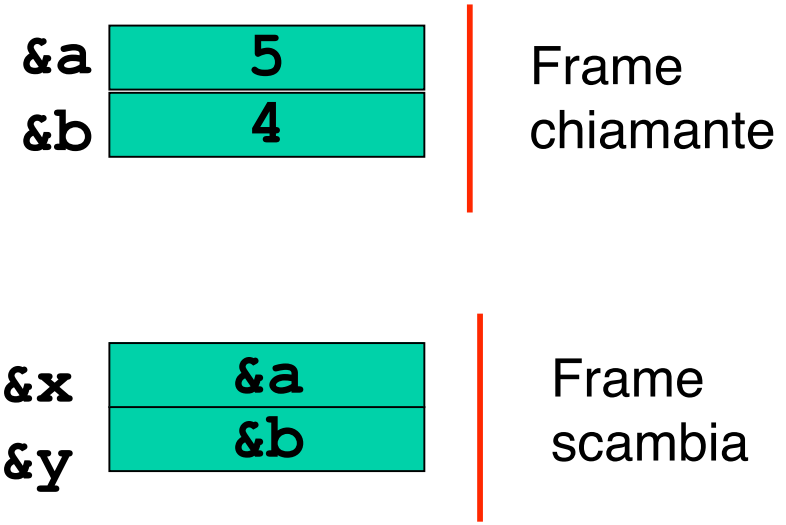
```
int a = 4, b = 5;
scambia(&a,&b);
```



Passaggio di parametri per riferimento (4.2)

- Esempio : versione corretta di scambia ...
  - alla fine dell'esecuzione di scambia (prima di ritornare al chiamante)

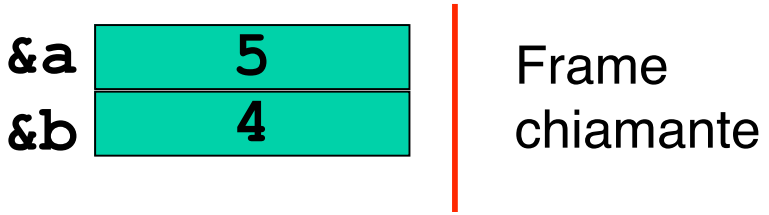
```
int a = 4, b = 5;
scambia(&a,&b);
```



Passaggio di parametri per riferimento (4.3)

- Esempio : versione corretta di scambia ...
  - al momento di eseguire la printf (il frame di scambia non è più significativo)

```
int a = 4, b = 5;
scambia(&a,&b);
printf("%d,%d",a,b);
```



Passaggio di parametri per riferimento (5)

- ATTENZIONE : gli array sono passati sempre per riferimento perché quello che si passa è il nome dell'array

```
void assegna (int x[ ]) {
  x[0] = 13;
}
```

  - con chiamata

```
int a[10];
assegna(a);
/* qua a[0] vale 13 */
```

Passaggio di parametri per riferimento (6)

- Inoltre : le due scritture

```
void assegna (int x[ ]){
  x[0] = 13;
}
```

e

```
void assegna (int *x){
  x[0] = 13;
}
```

  - sono del tutto equivalenti
  - si preferisce usare la prima per leggibilità

Passaggio di parametri per riferimento (7)

- Tipicamente le funzioni che lavorano su array hanno un secondo parametro che fornisce la lunghezza

```
int somma (int x[ ], int l){
  int i, s = 0;
  for(i = 0; i < l; i++)
    s += x[i];
  return s;
}
```

  - somma tutti gli elementi di un array intero di lunghezza `l`

Passaggio di parametri per riferimento (8)

- Per gli array multidimensionali la cosa è più complessa!!!!

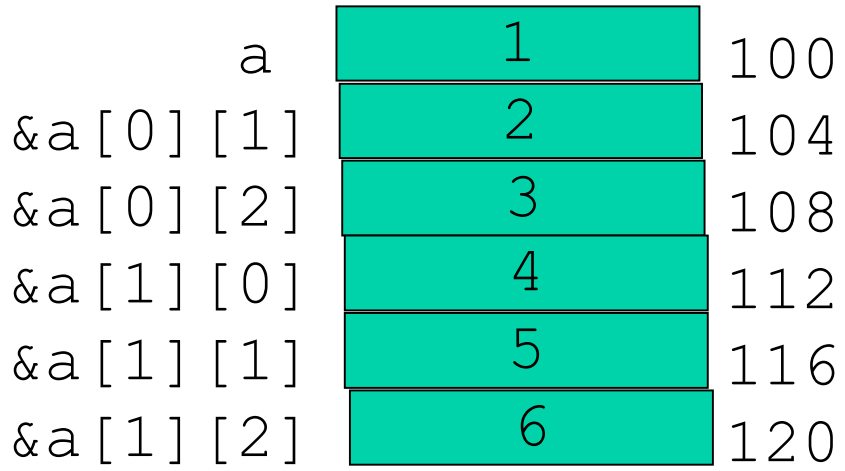
```
int somma (int x[ ][4], int l){
  int i, j, s = 0;
  for(i = 0; i < l; i++)
    for(j = 0; j < 4; j++)
      s += x[i][j];
  return s;
}
```

  - invece di 4 posso usare **N** costante

Passaggio di parametri per riferimento (9)

- Perché dobbiamo specificare l'ampiezza di una seconda dimensione di un array ?
  - Dipende dalla strategia di memorizzazione per gli array multidimensionali
    - es: **`int a[2][3]={{1,2,3},{4,5,6}};`**

`a[i][j]`
ha come indirizzo **`a+i*3+j`**



Passaggio di parametri per riferimento (10)

- Se non conosco la lunghezza della seconda dimensione il compilatore non riesce a generare codice corretto

```
int somma (int x[ ][4], int l){
  int i, j, s = 0;
  for(i = 0; i < l; i++)
    for(j = 0; j < 4; j++)
      s += x[i][j];
  return s;
}
```

Passaggio di parametri per riferimento (11)

- Esiste un modo migliore di quello appena esaminato per la rappresentazione degli array multidimensionali in C : lo vedremo in seguito

E le strutture ???

- Le strutture vengono sempre passate per valore
- Se una struttura contiene un array, allora l'array viene copiato!
  - Attenzione quando si passano strutture con campi array molto grandi!
- Se voglio modificare una struttura devo sempre utilizzare i puntatori!

E le strutture ??? (2)

- Esempio

```
typedef struct studente {
    char nom_cogn[40];
    unsigned int matricola;
} studente;

void scambia (studente * s1,
  studente * s2);
```

E le strutture ??? (3)

- Esempio

```
void scambia (studente * s1,
  studente *  s2) {
…
(*s1).matricola = 4;
 s1->matricola = 4;
/*sono equivalenti */
}
```

E le strutture ??? (4)

- Esempio : Come si dichiara una lista in C ?
  - Usando i puntatori

```
typedef struct nodo {
    struct nodo * next;
    int info;
} nodo;
```

Mi serve il nome della struttura !

Allocazione dinamica della memoria

- La creazione di nuove variabili a run time (tipo new in Java) viene effettuata in C utilizzando le funzioni di libreria standard che permettono l'allocazione dinamica di porzioni contigue di memoria
  - `malloc, calloc, realloc`
  - `#include <stdlib.h>`

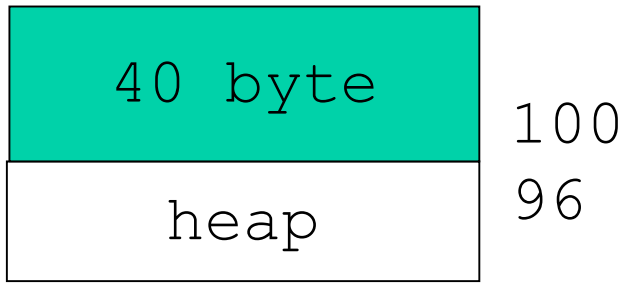
    Con queste primitive è possibile creare dinamicamente variabili e array di dimensione non nota a priori

Array dinamici -- malloc( )

- Vediamo come creare dinamicamente un array di 10 posizioni

```
int * a; /*conterrà il puntatore al
          primo elemento dell'array*/
a = malloc(10*sizeof(int));
```

Punta all'indirizzo iniziale della nuova area allocata



Array dinamici -- malloc( ) (2)

- Vediamo come creare dinamicamente un array di 10 posizioni

```
int * a; /*conterrà il puntatore al
          primo elemento dell'array*/
a = malloc(10*sizeof(int));
if (a == NULL) printf("fallimento!\n");
```

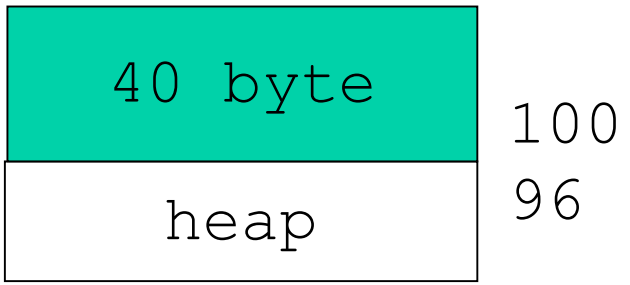
Se malloc non riesce ad allocare
l'area di memoria richiesta restituisce NULL
(verificare…)

heap

Array dinamici -- malloc( ) (3)

- Vediamo come creare dinamicamente un array di 10 posizioni

```
int * a; /*conterrà il puntatore al
          primo elemento dell'array*/
a = malloc(10*sizeof(int));
if (a == NULL) printf("fallimento!\n");
else {
  a[4] = 345;
… }
```



L'array si può accedere con i consueti operatori (come se fosse statico)

Array dinamici -- malloc( ) (4)

- malloc non inizializza la memoria a 0!
  - Possiamo inizializzarla esplicitamente o
  - usare `calloc`

```
int * a; /*conterrà il puntatore al
          primo elemento dell'array*/
a = calloc(10*sizeof(int));
If (a == NULL) printf("fallimento!\n");
else {
  a[4] = 345;
… }
```

Array dinamici -- realloc( )

- realloc modifica la lunghezza di un'area allocata precedentemente

```
int * a, * b; /*puntatori al
                primo elemento
                dell'array*/
a = malloc(10*sizeof(int));
…
b = realloc(a,20*sizeof(int));
/* adesso b punta ad un array di 20
 elementi */
```

Array dinamici -- realloc( ) (2)

- Meglio usare sempre due puntatori diversi (`a`,`b`) !
  - Altrimenti in caso di fallimento `NULL` sovrascrive il vecchio puntatore

Array dinamici -- free( )

- Lo spazio allocato sullo heap non viene deallocato all'uscita delle funzioni
- La deallocazione deve essere richiesta esplicitamente usando free

```
int * a;
a = malloc(10*sizeof(int));
…
free(a);
/* se qua accedo di nuovo ad a
può succedere di tutto */
```

tipo puntatore generico: `void *`

- non si può dereferenziare
- è necessario un cast prima di manipolare la variabile puntata
  - Es.

```
void * c;
int a;
c = &a;
*c = 5;  /* scorretto*/
*(int *)c = 5;    /* corretto*/
```

tipo puntatore generico: `void *` (2)

- Serve a scrivere funzioni 'polimorfe' in una maniera un po' brutale
- Esempio
  - il tipo della malloc è

```
void * malloc (unsigned int size);
```

  - quando scrivo

```
int * a;
a = malloc(10*sizeof(int));
```

    viene effettuato un cast implicito a (int *)

tipo puntatore generico: `void *` (3)

- Tipi delle altre funzioni di allocazione e deallocazione

```
void * calloc (unsigned int size);
void * realloc (void * ptr,
                unsigned int size);
void free (void * ptr);
```