Introduzione al C

Lezione 2

Funzioni e Puntatori

Rossano Venturini

rossano@di.unipi.it

Pagina web del corso

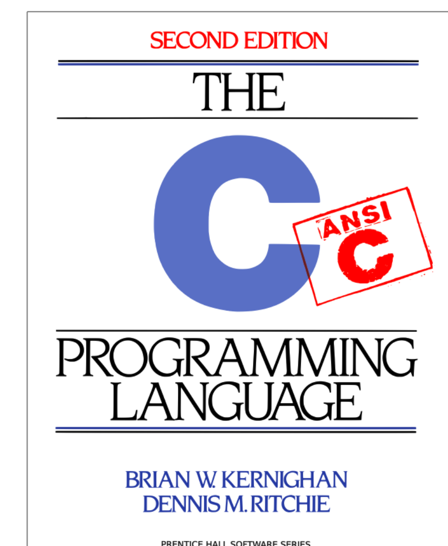
http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start

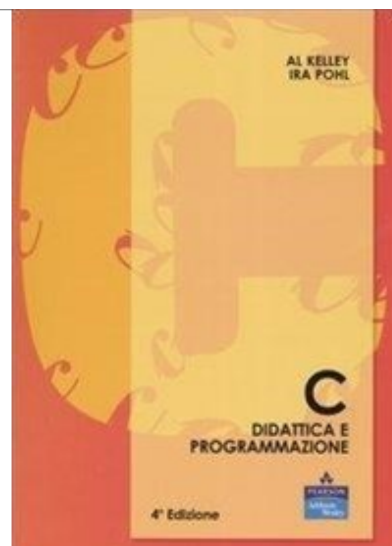
Lezioni di ripasso C

Giovedì 20	16-18	Aula A-B
Mercoledì 26	11-13	Aula A-B
Giovedì 27	16-18	Aula A-B

Le successive lezioni di laboratorio saranno

Corso B Giovedì	14-16	Aula H-M
Corso A Giovedì	16-18	Aula H-M





Esercizio 2

Primo

Esercizio

Scrivere un programma che legga da tastiera un intero e stabilisca se il numero è primo.
L'input consiste di una sola riga conenente l'intero x.
Il programma stampa in output 1 se x è primo, 0 altrimenti.

Esempi

Input	**Output**
`226`	`0`

Input	**Output**
`13`	`1`

Esercizio 2

```
#include <stdio.h>

int main()
{
  int x, fattore = 2, primo = 1;

  scanf("%d", &x);

  while( primo && fattore < x ){
    if ( (x % fattore) == 0 ) primo = 0;
    fattore++;
  }

   printf("%d\n",primo);
   return 0;
}
```

Esercizio 2 - soluzione più efficiente

```
#include <stdio.h>
#include <math.h>

int main()
{
  int x, fattore = 2, primo = 1;
  int limite;

  scanf("%d", &x);
  limite = (int) sqrt(x);

  while( primo && fattore <= limite ){
    if ( (x % fattore) == 0 ) primo = 0;
    fattore++;
  }

   printf("%d\n",primo);
   return 0;
}
```

Esercizio 4

Invertire un array

Esercizio

Scrivere un programma che legga da input gli N interi di un array A. Il programma deve invertire A in loco (cioè senza utilizzare un array di appoggio), ossia scambiare il contenuto della prima e dell'ultima cella, della seconda e della penultima, ecc.
Si assuma che $N \leq 10000$.
La prima riga dell'input è il valore N. Seguono N interi, uno per riga.
Il programma stampa in output gli elementi dell'array invertito, uno per riga.

Esempi

Input

```
5
3
1
4
0
4
```

Output

```
4
0
4
1
3
```

Esercizio 4

```
#include <stdio.h>
#define MAXSIZE (10000)

int main() {
 int n, i, j, scambio;
 int a[MAXSIZE];

 scanf("%d", &n );
 for ( i = 0; i < n; i++ )
   scanf("%d", &a[i]);

 /* Inversione in loco */
 for ( i = 0; i < n/2; i++ ) {
   j = (n-1)-i;
   scambio = a[i];
   a[i] = a[j];
   a[j] = scambio;
  }

 /* Output */
 for ( i = 0; i < n; i++ )   printf("%d\n", a[i]);
 printf("\n");
 return 0;
}
```

Esercizio 4

```
#include <stdio.h>
#define MAXSIZE (10000)

int main() {
 int n, i, j, scambio;
 int a[MAXSIZE];

 scanf("%d", &n );
 for ( i = 0; i < n; i++ )
   scanf("%d", &a[i]);

 /* Inversione in loco */
 for ( i = 0; i < n/2; i++ ) {
   j = (n-1)-i;
   scambio = a[i];
   a[i] = a[j];
   a[j] = scambio;
  }

 /* Output */
 for ( i = 0; i < n; i++ )   printf("%d\n", a[i]);
 printf("\n");
 return 0;
}
```

Testing

Testing

Compilazione, esecuzione e testing

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
```

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
```

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
```

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
3
4
```

Da confrontare con output0.txt

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
3
4
$ ./inverti < input0.txt
```

Da confrontare con output0.txt

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
3
4
$ ./inverti < input0.txt
3
4
```

Da confrontare con output0.txt

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
3
4
$ ./inverti < input0.txt
3
4

$ ./inverti < input0.txt | diff - output0.txt
```

Da confrontare con output0.txt

Testing

Compilazione, esecuzione e testing

```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
3
4
$ ./inverti < input0.txt
3
4

$ ./inverti < input0.txt | diff - output0.txt
1d0
< 3
2a2
> 3
```

Da confrontare con output0.txt

Testing

Compilazione, esecuzione e testing

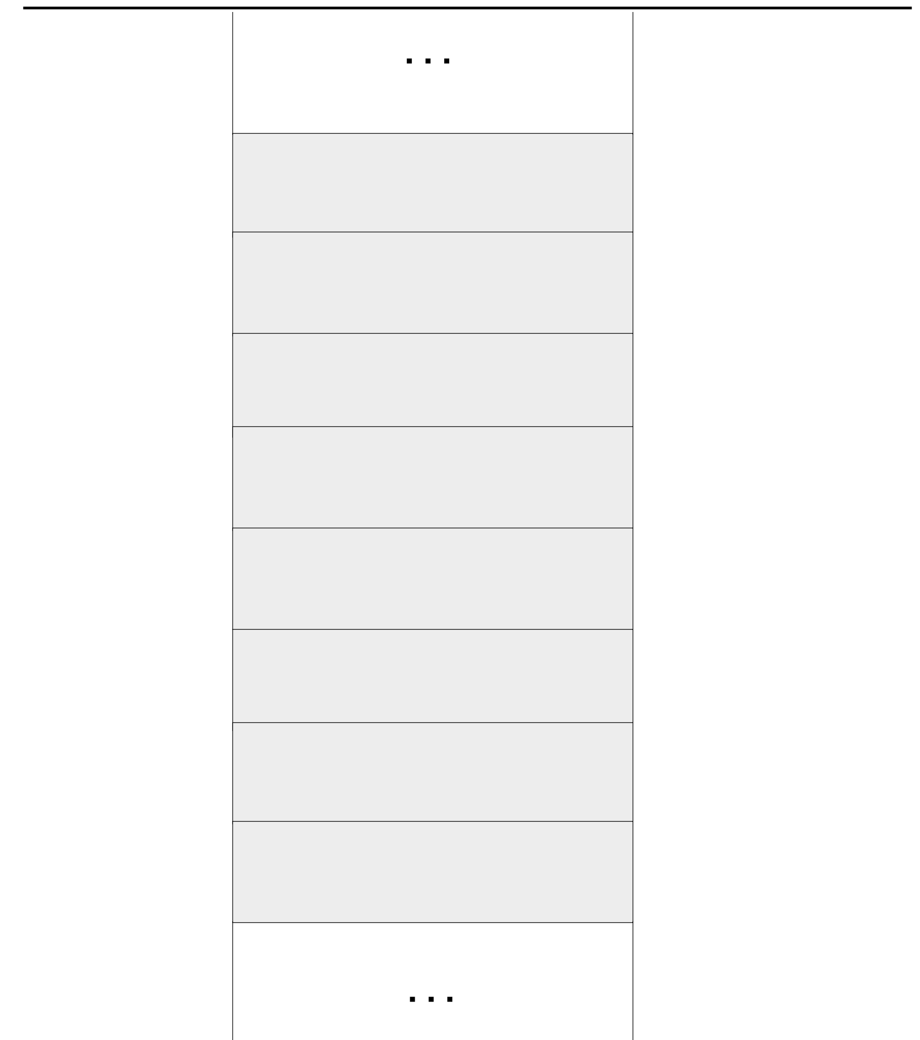
```
$ gcc -o inverti inverti.c
$ ./inverti
2
3
4
3
4
$ ./inverti < input0.txt
3
4

$ ./inverti < input0.txt | diff - output0.txt
1d0
< 3
2a2
> 3
```

Da confrontare con output0.txt

Niente in output se l'output è corretto

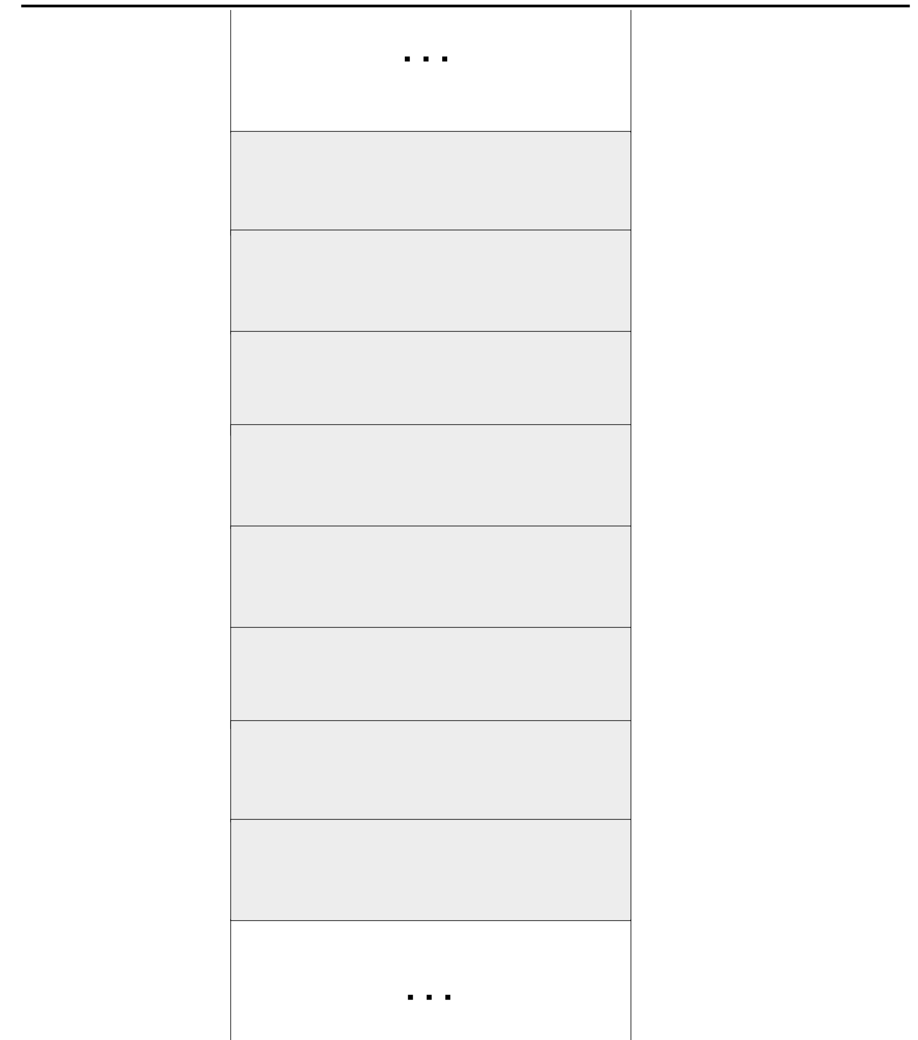
Un primo sguardo dietro le quinte

Un primo sguardo dietro le quinte



Un primo sguardo dietro le quinte

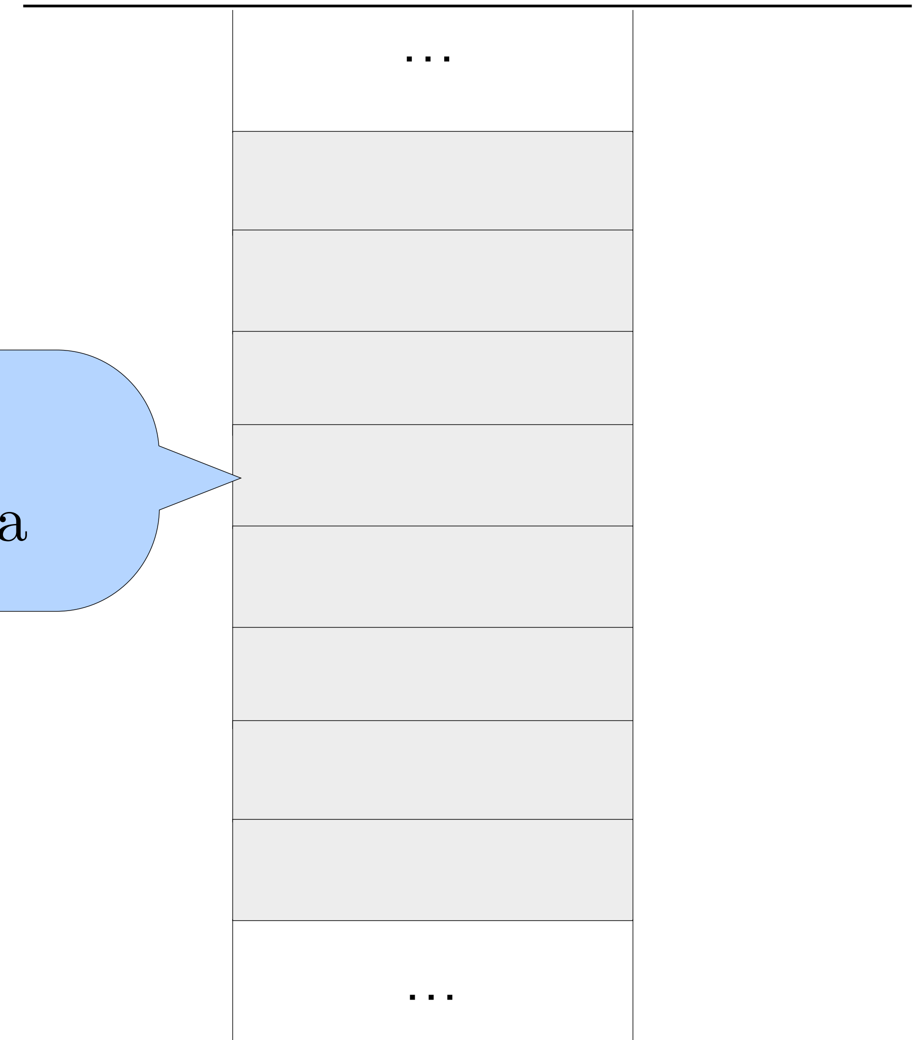
Attenzione, il modello di memoria che presenteremo è una versione semplificata di quello reale.



Un primo sguardo dietro le quinte

Memoria

celle



Semplificazione!
da 4 byte ciascuna

Un primo sguardo dietro le quinte

Memoria

	celle	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
```

Memoria

	celle	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Riservata per contenere il valore di x

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
y		0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
y	10	0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
printf("%d + %d = %d, x , y, x+y");
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
y	10	0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
printf("%d + %d = %d, x , y, x+y");
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
y	10	0x104
		0x108
		0x112
		0x116
x	?	0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
printf("%d + %d = %d, x , y, x+y");
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
y	10	0x104
		0x108
		0x112
		0x116
x	?	0x120
		0x124
		0x128
	...	

Un qualunque valore!
Inizializzate sempre!

Un primo sguardo dietro le quinte

```
int x;
int y = 10;
printf("%d + %d = %d, x , y, x+y");
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
y	10	0x104
		0x108
		0x112
		0x116
x	?	0x120
		0x124
		0x128
	...	

Funzioni (1)

Tutti i moderni linguaggi di programmazione permettono di strutturare un programma in funzioni.

Funzioni (1)

Tutti i moderni linguaggi di programmazione permettono di strutturare un programma in funzioni.

La definizione di una funzione si compone di:

- *intestazione*: nome della funzione, tipo del risultato e dichiarazione dei parametri;
- *corpo*: blocco di istruzioni che verranno eseguite alla chiamata.

Esempio

```
int somma(int x, int y) {
    return x + y;
}
```

Funzioni (1)

Tutti i moderni linguaggi di programmazione permettono di strutturare un programma in funzioni.

La definizione di una funzione si compone di:

- *intestazione*: nome della funzione, tipo del risultato e dichiarazione dei parametri;
- *corpo*: blocco di istruzioni che verranno eseguite alla chiamata.

Esempio

```
int somma(int x, int y) {
    return x + y;
}
```

Funzioni (1)

Tutti i moderni linguaggi di programmazione permettono di strutturare un programma in funzioni.

La definizione di una funzione si compone di:

- *intestazione*: nome della funzione, tipo del risultato e dichiarazione dei parametri;
- *corpo*: blocco di istruzioni che verranno eseguite alla chiamata.

Esempio

```
int somma(int x, int y) {
    return x + y;
}
```

Funzioni (1)

Tutti i moderni linguaggi di programmazione permettono di strutturare un programma in funzioni.

La definizione di una funzione si compone di:

- *intestazione*: nome della funzione, tipo del risultato e dichiarazione dei parametri;
- *corpo*: blocco di istruzioni che verranno eseguite alla chiamata.

Esempio

```
int somma(int x, int y) {
    return x + y;
}
```

Il tipo del risultato è void se la funzione non restituisce nessun valore.

Funzioni (2)

Per invocare una funzione è necessario che essa sia stata precedentemente definita (o soltanto dichiarata).

Funzioni (2)

Per invocare una funzione è necessario che essa sia stata precedentemente definita (o soltanto dichiarata).

Esempio

```
int somma(int x, int y) {
    return x + y;
}

int main () {
    int x = 10, y = 5;
    ...
    printf("%d\n", somma(y, x));
    return 0;
}
```

Funzioni (2)

Per invocare una funzione è necessario che essa sia stata precedentemente definita (o soltanto dichiarata).

Esempio

```
int somma(int x, int y) {
    return x + y;
}

int main () {
    int x = 10, y = 5;
    ...
    printf("%d\n", somma(y, x));
    return 0;
}
```

?

Funzioni (3)

Il C permette che una funzioni chiami se stessa ricorsivamente.

Funzioni (3)

Il C permette che una funzioni chiami se stessa ricorsivamente.

Esempio

```
int power(int n, int m) {
    if (n == 0)
        return 1;
    else
        return m * power(n-1, m);
}

int main () {
    int x = 10, y = 5;
    ...
    printf("%d\n", power(x, y)); // stampa 5^10
    return 0;
}
```

Funzioni (3)

Il C permette che una funzioni chiami se stessa ricorsivamente.

Esempio

```
int power(int n, int m) {
    if (n == 0)
        return 1;
    else
        return m * power(n-1, m);
}

int main () {
    int x = 10, y = 5;
    ...
    printf("%d\n", power(x, y)); // stampa 5^10
    return 0;
}
```

Funzioni (3)

Il C permette che una funzioni chiami se stessa ricorsivamente.

Esempio

```
int power(int n, int m) {
    if (n == 0)
        return 1;
    else
        return m * power(n-1, m);
}

int main () {
    int x = 10, y = 5;
    ...
    printf("%d\n", power(x, y)); // stampa 5^10
    return 0;
}
```

Funzioni (4)

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

```
void scambia(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

```
void scambia(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

```
void scambia(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

```
void scambia(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

```
void scambia(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

```
x=5 y=10
```

?

```
x=10 y=5
```

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

```
void scambia(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

x=5 y=10

?

x=10 y=5

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Cambia qualcosa così?

~~x=5 y=10~~

?

x=10 y=5

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
      int x = 10, y = 5;
      scambia(x, y);
      printf("x=%d y=%d", x, y);
      return 0;
}
```

	nome	*celle*	*indirizzo*
		...	
Ambiente locale di scambia()	x	10	0x100
	y	5	0x104
			0x108
			0x112
			0x116
	x	10	0x120
	y	5	0x124
			0x128
		...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

	nome	*celle*	*indirizzo*
		...	
Ambiente locale di scambia()	x	10	0x100
	y	5	0x104
			0x108
			0x112
			0x116
	x	10	0x120
	y	5	0x124
			0x128
		...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

	nome	*celle*	*indirizzo*
		...	
Ambiente locale di scambia()	x	10	0x100
	y	5	0x104
	tmp	10	0x108
			0x112
			0x116
	x	10	0x120
	y	5	0x124
			0x128
		...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
x	10	0x100
y	5	0x104
tmp	10	0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Ambiente locale di scambia()

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

	nome	*celle*	*indirizzo*
		...	
Ambiente locale di scambia()	x	5	0x100
	y	10	0x104
	tmp	10	0x108
			0x112
			0x116
	x	10	0x120
	y	5	0x124
			0x128
		...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Funzioni (4)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		08
		112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Vedremo in seguito come simulare il passaggio per riferimento attraverso l'uso dei puntatori.

Puntatori (1)

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

```
int x = 10;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

```
int x = 10;
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
```

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
p		0x108
		0x112
		0x116
x	10	0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
```

Specifica che si tratta di una variabile puntatore.

Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
p		0x108
		0x112
		0x116
x	10	0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
```

Specifica che si tratta di una variabile puntatore ad intero.

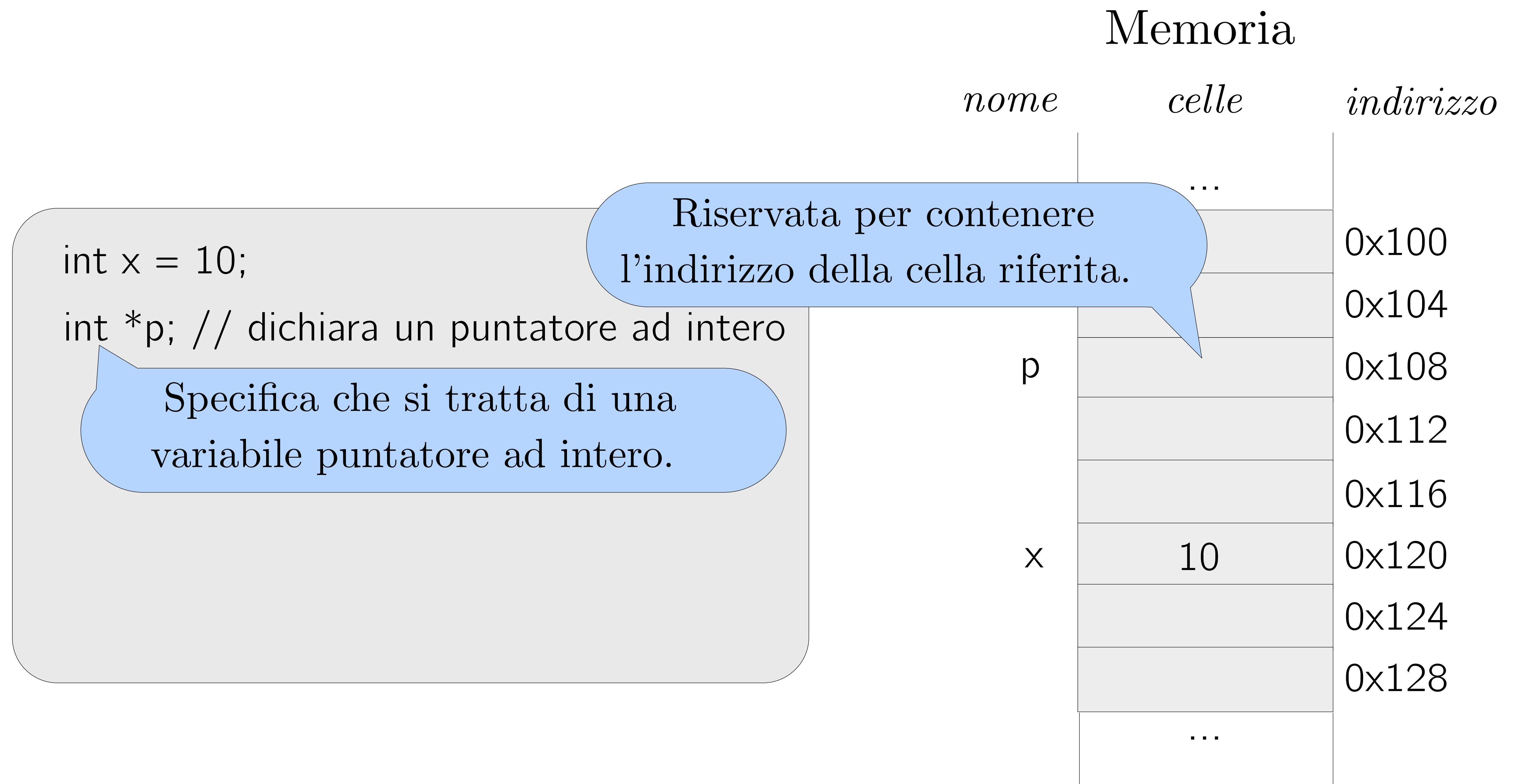
Memoria

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
p		0x108
		0x112
		0x116
x	10	0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.



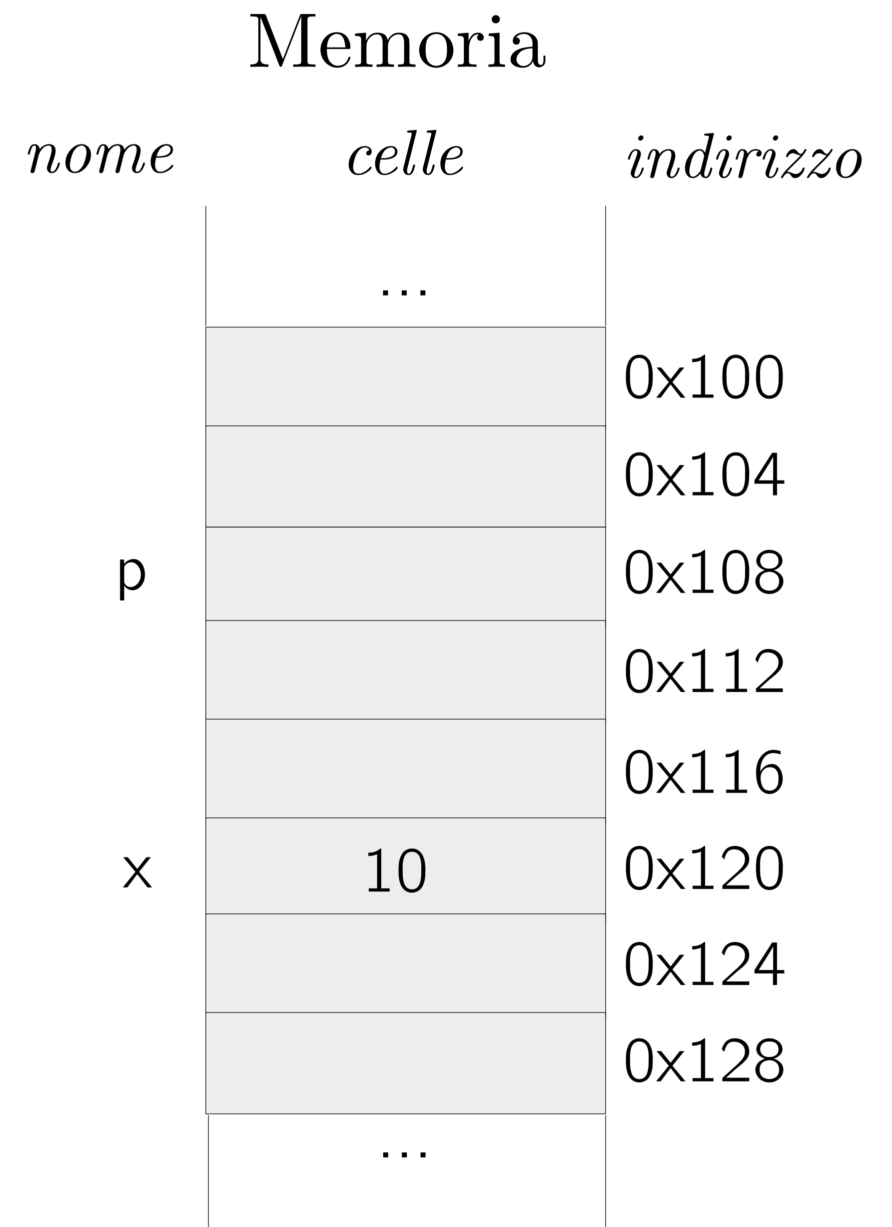
Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

Come posso far puntare p alla cella di x?

```
int x = 10;
int *p; // dichiara un puntatore ad intero
```



Puntatori (1)

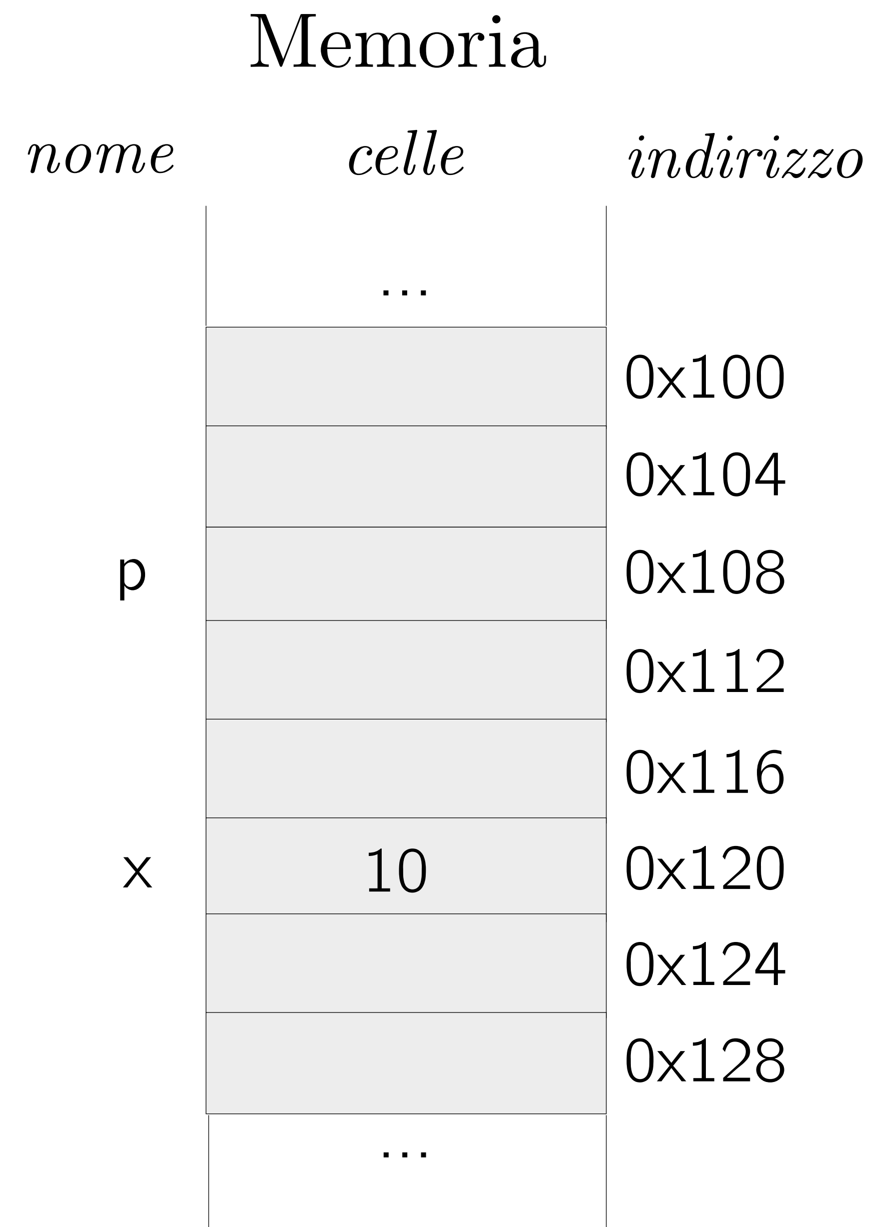
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

Come posso far puntare p alla cella di x?

Assegnando a p l'indirizzo di x. Denotato con &x

```
int x = 10;
int *p; // dichiara un puntatore ad intero
```



Puntatori (1)

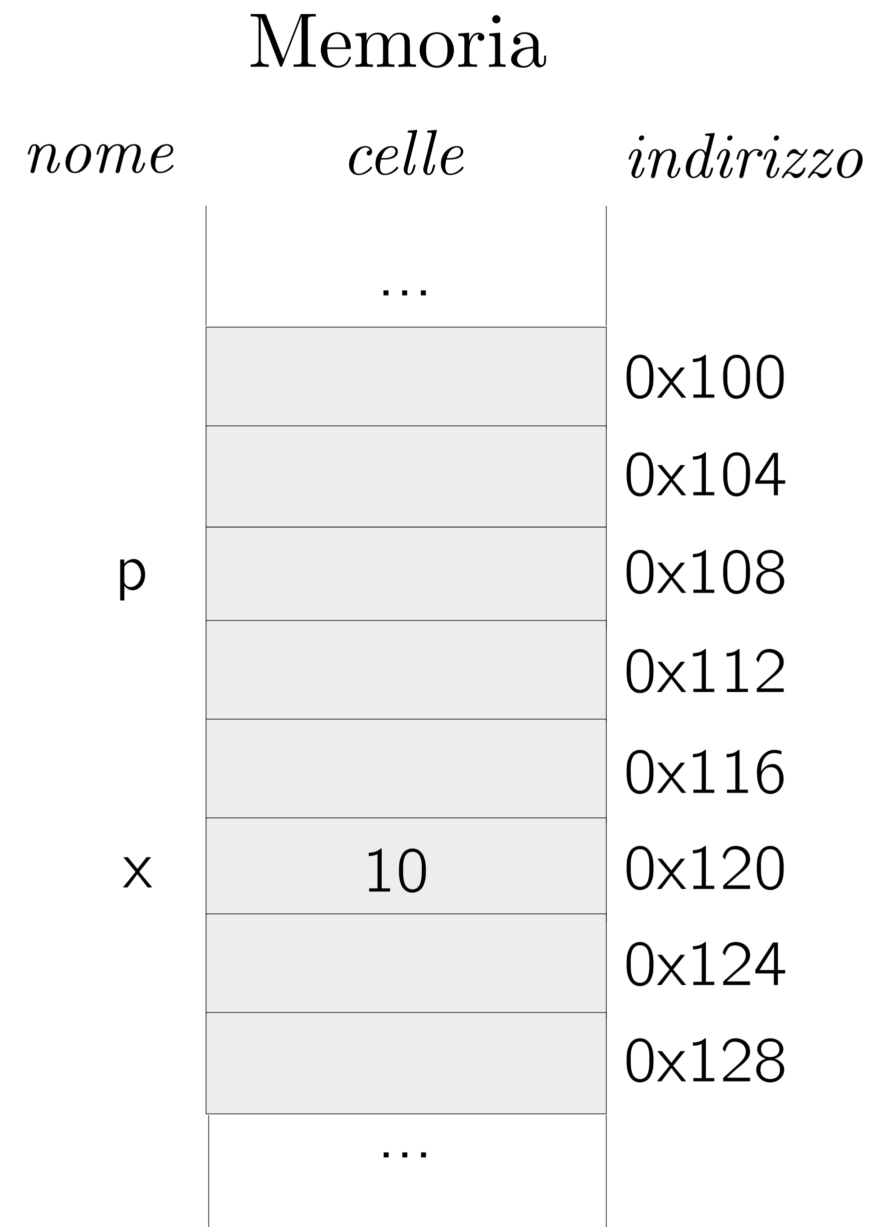
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

Come posso far puntare p alla cella di x?

Assegnando a p l'indirizzo di x. Denotato con &x

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



Puntatori (1)

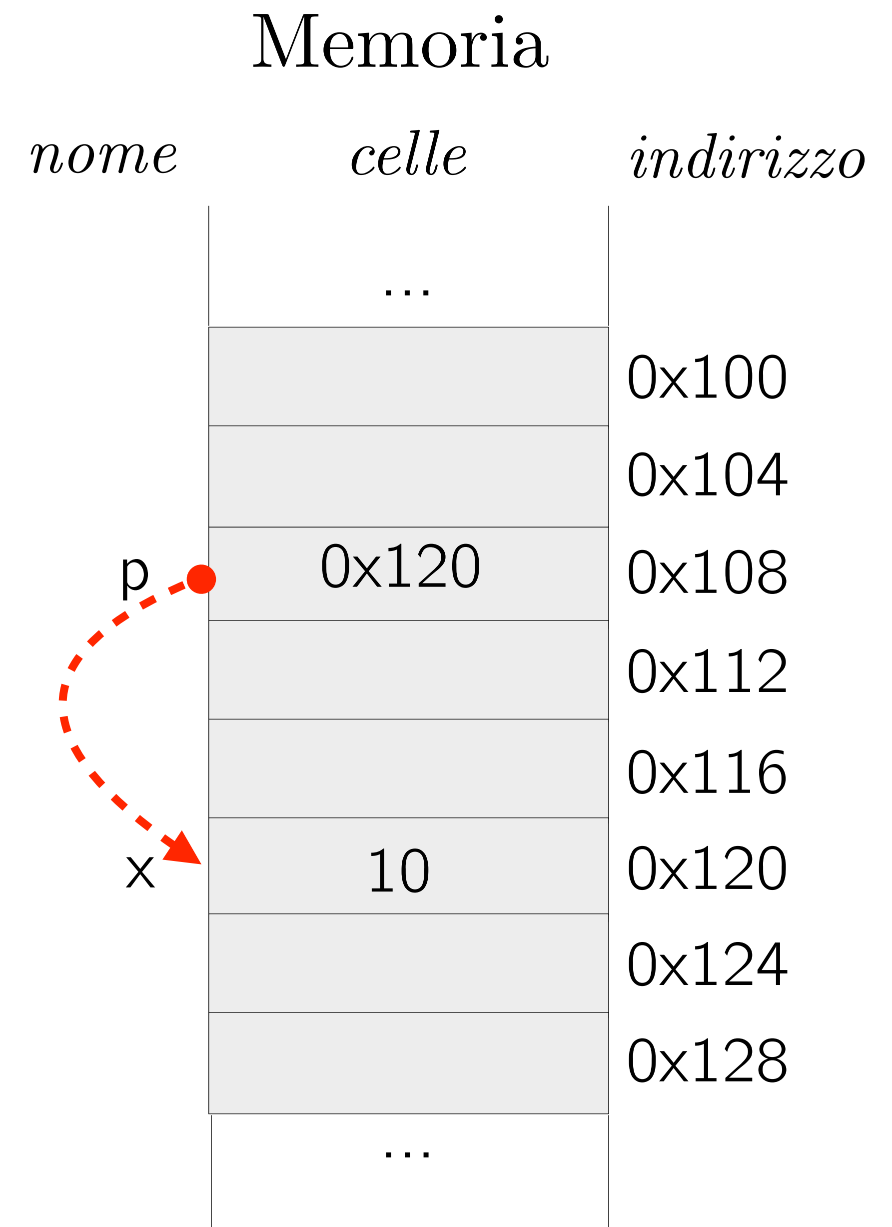
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

Come posso far puntare p alla cella di x?

Assegnando a p l'indirizzo di x. Denotato con &x

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



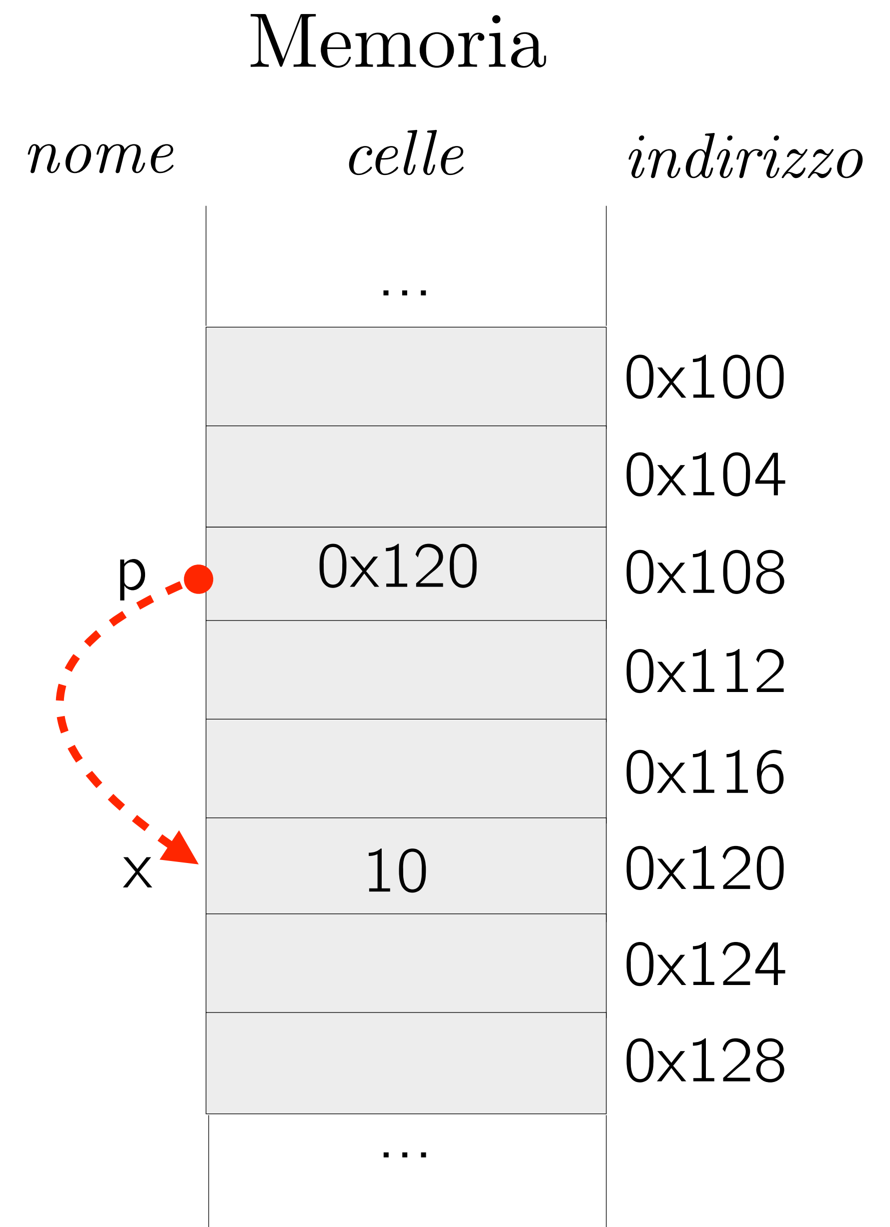
Puntatori (1)

Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



Puntatori (1)

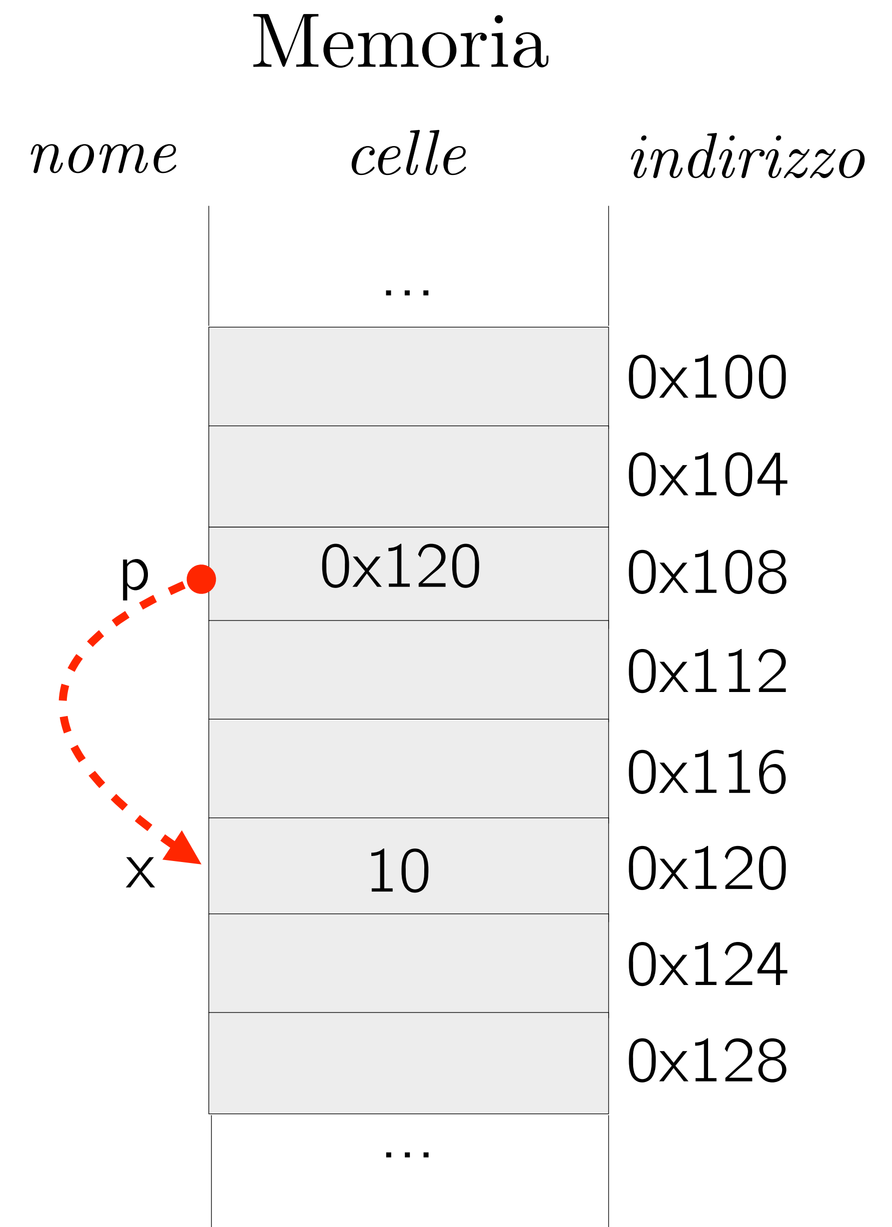
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

Si può accedere e modificare il valore in x attraverso p.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



Puntatori (1)

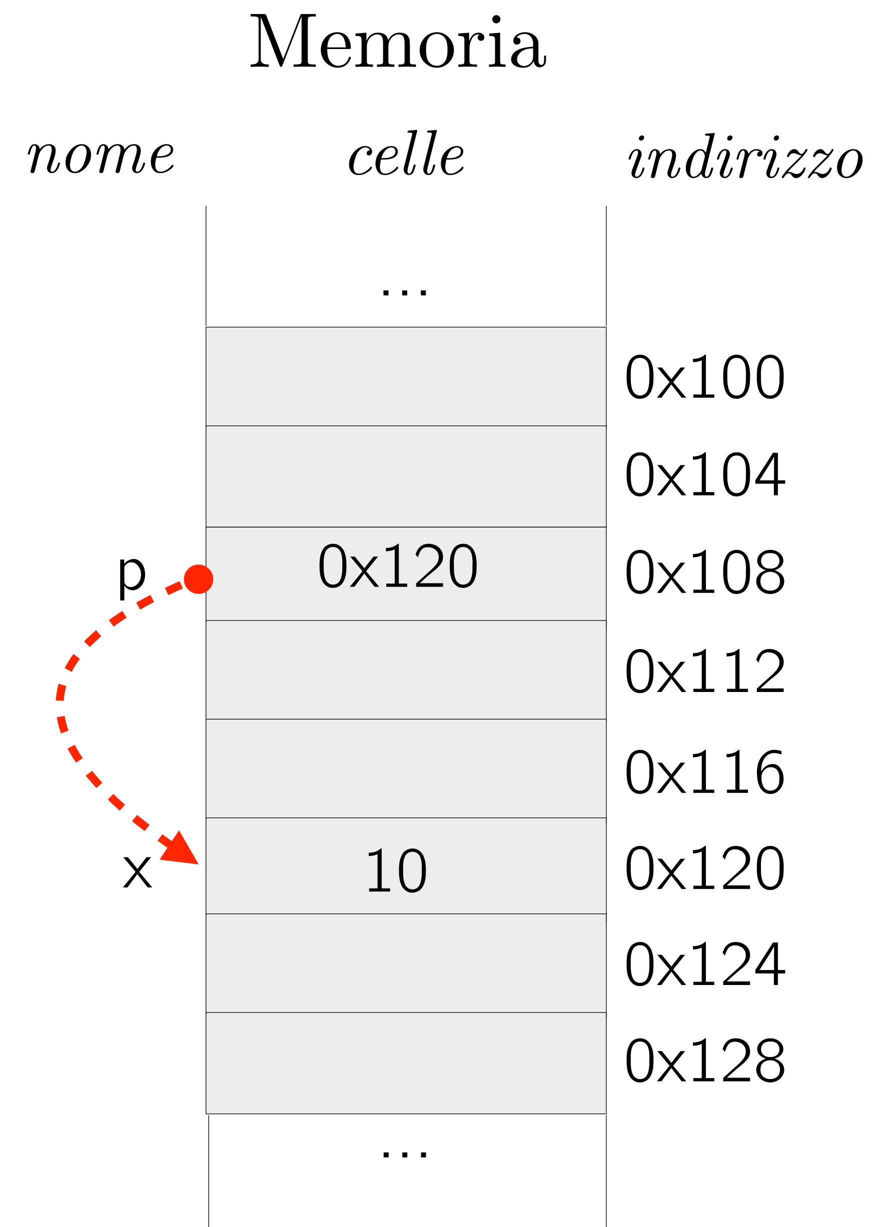
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

Si può accedere e modificare il valore in x attraverso p.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
y = *p+3 // equivalente a y = x +3;
```



Puntatori (1)

Una variabile di tipo int memorizza un valore.

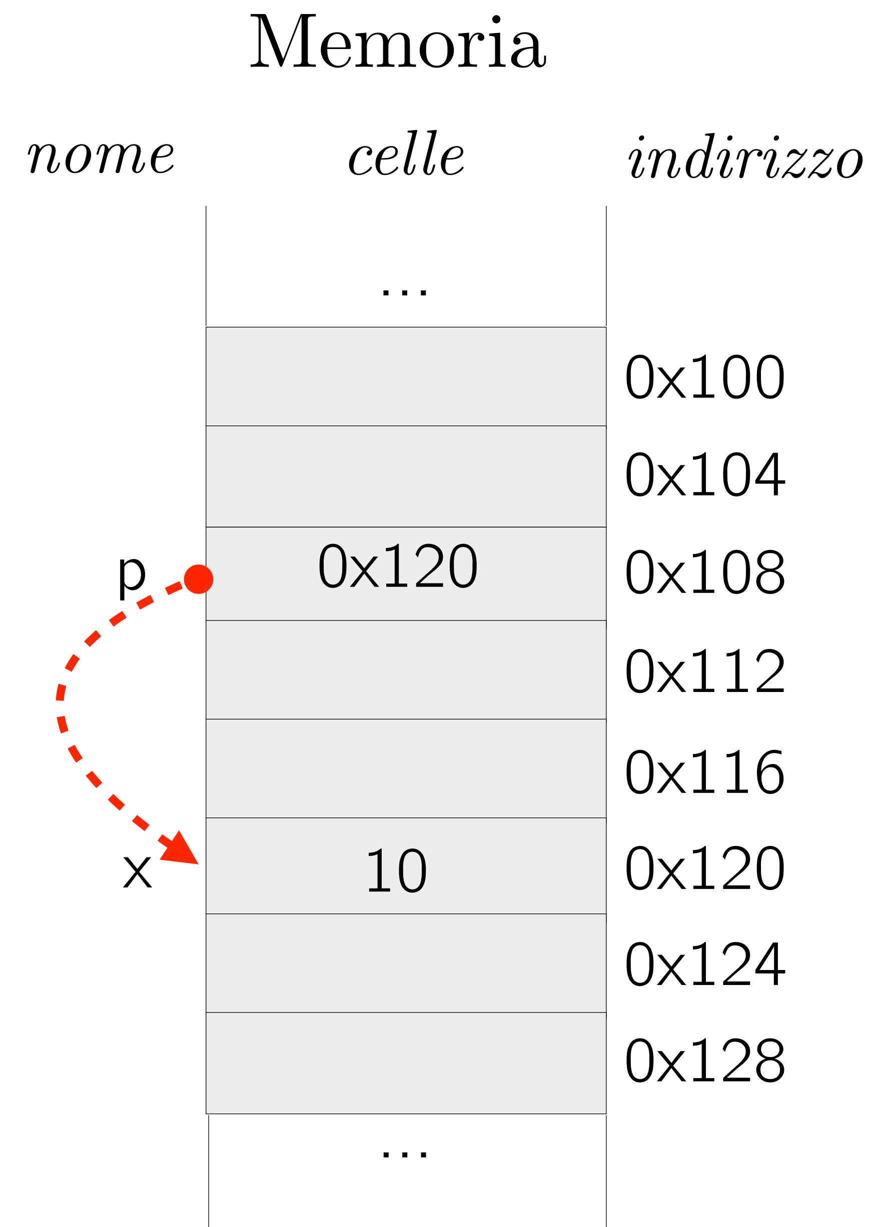
Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

Si può accedere e modificare il valore in x attraverso p.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
y = *p+3 // equivalente a y = x +3;
```

Segue il puntatore e denota la cella puntata.



Puntatori (1)

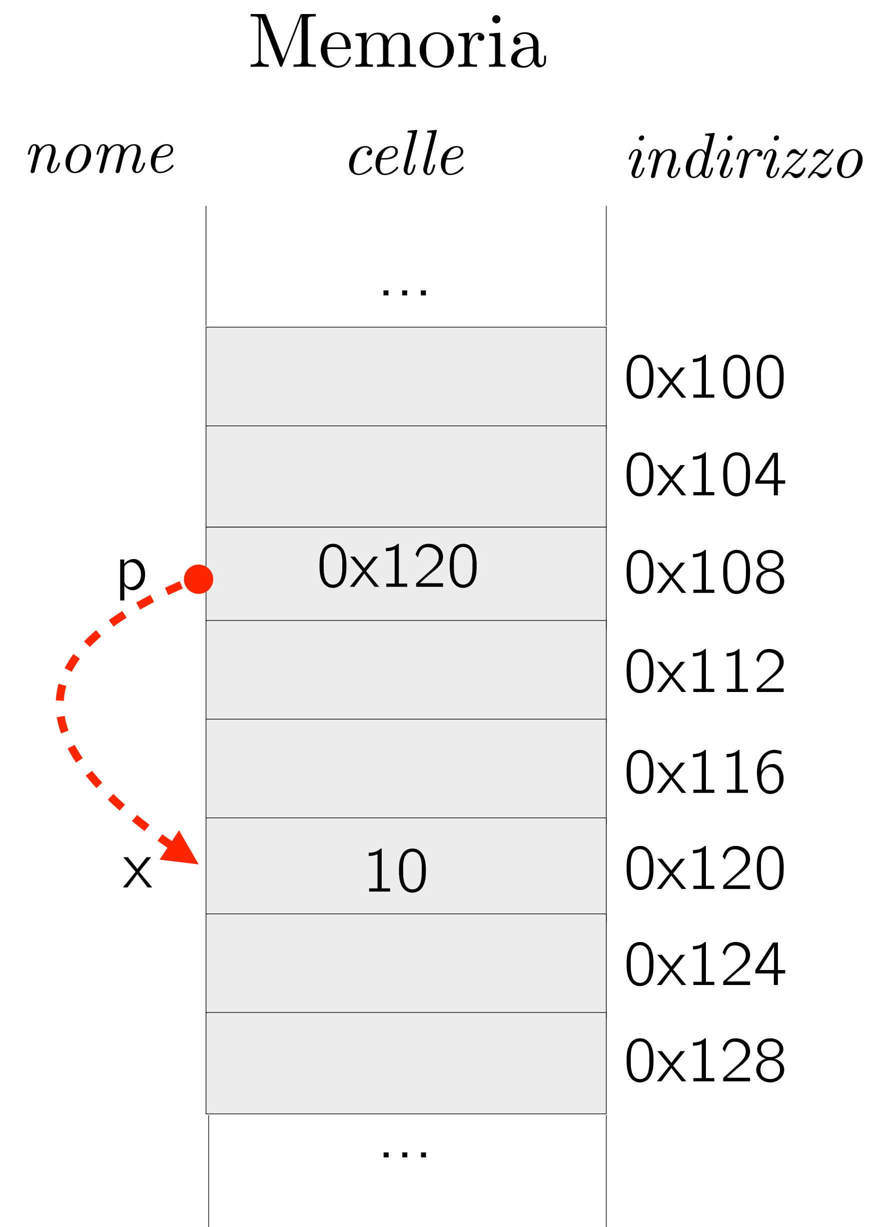
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

Si può accedere e modificare il valore in x attraverso p.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
y = *p+3 // equivalente a y = x +3;
*p = 22; // equivalente a x = 22;
```



Puntatori (1)

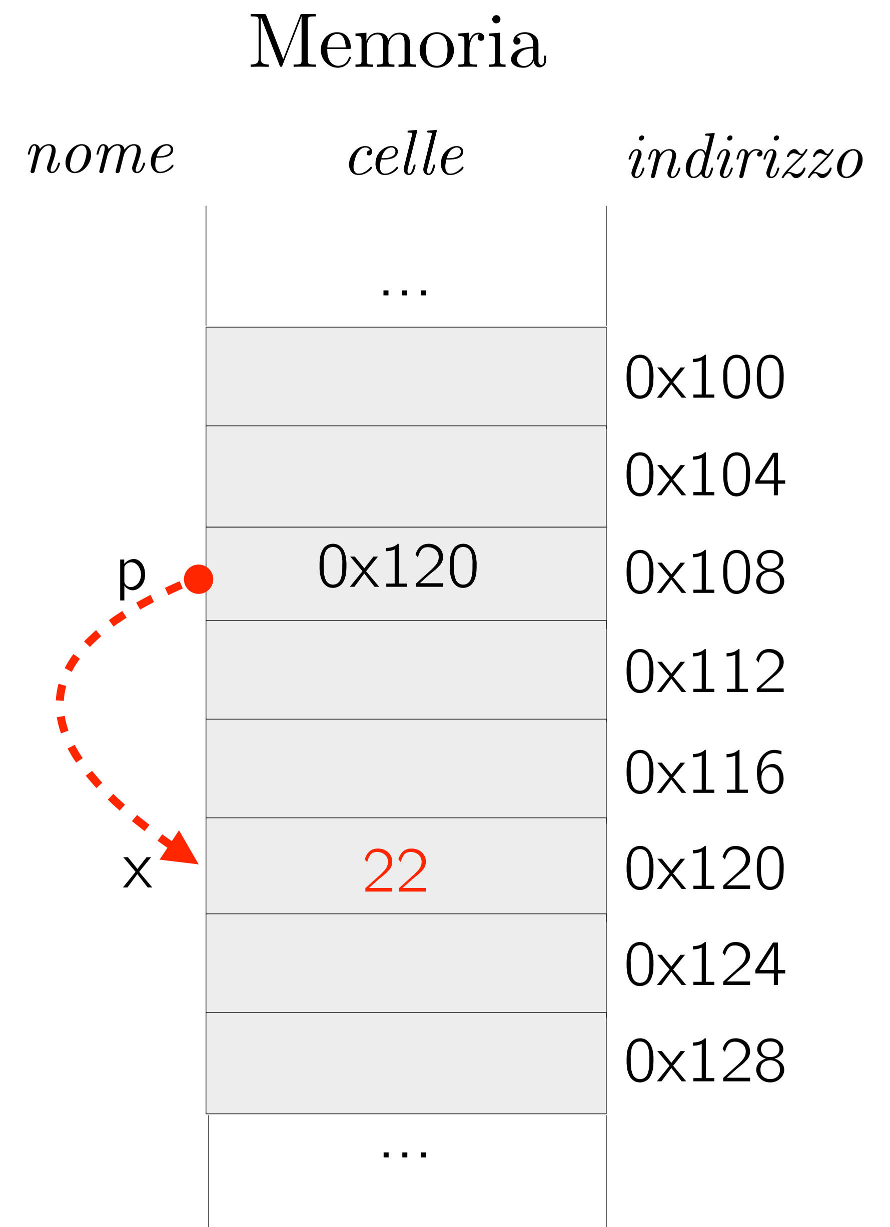
Una variabile di tipo int memorizza un valore.

Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

Si può accedere e modificare il valore in x attraverso p.

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
y = *p+3 // equivalente a y = x +3;
*p = 22; // equivalente a x = 22;
```



Puntatori (1)

Una variabile di tipo int memorizza un valore.

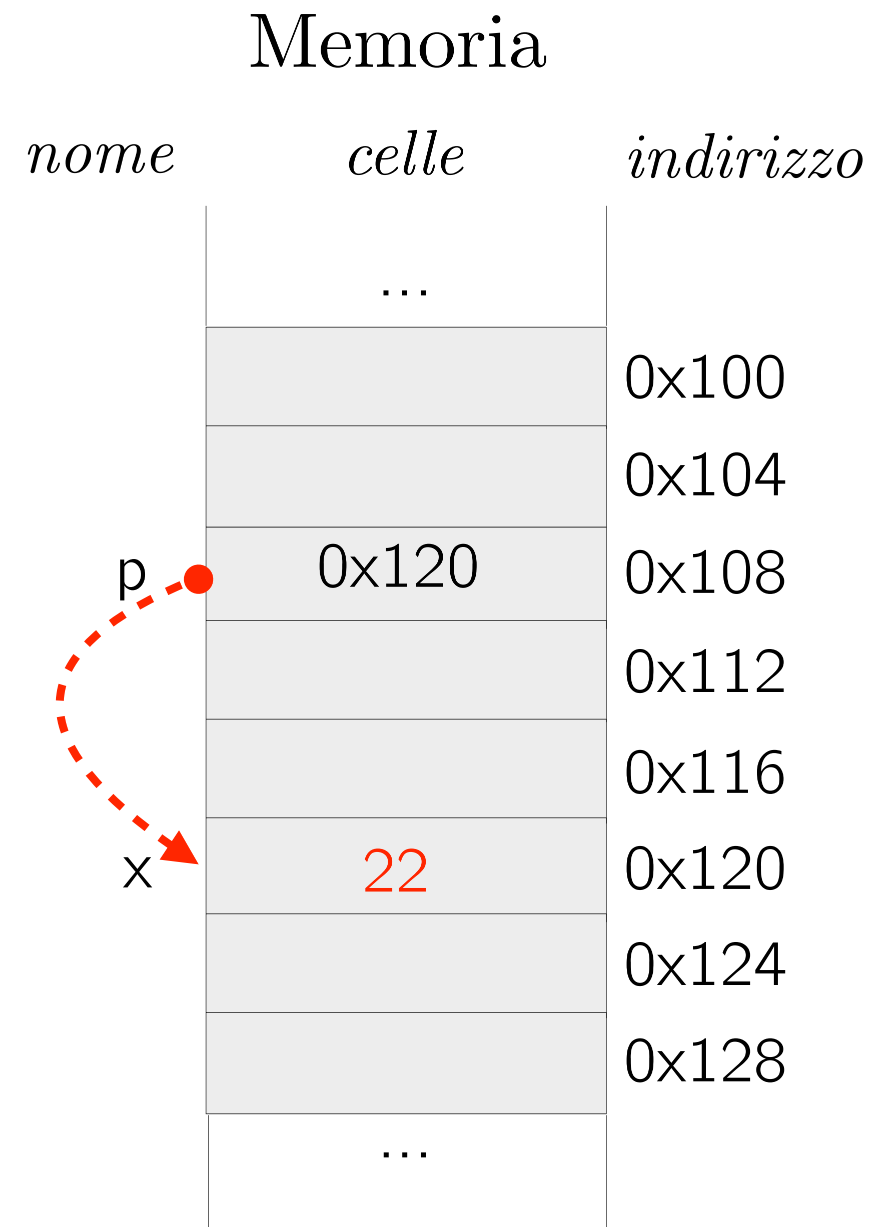
Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

Si può accedere e modificare il valore in x attraverso p.

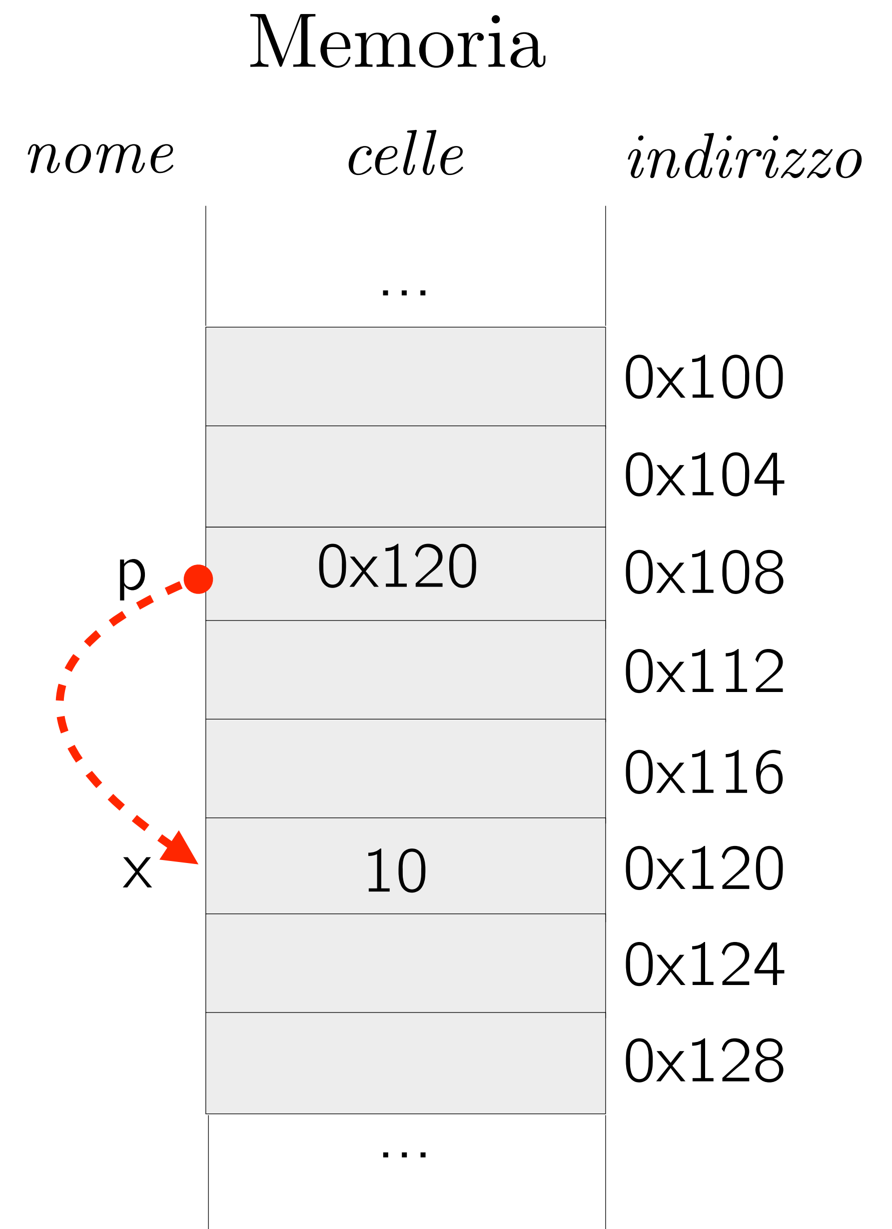
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
y = *p+3 // equivalente a y = x +3;
*p = 22; // equivalente a x = 22;
```

Si può usare *p proprio come useremmo x



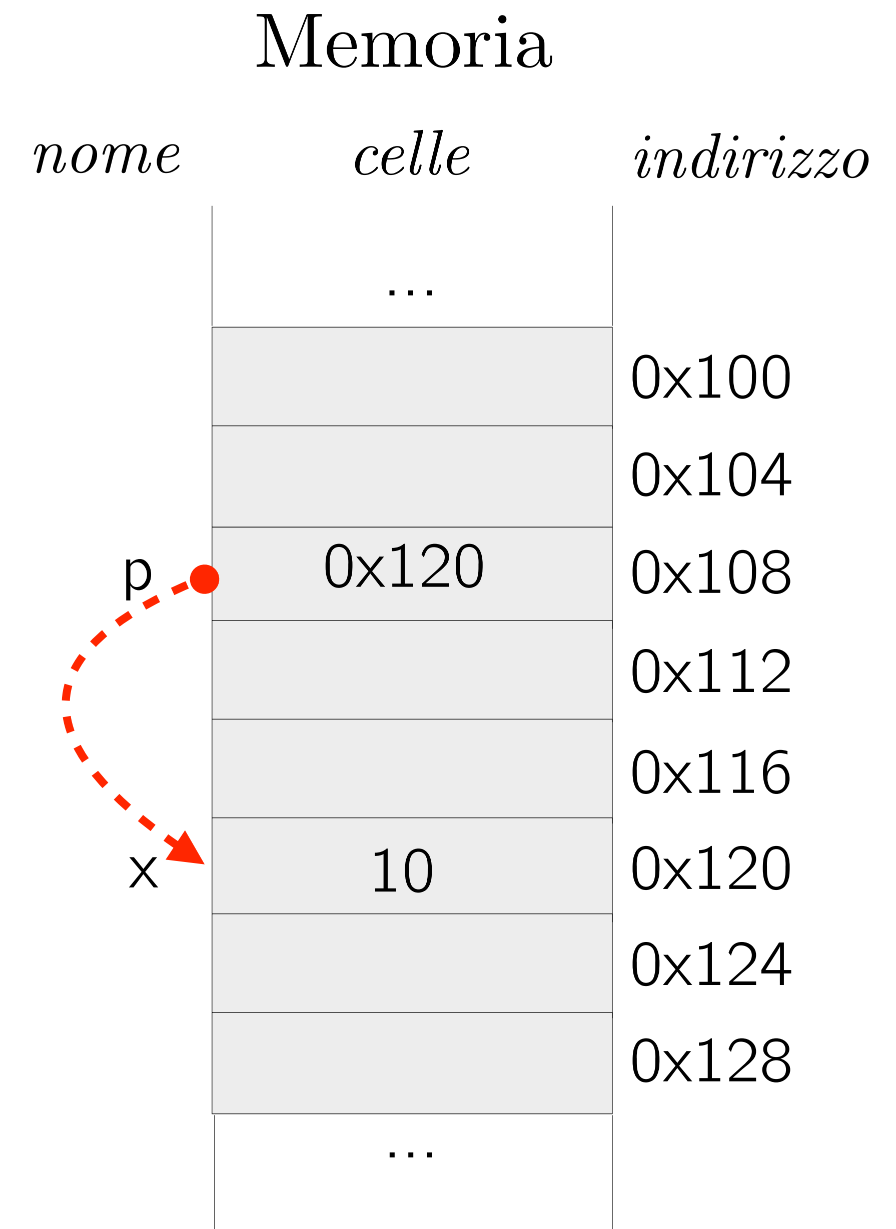
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



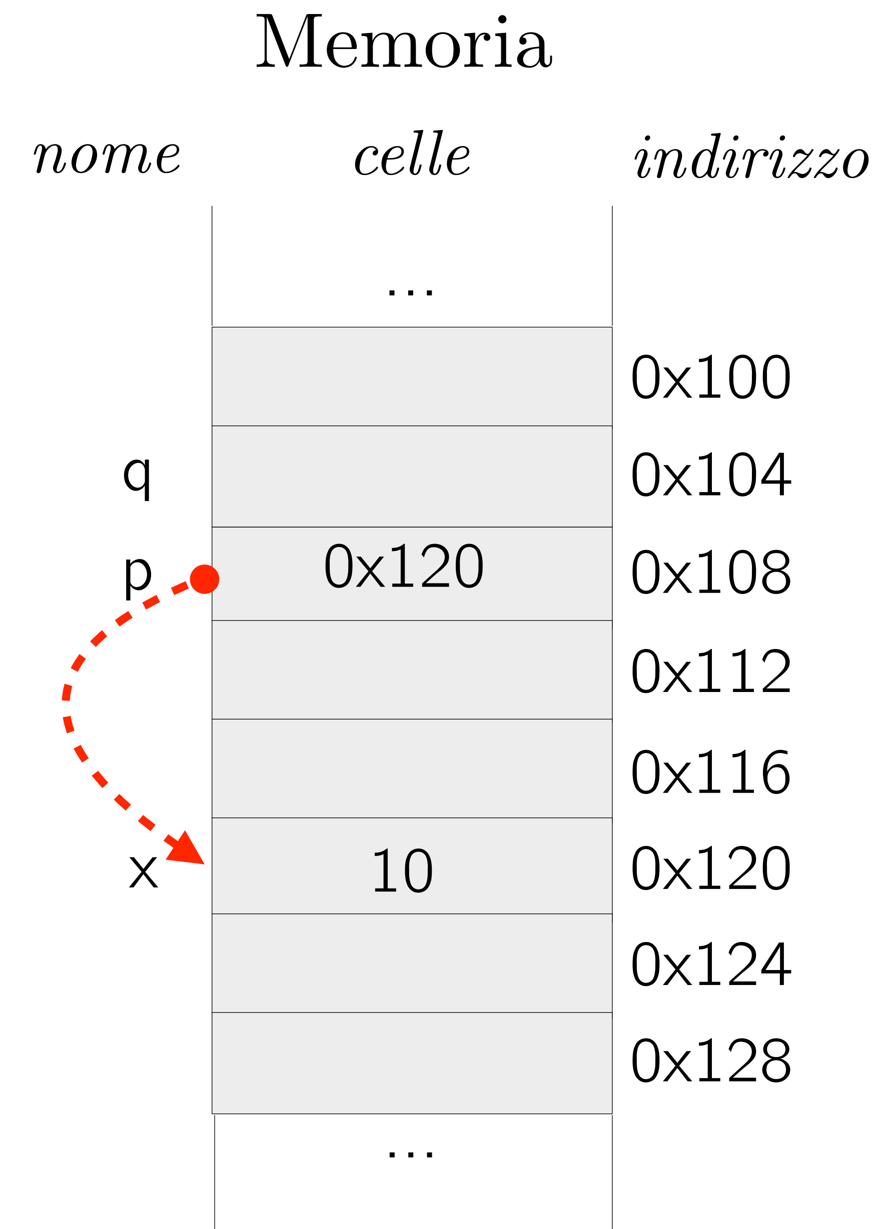
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
```



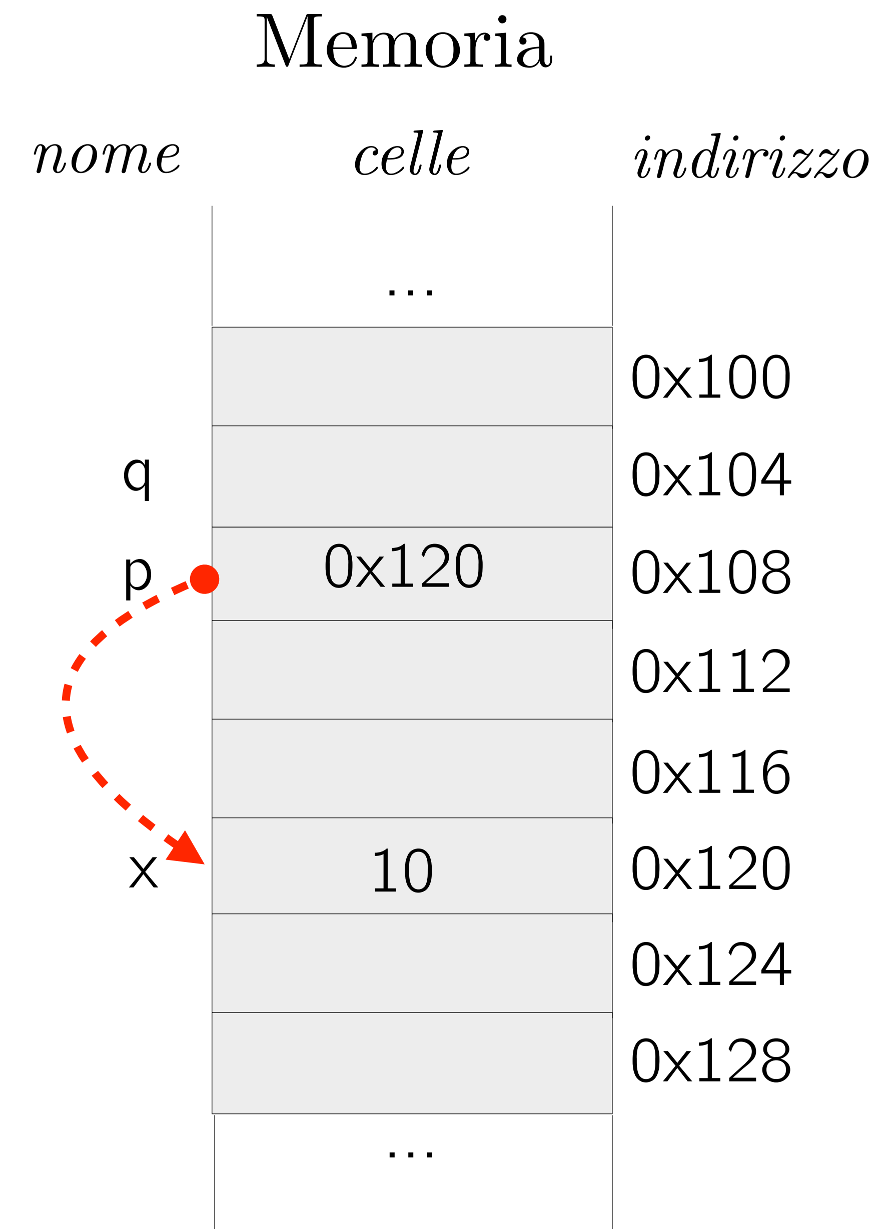
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
```



Puntatori (2)

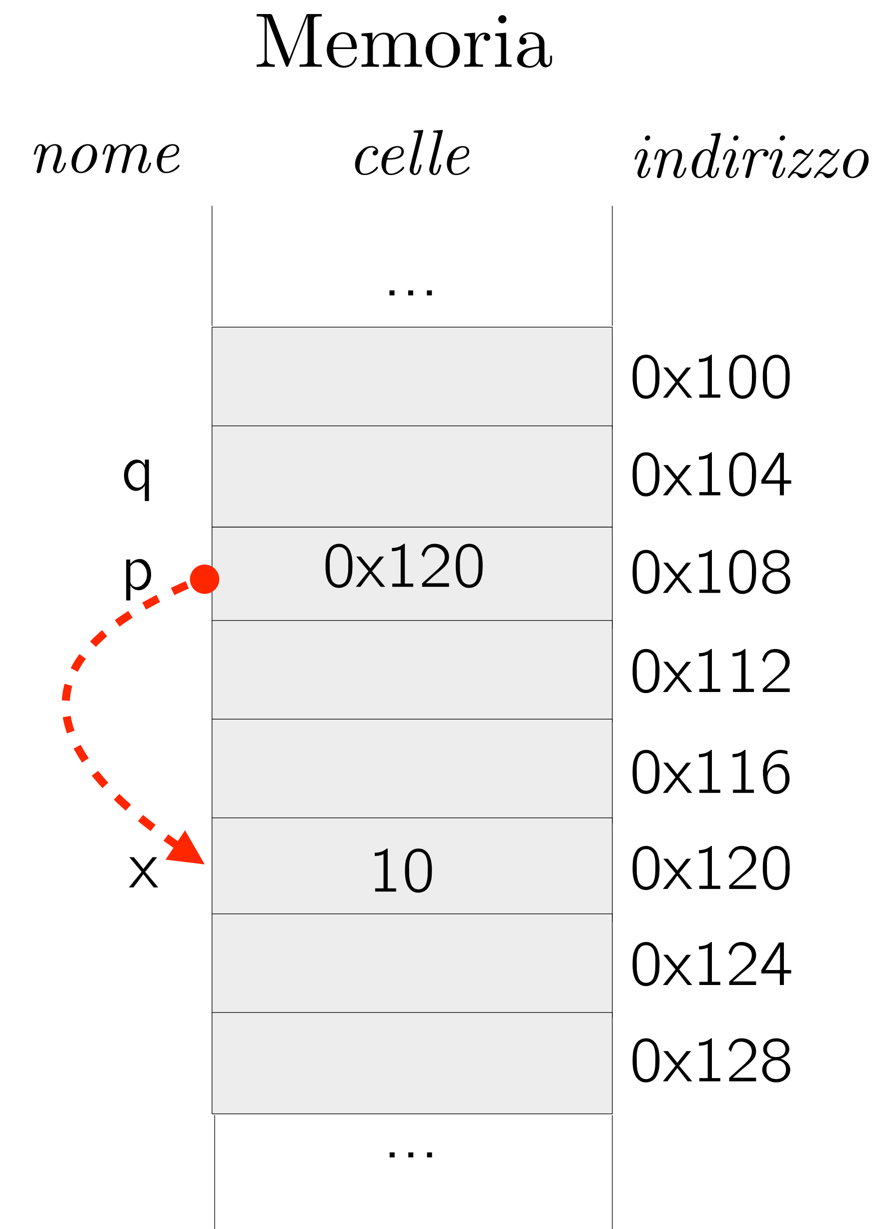
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```



Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

NO! q non è inizializzato.
Segmentation Fault!

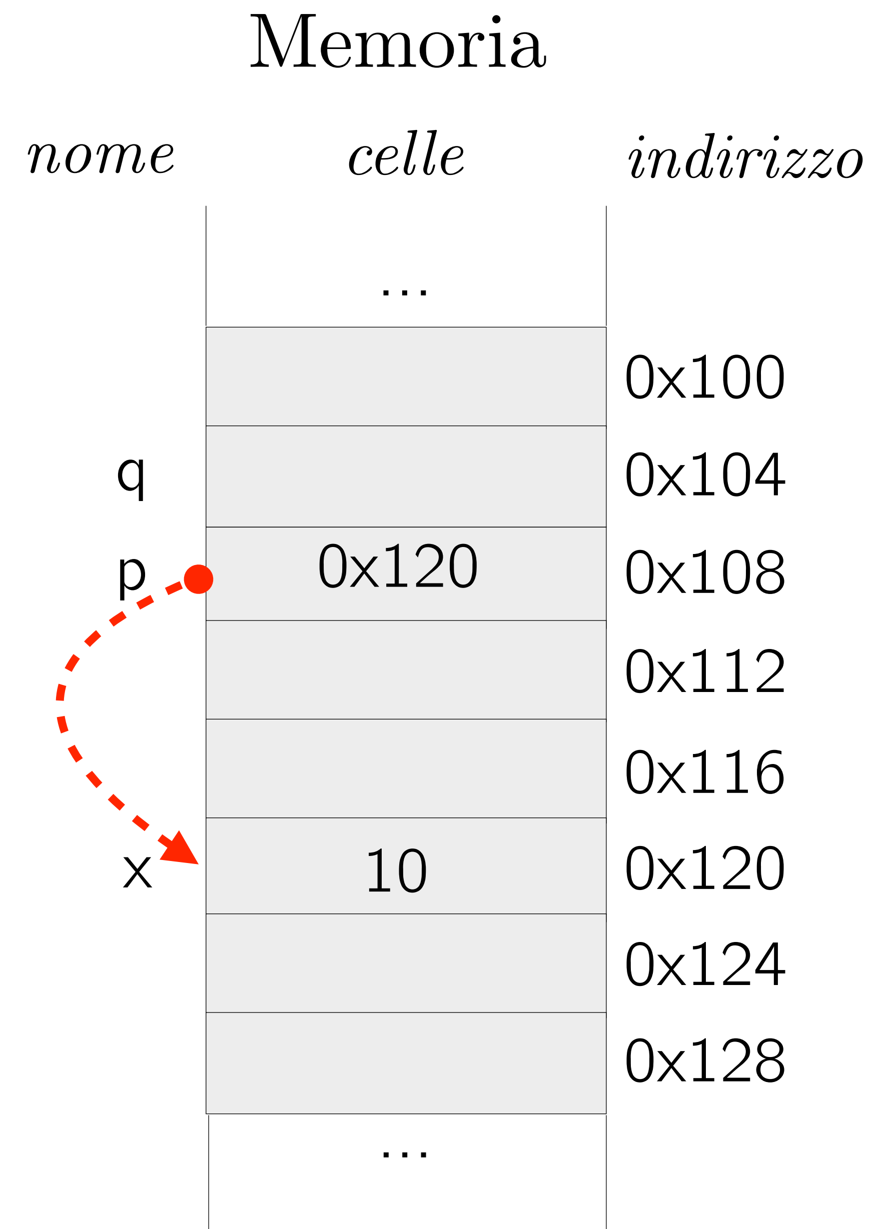


Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

NO! q non è inizializzato.
Segmentation Fault!

```
q = NULL;
```

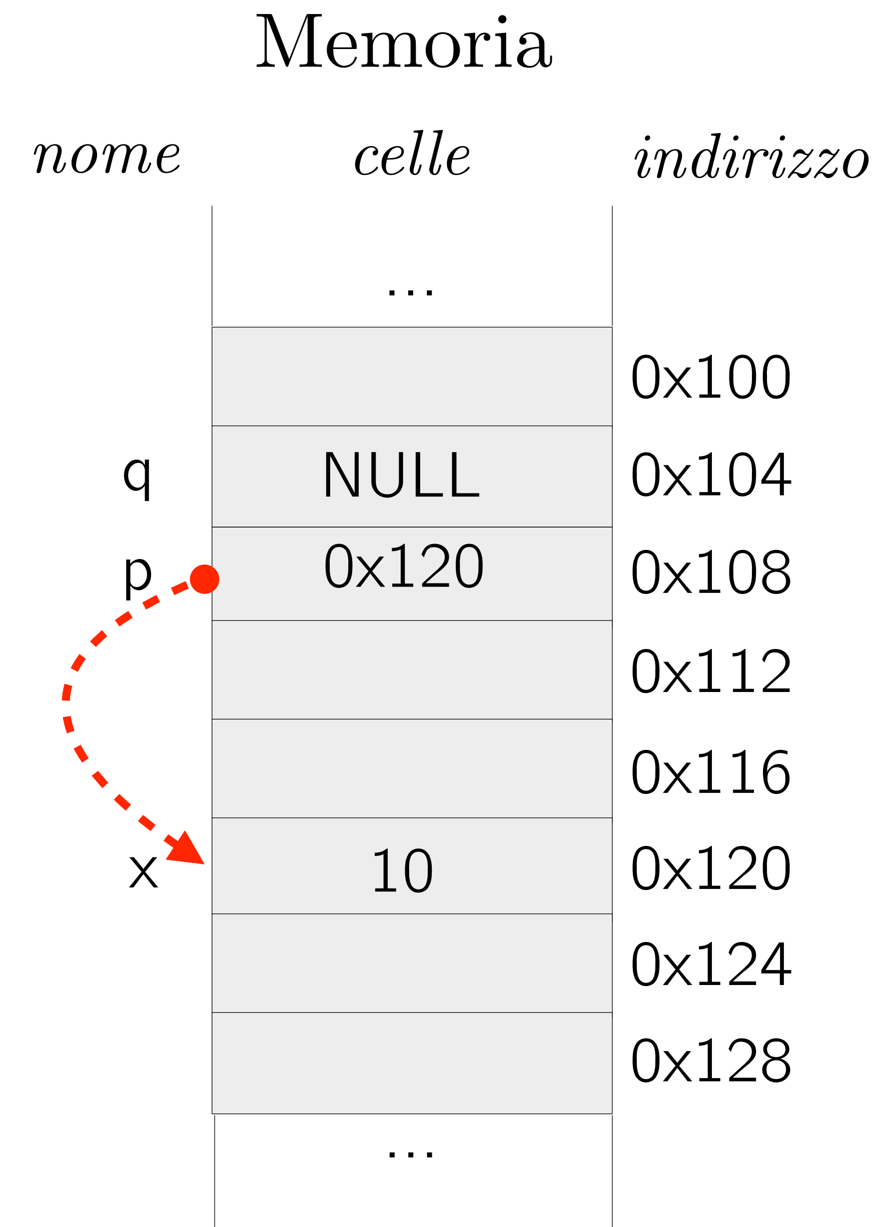


Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

NO! q non è inizializzato.
Segmentation Fault!

```
q = NULL;
```

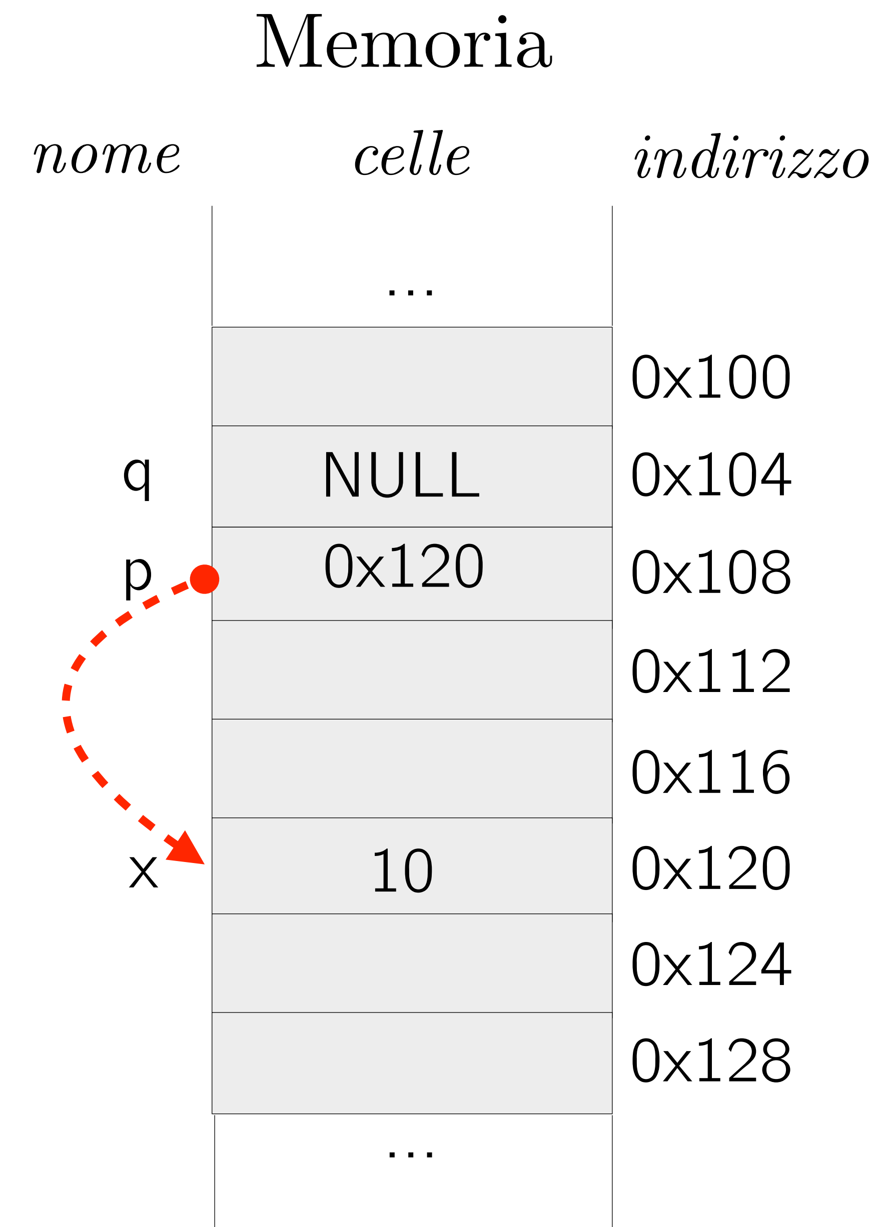


Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

NO! q non è inizializzato.
Segmentation Fault!

```
q = NULL;
*q = 5;
```



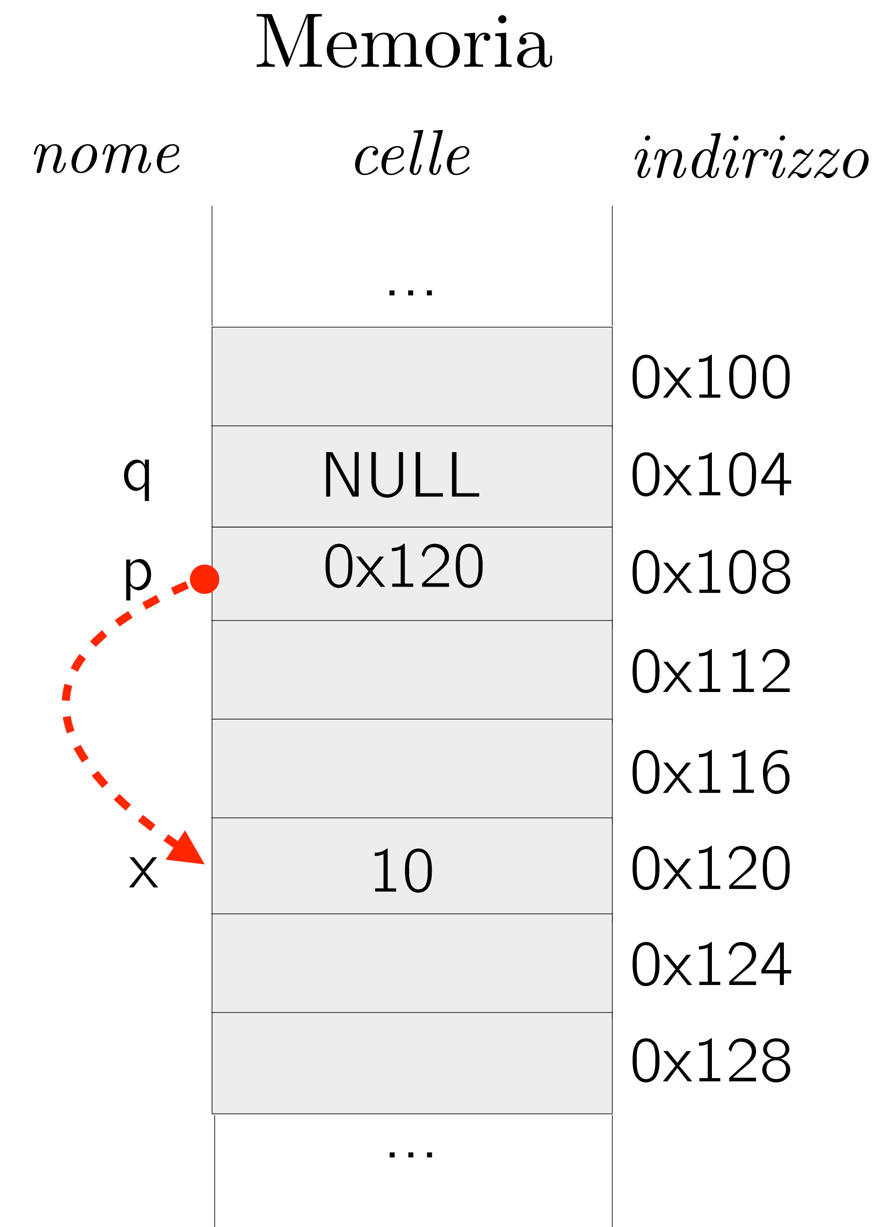
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

NO! q non è inizializzato.
Segmentation Fault!

```
q = NULL;
*q = 5;
```

NO! NULL non è un indirizzo valido.
Segmentation Fault!



Puntatori (2)

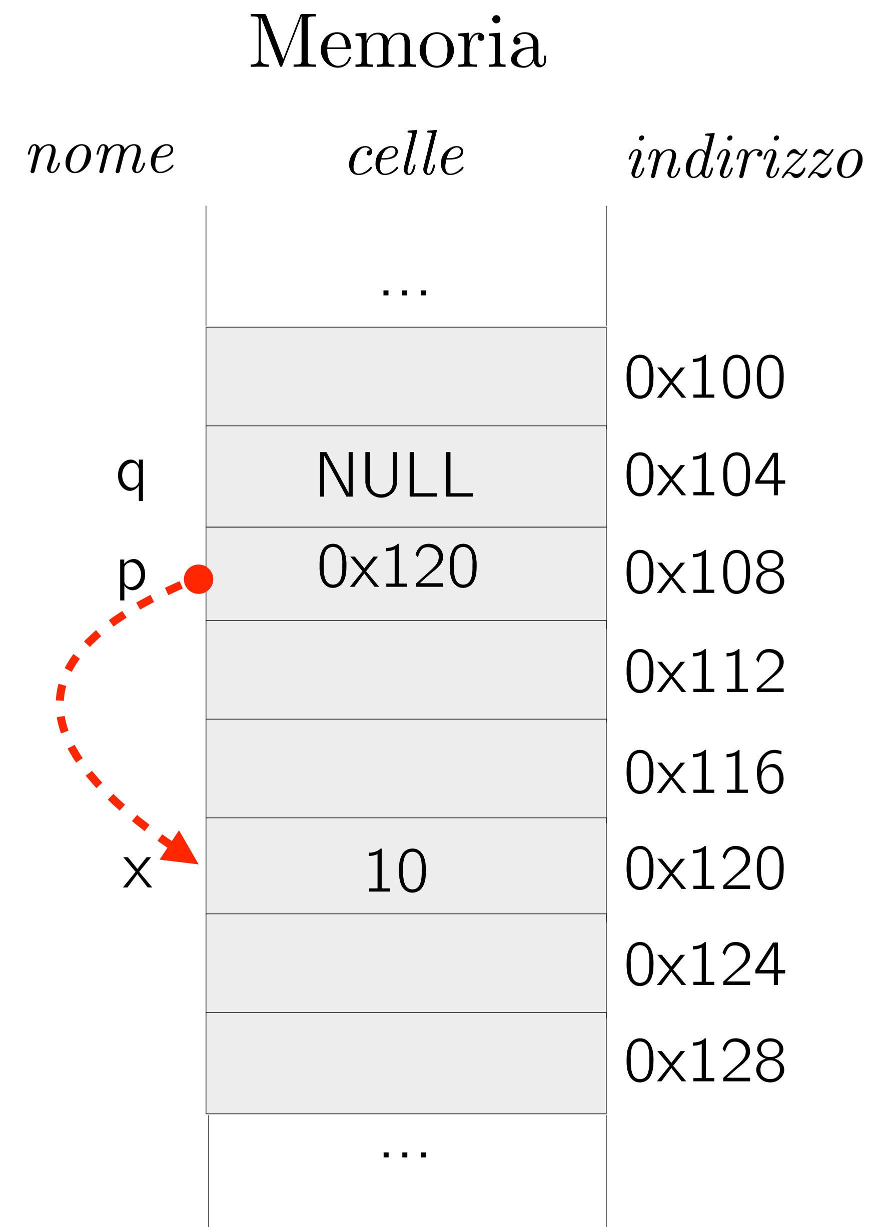
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

NO! q non è inizializzato.
Segmentation Fault!

```
q = NULL;
*q = 5;
```

NO! NULL non è un indirizzo valido.
Segmentation Fault!

```
q = x;
```



Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```

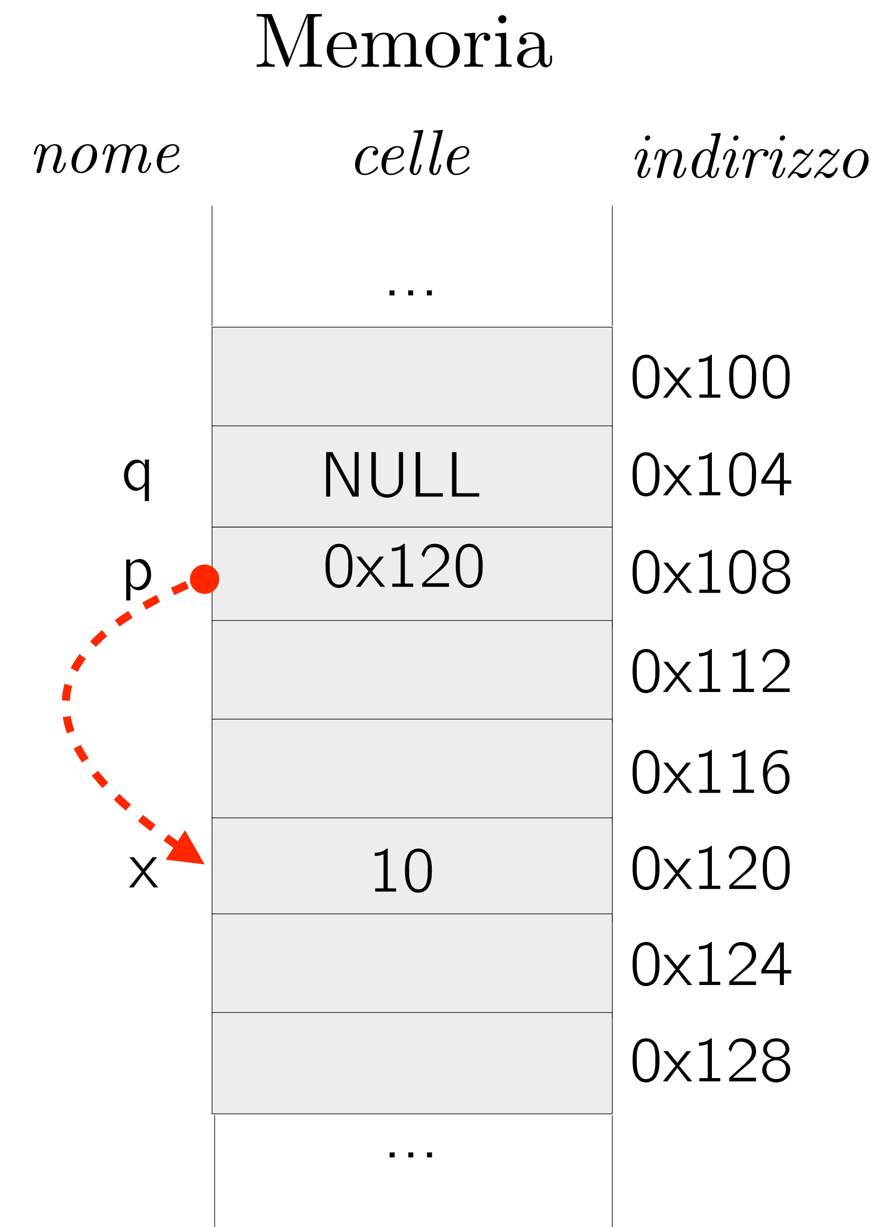
NO! q non è inizializzato.
Segmentation Fault!

```
q = NULL;
*q = 5;
```

NO! NULL non è un indirizzo valido.
Segmentation Fault!

```
q = x;
```

Errore di tipo



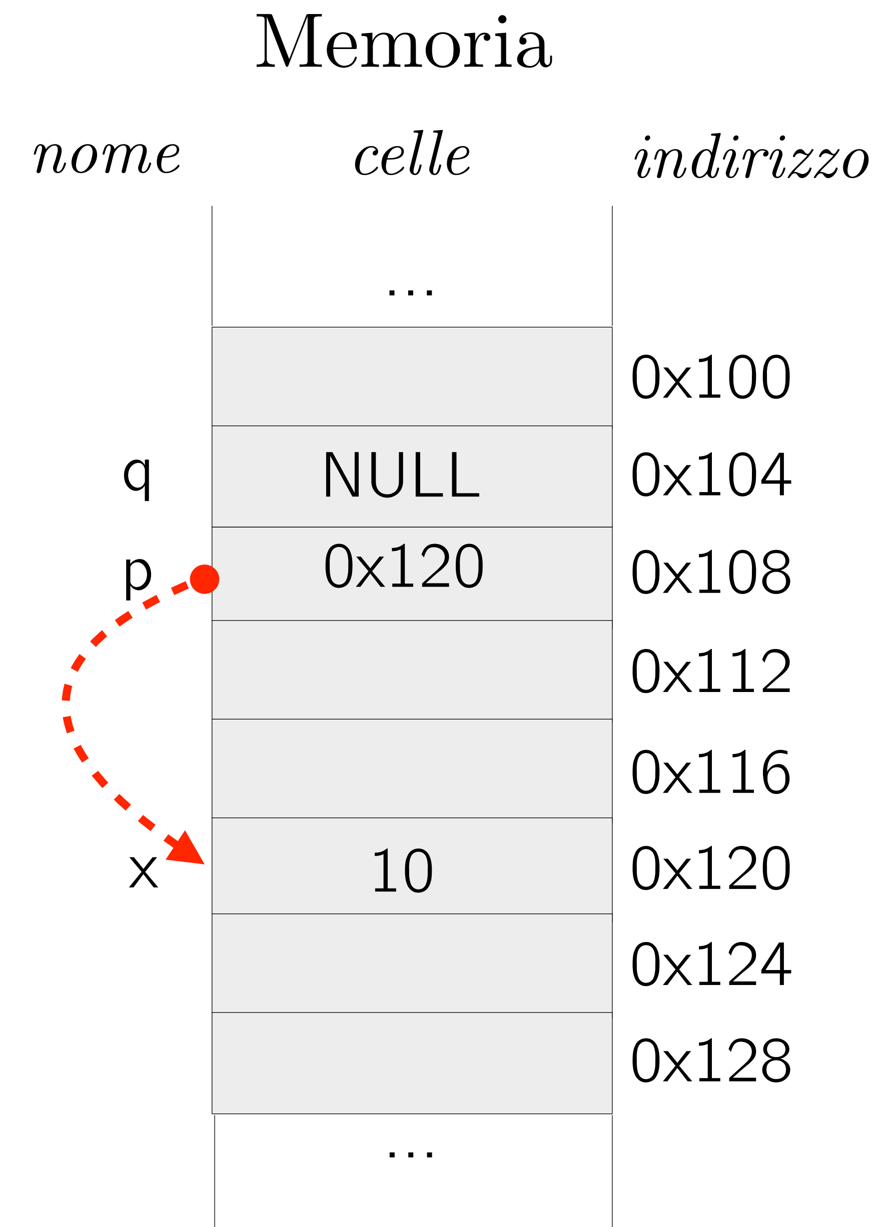
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```
NO! q non è inizializzato. Segmentation Fault!

```
q = NULL;
*q = 5;
```
NO! NULL non è un indirizzo valido. Segmentation Fault!

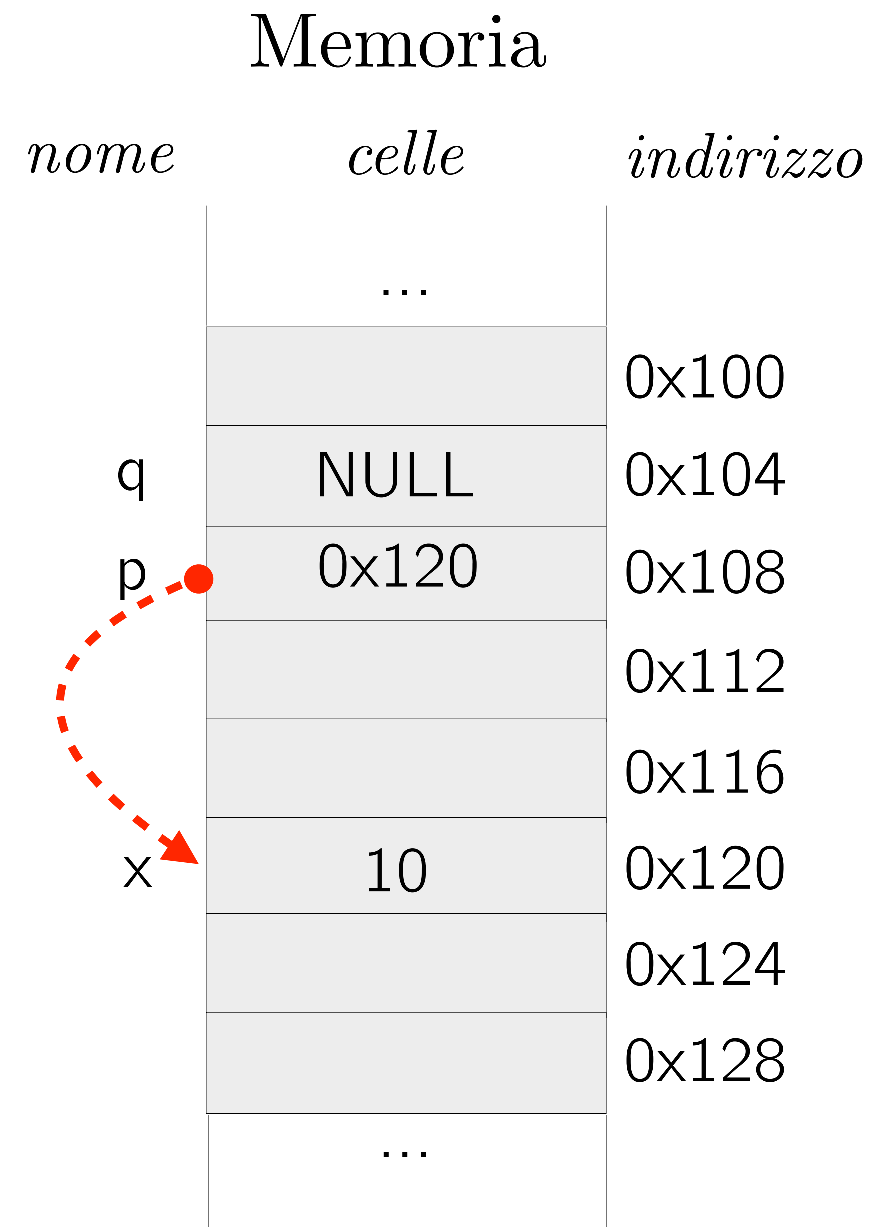
```
q = x;
```
Errore di tipo

```
q = &p;
```



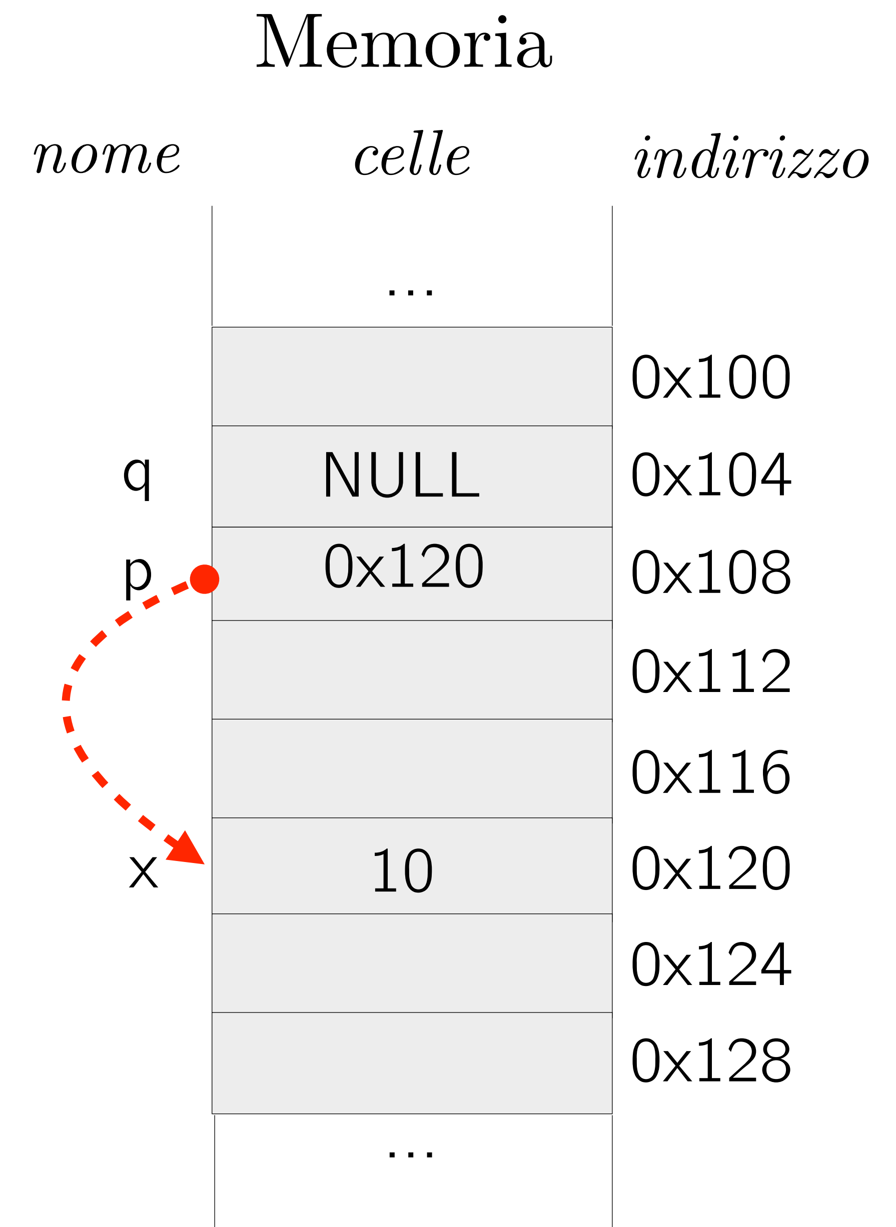
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;        NO! q non è inizializzato.
               Segmentation Fault!
q = NULL;      NO! NULL non è
               un indirizzo valido.
*q = 5;        Segmentation Fault!
q = x;         Errore di tipo
q = &p;        Errore di tipo
```



Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;        NO! q non è inizializzato.
               Segmentation Fault!
q = NULL;      NO! NULL non è
               un indirizzo valido.
*q = 5;        Segmentation Fault!
q = x;         Errore di tipo
q = &p;        Errore di tipo
q = &x;
```



Puntatori (2)

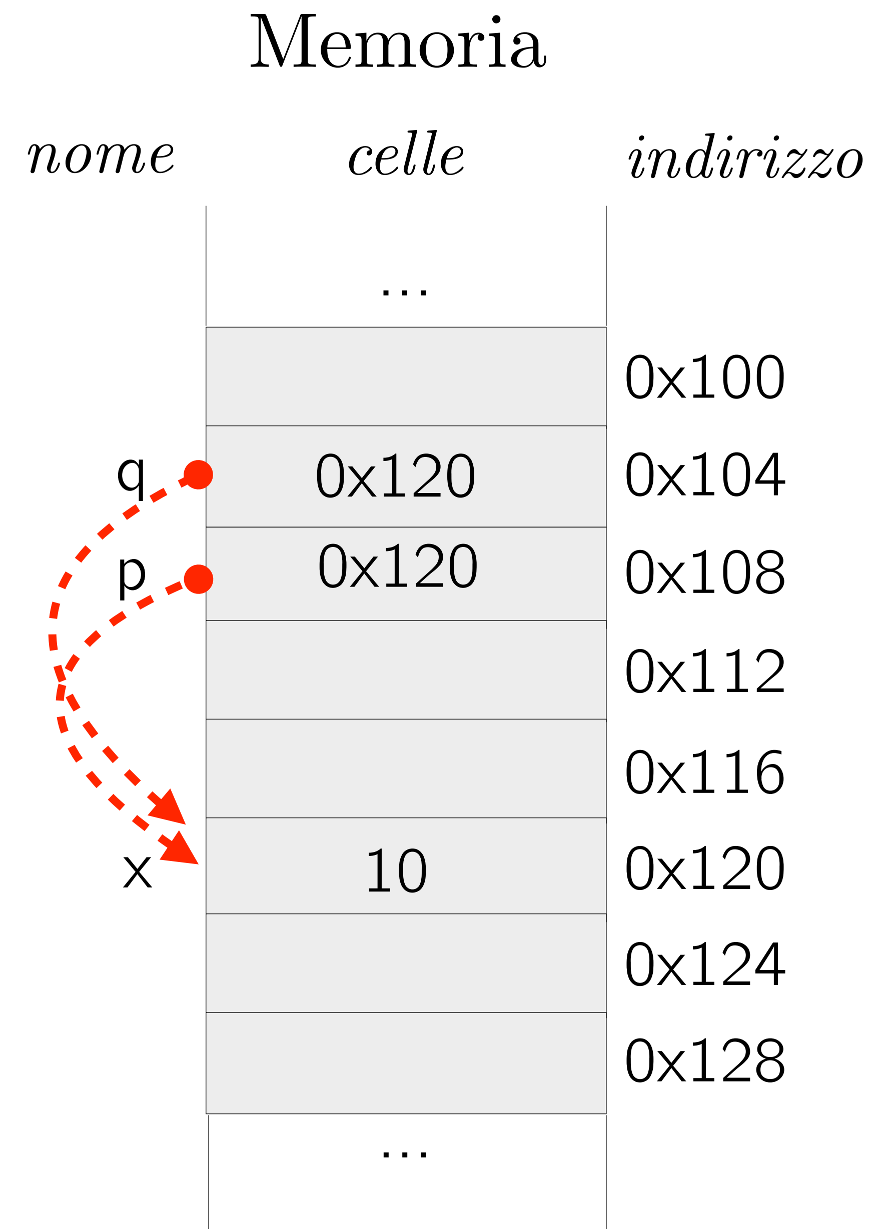
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
```
NO! q non è inizializzato. Segmentation Fault!

```
q = NULL;
*q = 5;
```
NO! NULL non è un indirizzo valido. Segmentation Fault!

```
q = x;
```
Errore di tipo

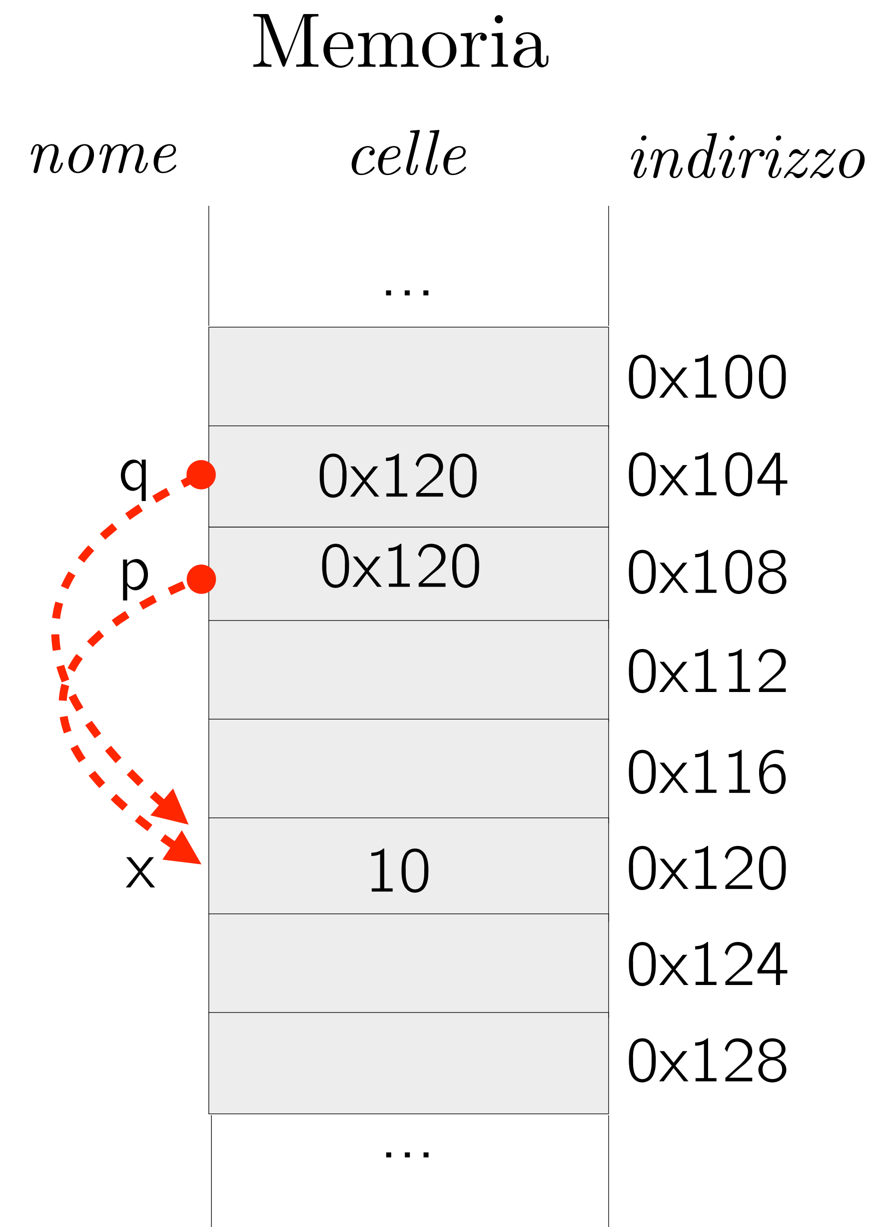
```
q = &p;
```
Errore di tipo

```
q = &x;
```



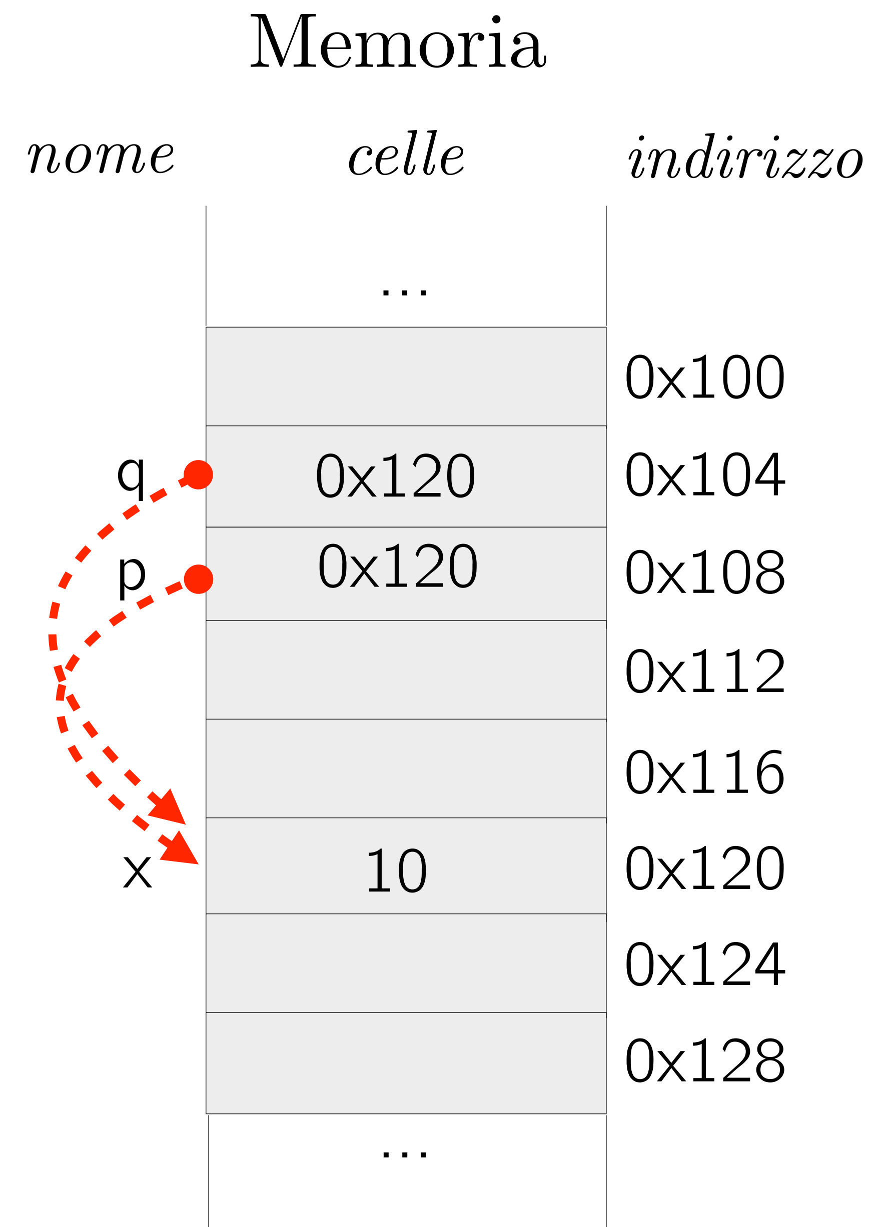
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;      NO! q non è inizializzato.
             Segmentation Fault!
q = NULL;    NO! NULL non è
             un indirizzo valido.
*q = 5;      Segmentation Fault!
q = x;       Errore di tipo
q = &p;      Errore di tipo
q = &x;
q = p;
```



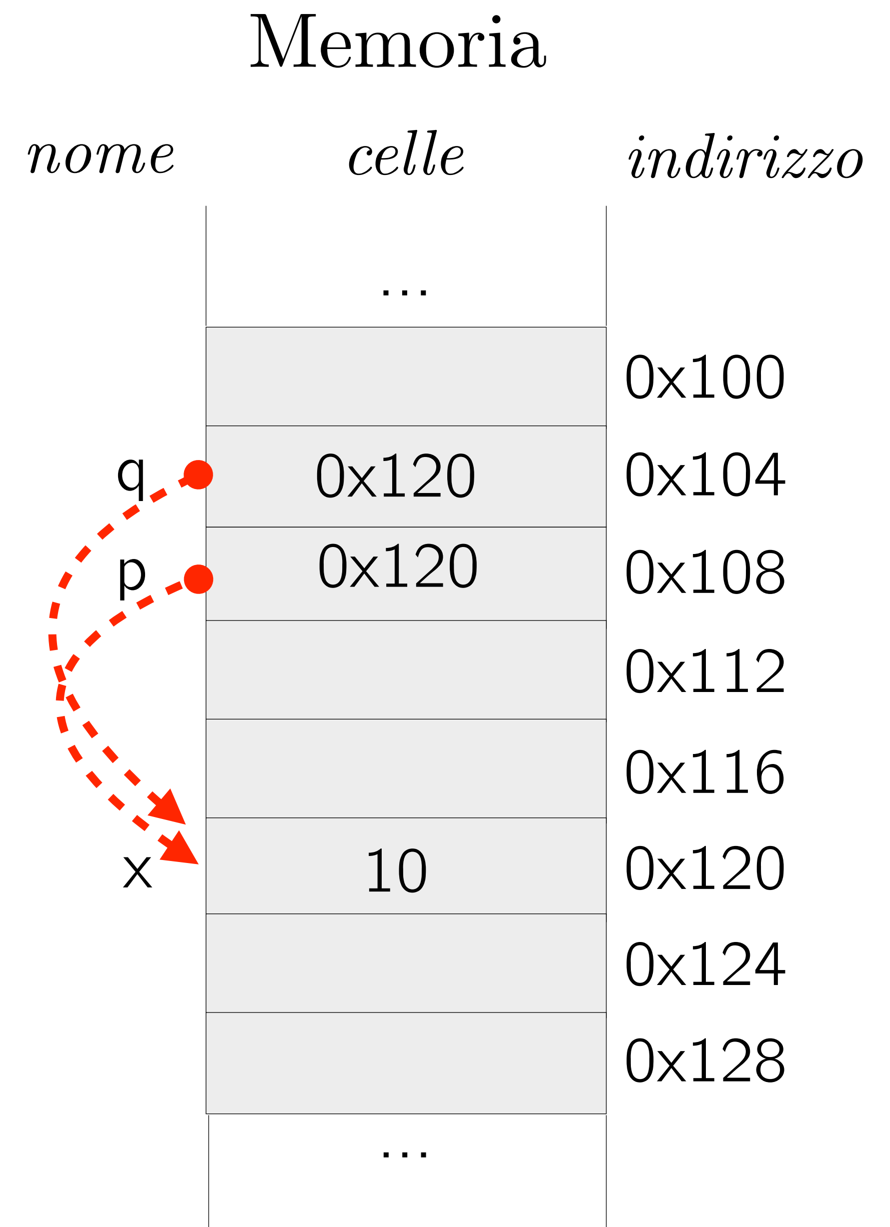
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;        NO! q non è inizializzato.
               Segmentation Fault!

q = NULL;      NO! NULL non è
*q = 5;        un indirizzo valido.
               Segmentation Fault!
q = x;         Errore di tipo
q = &p;        Errore di tipo
q = &x;
q = p;         Equivalente al precedente
```



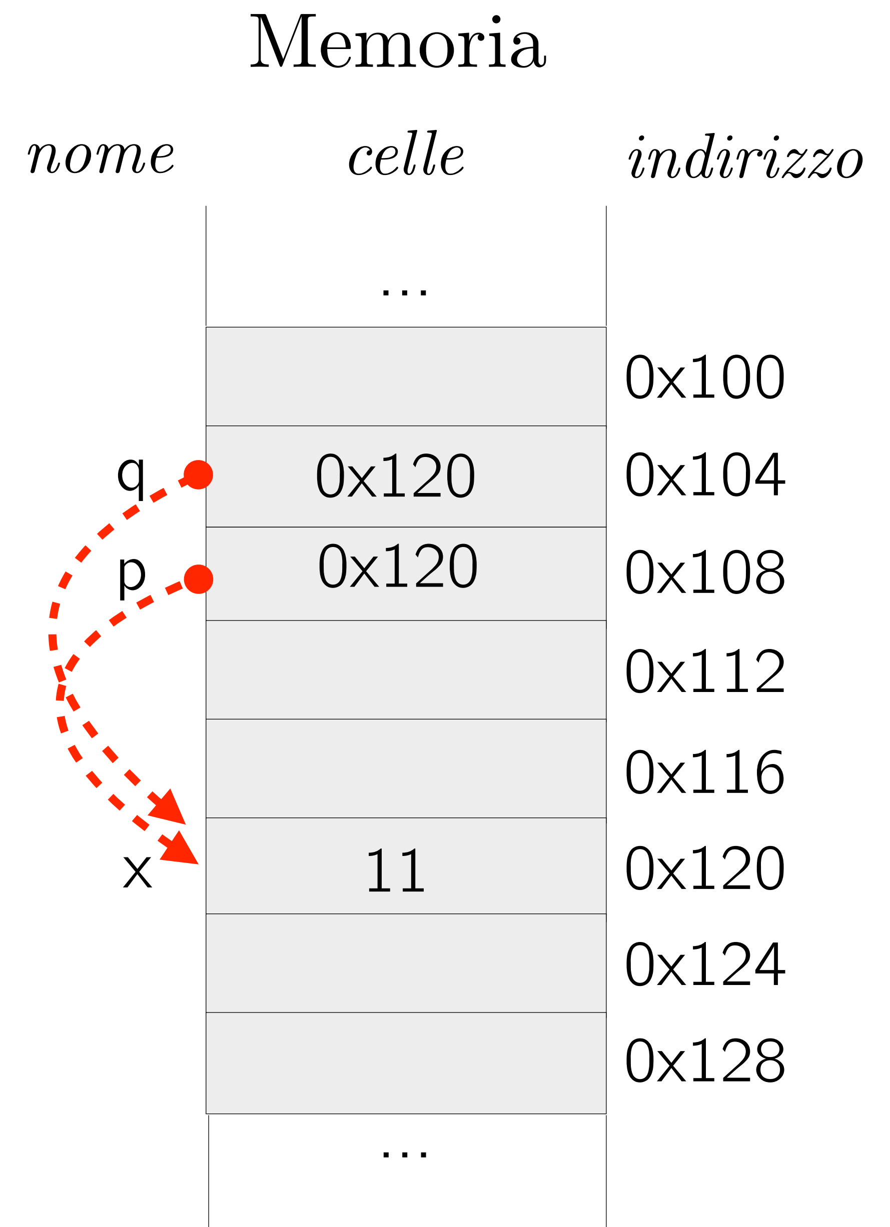
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;        NO! q non è inizializzato.
               Segmentation Fault!

q = NULL;      NO! NULL non è
               un indirizzo valido.
*q = 5;        Segmentation Fault!
q = x;         Errore di tipo
q = &p;        Errore di tipo
q = &x;
q = p;         Equivalente al precedente
*q = 11;
```



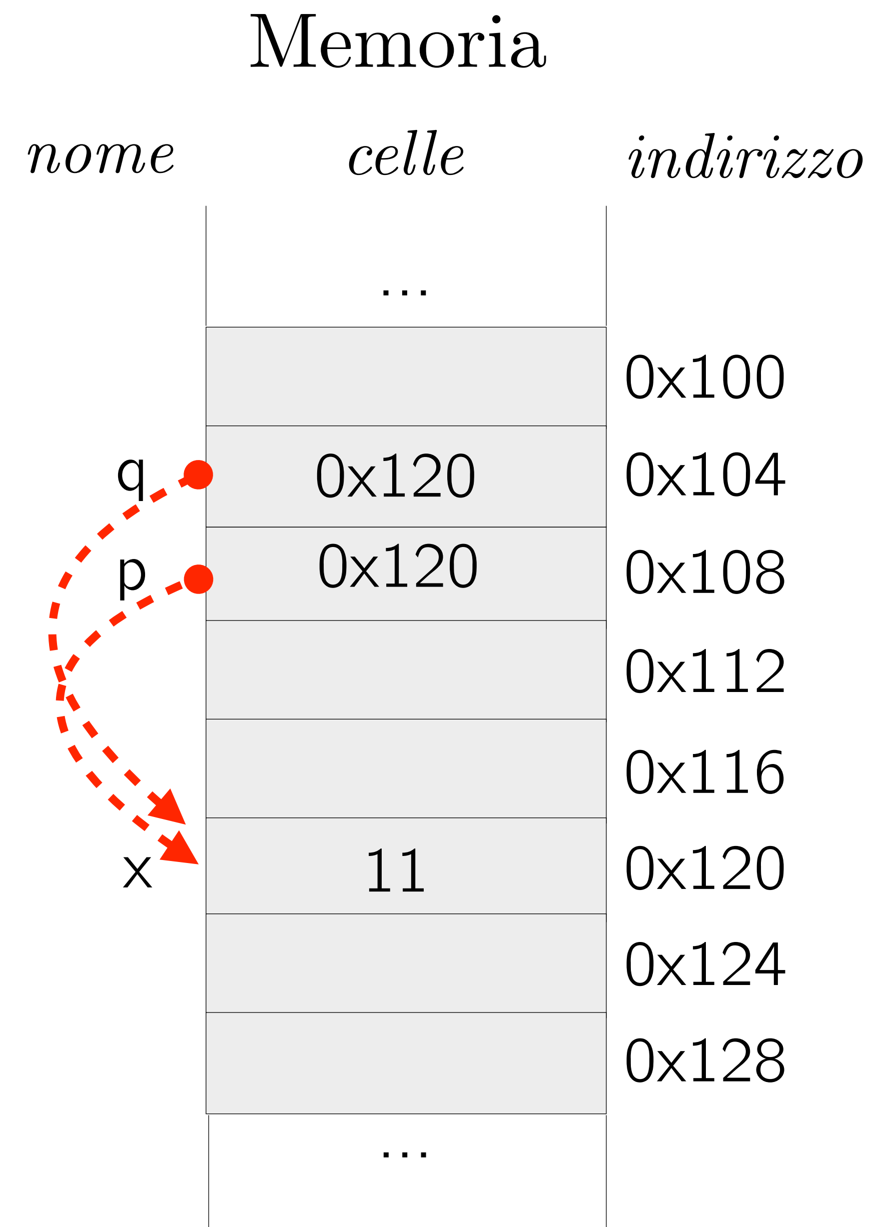
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;        NO! q non è inizializzato.
               Segmentation Fault!
q = NULL;      NO! NULL non è
               un indirizzo valido.
*q = 5;        Segmentation Fault!
q = x;         Errore di tipo
q = &p;        Errore di tipo
q = &x;
q = p;         Equivalente al precedente
*q = 11;
```



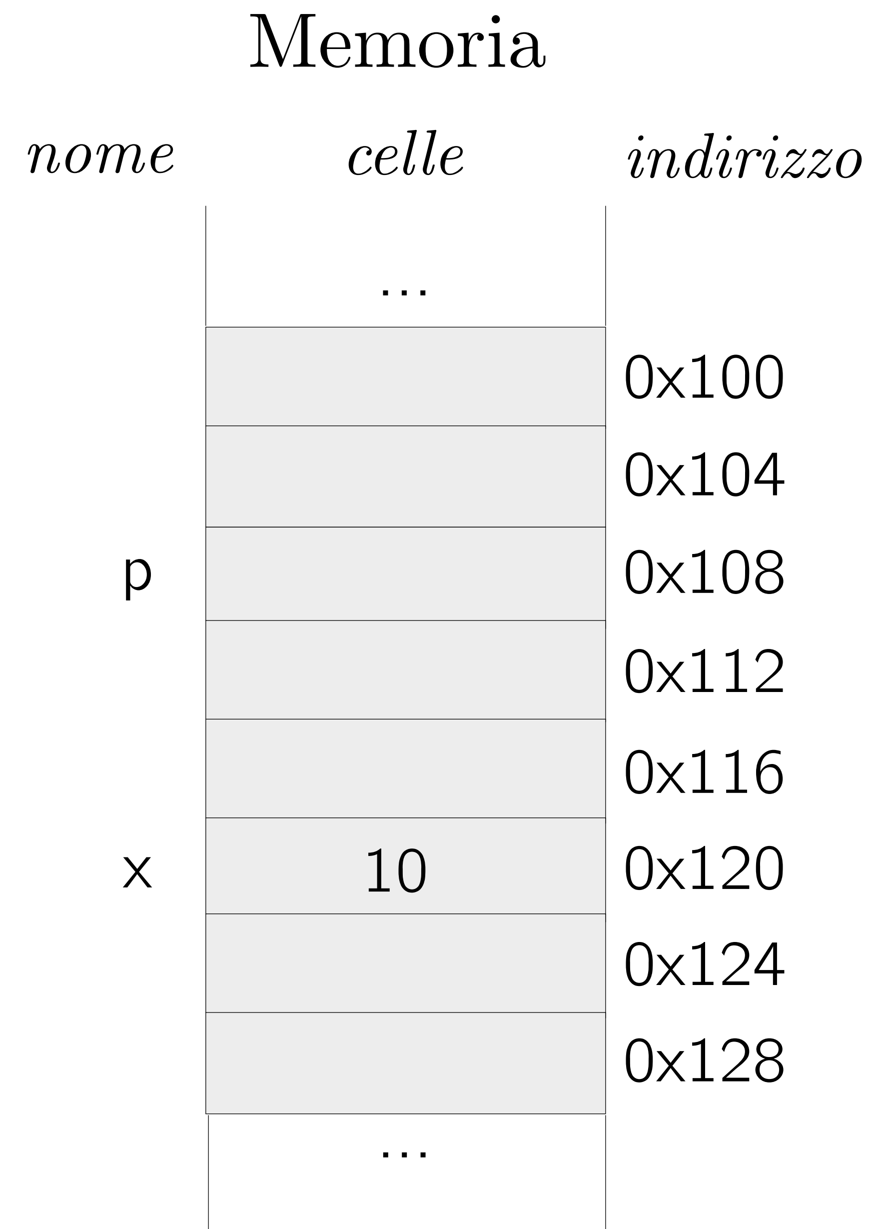
Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;        NO! q non è inizializzato.
               Segmentation Fault!

q = NULL;      NO! NULL non è
               un indirizzo valido.
*q = 5;        Segmentation Fault!
q = x;         Errore di tipo
q = &p;        Errore di tipo
q = &x;
q = p;         Equivalente al precedente
*q = 11;
printf("%p %d %d", q, *q, *p);
```



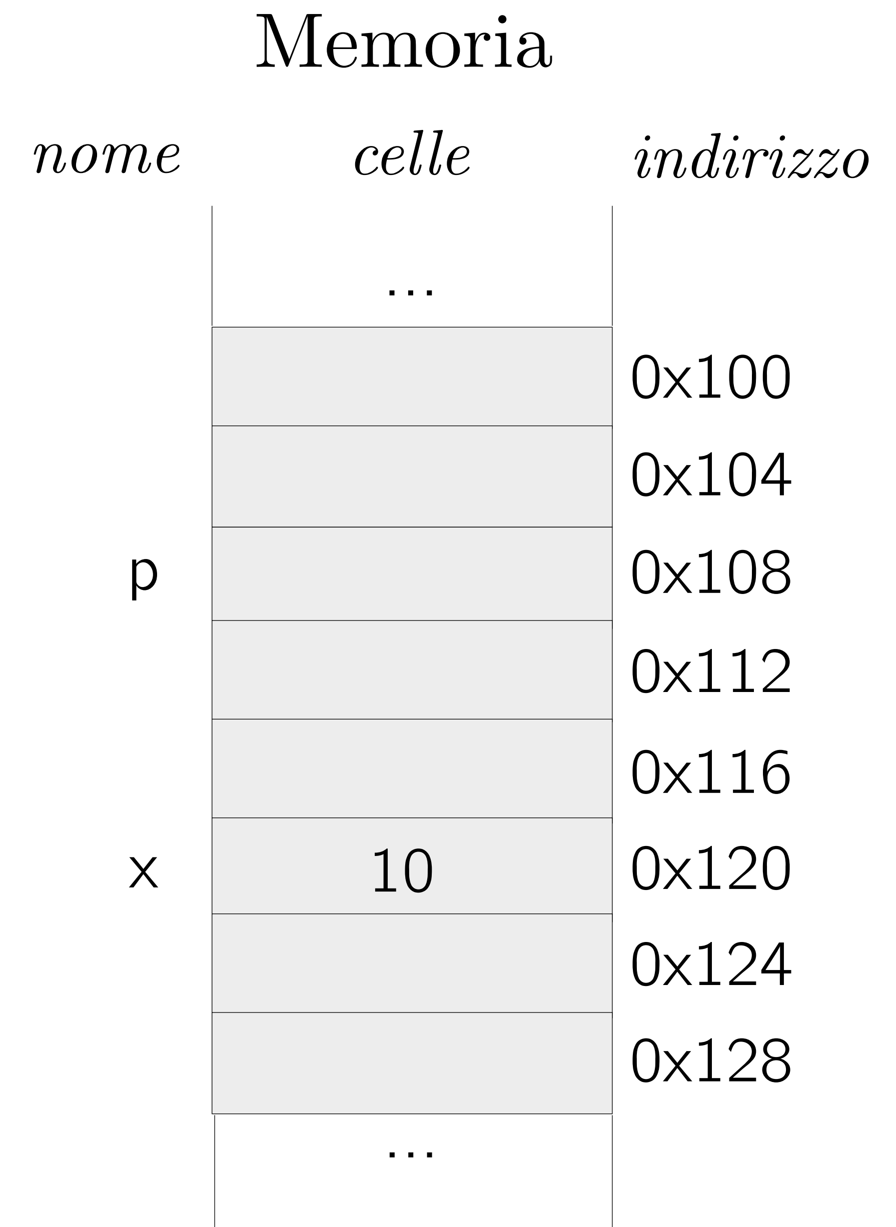
Puntatori (3)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



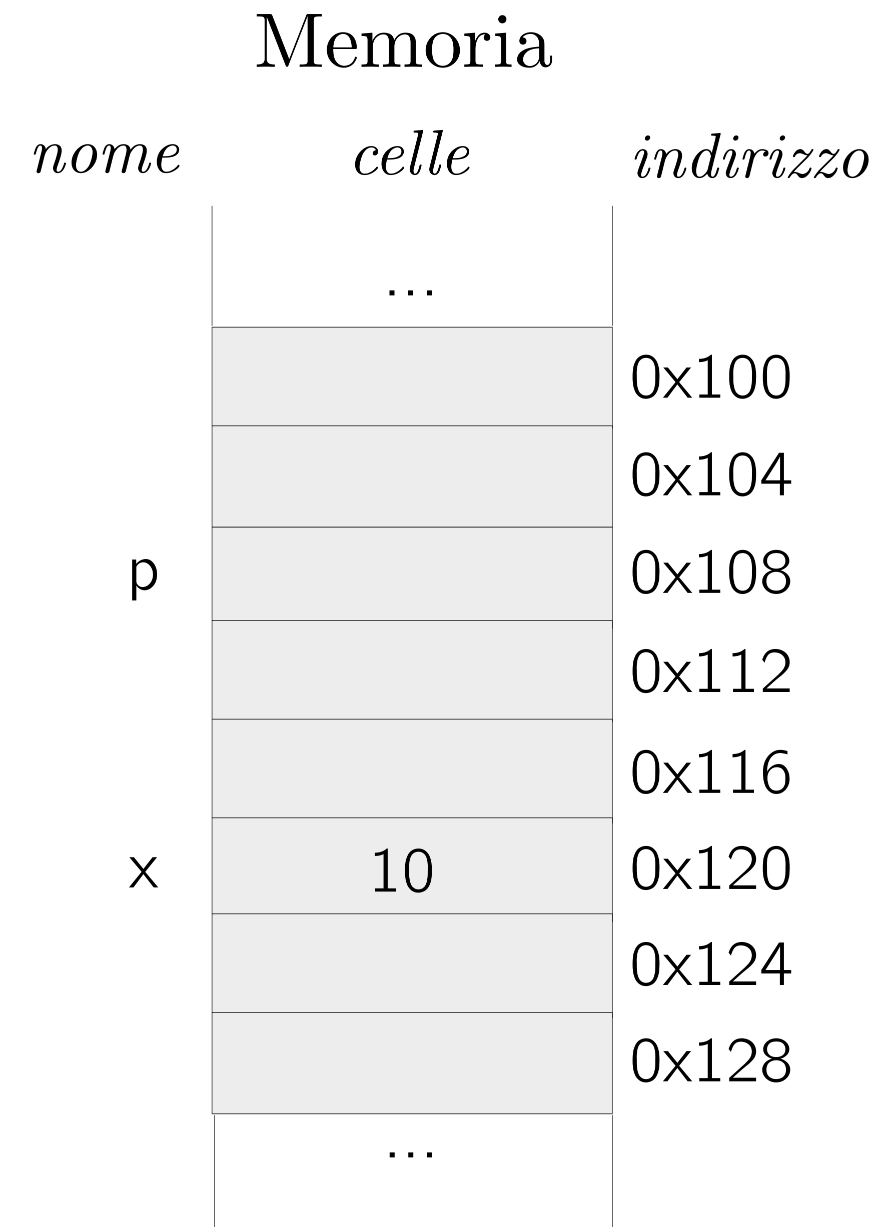
Puntatori (3)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;

char x = 'c';
char *p = &x;
```



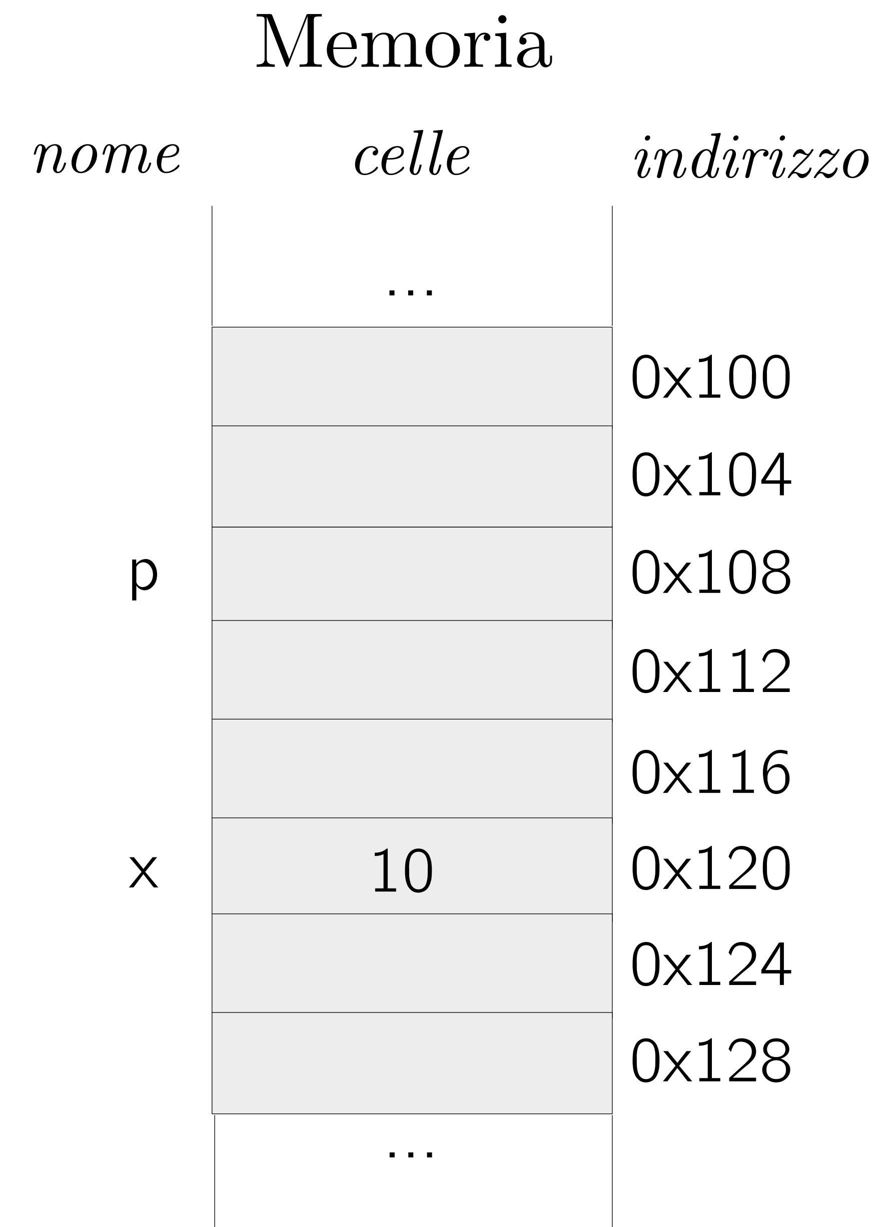
Puntatori (3)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;

char x = 'c';
char *p = &x;

float x = 3.14f;
float *p = &x;
```



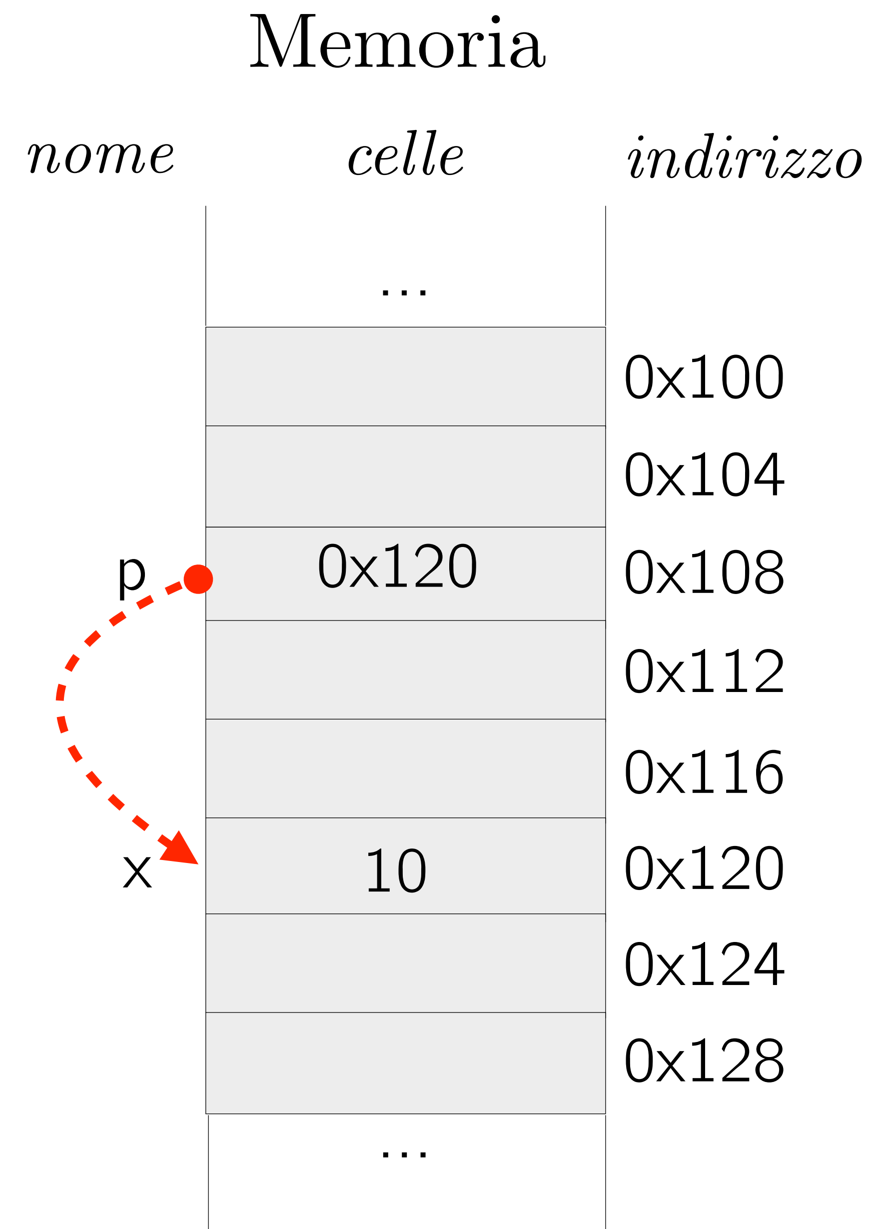
Puntatori (3)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;

char x = 'c';
char *p = &x;

float x = 3.14f;
float *p = &x;

double x = 3.14lf;
double *p = &x;
```



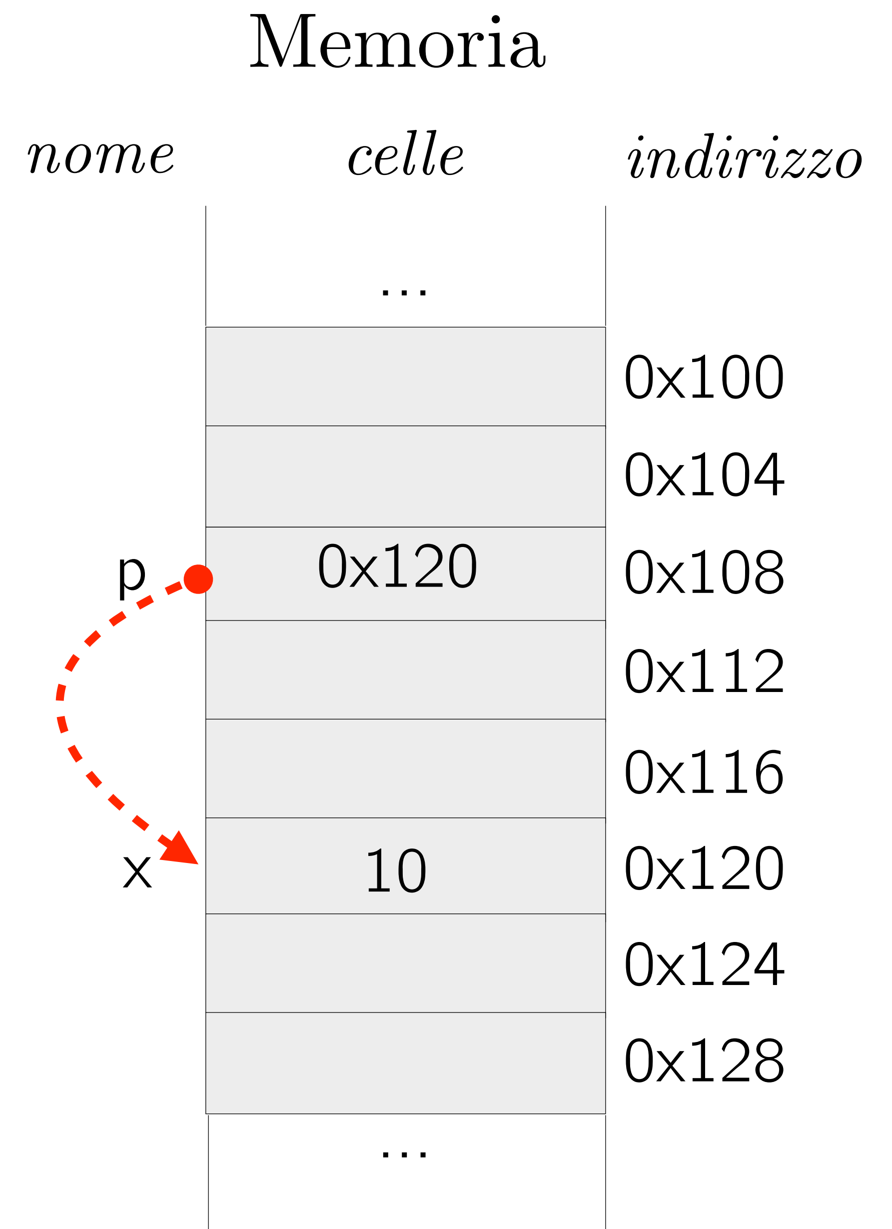
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
```



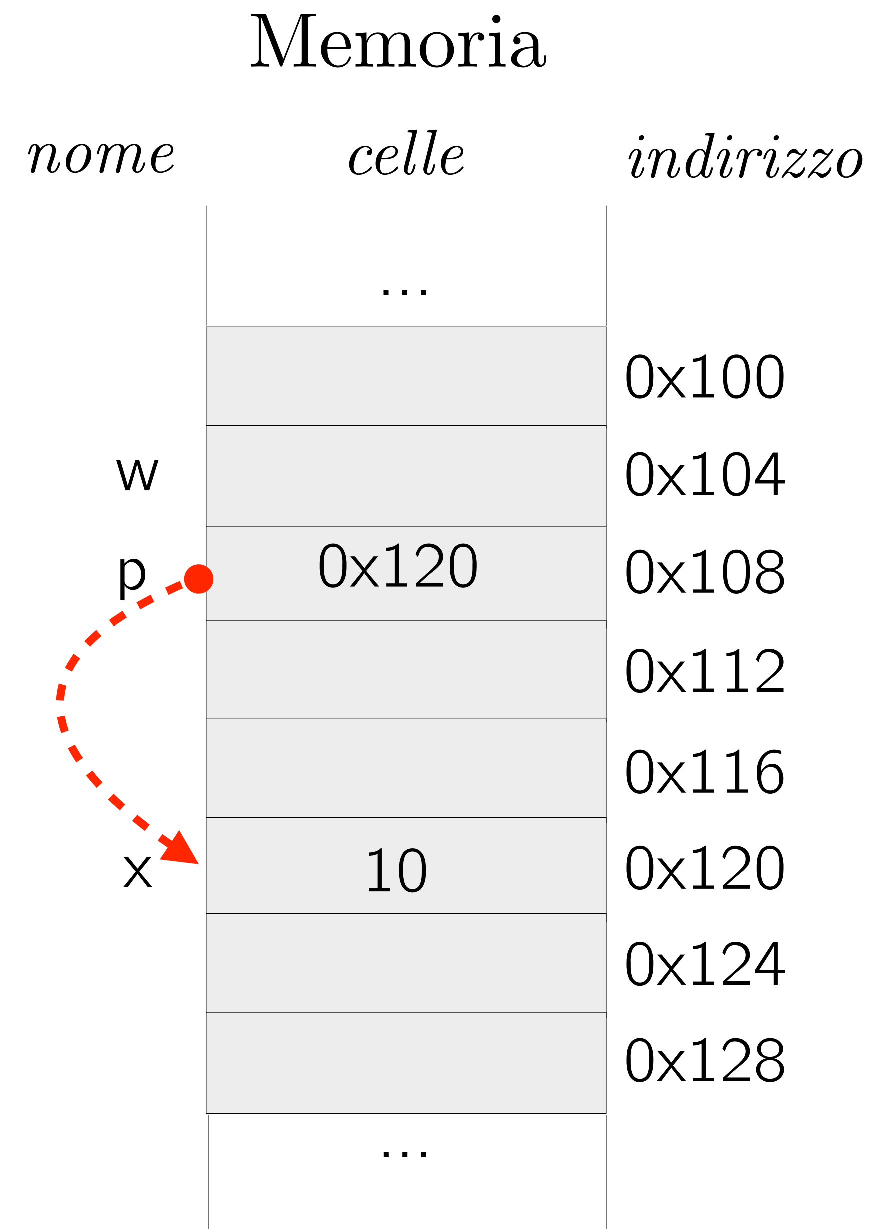
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
```



Puntatori (4)

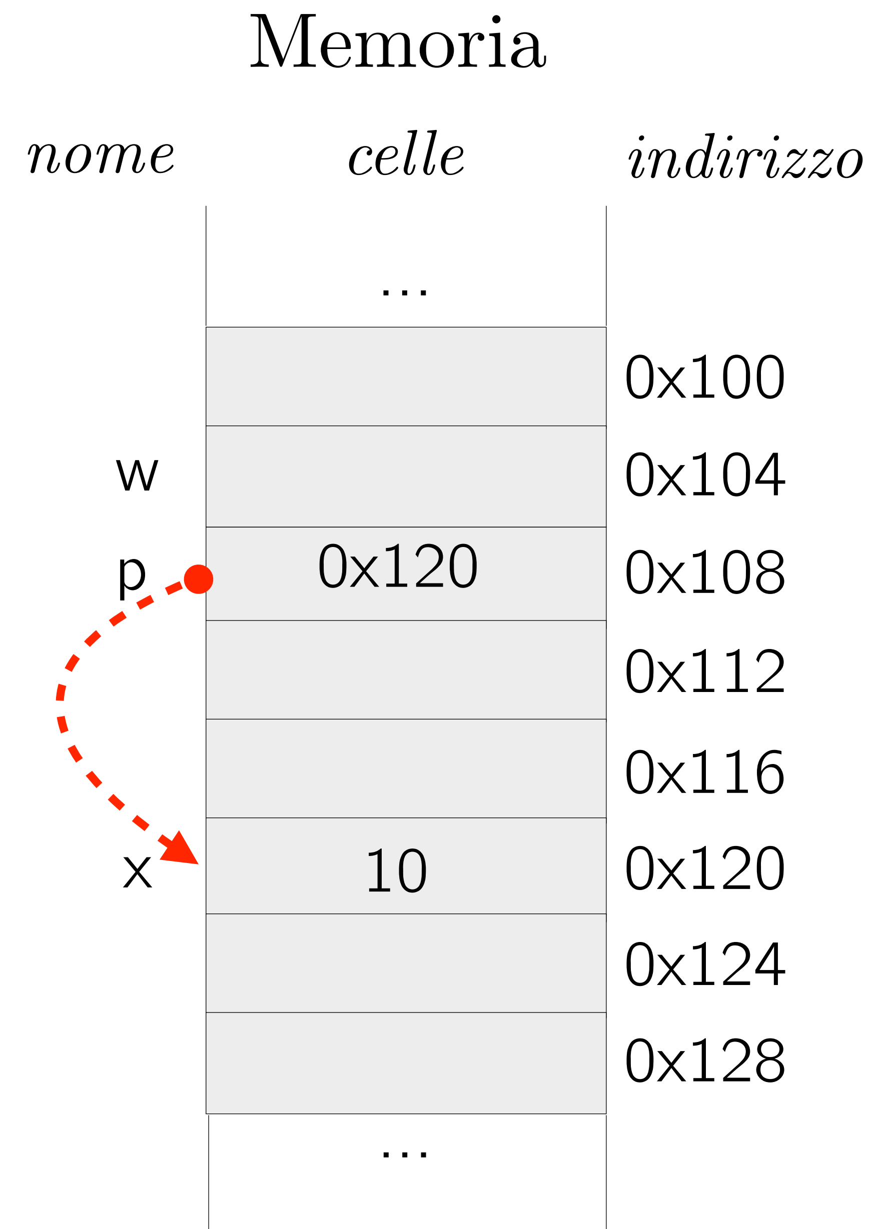
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
```



Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
```

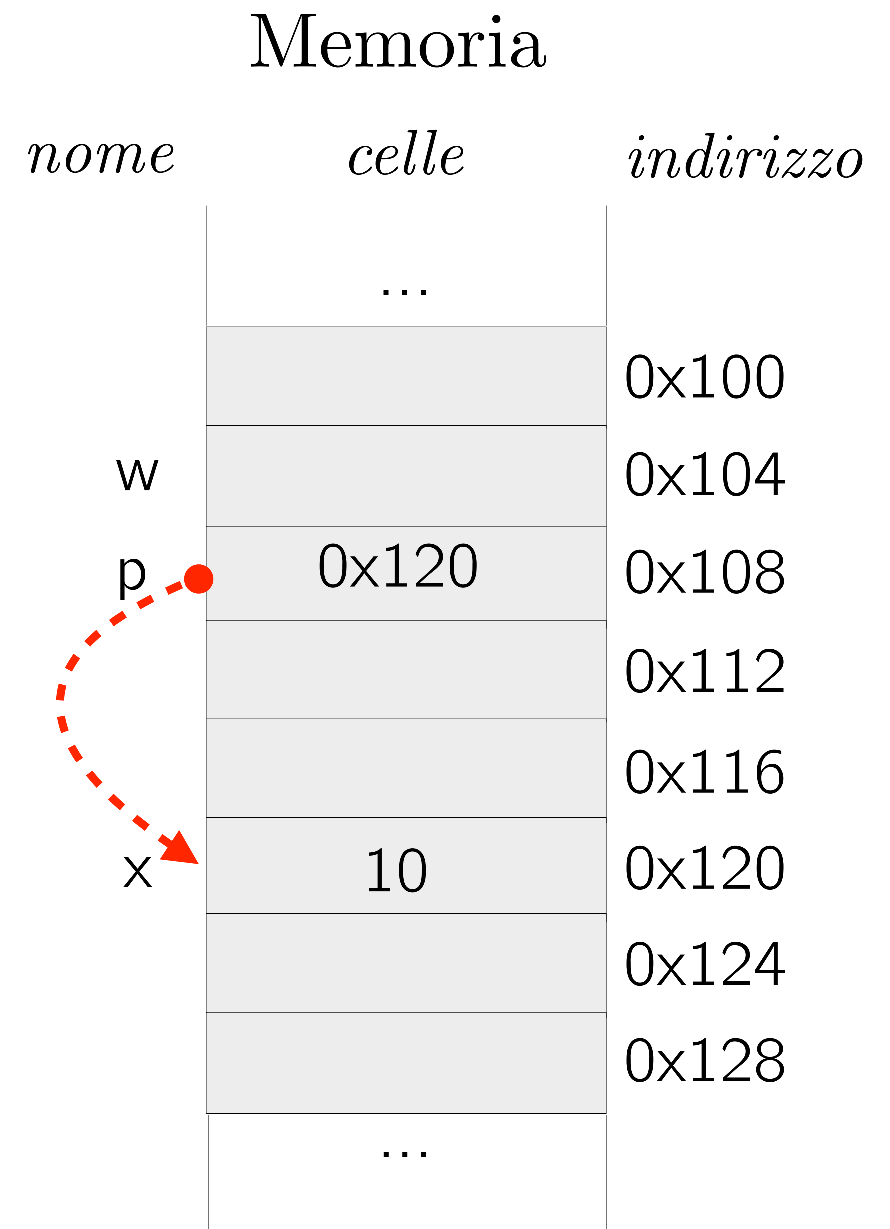
Puntatore a puntatore ad intero



Puntatori (4)

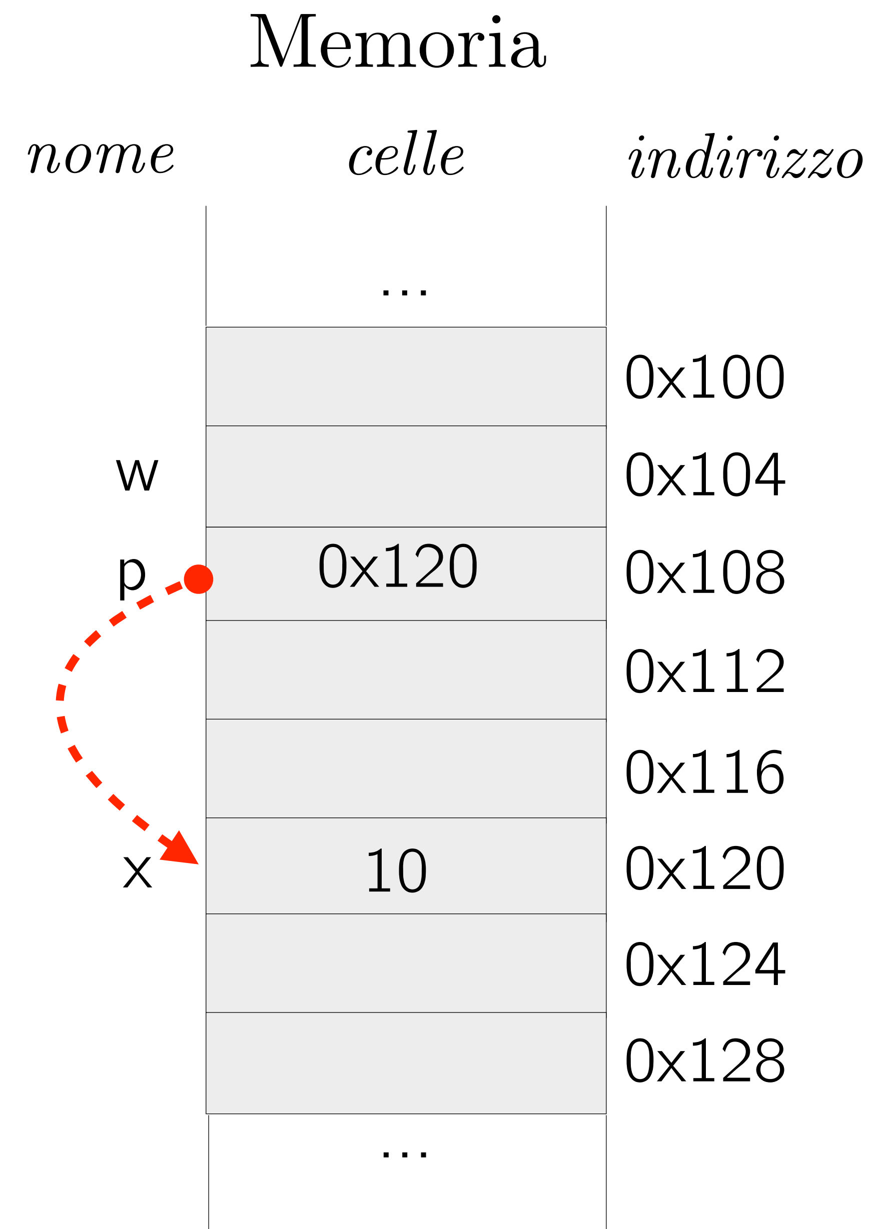
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
w = &x;
```

Puntatore a puntatore ad intero



Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
```

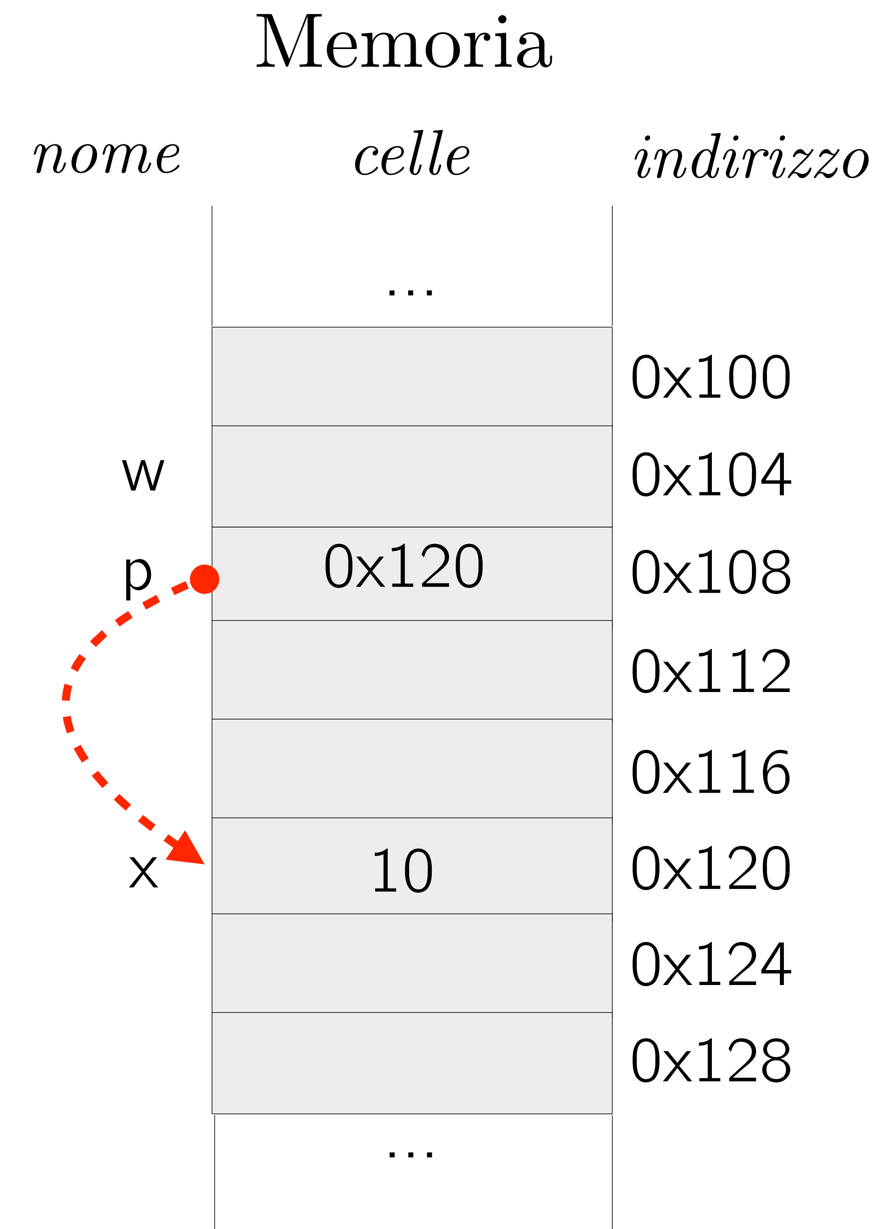


Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
w = &x;
w = p;
```

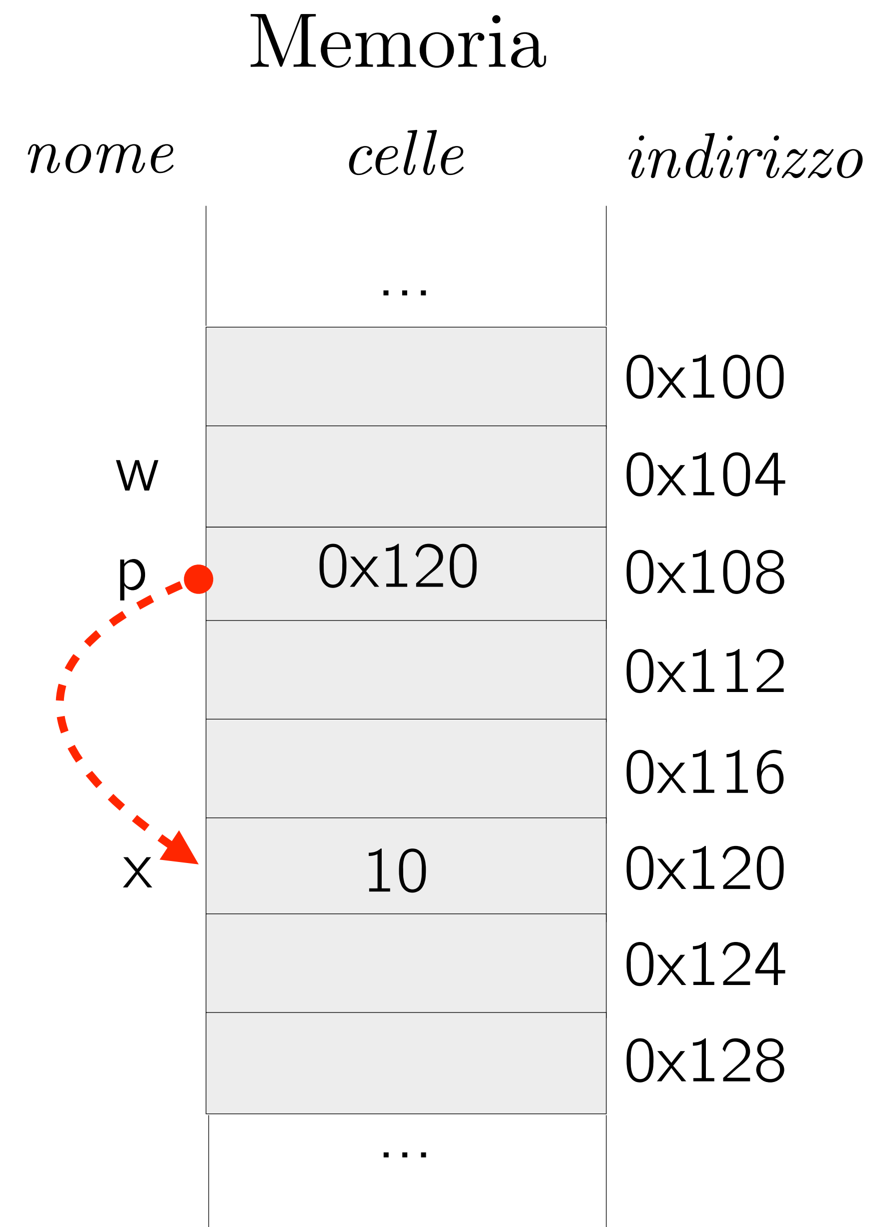
Puntatore a puntatore ad intero

Errore di tipo



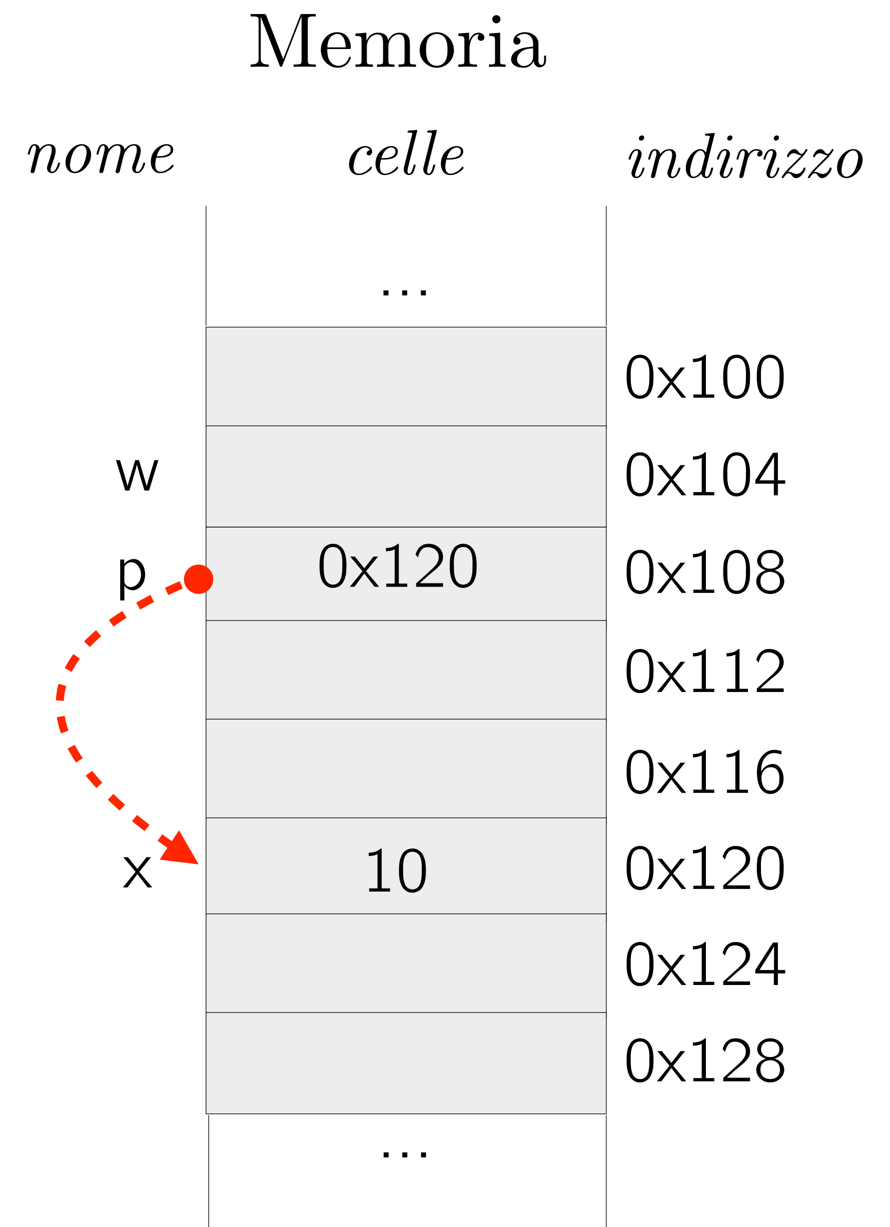
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
```



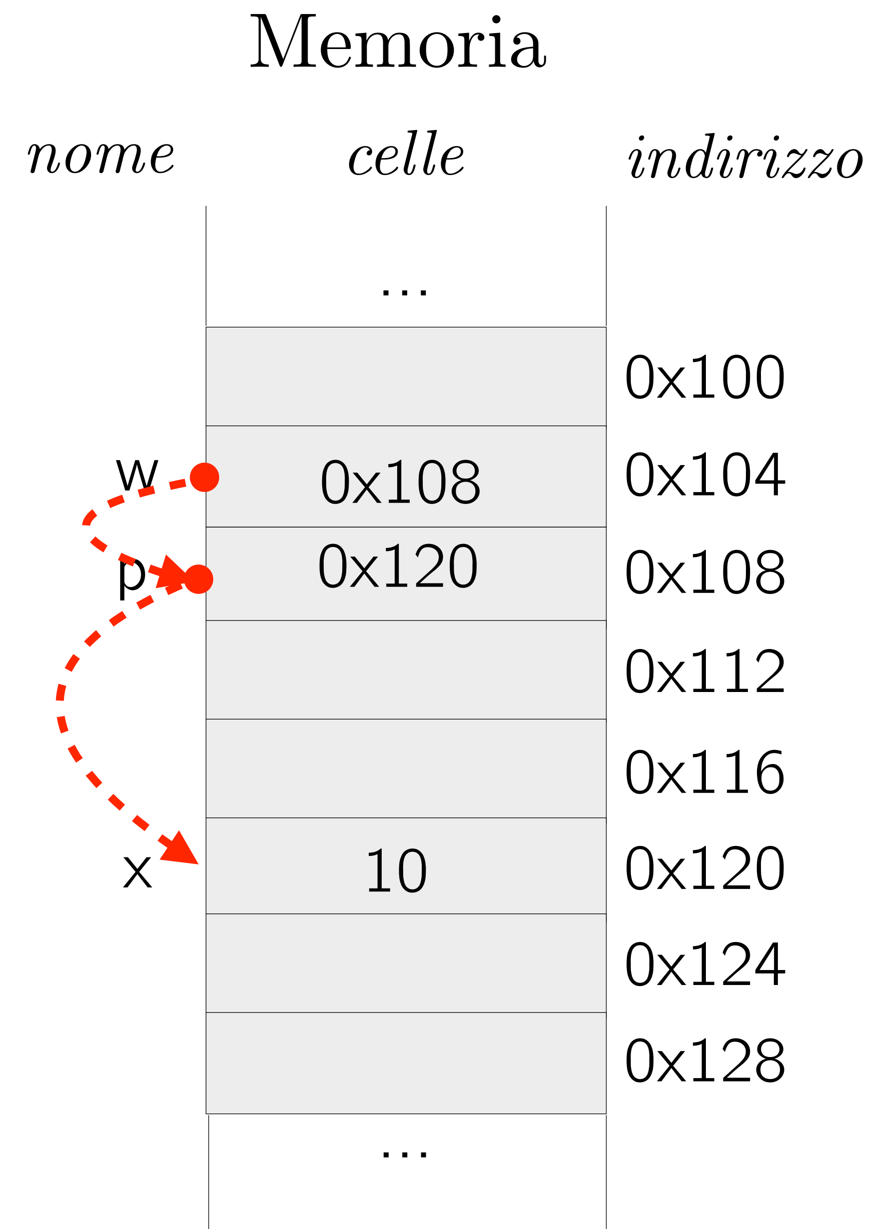
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
```



Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
```



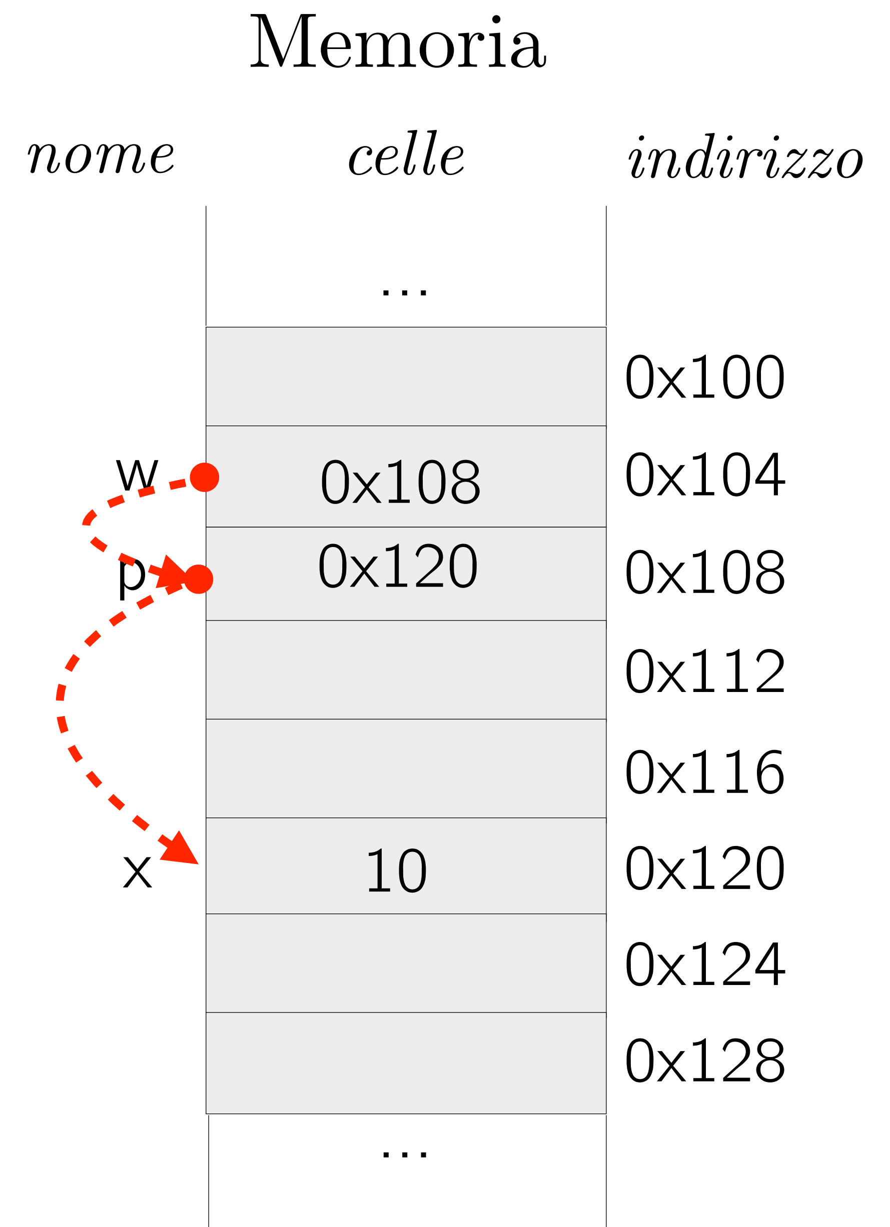
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
w = &x;
w = p;
w = &p;
*w
```

Puntatore a puntatore ad intero

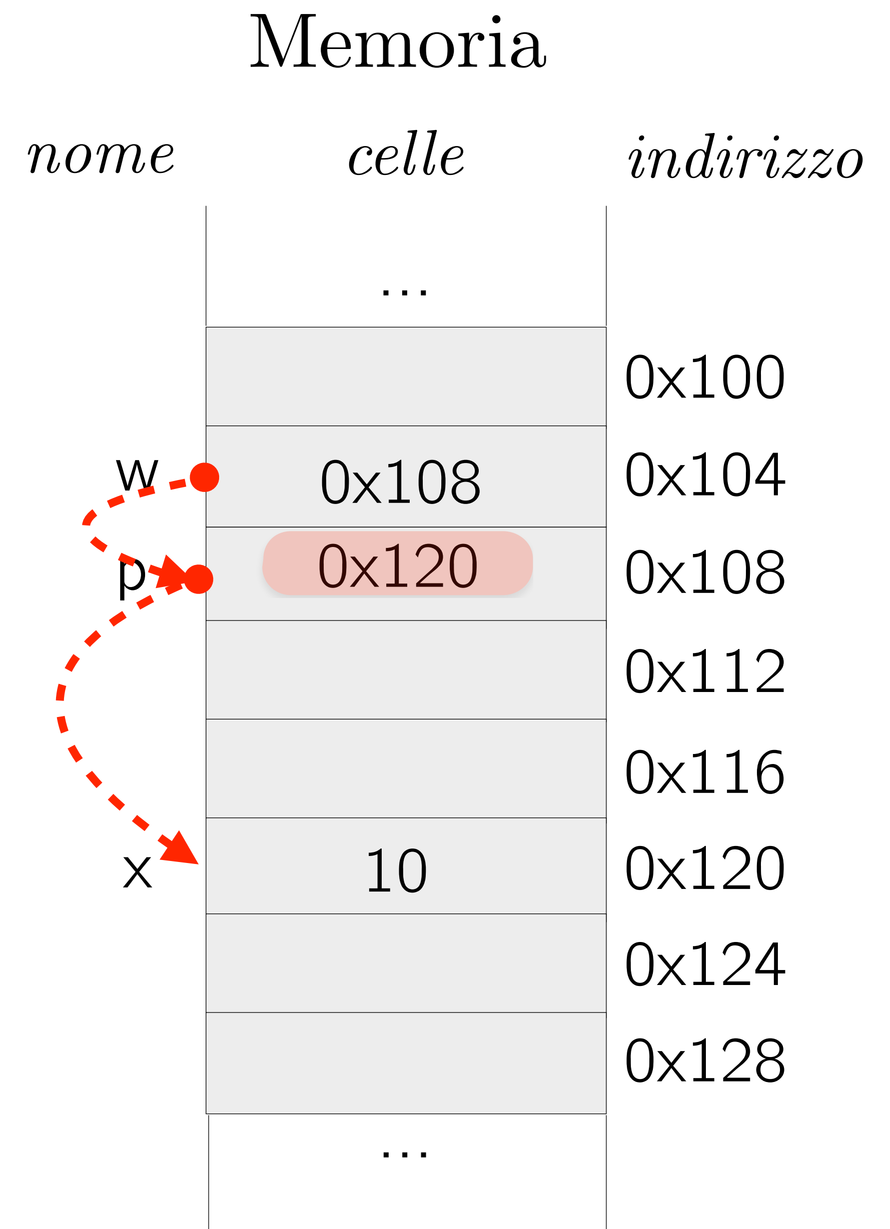
Errore di tipo

Errore di tipo



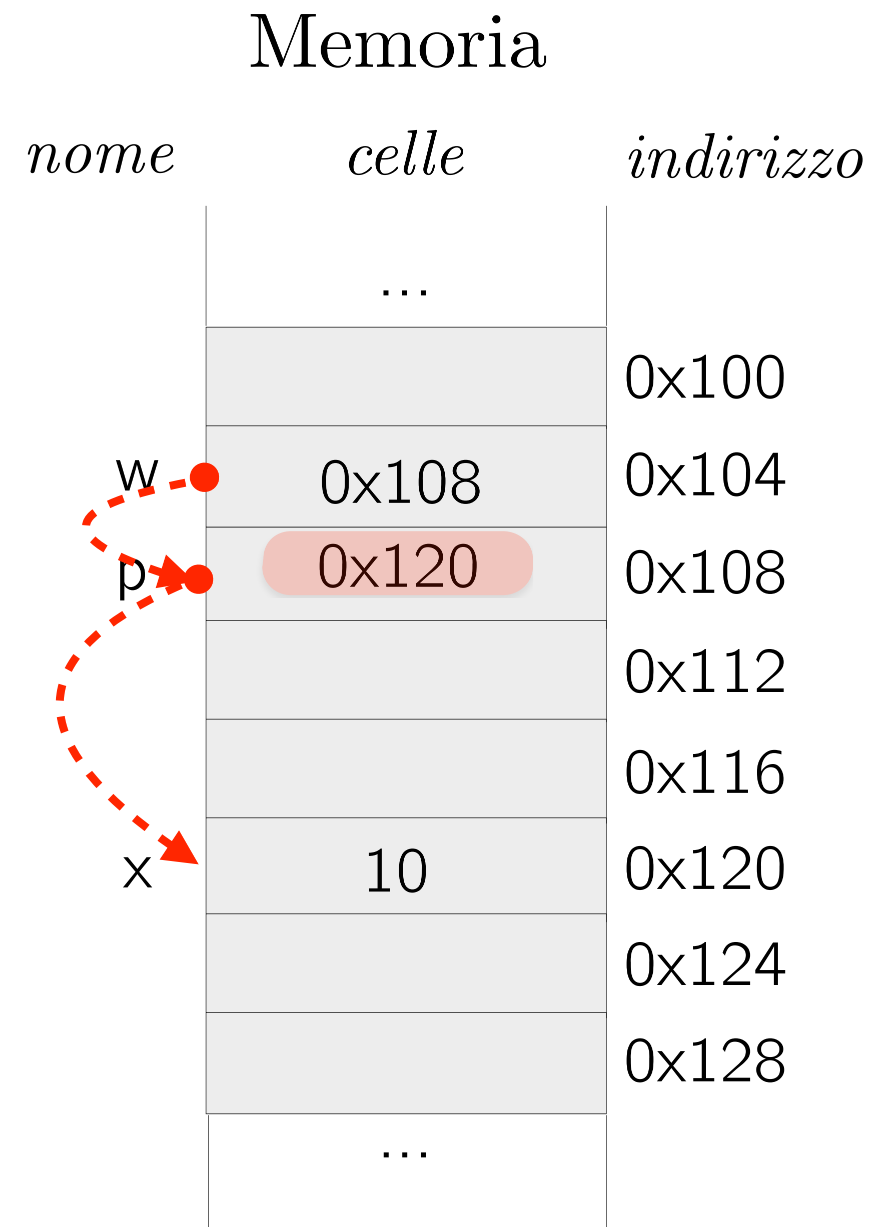
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
*w
```



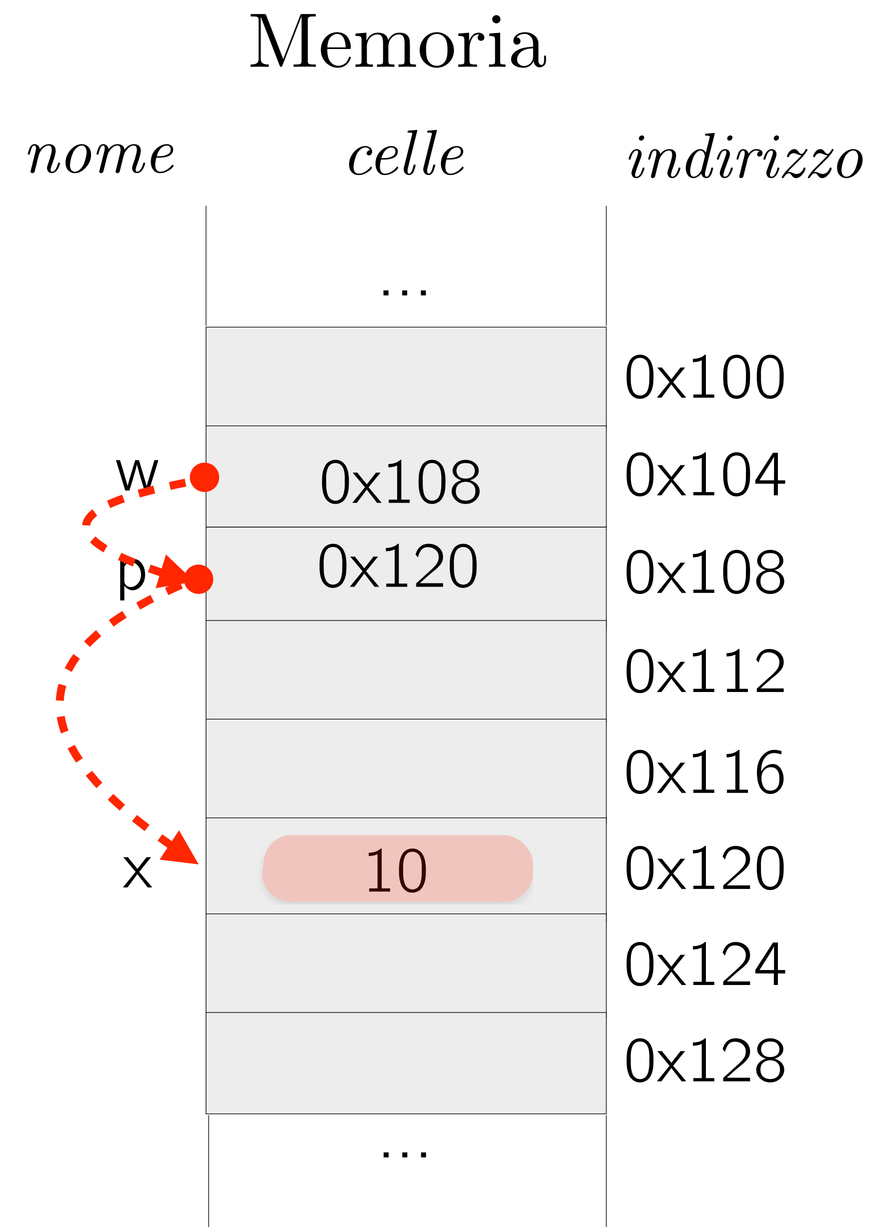
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
*w
**w
```



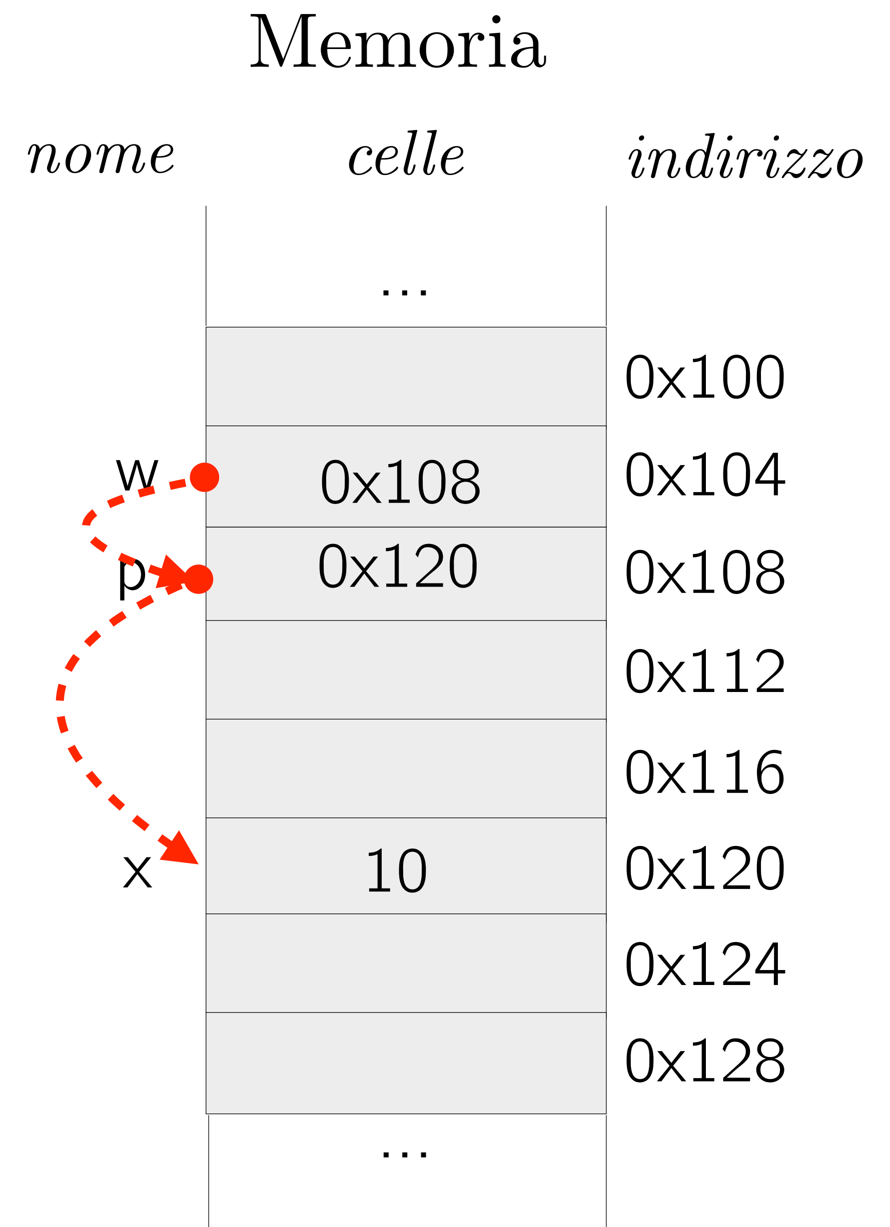
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
*w
**w
```



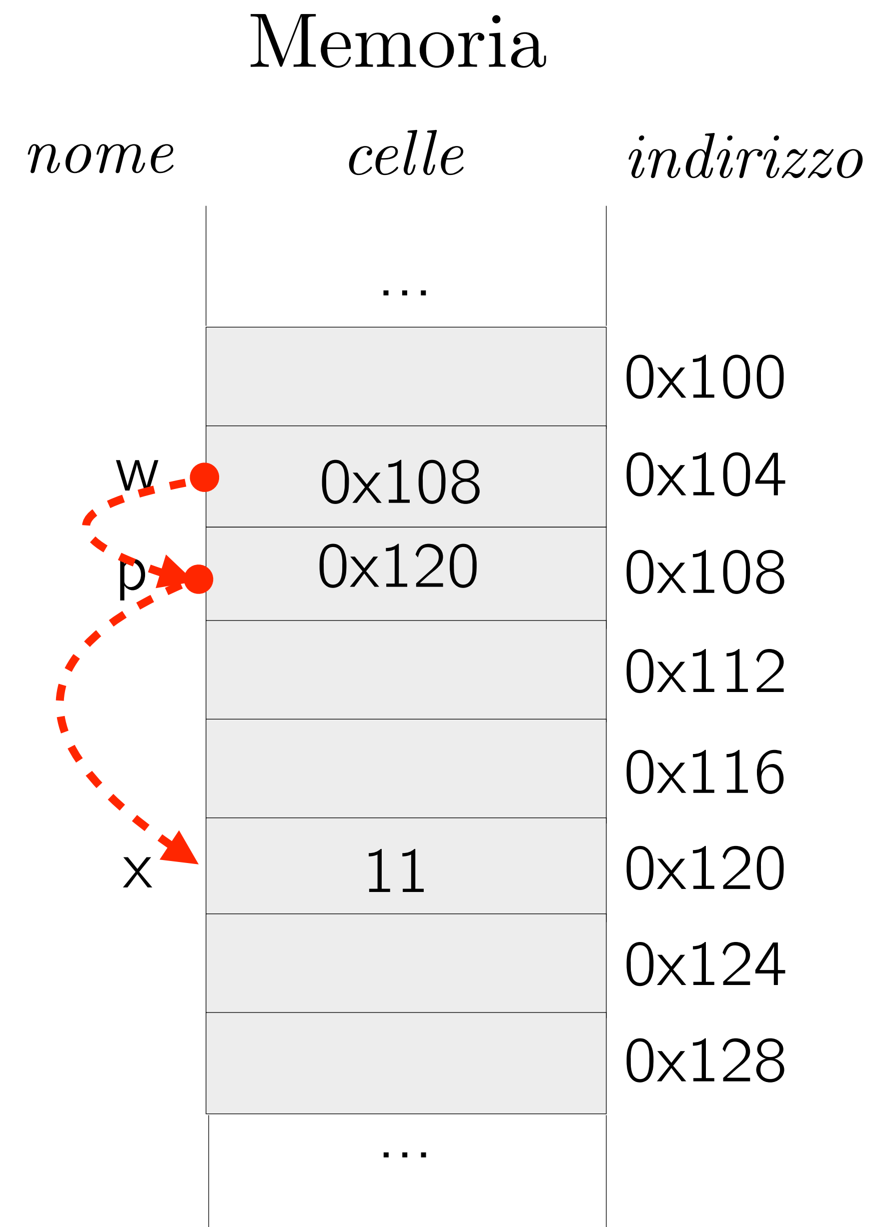
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
*w
**w
**w = 11
```



Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
*w
**w
**w = 11
```



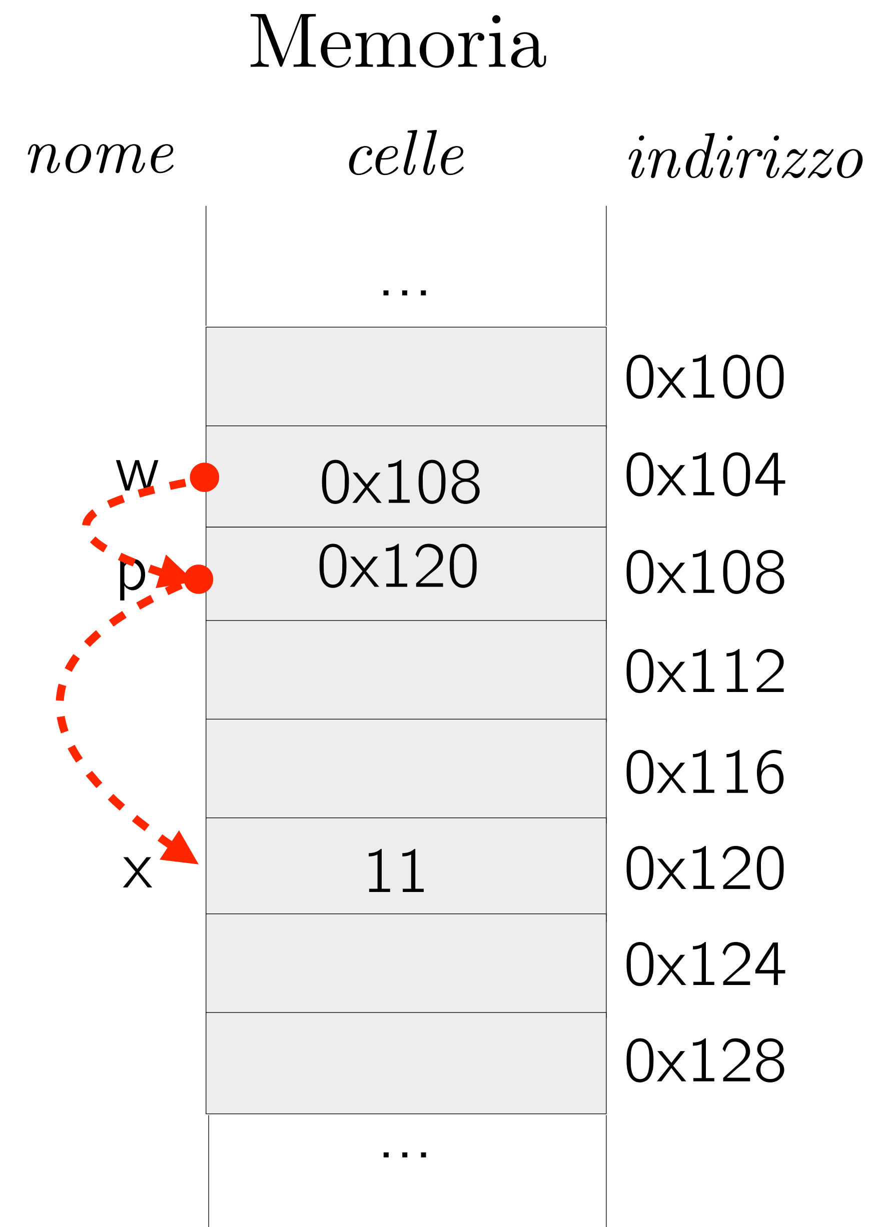
Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
w = &x;
w = p;
w = &p;
*w
**w
**w = 11
int ***z;
```

Puntatore a puntatore ad intero (riferito a `int **w;`)

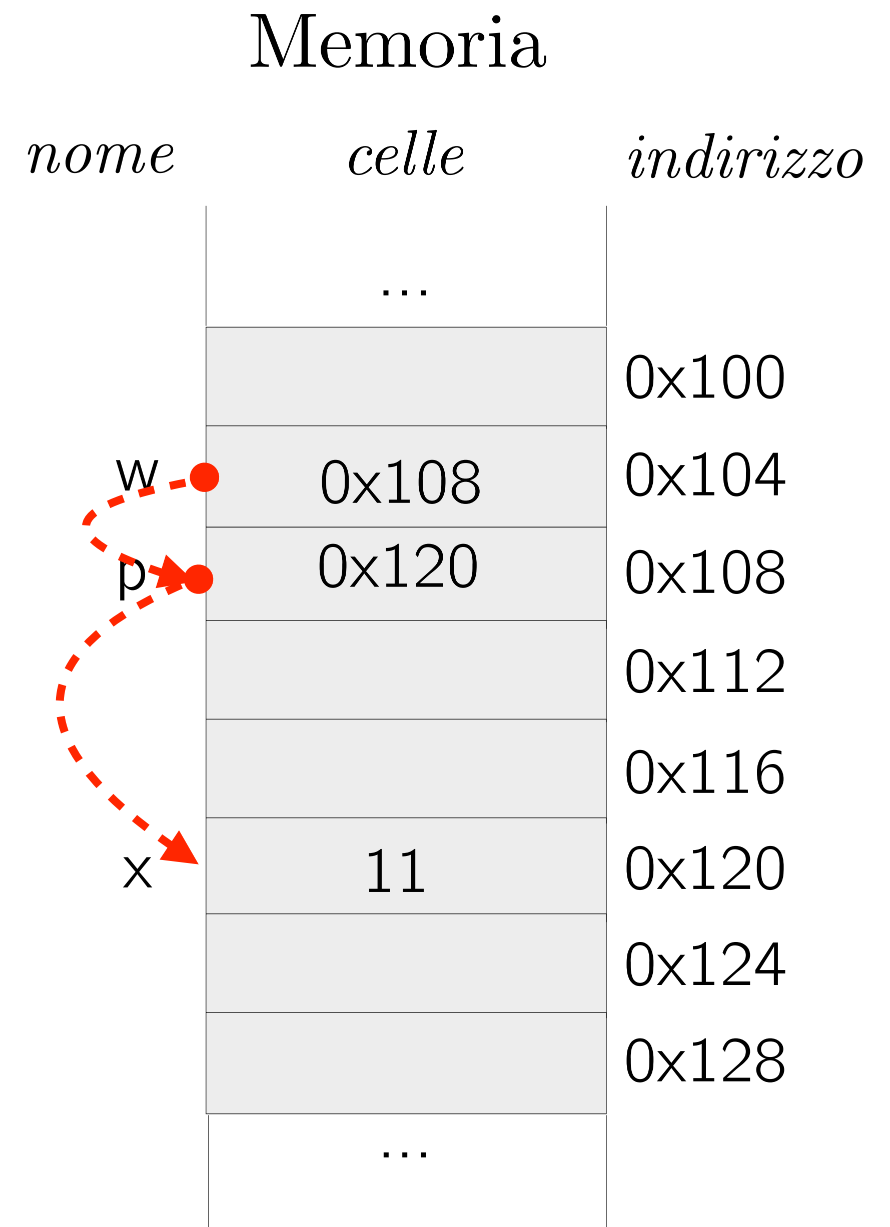
Errore di tipo (riferito a `w = &x;`)

Errore di tipo (riferito a `w = p;`)



Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;   Puntatore a puntatore ad intero
w = &x;    Errore di tipo
w = p;     Errore di tipo
w = &p;
*w
**w
**w = 11
int ***z;  Puntatore a puntatore
           a puntatore ad intero
```



Puntatori (4)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w;
w = &x;
w = p;
w = &p;
*w
**w
**w = 11
int ***z;
```

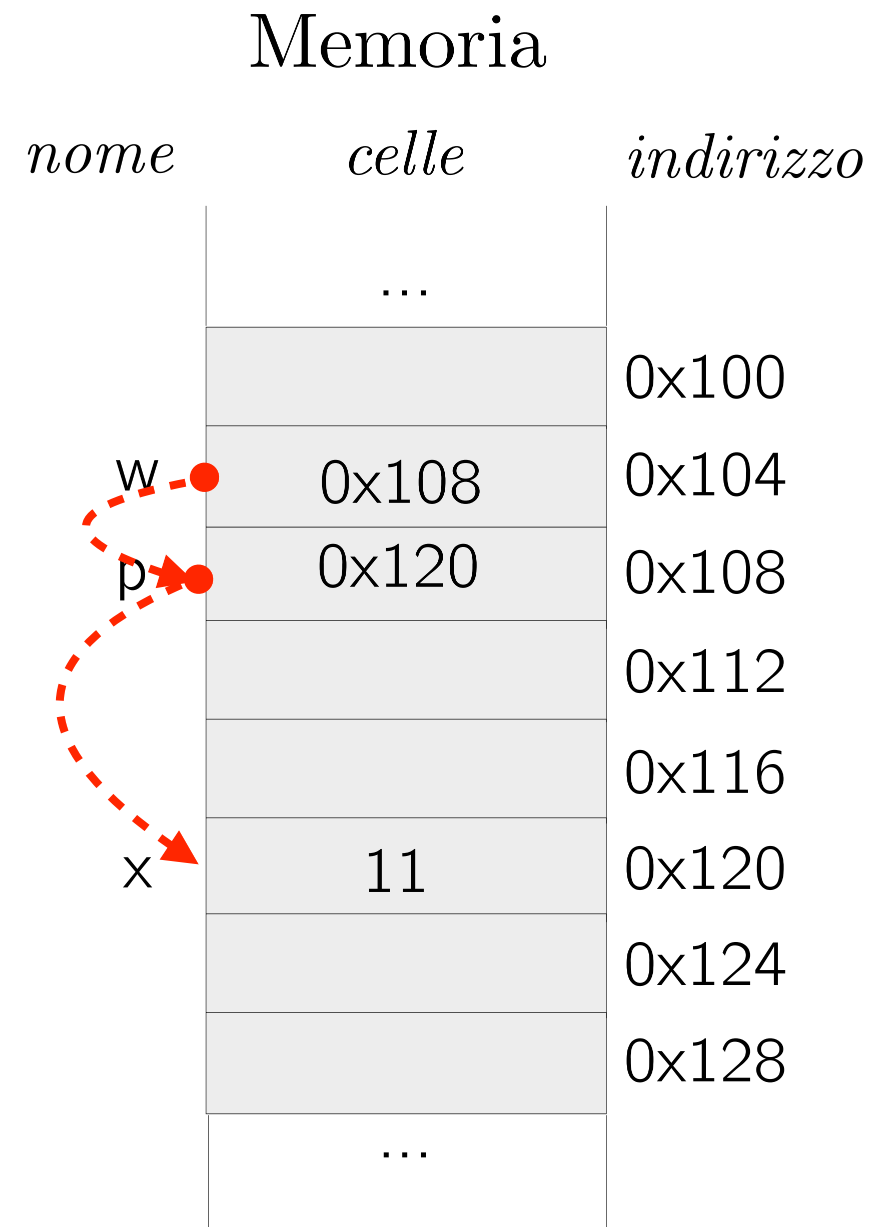
int **w; Puntatore a puntatore ad intero

w = &x; Errore di tipo

w = p; Errore di tipo

int ***z; Puntatore a puntatore a puntatore ad intero

Se dovete usarlo in un vostro programma, c'è un problema nel programma



Funzioni (5)

Ogni funzione è dotata di un *ambiente locale* che memorizza variabili locali (e parametri)

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dot… ariabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dall'esterno
- allocate/deallocate automaticamente all'ingresso/uscita dalla funzione

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	…	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	…	

Funzioni (5)

Ogni funzione è dot ... ariabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili da
- allocate/deall ... lla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri è *sempre* per valore.

La funzione riceve una copia del parametro passato. Quindi, eventuali modifiche *non* si propagano al chiamante.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dot… …ariabili locali (e parametri)

- *non* visibili da…
- allocate/deall… …lla funzione

Il passaggio dei pa…

La funzione riceve… …ntuali modifiche *non* si pro…

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(x, y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotat[…] variabili locali (e parametri)

- *non* visibili da[…]
- allocate/deall[…]lla funzione

Il passaggio dei pa[…]

La funzione riceve[…]ntuali modifiche *non* si pro[…]

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dot[...] variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili da[...]
- allocate/deall[...] lla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei pa[...]

La funzione riceve[...] ntuali modifiche *non* si pro[...]

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotat[...] variabili locali (e parametri)

- *non* visibili da[...]
- allocate/deall[...] lla funzione

Il passaggio dei pa[...]

La funzione riceve [...] ntuali modifiche *non* si pro[...]

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotat[...] variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili da[...]
- allocate/deall[...] lla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei pa[...]

La funzione riceve[...] ntuali modifiche *non* si pro[...]

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotat[...] variabili locali (e parametri)

- *non* visibili da[...]
- allocate/deall[...]lla funzione

Il passaggio dei pa[...]

La funzione riceve[...]ntuali modifiche *non* si pro[...]

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
x	0x120	0x100
y	0x124	0x104
		0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Ambiente locale di scambia()

Funzioni (5)

Ogni funzione è dot[...] variabili locali (e parametri)

- *non* visibili da[...]
- allocate/deall[...]lla funzione

Come può una funzione propagare le modifiche al chiamante?

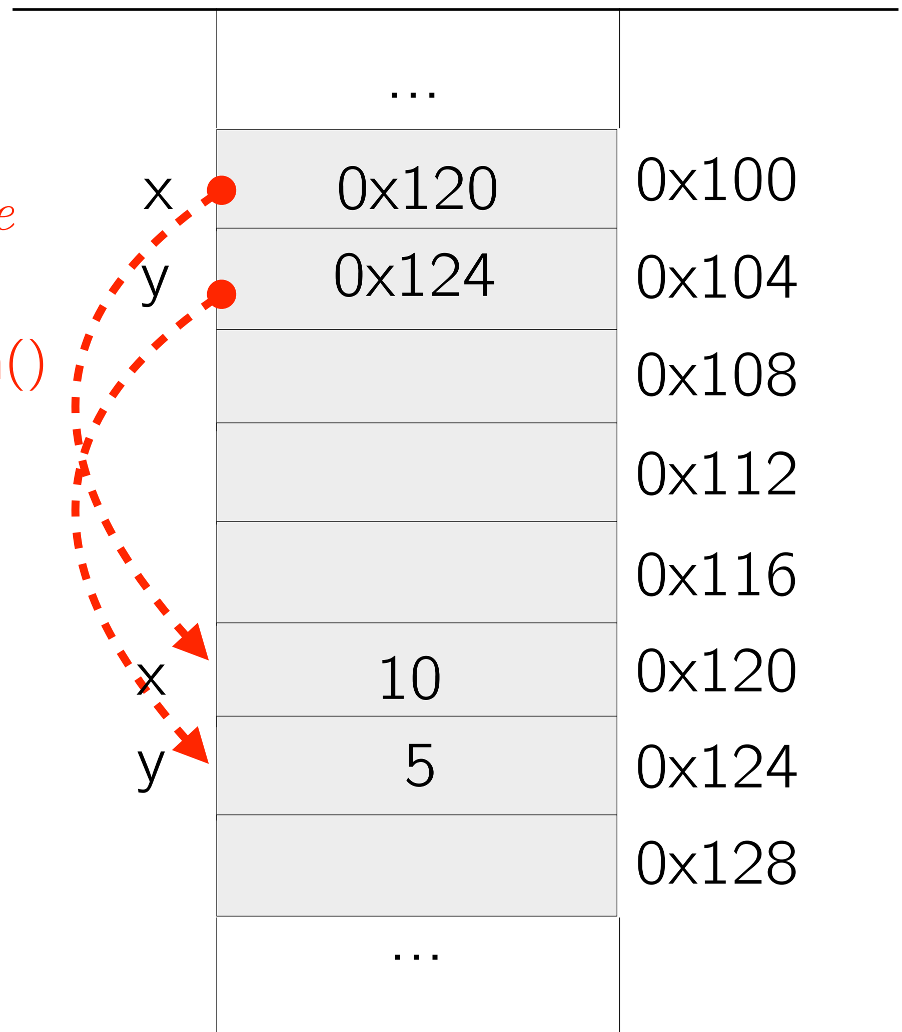
Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

Il passaggio dei pa[...]

La funzione riceve[...]ntuali modifiche *non* si pro[...]

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```



Funzioni (5)

Ogni funzione è dotat... variabili locali (e parametri)

- *non* visibili da...
- allocate/deall... lla funzione

Il passaggio dei pa...

La funzione riceve ... ntuali modifiche *non* si pro...

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
      int x = 10, y = 5;
      scambia(&x, &y);
      printf("x=%d y=%d", x, y);
      return 0;
}
```

nome	celle	indirizzo
	...	
x	0x120	0x100
y	0x124	0x104
tmp	10	0x108
		0x112
		0x116
x	10	0x120
y	5	0x124
		0x128
	...	

Ambiente locale di scambia()

Funzioni (5)

Ogni funzione è dotat… variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili da…
- allocate/deall… …lla funzione

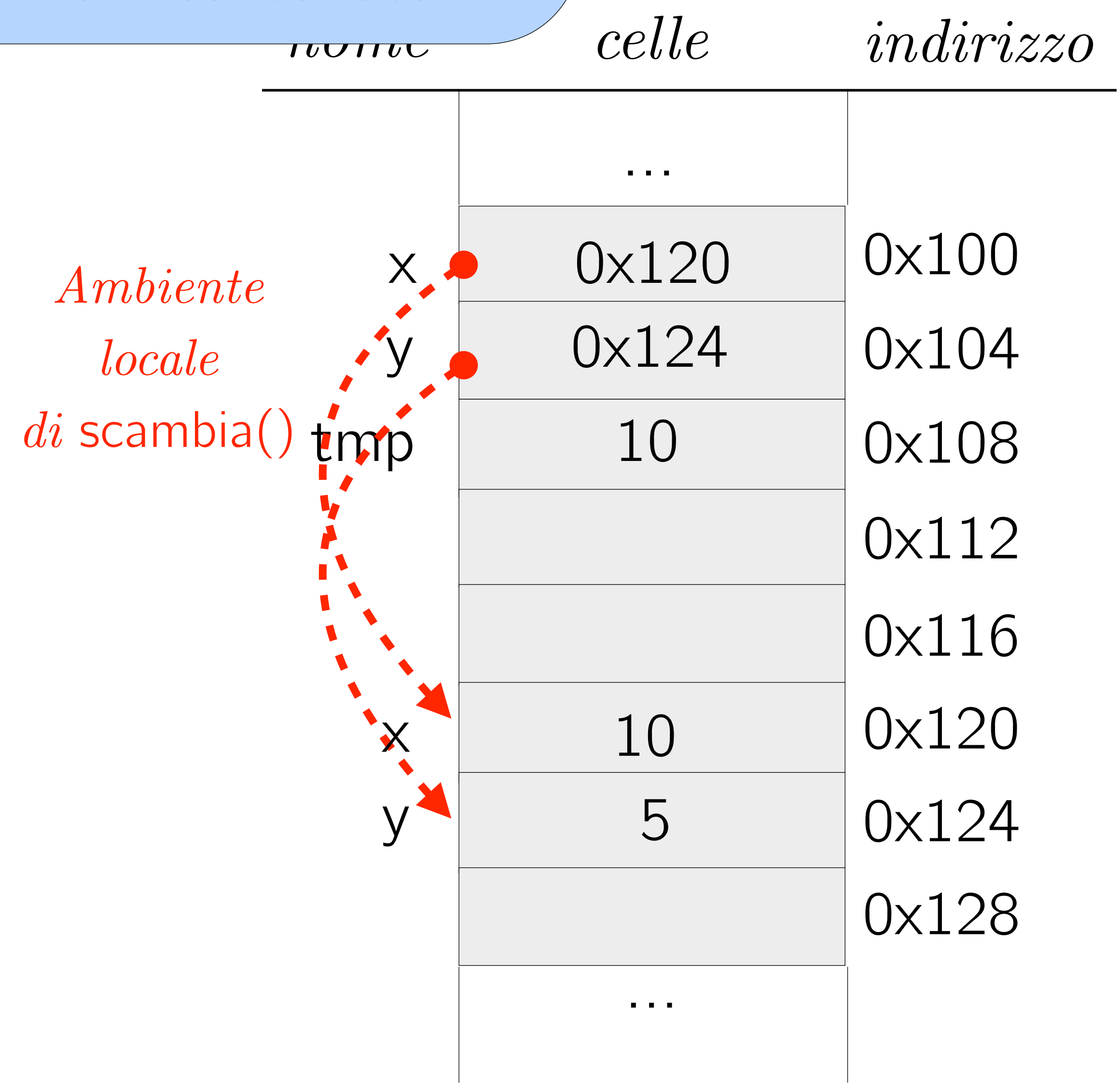
Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei pa…

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

La funzione riceve… …ntuali modifiche *non* si pro…

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

int main () {
      int x = 10, y = 5;
      scambia(&x, &y);
      printf("x=%d y=%d", x, y);
      return 0;
}
```



Funzioni (5)

Ogni funzione è dotat[…] variabili locali (e parametri)

- *non* visibili da[…]
- allocate/deall[…]lla funzione

Il passaggio dei pa[…]

La funzione riceve[…]ntuali modifiche *non* si pro[…]

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Ambiente locale di scambia()

nome	celle	indirizzo
	...	
x	0x120	0x100
y	0x124	0x104
tmp	10	0x108
		0x112
		0x116
x	5	0x120
y	10	0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotat[...] variabili locali (e parametri)

- *non* visibili da[...]
- allocate/deall[...]lla funzione

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei pa[...]

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

La funzione riceve[...]ntuali modifiche *non* si pro[...]

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

nome	celle	indirizzo
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	5	0x120
y	10	0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dot... variabili locali (e parametri)

- *non* visibili da...
- allocate/deall... lla funzione

Il passaggio dei pa...

La funzione riceve ... ntuali modifiche *non* si pro...

Come può una funzione propagare le modifiche al chiamante?

Simulando il passaggio per riferimento con l'uso dei puntatori

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamato può modificarne il contenuto.

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

int main () {
    int x = 10, y = 5;
    scambia(&x, &y);
    printf("x=%d y=%d", x, y);
    return 0;
}
```

Capito perché si usa scanf("%d", &x)?

nome	*celle*	*indirizzo*
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x	5	0x120
y	10	0x124
		0x128
	...	

Esercizio 1

Somma dispari

Esercizio

Scrivere una funzione ricorsiva f che, dato un intero N, restituisca la somma dei primi N interi dispari. Scrivere un programma che prenda in input un intero x e stampi il valore di $f(x)$.
L'unica riga dell'input contiene il valore x.
L'unica riga dell'output contiene la somma dei primi x numeri dispari.

Esempio

Input	**Output**
6	36

Esercizio 2

Scambia

Esercizio

Implementare la funzione `swap(int *a, int *b)` che scambi il contenuto delle due variabili.

Scrivere poi un programma che legga da input il valore di due variabili intere a e b e utilizzi la funzione `swap` per scambiare i loro valori. Il programma deve quindi stampare il valore di a e b dopo questa operazione.
Le due righe dell'input contengono il valore di a e b.
Le due righe dell'output contengono il valore di a e b dopo lo scambio.

Esempio

Input

```
2
4
```

Output

```
4
2
```

Esercizio 3

Triplo scambia

Esercizio

Implementare una funzione `tswap(int *x, int *y, int *z)` che riceva in input tre variabili e ne scambi i valori in modo che

- x prenda il valore di z;
- y prenda il valore di x;
- z prenda il valore di y.

Leggere da input un array di 3 interi e invocare la funzione passando gli indirizzi delle 3 celle in ordine.

Scrivere poi un programma che legga da input il valore di tre variabili intere a, b e c e utilizzi la funzione `tswap` per scambiare i loro valori.
Le tre righe dell'input contengono il valore di a, b e c .
Le tre righe dell'output contengono il valore di a, b e c dopo lo scambio.

Esempio

Input

```
2
4
1
```

Output

```
1
2
4
```

Esercizio 4

MinMax

Esercizio

Scrivere una funzione `minmax` avente i seguenti parametri

- un array di interi;
- la lunghezza dell'array;
- un puntatore a una variabile intera `min`;
- un puntatore a una variabile intera `max`.

La funzione scandisce l'array e salva in `min` la posizione in cui si trova l'elemento minimo e in `max` la posizione in cui si trova l'elemento massimo. Si può assumere che l'array contenga valori distinti.

Scrivere poi un programma che

- legga 10 interi da tastiera;
- invochi `minmax` sull'array letto;
- produca in output: la posizione dell'elemento minimo, il valore dell'elemento minimo, la posizione dell'elemento massimo, il valore dell'elemento massimo.