

Introduzione al C

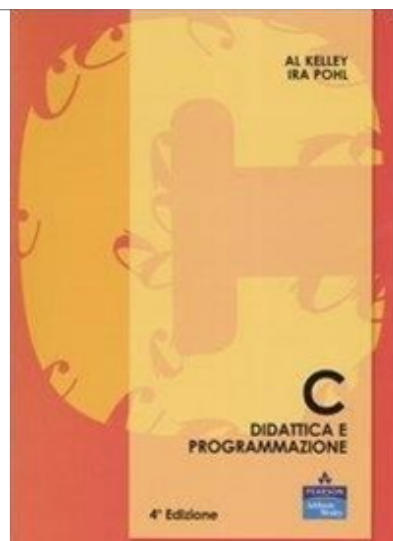
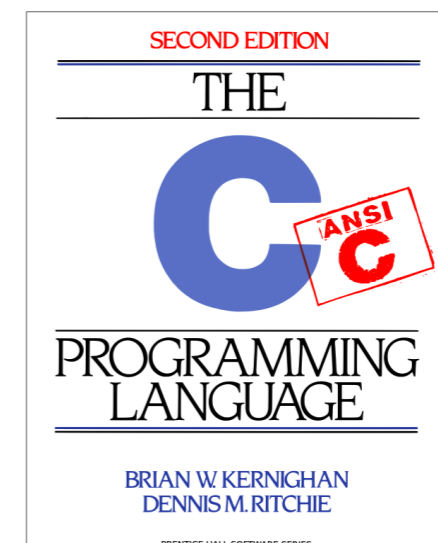
Parte 1 Elementi

Rossano Venturini

rossano.venturini@unipi.it

Pagina web del corso

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



Esame

Esame

Scritto Algoritmica

Esame

Scritto Algoritmica



voto \geq 18

Prova Laboratorio

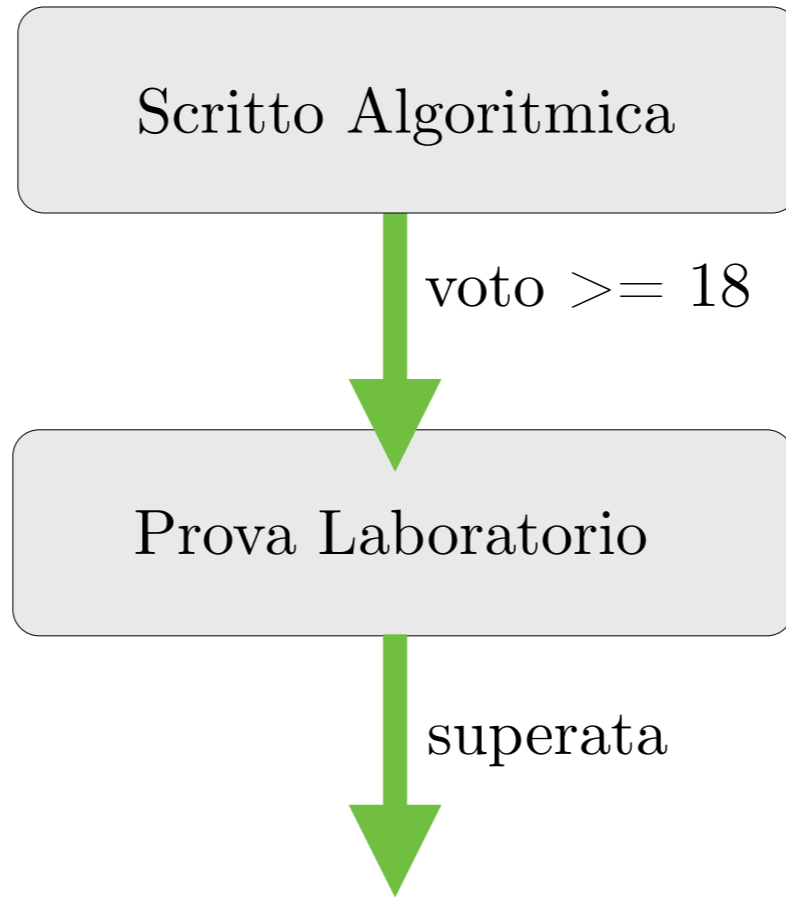
Esame

Scritto Algoritmica

voto \geq 18

Prova Laboratorio

superata



Esame

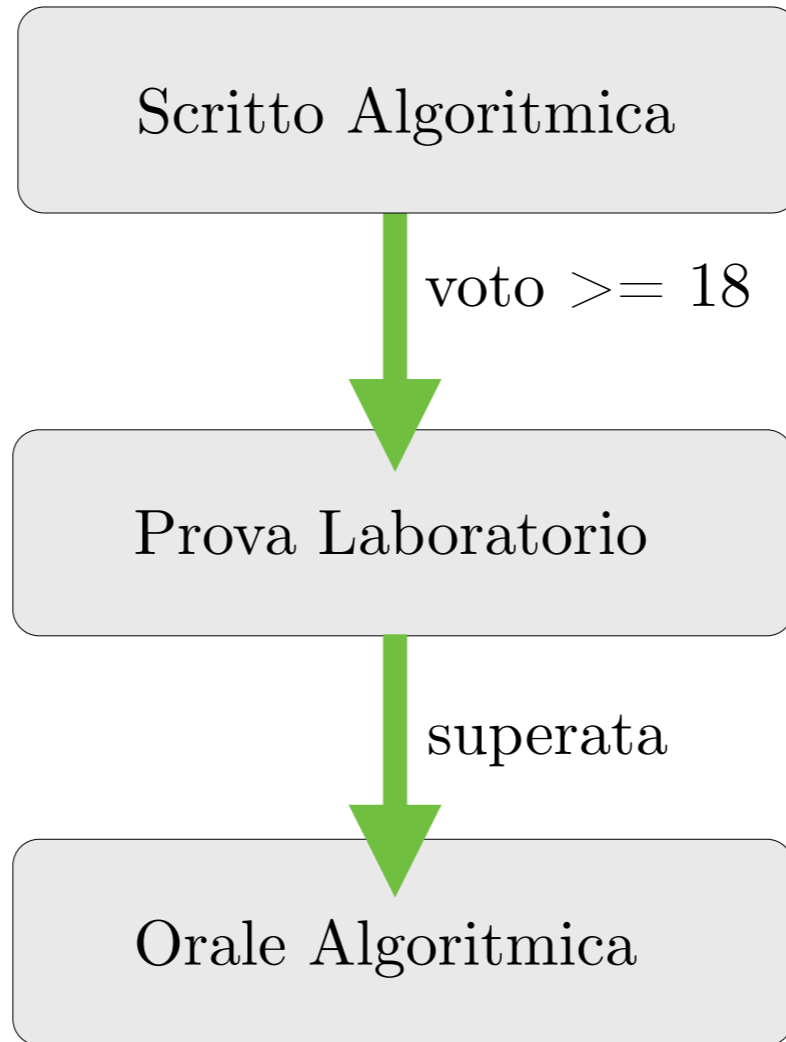
Scritto Algoritmica

voto \geq 18

Prova Laboratorio

superata

Orale Algoritmica



Esame

Scritto Algoritmica

voto \geq 18

Prova Laboratorio

superata

Orale Algoritmica

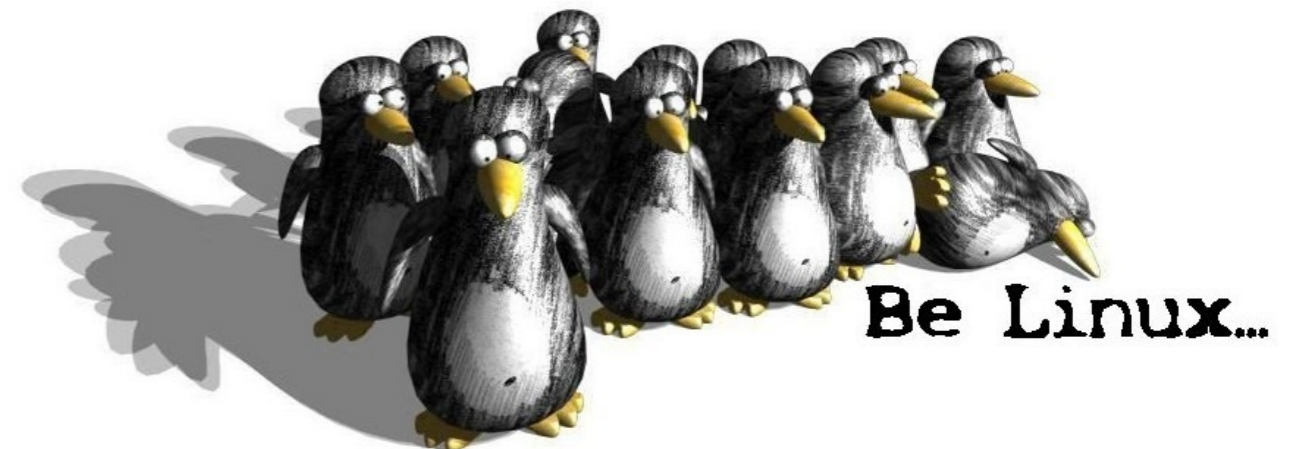


Introduzione al C

- Strumenti che utilizzeremo nelle esercitazioni
- **Editare sorgenti:** editor di testo generico (ad esempio, gedit per Linux)
- **Compilare:** gcc per Linux, o tool di pubblico dominio per Windows (vedere la pagina Web del corso)

Introduzione al C

- Strumenti che utilizzeremo nelle esercitazioni
- **Editare sorgenti:** editor di testo generico (ad esempio, gedit per Linux)
- **Compilare:** gcc per Linux, o tool di pubblico dominio per Windows (vedere la pagina Web del corso)



Struttura di un programma C

Direttive al preprocessore

```
#include <*.h>  
#define MAX (100)  
#include <stdio.h> // generalmente necessaria
```

Dichiarazioni di variabili globali

```
char pippo;  
...  
int pluto;
```

Definizioni/dichiarazioni di funzioni

```
void foo();  
  
int main () {  
    /* esecuzione inizia da qui */  
    ...  
    return 0; // necessario! Valore diverso da 0 indica un errore.  
}
```

Scheletro di un programma C

```
#include <stdio.h>

int main () {
    /* esecuzione inizia da qui */

    // corpo della funzione
    // scrivi qui il tuo codice
    // e spera per il meglio!

    return 0;
}
```

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria `printf()` (definita in `stdio.h`) per stampare una stringa.

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria `printf()` (definita in `stdio.h`) per stampare una stringa.

```
$ gedit hw.c &
```

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

```
#include <stdio.h>

int main () {
    printf("Hello World!\n");
    return 0;
}
```

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

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

```
int main () {  
    printf("Hello World!\n");  
    return 0;  
}
```

Le costanti stringa in C sono sempre specificate tra una coppia di apici

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

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

```
int main () {  
    printf("Hello World!\n");  
    return 0;  
}
```

Le costanti stringa in C sono sempre specificate tra una coppia di apici

`\n` è la *sequenza di escape* che codifica il carattere di fine linea

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

```
#include <stdio.h>

int main () {
    printf("Hello World!\n");
    return 0;
}
```

Compilazione ed esecuzione

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

```
#include <stdio.h>

int main () {
    printf("Hello World!\n");
    return 0;
}
```

Compilazione ed esecuzione

```
$ gcc -o hw hw.c
```

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

```
#include <stdio.h>

int main () {
    printf("Hello World!\n");
    return 0;
}
```

Compilazione ed esecuzione

```
$ gcc -o hw hw.c
```

```
$ ./hw
```

Esempio: hw.c

Aggiungiamo una chiamata alla funzione di libreria printf() (definita in stdio.h) per stampare una stringa.

```
$ gedit hw.c &
```

```
#include <stdio.h>

int main () {
    printf("Hello World!\n");
    return 0;
}
```

Compilazione ed esecuzione

```
$ gcc -o hw hw.c
```

```
$ ./hw
```

```
Hello World!
```

```
$
```

Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

```
int x;  
int y = 0;  
char pippo, pluto;
```

Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

```
int x;  
int y = 0;  
char pippo, pluto;
```

Il contenuto di una variabile si può inizializzare o modificare mediante *assegnamenti* e *pre/post-incrementi*.

Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

```
int x;  
int y = 0;  
char pippo, pluto;
```

Il contenuto di una variabile si può inizializzare o modificare mediante *assegnamenti* e *pre/post-incrementi*.

```
x = 0;                      (forme equivalenti)
```


Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

```
int x;  
int y = 0;  
char pippo, pluto;
```

Il contenuto di una variabile si può inizializzare o modificare mediante *assegnamenti* e *pre/post-incrementi*.

```
x = 0;                (forme equivalenti)  
x = x + 1;           ++x; oppure x++;
```

Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

```
int x;  
int y = 0;  
char pippo, pluto;
```

Il contenuto di una variabile si può inizializzare o modificare mediante *assegnamenti* e *pre/post-incrementi*.

$x = 0;$	(forme equivalenti)
$x = x + 1;$	$++x;$ oppure $x++;$
$x = y + x;$	$x = x + y;$ oppure $x += y;$

Dichiarazione e assegnamento

Una dichiarazione introduce il nome di una variabile e specifica il tipo di dato che essa conterrà.

```
int x;  
int y = 0;  
char pippo, pluto;
```

Il contenuto di una variabile si può inizializzare o modificare mediante *assegnamenti* e *pre/post-incrementi*.

$x = 0;$	(forme equivalenti)
$x = x + 1;$	$++x;$ oppure $x++;$
$x = y + x;$	$x = x + y;$ oppure $x += y;$
$x = x * 3;$	$x *= 3;$

Tipi di dato primitivi

Tutti i tipi primitivi in C sono numerici e gli operatori standard (+, -, *, /, %) sono definiti su di essi.

Interi: char, short, int, long

Si differenziano in base alla loro occupazione e, quindi, al range di valori rappresentabili.

char	1 byte	usato per rappresentare caratteri ASCII
short	2 byte	
int	4 byte	
long	8 byte	

Il modificatore unsigned restringe i valori rappresentabili ai soli positivi.

Floating-point: float, double

Si differenziano in base alla loro occupazione e, quindi, alla precisione della rappresentazione.

float	4 byte
double	8 byte

Costrutto condizionale: if-else

Costrutto condizionale: if-else

```
if ( guardia ) {  
    // blocco 1  
} else { // opzionale  
    // blocco 2  
}
```

Costrutto condizionale: if-else

```
if ( guardia ) {  
    // blocco 1  
} else { // opzionale  
    // blocco 2  
}
```

Indentate correttamente il vostro codice!

Costrutto condizionale: if-else

```
if ( guardia ) {  
    // blocco 1  
} else { // opzionale  
    // blocco 2  
}
```

Esempio

```
if ( voto >= 18 ) {  
    printf("Promosso!");  
} else {  
    printf("Chi è il prossimo?");  
}
```


Operatori logici e di confronto

Le guardie sono ottenute tipicamente combinando espressioni attraverso operatori logici e di confronto.

Operatori logici e di confronto

Le guardie sono ottenute tipicamente combinando espressioni attraverso operatori logici e di confronto.

Operatori logici

&&	And	!	Not
	Or	^	Xor

Operatori logici e di confronto

Le guardie sono ottenute tipicamente combinando espressioni attraverso operatori logici e di confronto.

Operatori logici

&&	And	!	Not
	Or	^	Xor

Operatori di confronto

<	Minore	<=	Minore o uguale
>	Maggiore	>=	Maggiore o uguale
==	Uguaglianza	!=	Diverso da

Operatori logici e di confronto

Le guardie sono ottenute tipicamente combinando espressioni attraverso operatori logici e di confronto.

Operatori logici

&&	And	!	Not
	Or	^	Xor

Operatori di confronto

<	Minore	<=	Minore o uguale
>	Maggiore	>=	Maggiore o uguale
==	Uguaglianza	!=	Diverso da

Attenzione a non confondere

`x==y` (confronto) e

`x=y` (assegnamento)

Operatori logici e di confronto

Le guardie sono ottenute tipicamente combinando espressioni attraverso operatori logici e di confronto.

Operatori logici

&&	And	!	Not
	Or	^	Xor

Operatori di confronto

<	Minore	<=	Minore o uguale
>	Maggiore	>=	Maggiore o uguale
==	Uguaglianza	!=	Diverso da

Esempio di guardia complessa

```
if ( anno%400 == 0 ||  
    ( anno%100 != 0 && anno%4 == 0 ) ) {  
    printf("Bisestile");  
}
```

Booleani in C

In C non esiste un tipo primitivo per i booleani.
Questi vengono codificati con interi.

Booleani in C

In C non esiste un tipo primitivo per i booleani.
Questi vengono codificati con interi.

Un qualsiasi valore è interpretato come

Falso se == 0

Vero se != 0

Booleani in C

In C non esiste un tipo primitivo per i booleani.
Questi vengono codificati con interi.

Un qualsiasi valore è interpretato come

Falso	se == 0
Vero	se != 0

Gli operatori logici e di confronto sono a tutti gli effetti operatori aritmetici che restituiscono

0	se Falsi
1	se Veri

Booleani in C

In C non esiste un tipo primitivo per i booleani.
Questi vengono codificati con interi.

Un qualsiasi valore è interpretato come

Falso	se == 0
Vero	se != 0

Gli operatori logici e di confronto sono a tutti gli effetti operatori aritmetici che restituiscono

0	se Falsi
1	se Veri

Esempio

```
int x =67947;  
if ( x ) {  
    printf("x diverso da 0");  
}
```

Costrutto iterativo: while

Costrutto iterativo: while

```
while ( guardia ) {  
    // corpo del while  
}
```

Costrutto iterativo: while

```
while ( guardia ) {  
    // corpo del while  
}
```

Esempio

```
int counter = 0;  
while ( counter < 1000 ) {  
    printf("Una corretta indentazione favorisce la leggibilità del codice");  
    counter++;  
}
```

Costrutto iterativo: for

Costrutto iterativo: for

```
for( inizializzazione; guardia; incremento ) {  
    // corpo del for  
}
```

Costrutto iterativo: for

```
for( inizializzazione; guardia; incremento ) {  
    // corpo del for  
}
```

Esempio equivalente al precedente

```
int counter;  
for ( counter = 0; counter < 1000; counter++ ) {  
    printf("Una corretta indentazione favorisce la leggibilità del codice");  
}
```

Costrutto iterativo: for

```
for( inizializzazione; guardia; incremento ) {  
    // corpo del for  
}
```

Esempio equivalente al precedente

```
int counter;  
for ( counter = 0; counter < 1000; counter++ ) {  
    printf("Una corretta indentazione favorisce la leggibilità del codice");  
}
```

Il for può essere sempre riscritto come

```
inizializzazione;  
while( guardia ) {  
    // corpo del for  
    incremento;  
}
```


Array (1)

Ad alto livello, un array è una *collezione di oggetti dello stesso tipo*, raccolti sotto un unico nome e identificati da un indice intero compreso tra 0 e $n-1$, con n dimensione dell'array.

Array (1)

Ad alto livello, un array è una *collezione di oggetti dello stesso tipo*, raccolti sotto un unico nome e identificati da un indice intero compreso tra 0 e n-1, con n dimensione dell'array.

La sintassi per dichiarare array di *dimensione costante* in C è

tipo nome-array [dimensione]

Array (1)

Ad alto livello, un array è una *collezione di oggetti dello stesso tipo*, raccolti sotto un unico nome e identificati da un indice intero compreso tra 0 e n-1, con n dimensione dell'array.

La sintassi per dichiarare array di *dimensione costante* in C è

tipo nome-array [dimensione]

Dimensione è costante (ad es. 10, 100, 34526 ecc.) non una variabile (ad es. n, m, k, pippo ecc.).

Array (1)

Ad alto livello, un array è una *collezione di oggetti dello stesso tipo*, raccolti sotto un unico nome e identificati da un indice intero compreso tra 0 e n-1, con n dimensione dell'array.

La sintassi per dichiarare array di *dimensione costante* in C è

tipo nome-array [dimensione]

Alcuni esempi

```
int a[10];
```

```
char s[24];
```

```
int b[5] = {55,3,77,14,22}; // dichiara array e lo inizializza
```

Array (1)

Ad alto livello, un array è una *collezione di oggetti dello stesso tipo*, raccolti sotto un unico nome e identificati da un indice intero compreso tra 0 e n-1, con n dimensione dell'array.

La sintassi per dichiarare array di *dimensione costante* in C è

tipo nome-array [dimensione]

Alcuni esempi

```
int a[10];  
char s[24];  
int b[5] = {55,3,77,14,22}; // dichiara array e lo inizializza
```

Cosa non fare

```
int n = 20;  
char s[n];
```

Array (2)

Ovviamente è possibile accedere/modificare il valore di un qualunque elemento di un array specificando il suo indice.

Array (2)

Ovviamente è possibile accedere/modificare il valore di un qualunque elemento di un array specificando il suo indice.

Alcuni esempi

```
int b[5] = {55,3,77,14,22};
```

```
b[0] = 6;
```

```
int x = b[4] + 1;
```

Array (2)

Ovviamente è possibile accedere/modificare il valore di un qualunque elemento di un array specificando il suo indice.

Alcuni esempi

```
int b[5] = {55,3,77,14,22};  
b[0] = 6;  
int x = b[4] + 1;
```

Cosa non fare

```
int b[5] = {55,3,77,14,22};  
b[5] = 4; // accesso out-of-bound
```


Array (2)

Ovviamente è possibile accedere/modificare il valore di un qualunque elemento di un array specificando il suo indice.

Alcuni esempi

```
int b[5] = {55,3,77,14,22};  
b[0] = 6;  
int x = b[4] + 1;
```

Cosa non fare

```
int b[5] = {55,3,77,14,22};  
b[5] = 4; // accesso out-of-bound
```

Nessun errore a tempo di compilazione, segmentation fault o comportamenti indesiderati a tempo di esecuzione.

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

```
#include <stdlib.h>
#include <time.h>

int main() {
    ...
    /* inizializza il generatore di numeri pseudocasuali con seme time(NULL).
       Usate un numero qualunque su Windows. */
    srand(time(NULL));

    int a[10];
    for ( int i = 0; i < 10; i++) {
        a[i] = rand() % 100; // intero tra 0 e 99
        // fai qualcosa con a[i]
    }
    return 0;
}
```

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

```
#include <stdlib.h>
```

```
#include <time.h>
```

```
int main() {
```

```
    ...
```

```
    /* inizializza il generatore di numeri pseudocasuali con seme time(NULL).
```

```
    Usate un numero qualunque su Windows. */
```

```
    srand(time(NULL));
```

```
    int a[10];
```

```
    for ( int i = 0; i < 10; i++) {
```

```
        a[i] = rand() % 100; // intero tra 0 e 99
```

```
        // fai qualcosa con a[i]
```

```
    }
```

```
    return 0;
```

```
}
```

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

```
#include <stdlib.h>
#include <time.h>

int main() {
    ...
    /* inizializza il generatore di numeri pseudocasuali con seme time(NULL).
       Usate un numero qualunque su Windows. */
    srand(time(NULL));

    int a[10];
    for ( int i = 0; i < 10; i++) {
        a[i] = rand() % 100; // intero tra 0 e 99
        // fai qualcosa con a[i]
    }
    return 0;
}
```

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

```
#include <stdlib.h>
#include <time.h>

int main() {
    ...
    /* inizializza il generatore di numeri pseudocasuali con seme time(NULL).
       Usate un numero qualunque su Windows. */
    srand(time(NULL));

    int a[10];
    for ( int i = 0; i < 10; i++) {
        a[i] = rand() % 100; // intero tra 0 e 99
        // fai qualcosa con a[i]
    }
    return 0;
}
```

Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

```
#include <stdlib.h>
#include <time.h>

int main() {
    ...
    /* inizializza il generatore di numeri pseudocasuali con seme time(NULL).
       Usate un numero qualunque su Windows. */
    srand(time(NULL));

    int a[10];
    for ( int i = 0; i < 10; i++) {
        a[i] = rand() % 100; // intero tra 0 e 99
        // fai qualcosa con a[i]
    }
    return 0;
}
```


Array (3)

Tipicamente si usa un ciclo for per scandire gli elementi di un array.

Esempio: inizializzare gli elementi di un array con interi (pseudo-)casuali

```
#include <stdlib.h>
#include <time.h>

int main() {
    ...
    /* inizializza il generatore di numeri pseudocasuali con seme time(NULL).
       Usate un numero qualunque su Windows. */
    srand(time(NULL));

    int a[10];
    for ( int i = 0; i < 10; i++) {
        a[i] = rand() % 100; // intero tra 0 e 99
        // fai qualcosa con a[i]
    }
    return 0;
}
```

Stampare: printf

printf è una funzione di libreria (in `stdio.h`) per stampare testo formattato sullo standard output.

Stampare: printf

printf è una funzione di libreria (in `stdio.h`) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

Stampare: printf

printf è una funzione di libreria (in `stdio.h`) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

printf rimpiazza in `formato-output` ogni place-holder con il corrispondente argomento e stampa il risultato sullo standard output.

Stampare: printf

printf è una funzione di libreria (in stdio.h) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

printf rimpiazza in **formato-output** ogni place-holder con il corrispondente argomento e stampa il risultato sullo standard output.

Esempio

```
int x = 5, y = 10;  
printf("%d + %d = %d\n", x, y, x+y);
```

Stampare: printf

printf è una funzione di libreria (in stdio.h) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

printf rimpiazza in **formato-output** ogni place-holder con il corrispondente argomento e stampa il risultato sullo standard output.

Esempio

```
int x = 5, y = 10;  
printf("%d + %d = %d\n", x, y, x+y);
```

```
5 + 10 = 15
```

Stampare: printf

printf è una funzione di libreria (in `stdio.h`) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

printf rimpiazza in `formato-output` ogni place-holder con il corrispondente argomento e stampa il risultato sullo standard output.

Esempio

```
int x = 5, y = 10;  
printf("%d + %d = %d\n", x, y, x+y);
```

Ogni tipo ha il suo place-holder.

`%d` è quello degli interi.

Stampare: printf

printf è una funzione di libreria (in stdio.h) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

printf rimpiazza in **formato-output** ogni place-holder con il corrispondente argomento e stampa il risultato sullo standard output.

Esempio

\n è il ritorno a capo

```
int x = 5, y = 10;  
printf("%d + %d = %d\n", x, y, x+y);
```

```
5 + 10 = 15
```


Stampare: printf

printf è una funzione di libreria (in stdio.h) per stampare testo formattato sullo standard output.

```
printf("formato-output", lista-argomenti)
```

printf rimpiazza in **formato-output** ogni place-holder con il corrispondente argomento e stampa il risultato sullo standard output.

Alcuni place-holder

%d	int	%c	char
%s	stringa	%p	indirizzo
%f	float	%lf	double

Leggere: scanf

scanf è una funzione di libreria (in `stdio.h`) per leggere valori o testo dallo standard input.

Leggere: scanf

scanf è una funzione di libreria (in `stdio.h`) per leggere valori o testo dallo standard input.

```
scanf("formato-input", lista-argomenti)
```

Leggere: scanf

scanf è una funzione di libreria (in stdio.h) per leggere valori o testo dallo standard input.

```
scanf("formato-input", lista-argomenti)
```

Sintassi simile a printf ma comportamento simmetrico.

Leggere: scanf

scanf è una funzione di libreria (in stdio.h) per leggere valori o testo dallo standard input.

`scanf("formato-input", lista-argomenti)`

Sintassi simile a printf ma comportamento simmetrico.

Esempio

```
int x = 5;  
printf("%d\n", x);  
scanf("%d", &x);  
printf("%d\n", x);
```

Leggere: scanf

scanf è una funzione di libreria (in stdio.h) per leggere valori o testo dallo standard input.

scanf(**“formato-input”, lista-argomenti**)

Sintassi simile a printf ma comportamento simmetrico.

Esempio

```
int x = 5;  
printf(“%d\n”, x);  
scanf(“%d”, &x);  
printf(“%d\n”, x);
```

Richiede l'indirizzo della variabile che conterrà il valore letto (per questo &x).
Più dettagli nella prossima lezione!

Leggere: scanf

scanf è una funzione di libreria (in stdio.h) per leggere valori o testo dallo standard input.

`scanf("formato-input", lista-argomenti)`

Sintassi simile a printf ma comportamento simmetrico.

Esempio

```
int x = 5;  
printf("%d\n", x);  
scanf("%d", &x);  
printf("%d\n", x);
```

Leggere: scanf

scanf è una funzione di libreria (in stdio.h) per leggere valori o testo dallo standard input.

`scanf("formato-input", lista-argomenti)`

Sintassi simile a printf ma comportamento simmetrico.

Esempio

```
int x = 5;  
printf("%d\n", x);  
scanf("%d", &x);  
printf("%d\n", x);
```

5

10

Leggere: scanf

scanf è una funzione di libreria (in stdio.h) per leggere valori o testo dallo standard input.

`scanf("formato-input", lista-argomenti)`

Sintassi simile a printf ma comportamento simmetrico.

Esempio

```
int x = 5;  
printf("%d\n", x);  
scanf("%d", &x);  
printf("%d\n", x);
```

5

10

10

Esempio printf e scanf

Esempio printf e scanf

Area di un cerchio di raggio dato

```
#include <stdio.h>
#define PI (3.1415f)

int main() {
    float r; // raggio del cerchio
    float a; // area del cerchio

    scanf("%f", &r);
    a = PI * r * r;
    printf("%f\n", a);

    return 0;
}
```

Esempio printf e scanf

Area di un cerchio di raggio dato

```
#include <stdio.h>
#define PI (3.1415f)

int main() {
    float r; // raggio del cerchio
    float a; // area del cerchio

    scanf("%f", &r);
    a = PI * r * r;
    printf("%f\n", a);

    return 0;
}
```

Esempio printf e scanf

Area di un cerchio di raggio dato

```
#include <stdio.h>
#define PI (3.1415f)

int main() {
    float r; // raggio del cerchio
    float a; // area del cerchio

    scanf("%f", &r);
    a = PI * r * r;
    printf("%f\n", a);

    return 0;
}
```

Introduzione al C

Parte 2

Funzioni e Puntatori

Rossano Venturini

rossano.venturini@unipi.it

Pagina web del corso

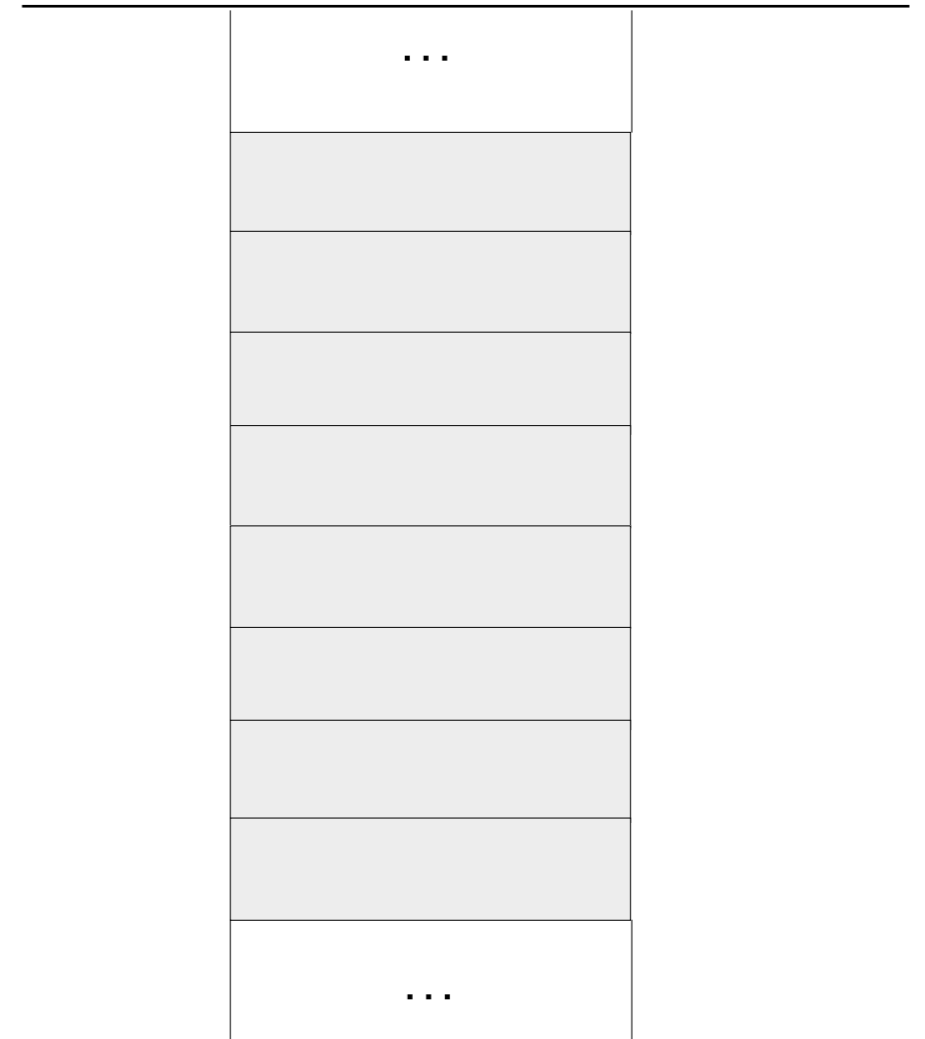
<http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start>

Un primo sguardo dietro le quinte

Un primo sguardo dietro le quinte

Memoria

celle

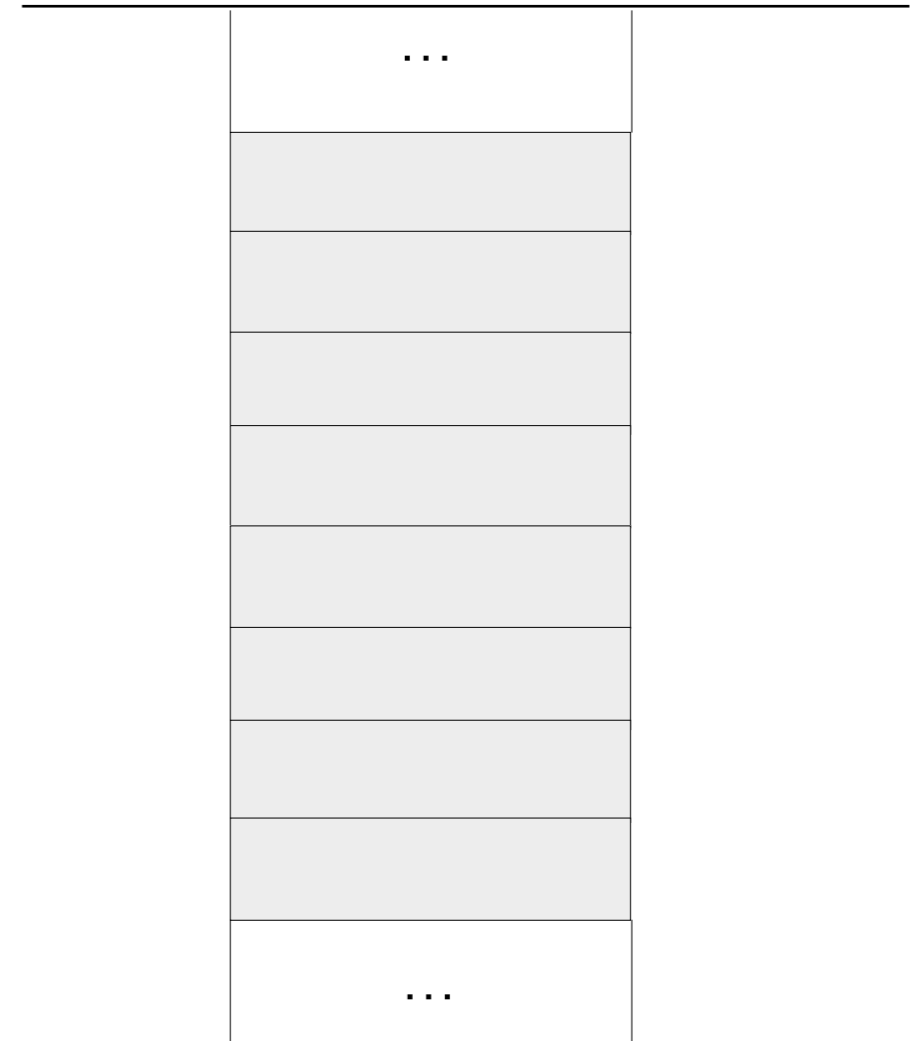


Un primo sguardo dietro le quinte

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

Memoria

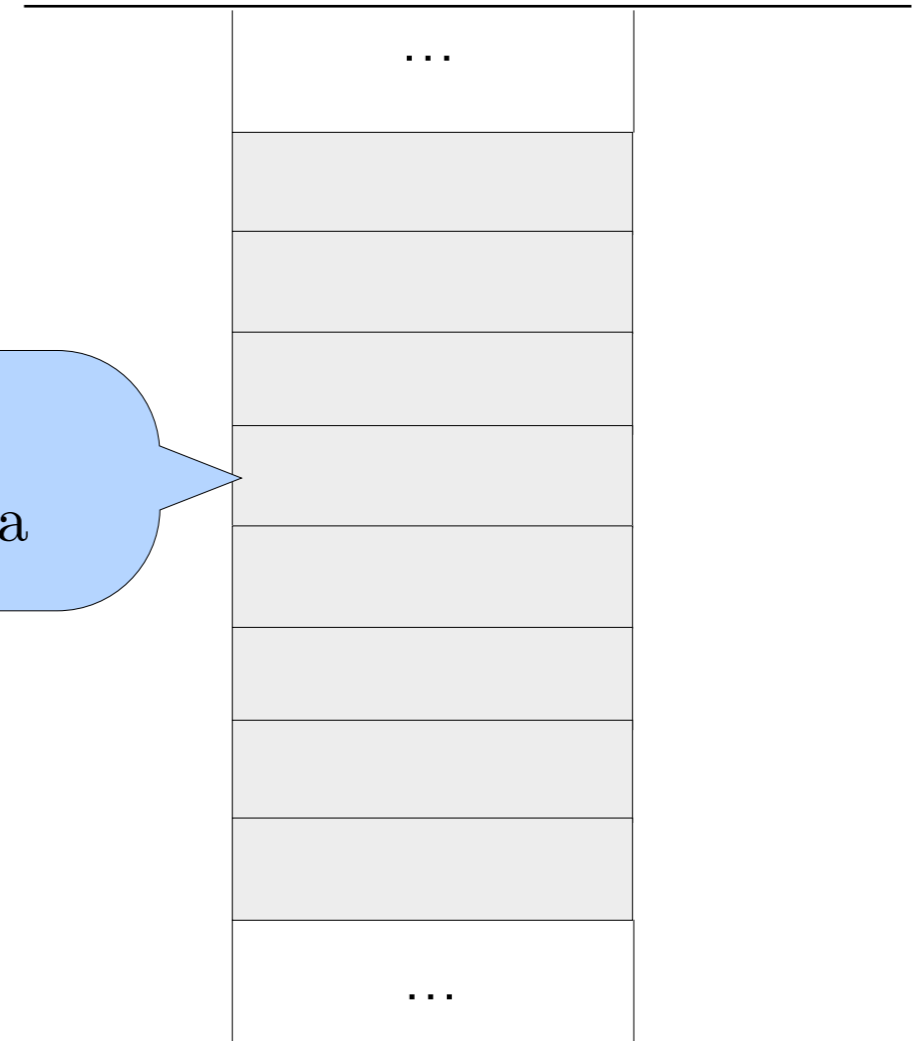
celle



Un primo sguardo dietro le quinte

Memoria

celle



Semplificazione!
da 4 byte ciascuna

Un primo sguardo dietro le quinte

Memoria

<i>celle</i>	<i>indirizzo</i>
...	
	0x100
	0x104
	0x108
	0x112
	0x116
	0x120
	0x124
	0x128
...	

Un primo sguardo dietro le quinte

```
int x;
```

Memoria

<i>celle</i>	<i>indirizzo</i>
...	
	0x100
	0x104
	0x108
	0x112
	0x116
	0x120
	0x124
	0x128
...	

Un primo sguardo dietro le quinte

```
int x;
```

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

```
int x;
```

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
x		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

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

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
y		0x100
		0x104
		0x108
		0x112
x		0x116
		0x120
		0x124
		0x128
	...	

Un primo sguardo dietro le quinte

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

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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 510
    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 510  
    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 510  
    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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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;  
}
```

	<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
		...	
<i>Ambiente locale di scambia()</i>	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;  
}
```

	<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
		...	
<i>Ambiente locale di scambia()</i>	x	10	0x100
	y	5	0x104
			0x108
			0x112
			0x116
			0x120
	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;  
}
```

	<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
		...	
<i>Ambiente locale di scambia()</i>	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;  
}
```

	<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
		...	
<i>Ambiente locale di scambia()</i>	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;  
}
```

	<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
		...	
<i>Ambiente locale di scambia()</i>	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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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.

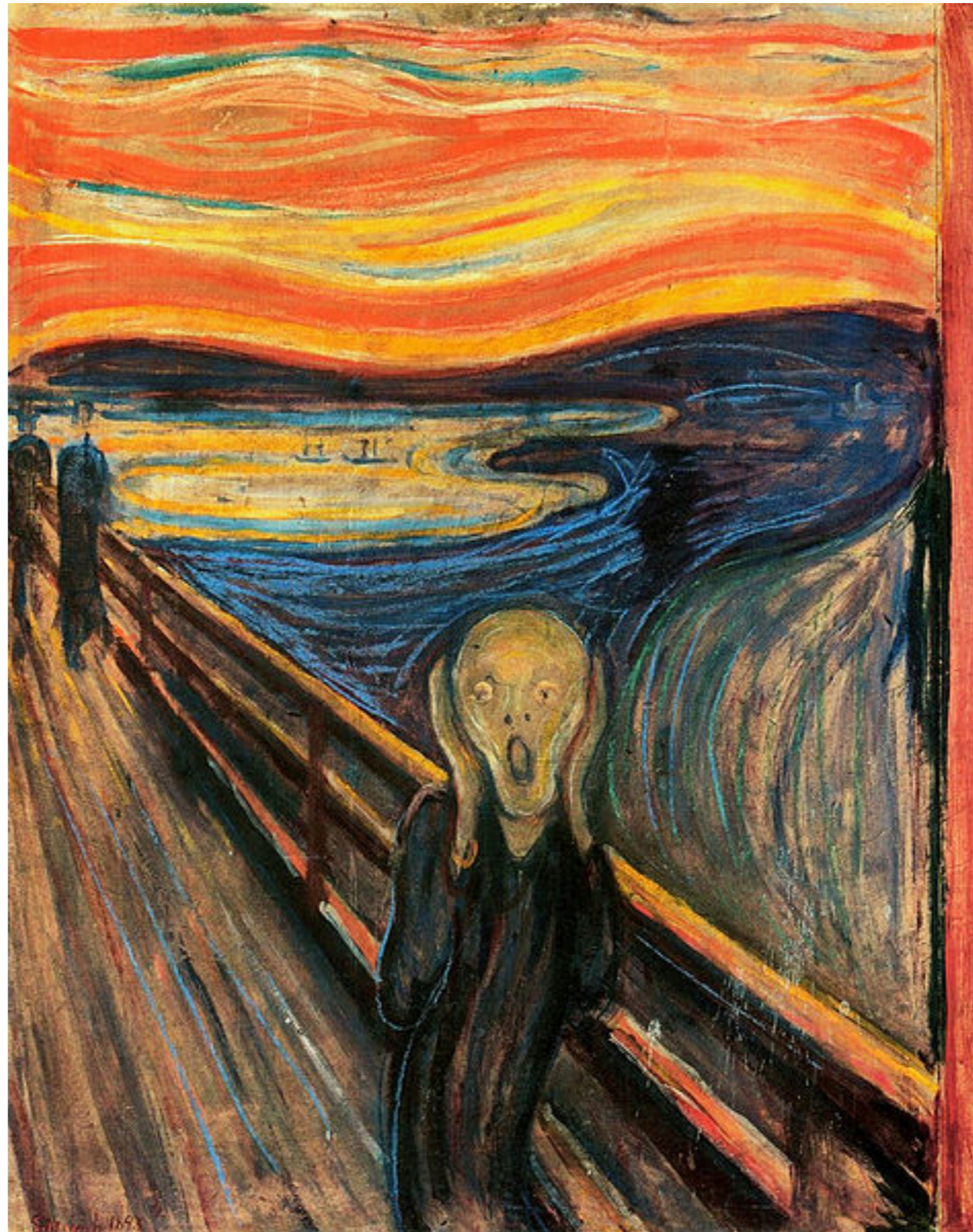
```
void scambia(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}  
  
int main () {  
    int x = 10, y = 5;  
    scambia(x, y);  
    printf("x=%d y=%d", x, y);  
    return 0;  
}
```

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

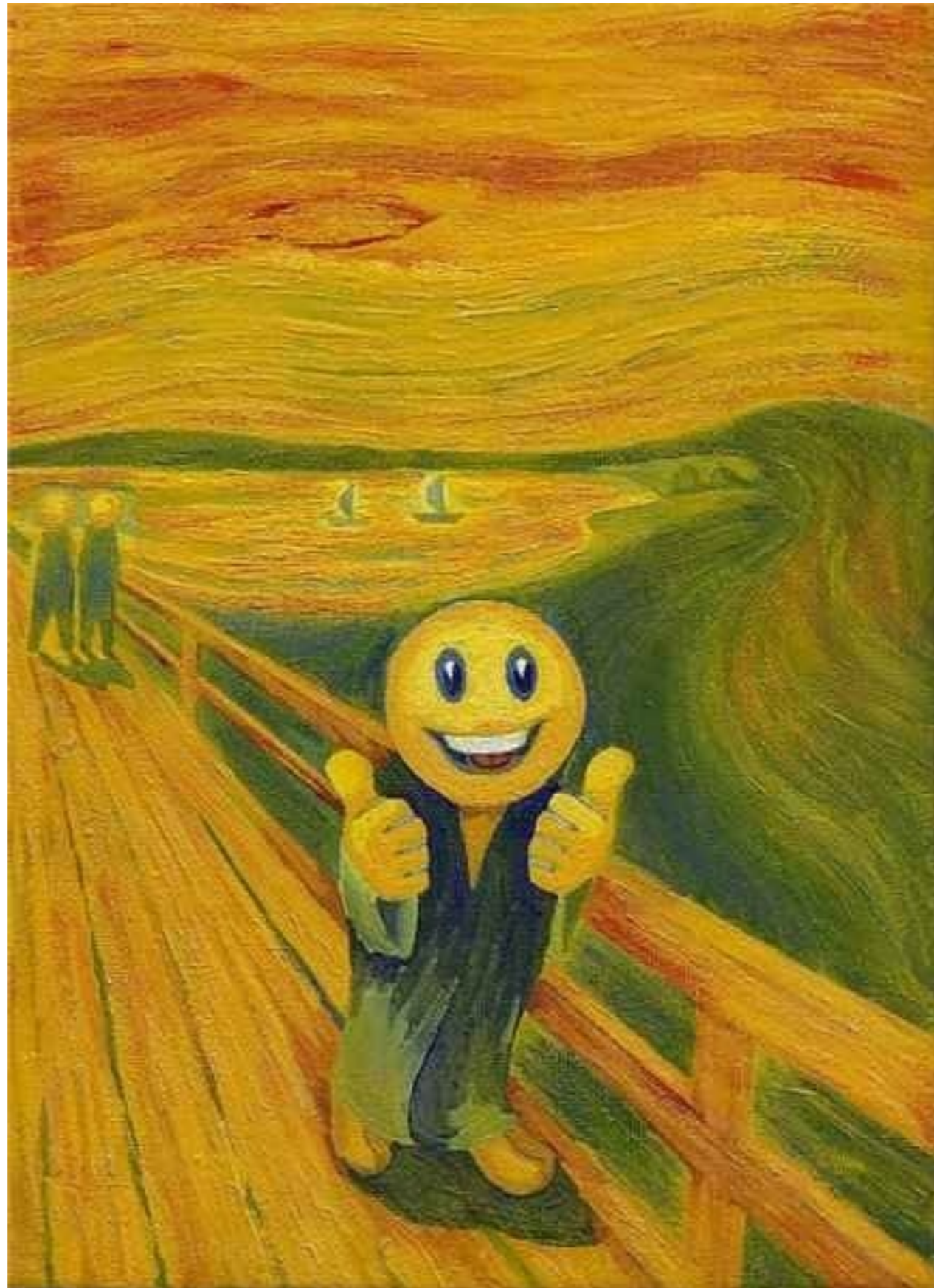
	<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
		...	
			0x100
			0x104
			0x108
			0x112
			0x116
	x	10	0x120
	y	5	0x124
			0x128
		...	

Puntatori (1)

Puntatori (1)



Puntatori (1)



Puntatori (1)

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Puntatori (1)

Una variabile di tipo int memorizza un valore.

```
int x = 10;
```

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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.

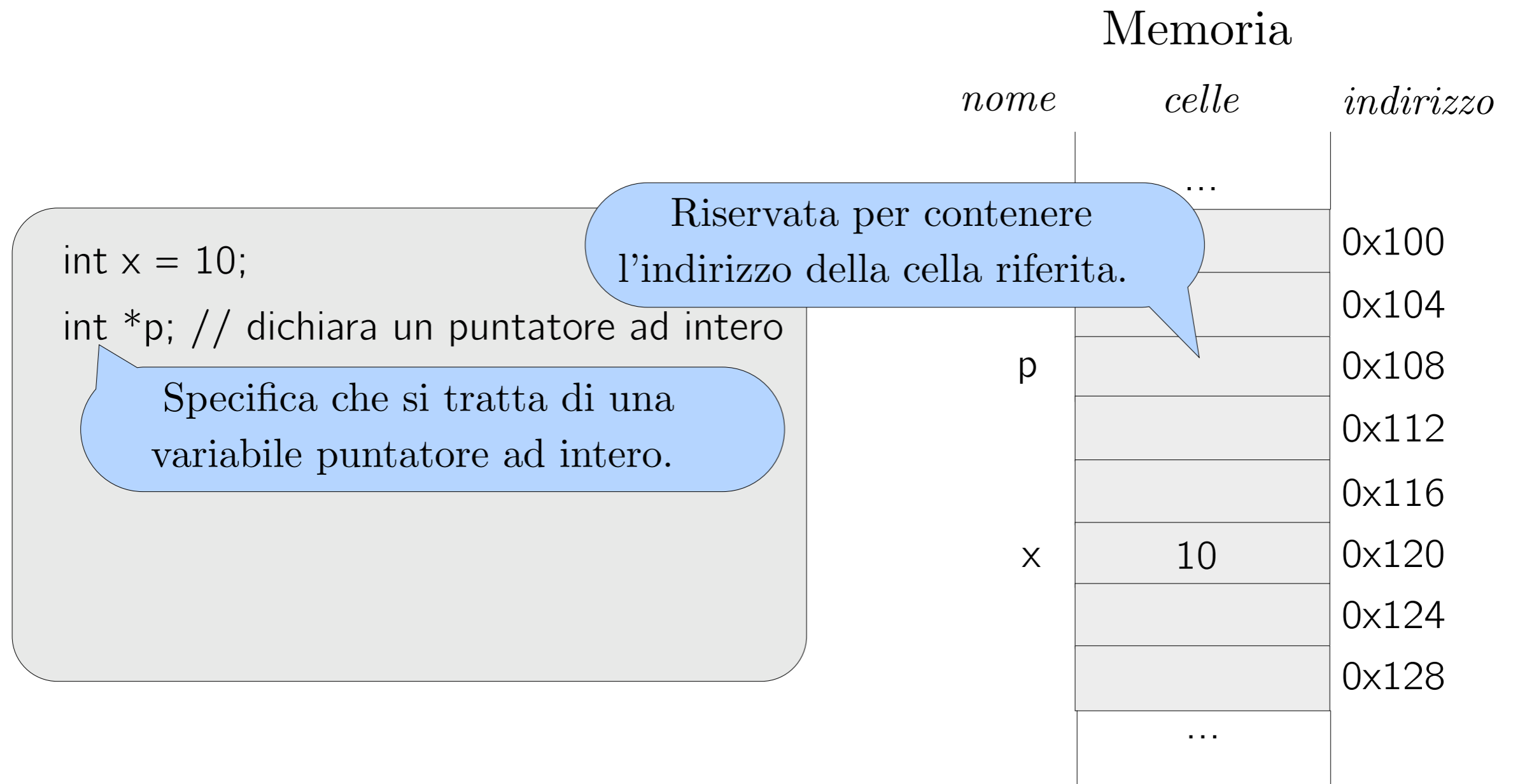
Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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
```

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		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.

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
```

Memoria

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
<i>p</i>		0x108
		0x112
		0x116
<i>x</i>	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.

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;
```

Memoria		
<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
<i>p</i>		0x108
		0x112
		0x116
<i>x</i>	10	0x120
		0x124
		0x128
	...	

Puntatori (1)

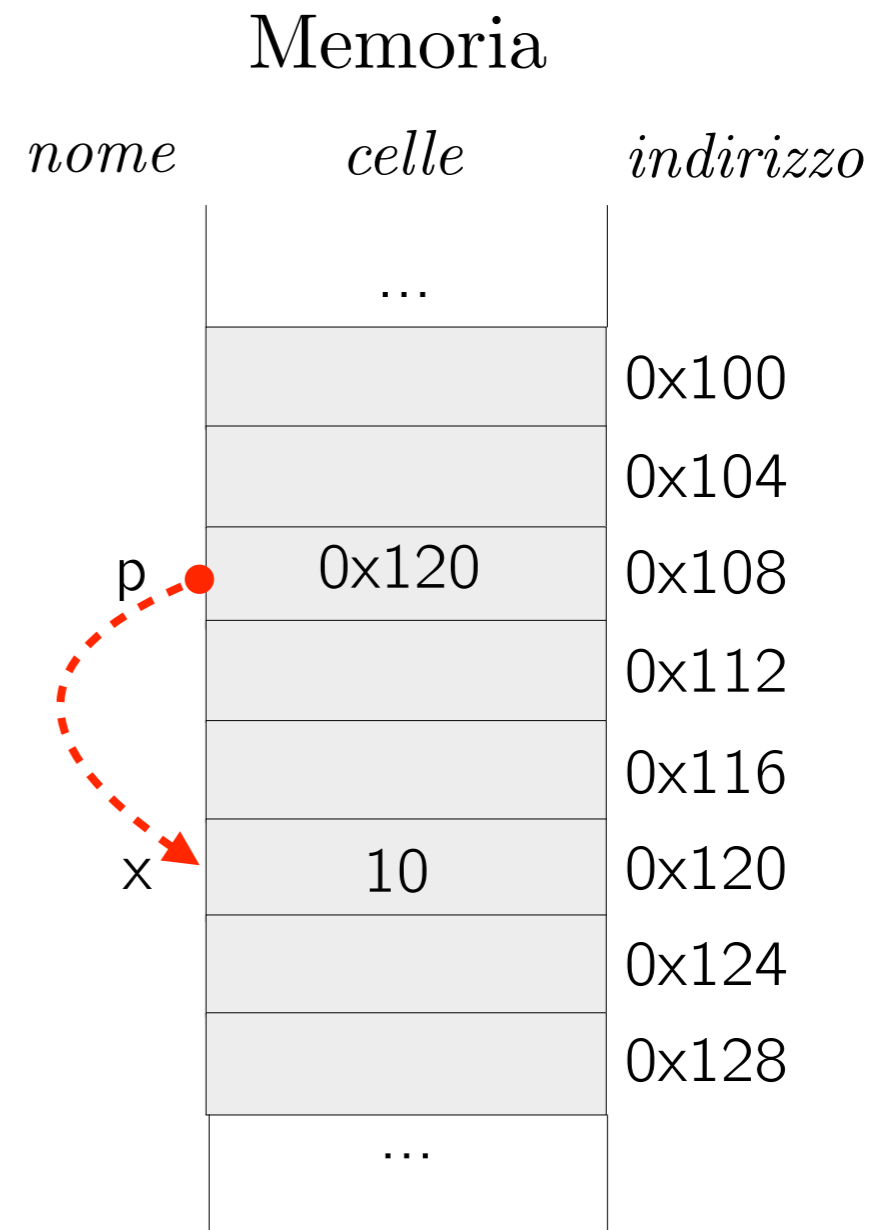
Una variabile di tipo `int` memorizza un valore.

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

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;
```



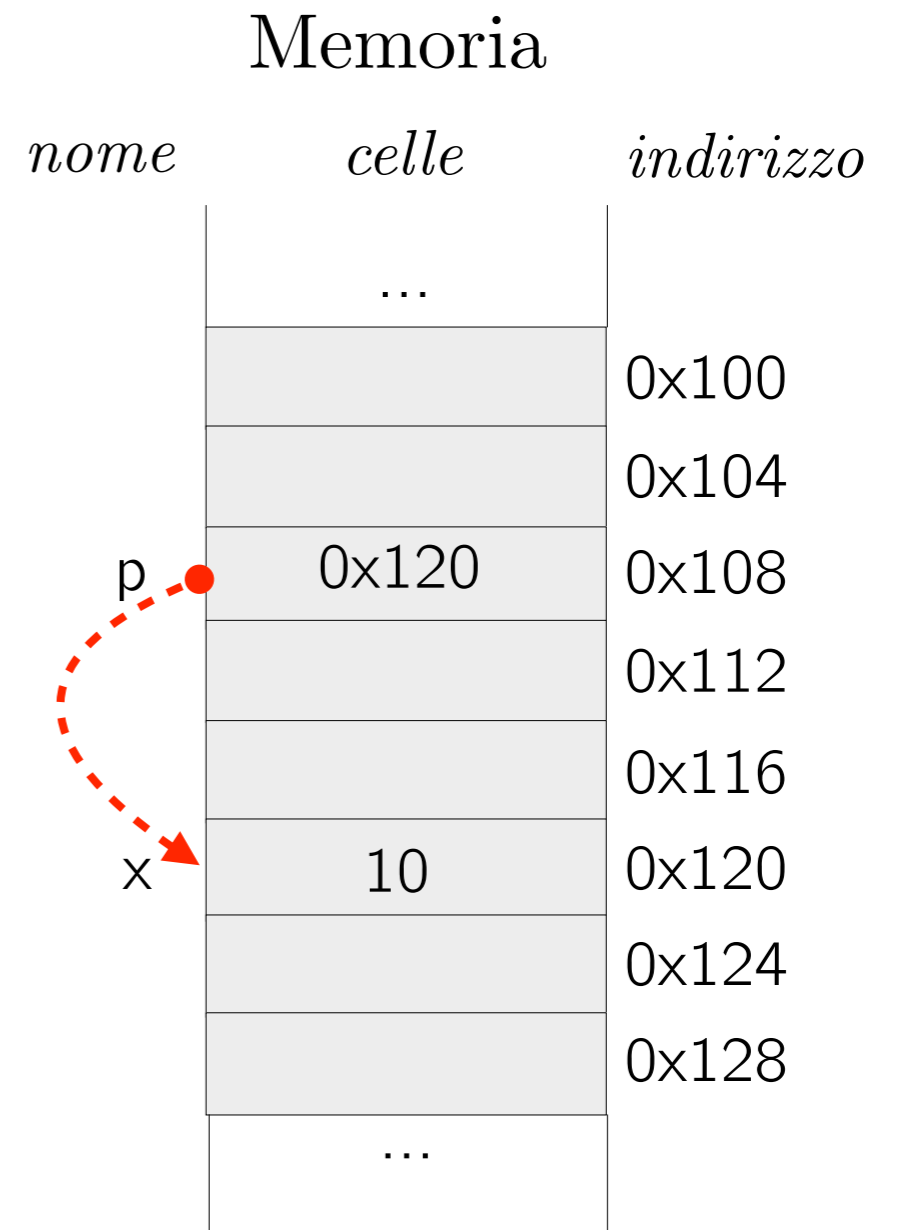
Puntatori (1)

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)

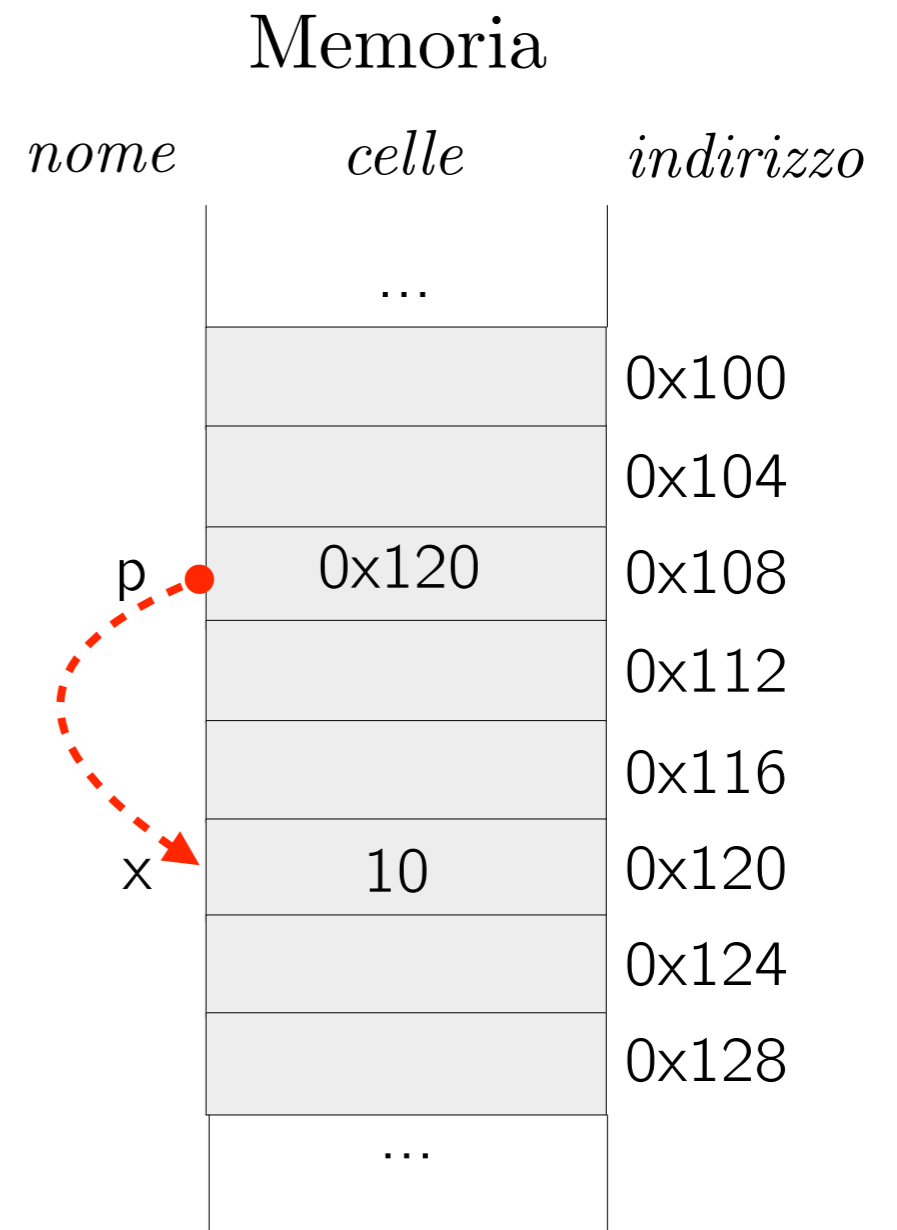
Una variabile di tipo `int` memorizza un valore.

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)

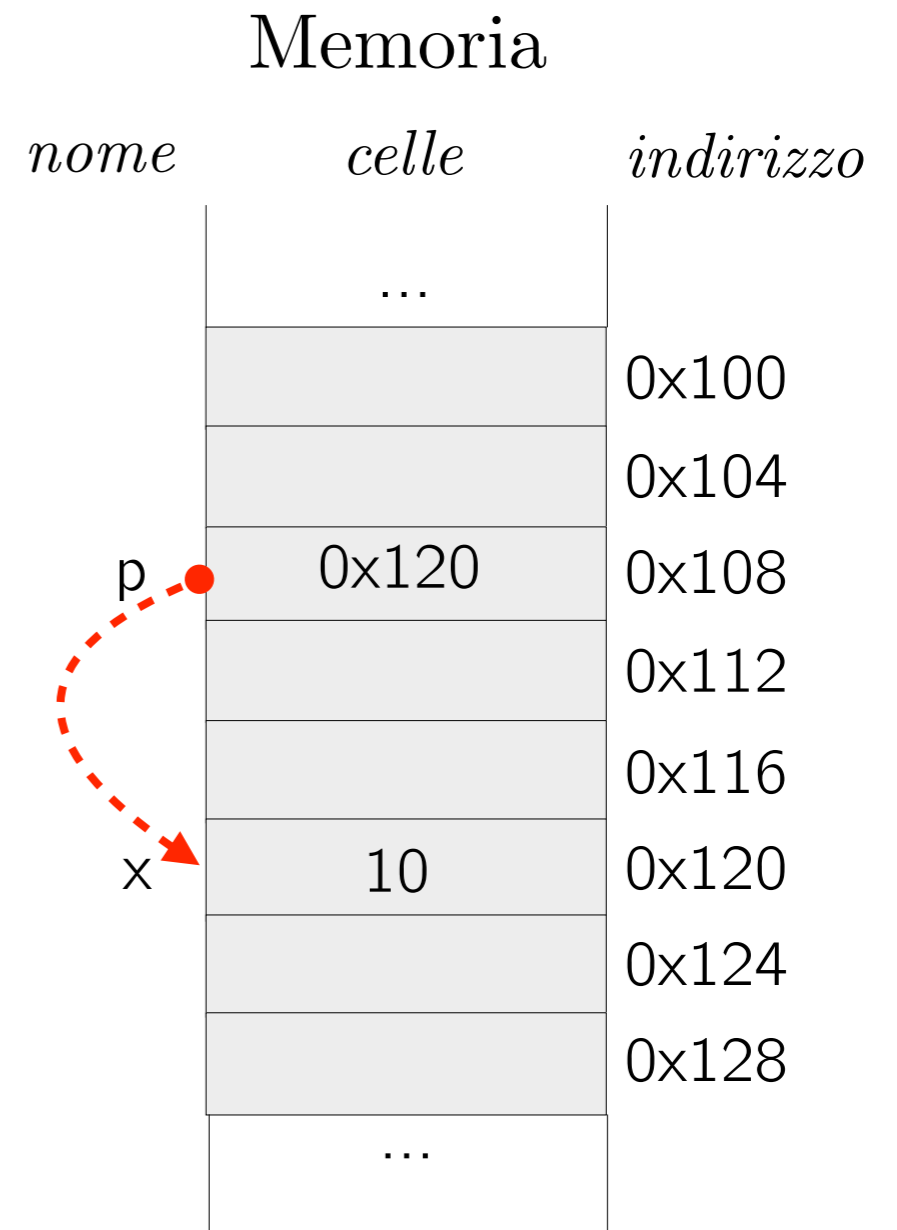
Una variabile di tipo `int` memorizza un valore.

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.

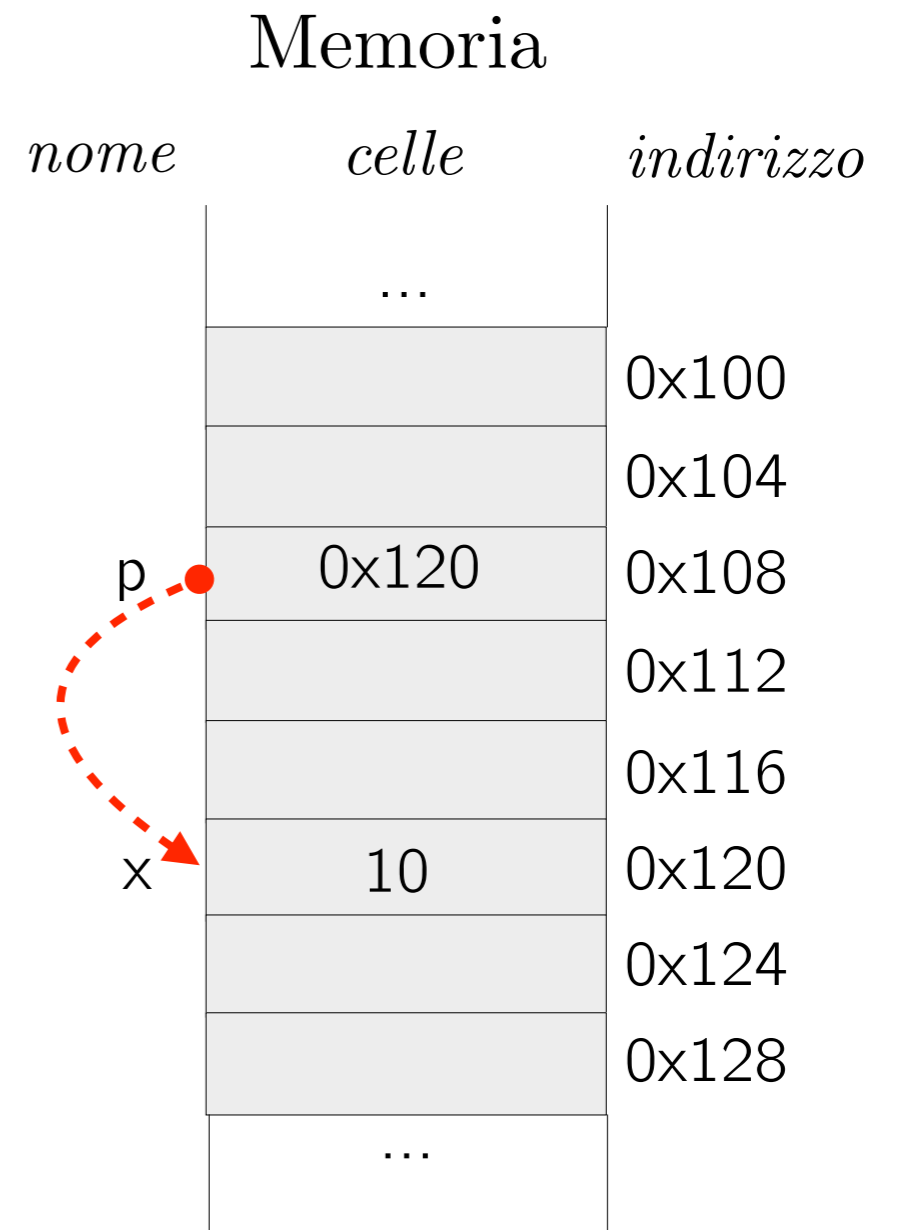
Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

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)

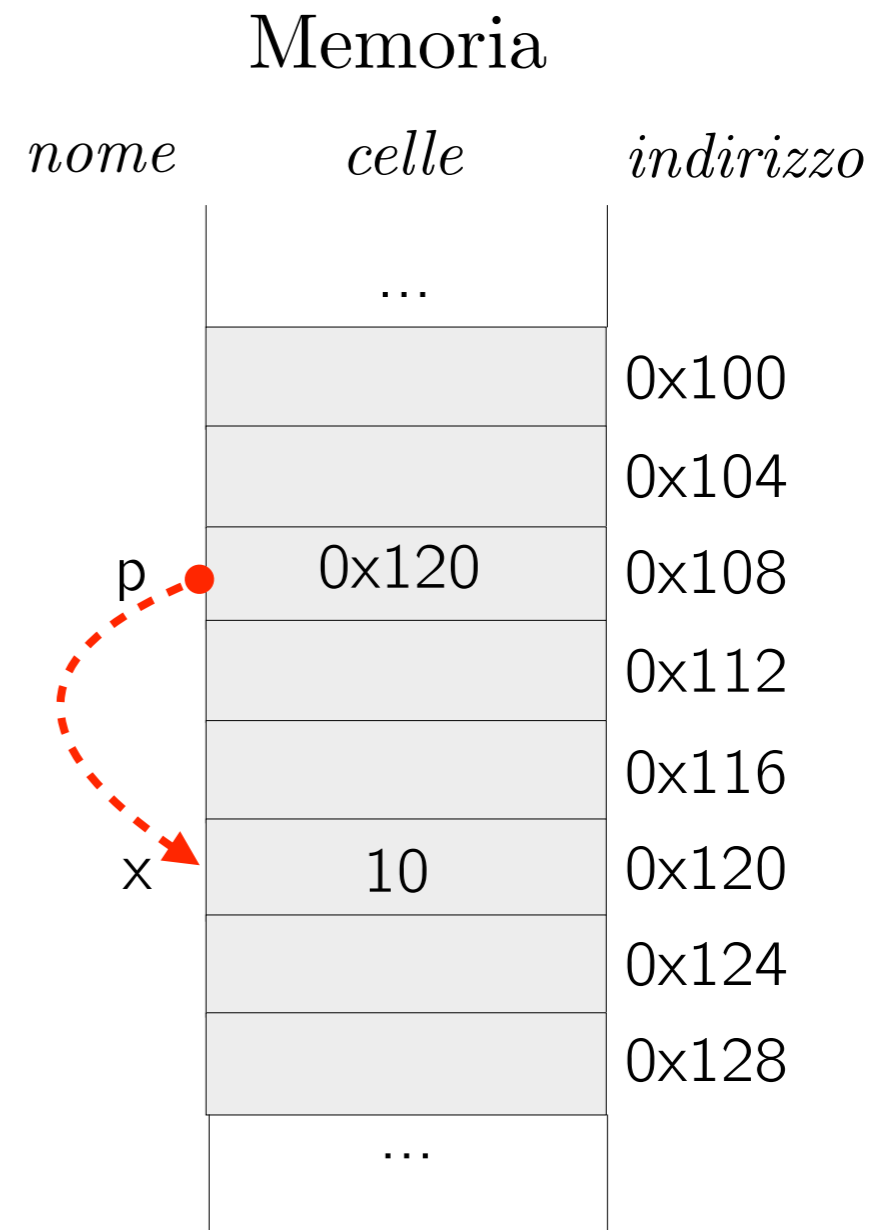
Una variabile di tipo `int` memorizza un valore.

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)

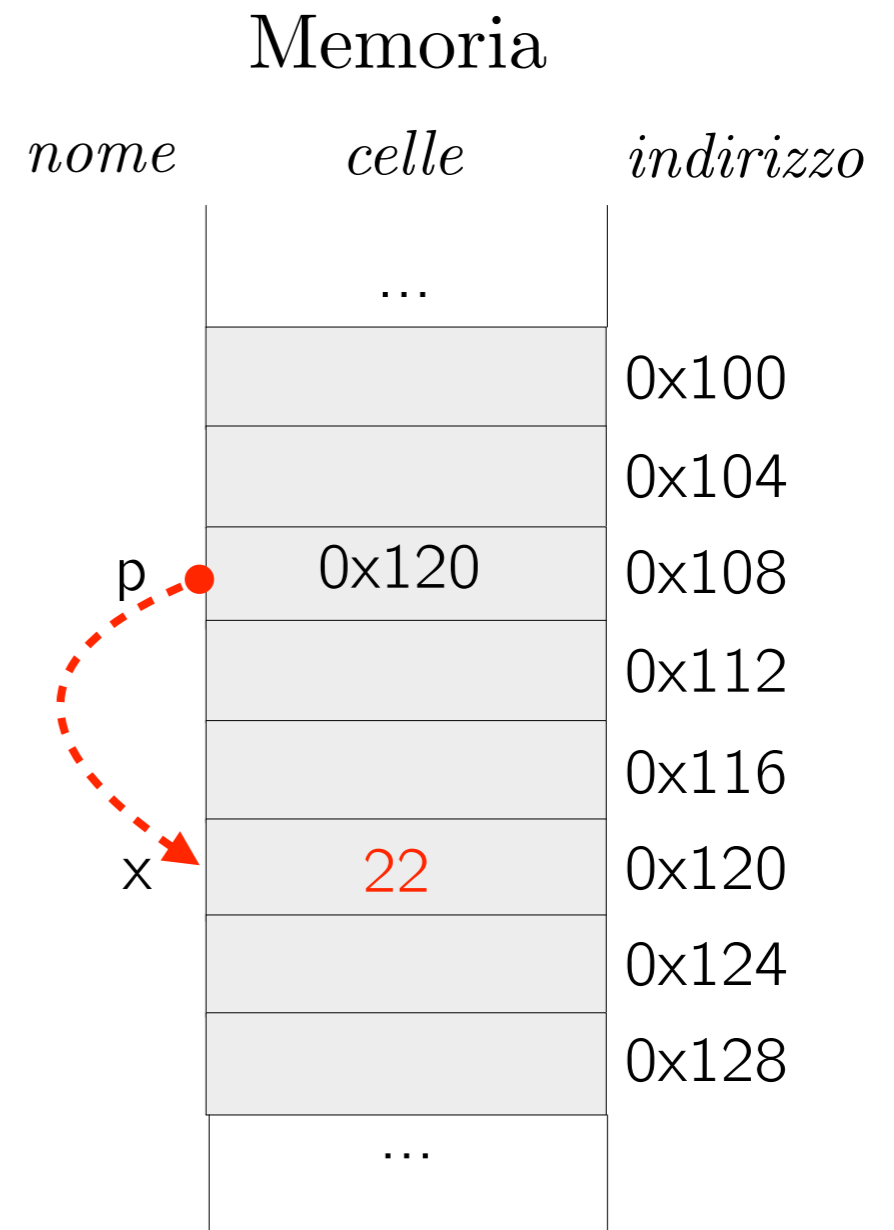
Una variabile di tipo `int` memorizza un valore.

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.

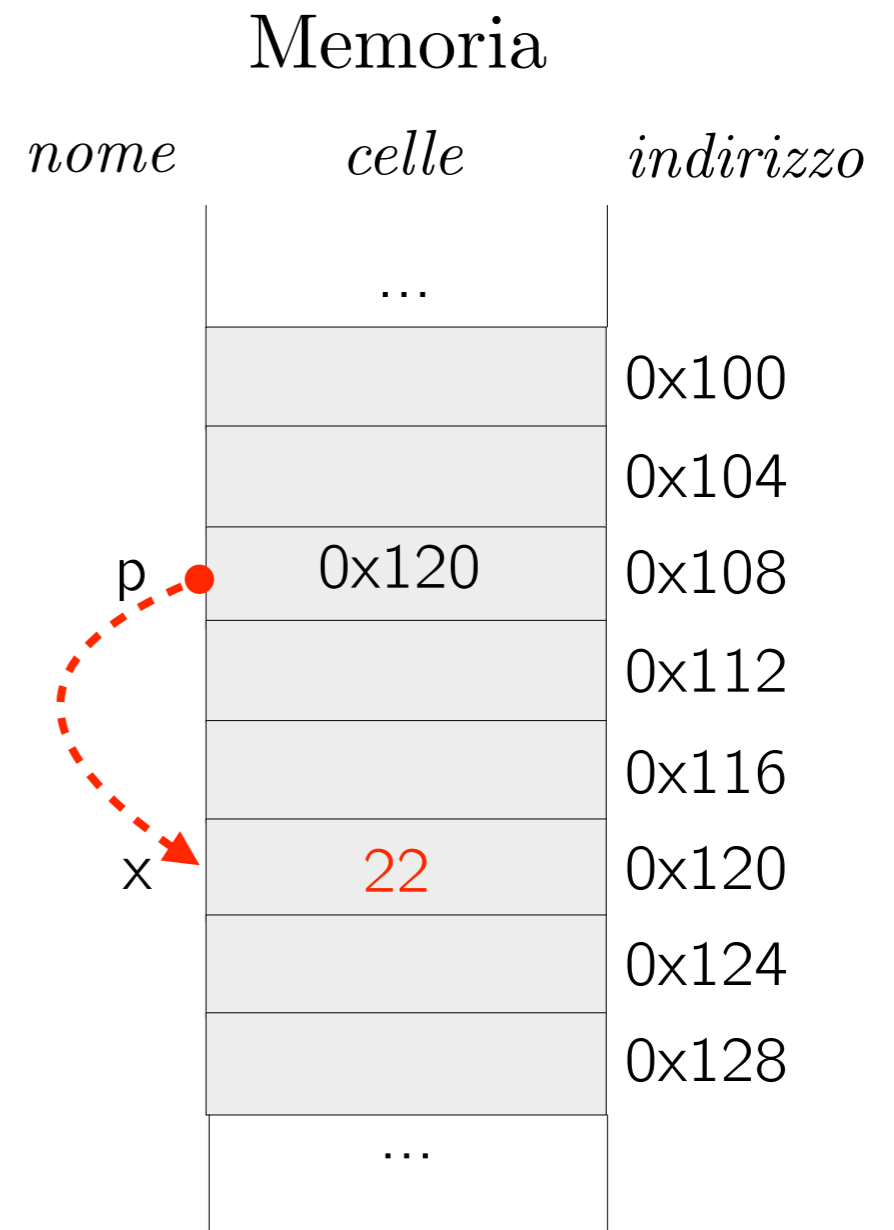
Un puntatore è una variabile che memorizza un riferimento (al valore) contenuto di un'altra cella.

e ora?

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

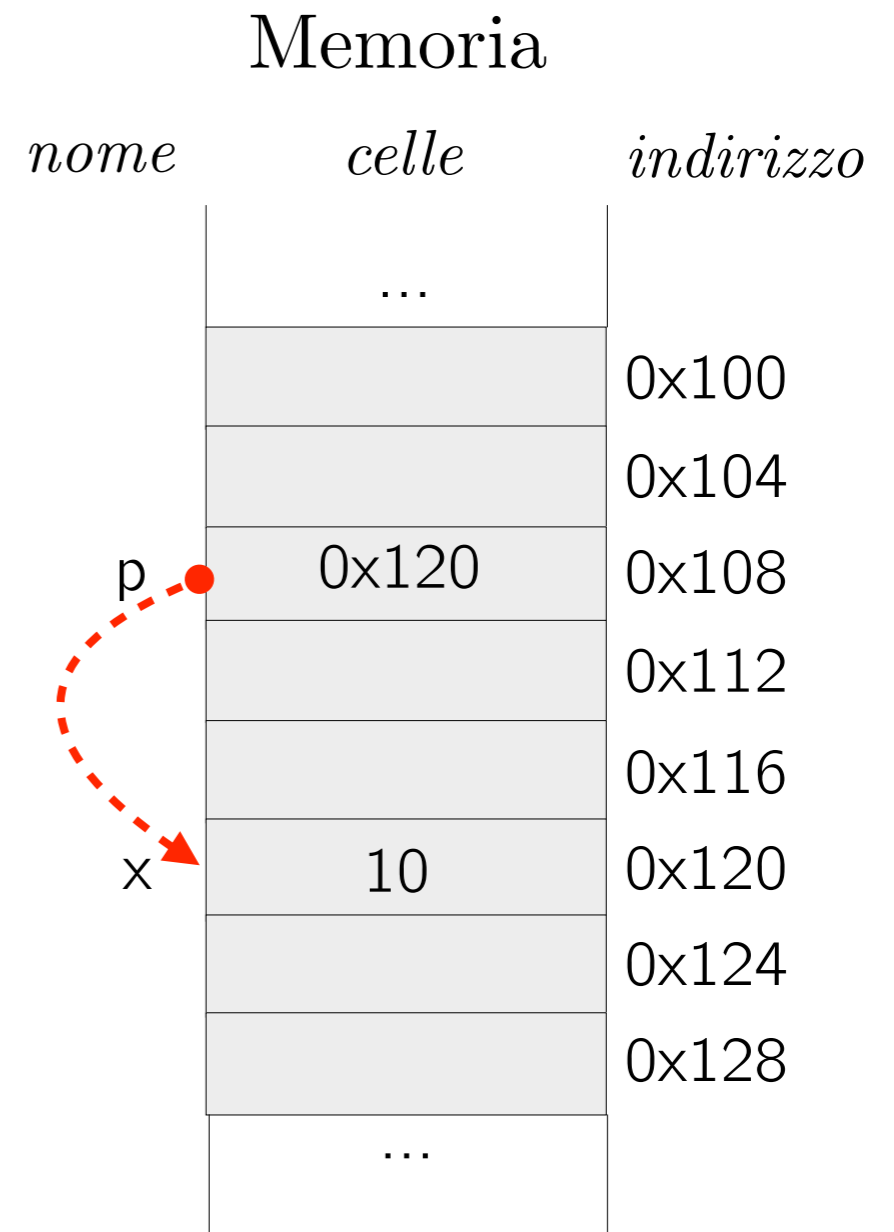
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
y = *p+3 // equivalente a y = x +3;
*p = 22; // equivalente a x = 22;
```

Si può usare `*p` proprio come useremmo `x`



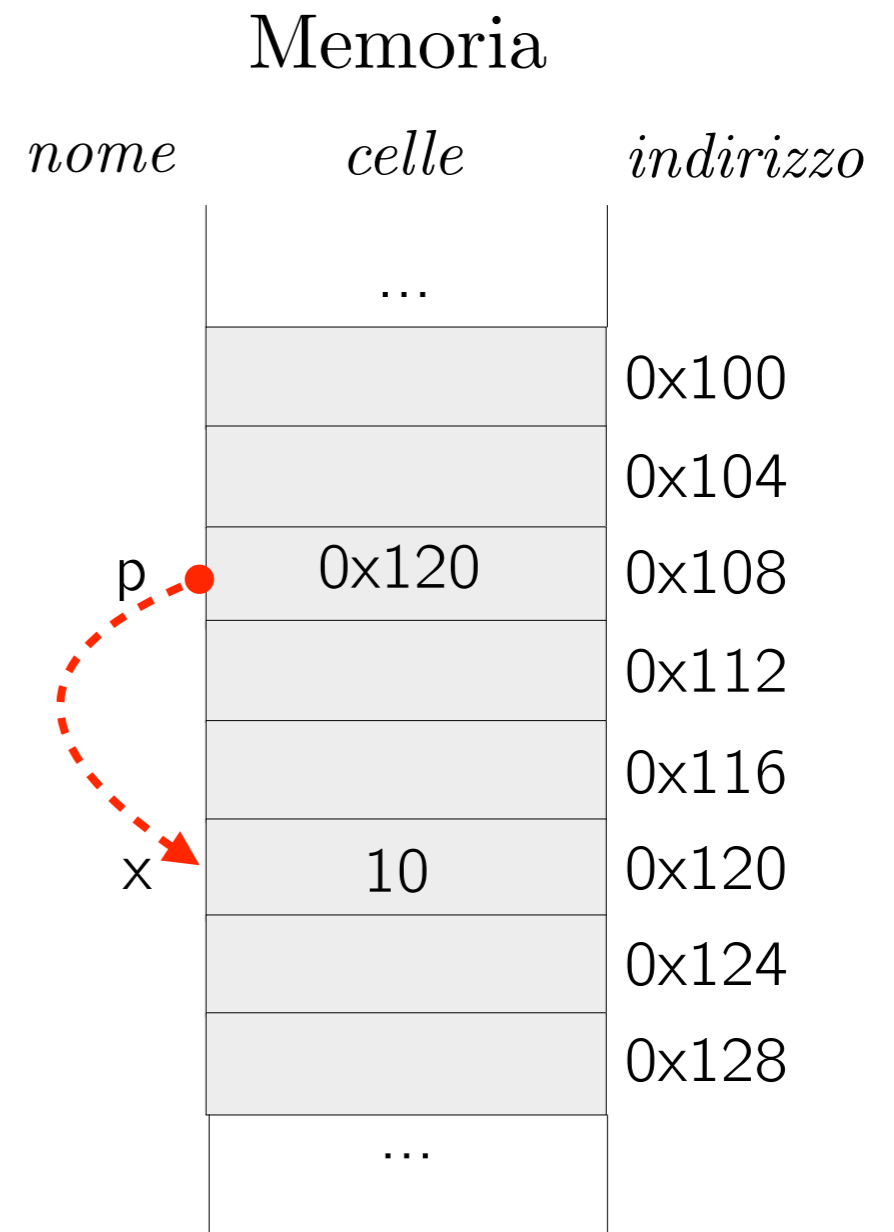
Puntatori (2)

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



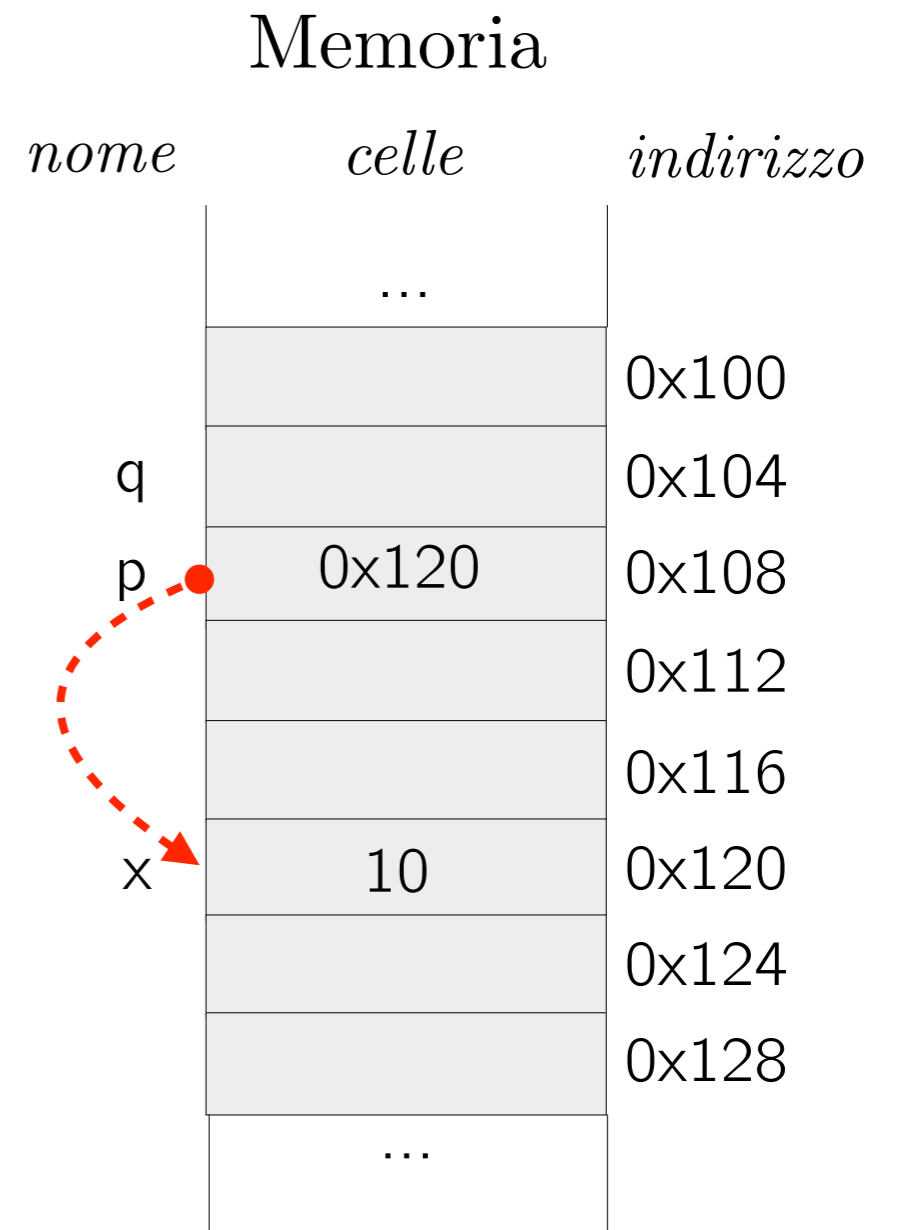
Puntatori (2)

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



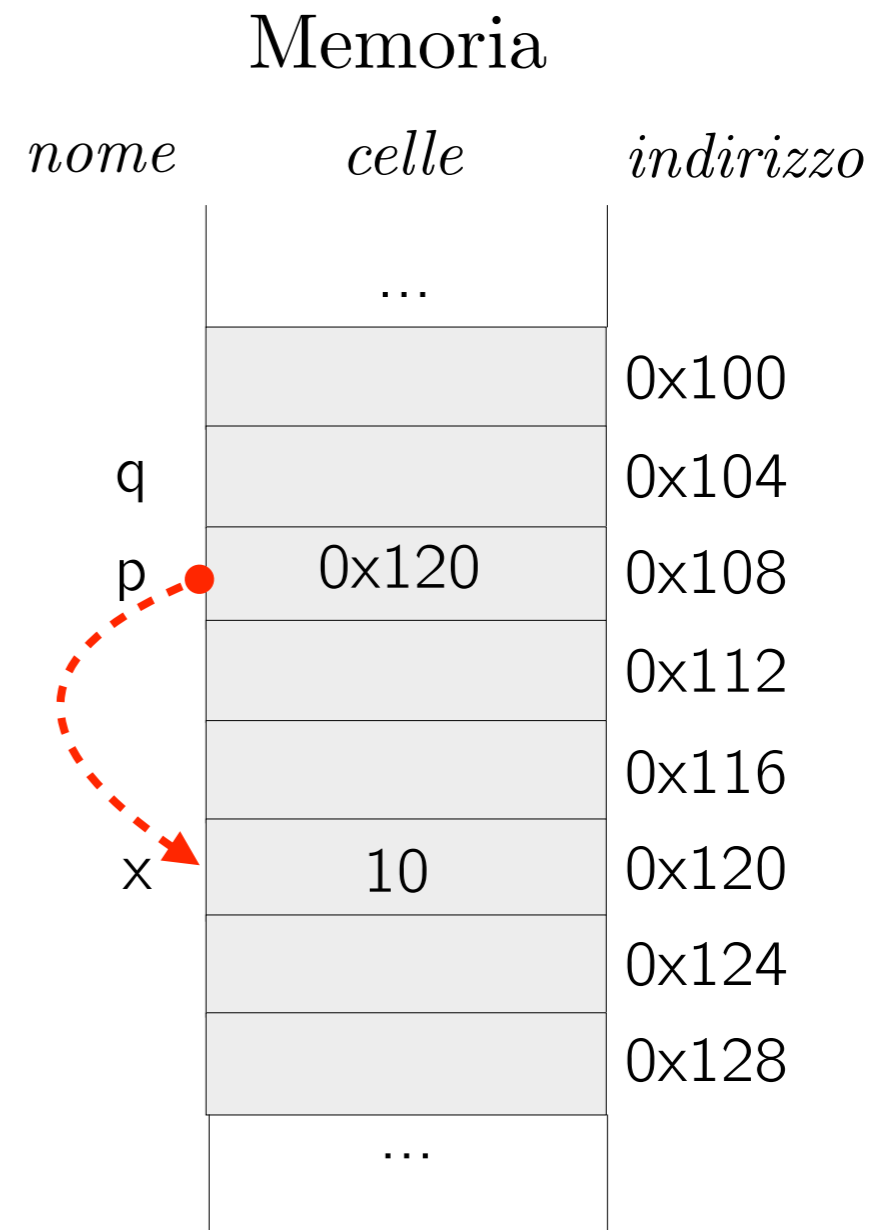
Puntatori (2)

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



Puntatori (2)

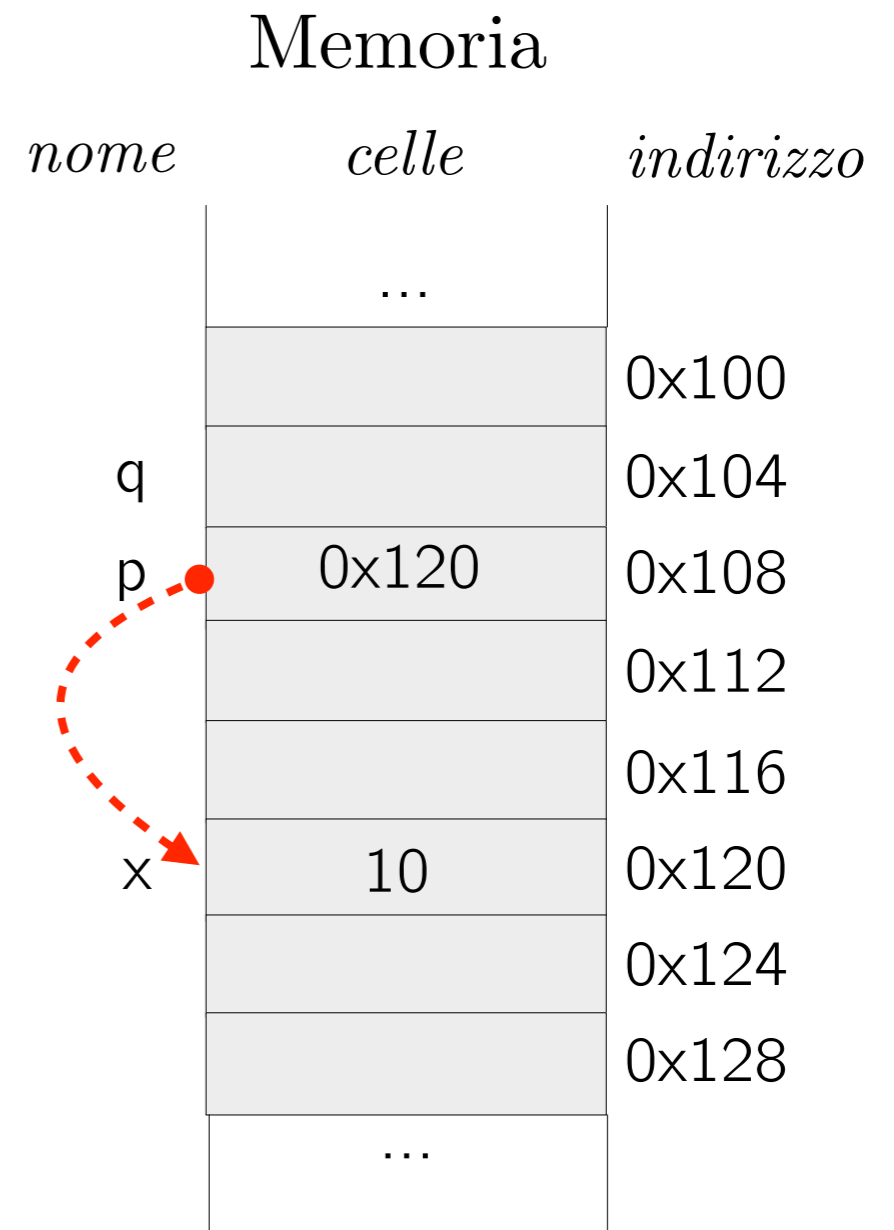
```
int x = 10;  
int *p; // dichiara un puntatore ad intero  
p = &x;  
int *q;  
*q = 5;
```



Puntatori (2)

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

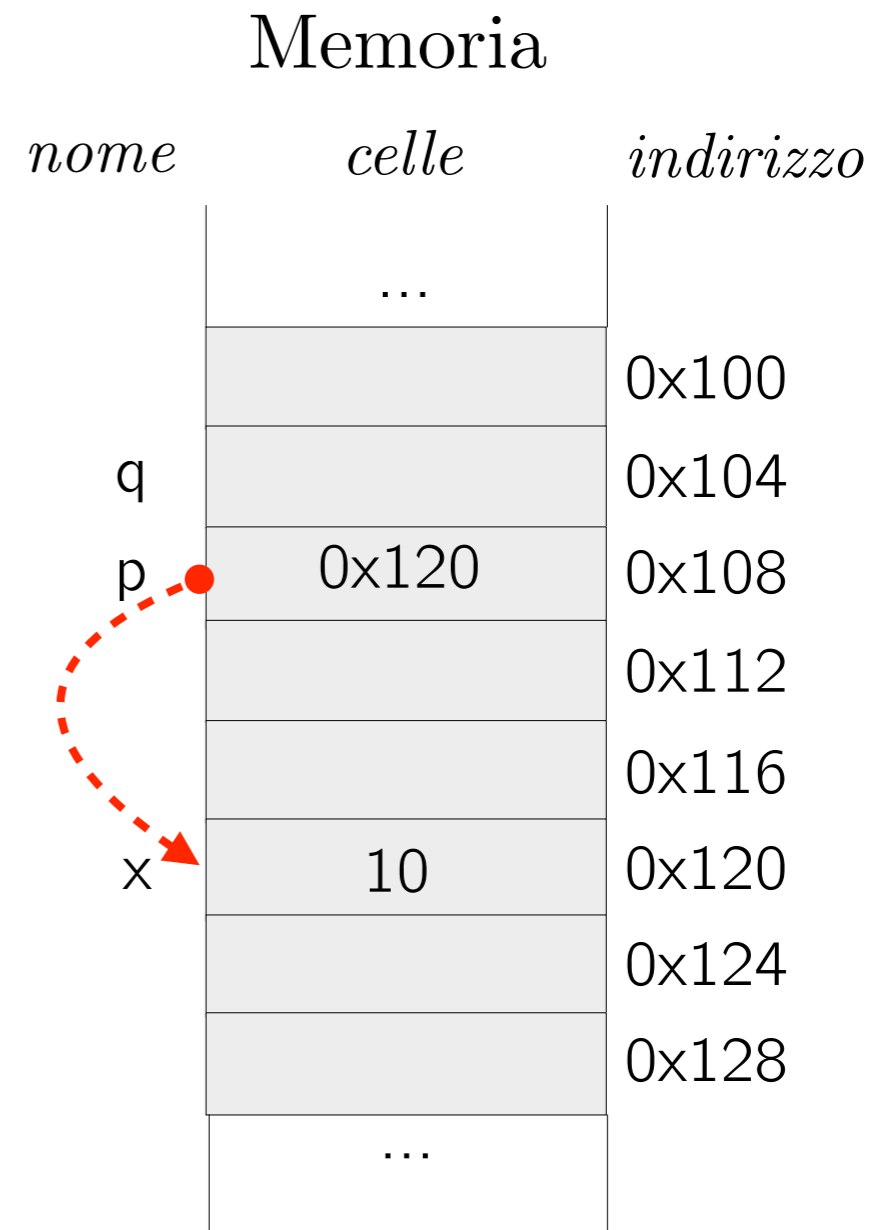
NO! q non è inizializzato.
Segmentation Fault!



Puntatori (2)

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

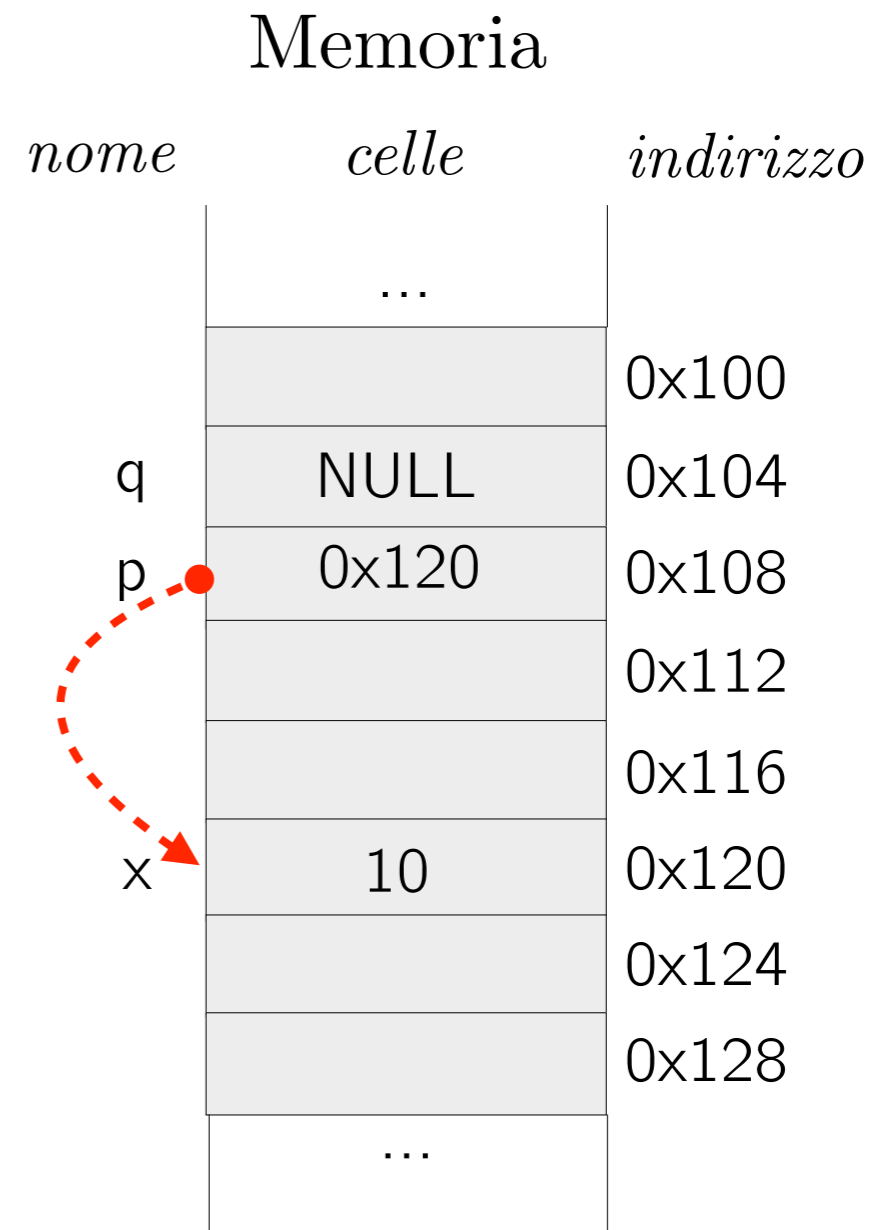
NO! q non è inizializzato.
Segmentation Fault!



Puntatori (2)

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

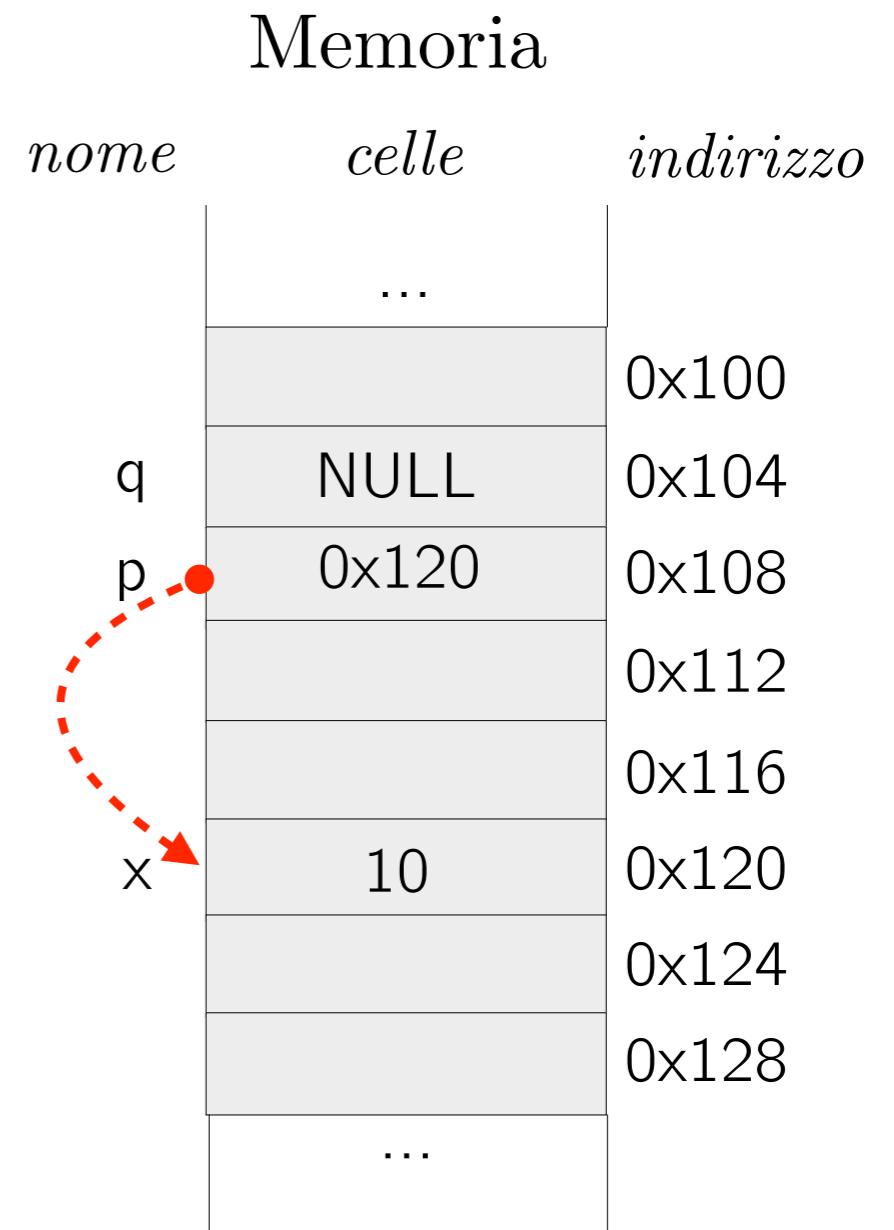
NO! q non è inizializzato.
Segmentation Fault!



Puntatori (2)

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

NO! q non è inizializzato.
Segmentation Fault!

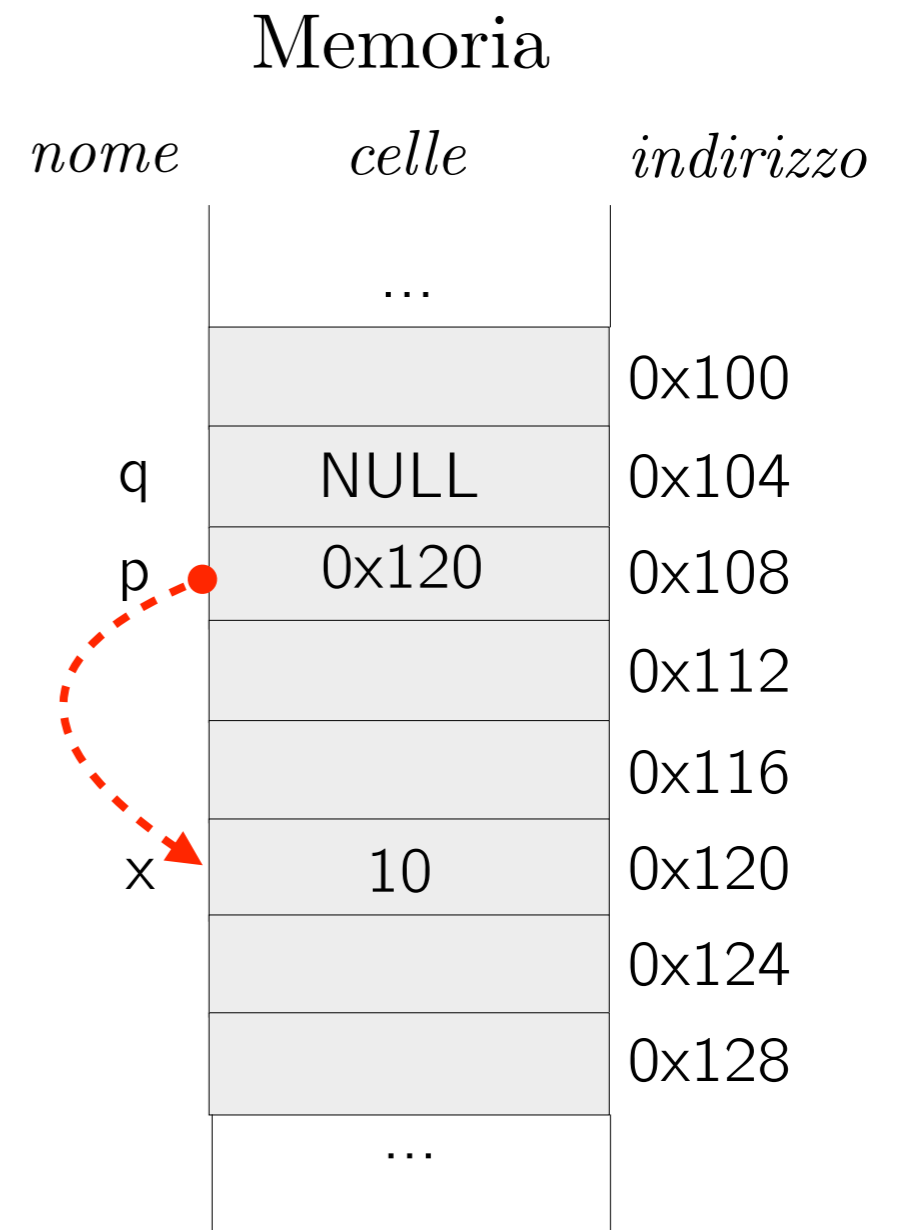


Puntatori (2)

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

NO! q non è inizializzato.
Segmentation Fault!

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

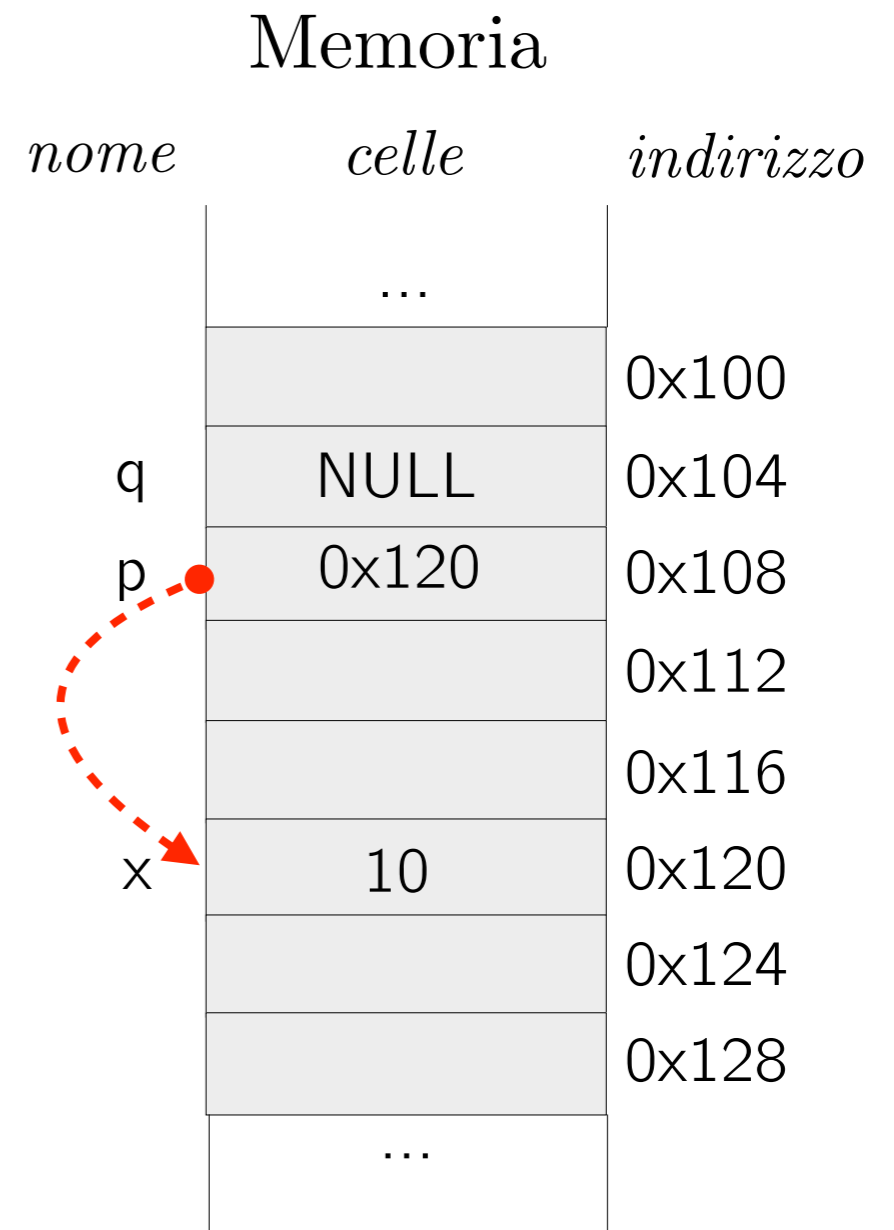


Puntatori (2)

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

NO! q non è inizializzato.
Segmentation Fault!

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



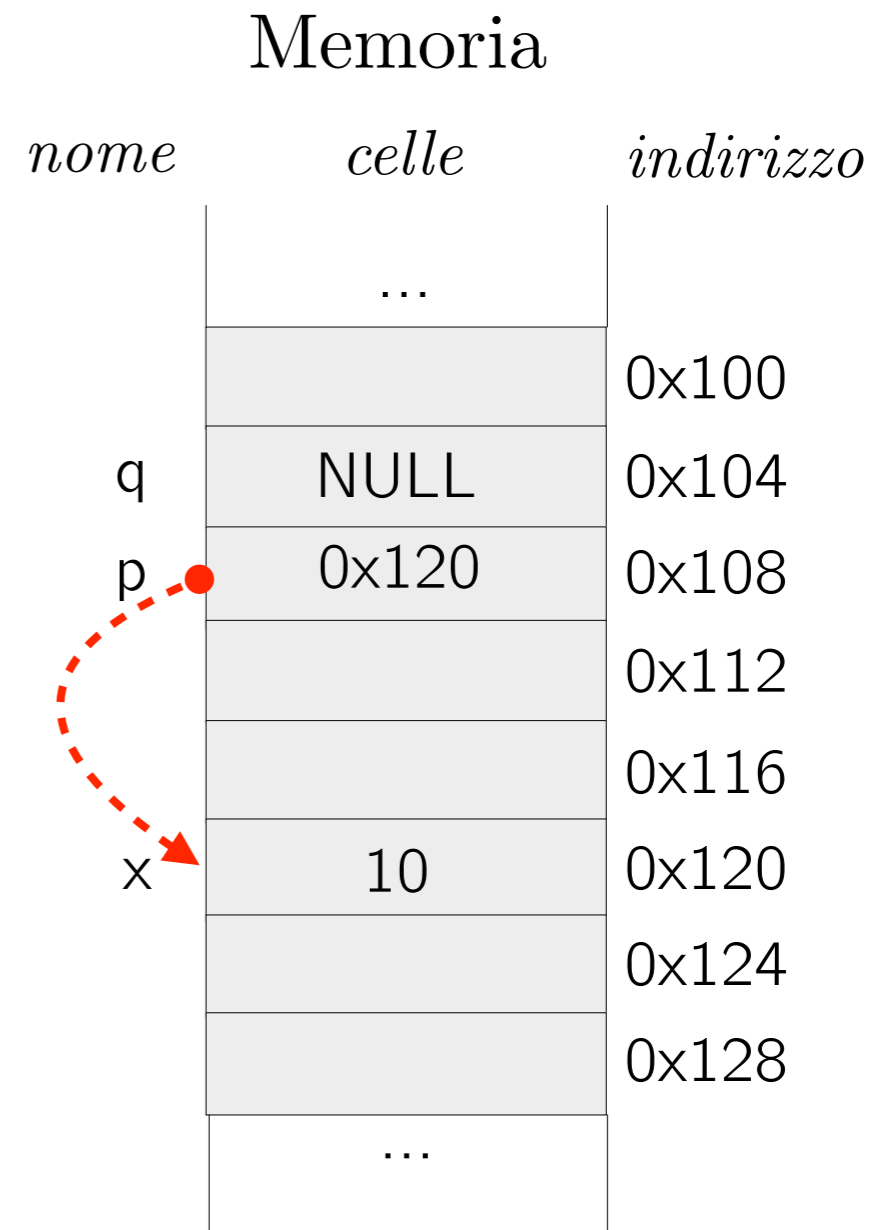
Puntatori (2)

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

NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo



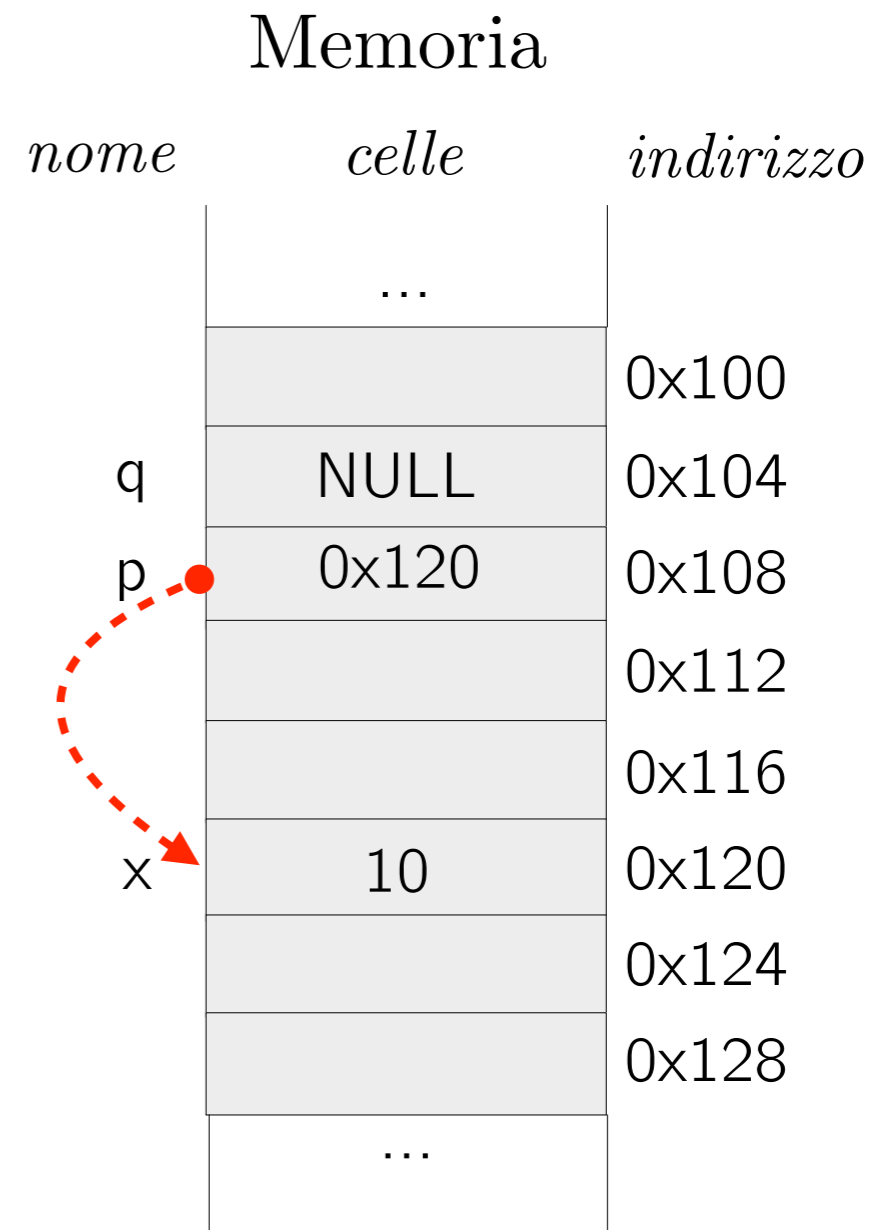
Puntatori (2)

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

NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo



Puntatori (2)

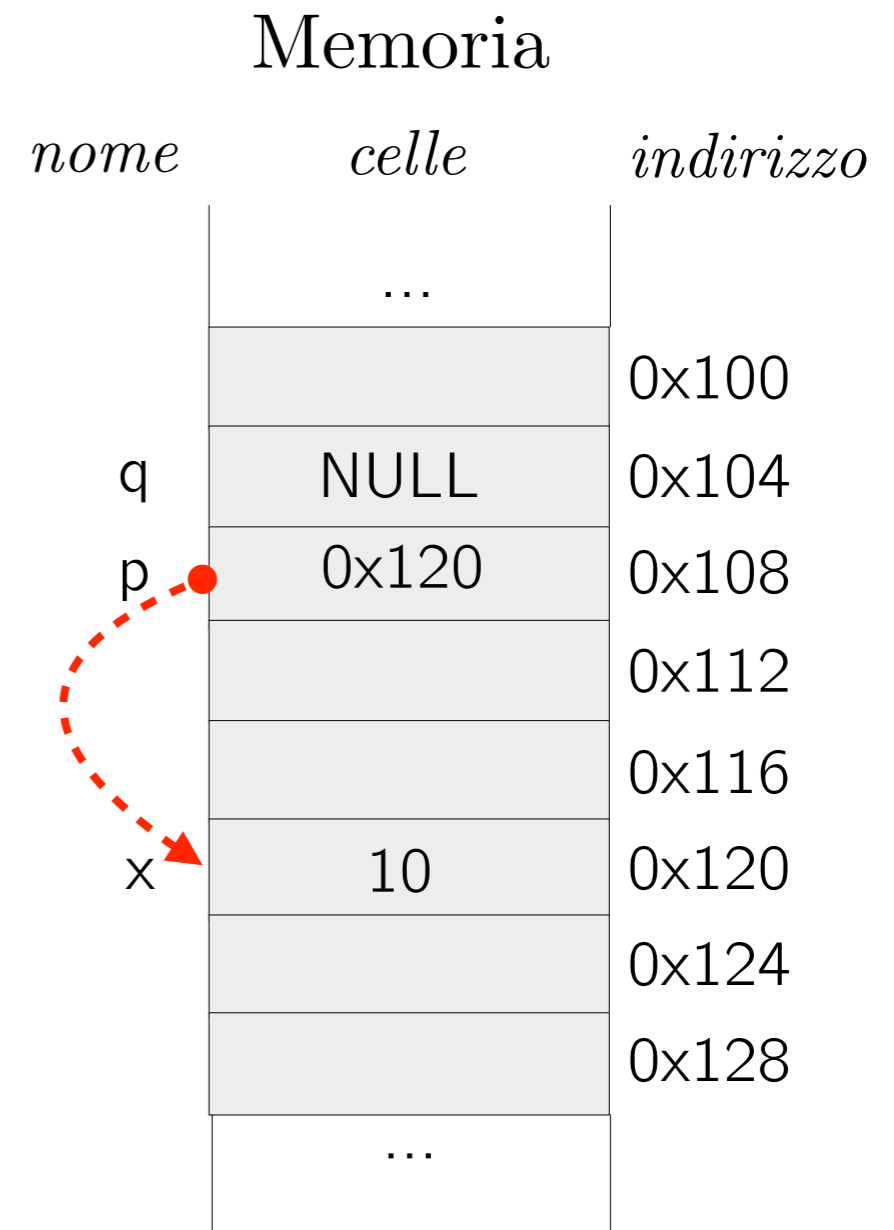
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
q = NULL;
*q = 5;
q = x;
q = &p;
```

NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

Errore di tipo



Puntatori (2)

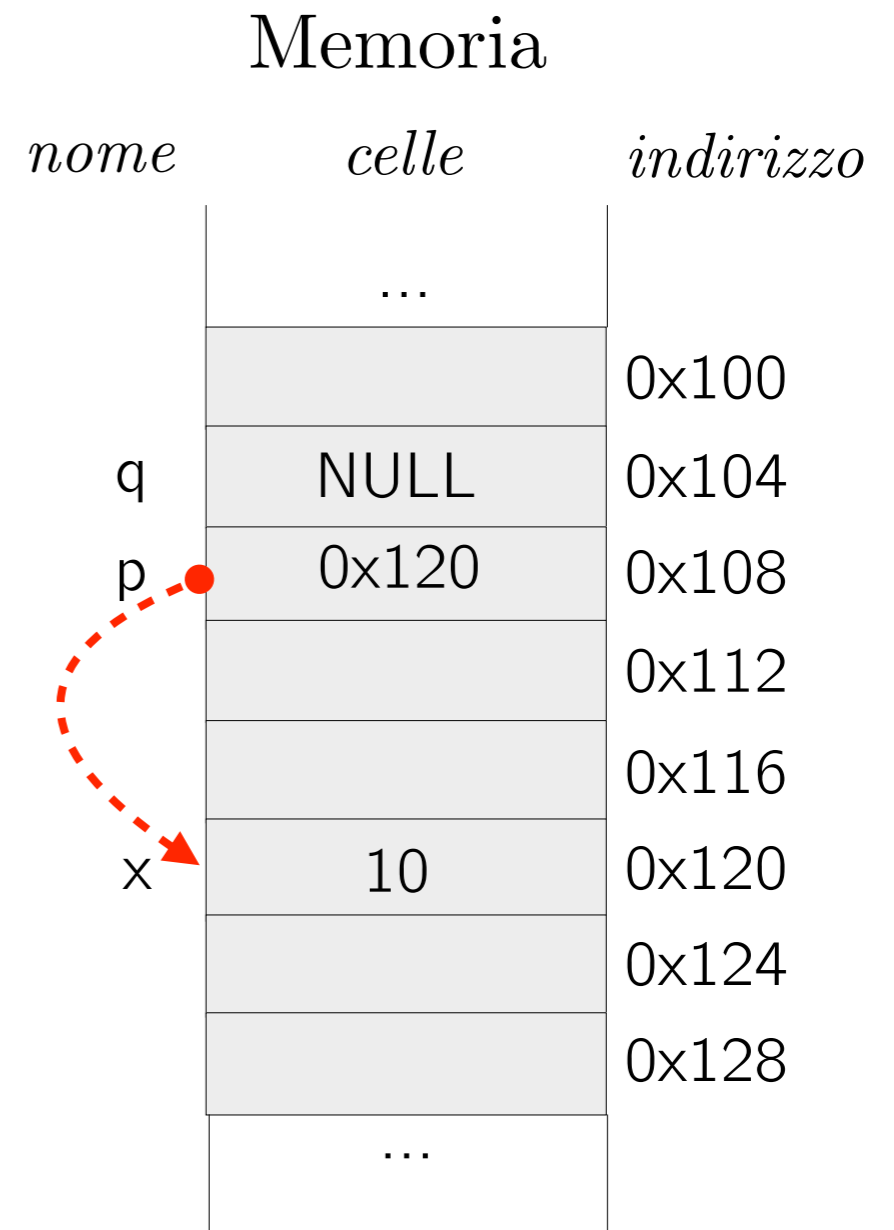
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
q = NULL;
*q = 5;
q = x;
q = &p;
q = &x;
```

NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

Errore di tipo



Puntatori (2)

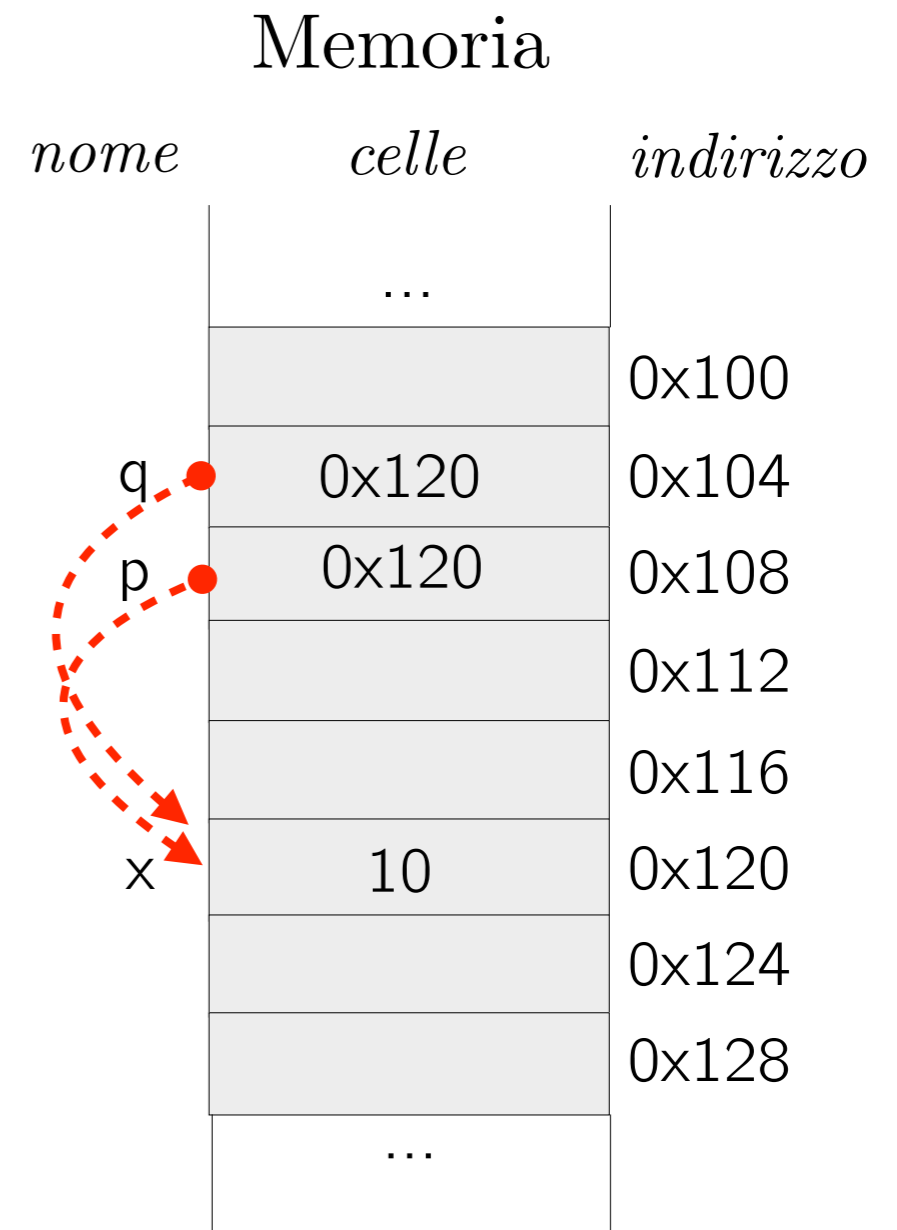
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
q = NULL;
*q = 5;
q = x;
q = &p;
q = &x;
```

NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

Errore di tipo



Puntatori (2)

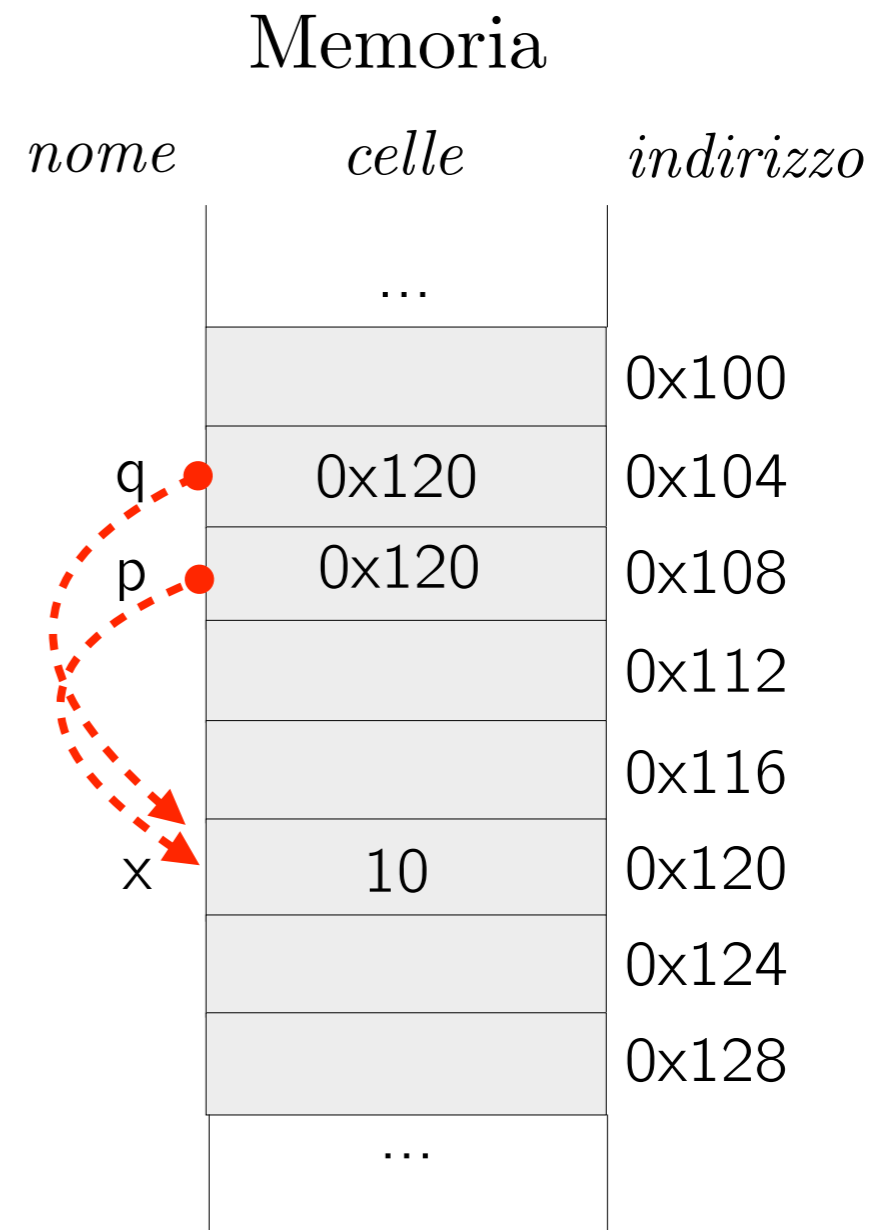
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
q = NULL;
*q = 5;
q = x;
q = &p;
q = &x;
q = p;
```

NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

Errore di tipo



Puntatori (2)

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

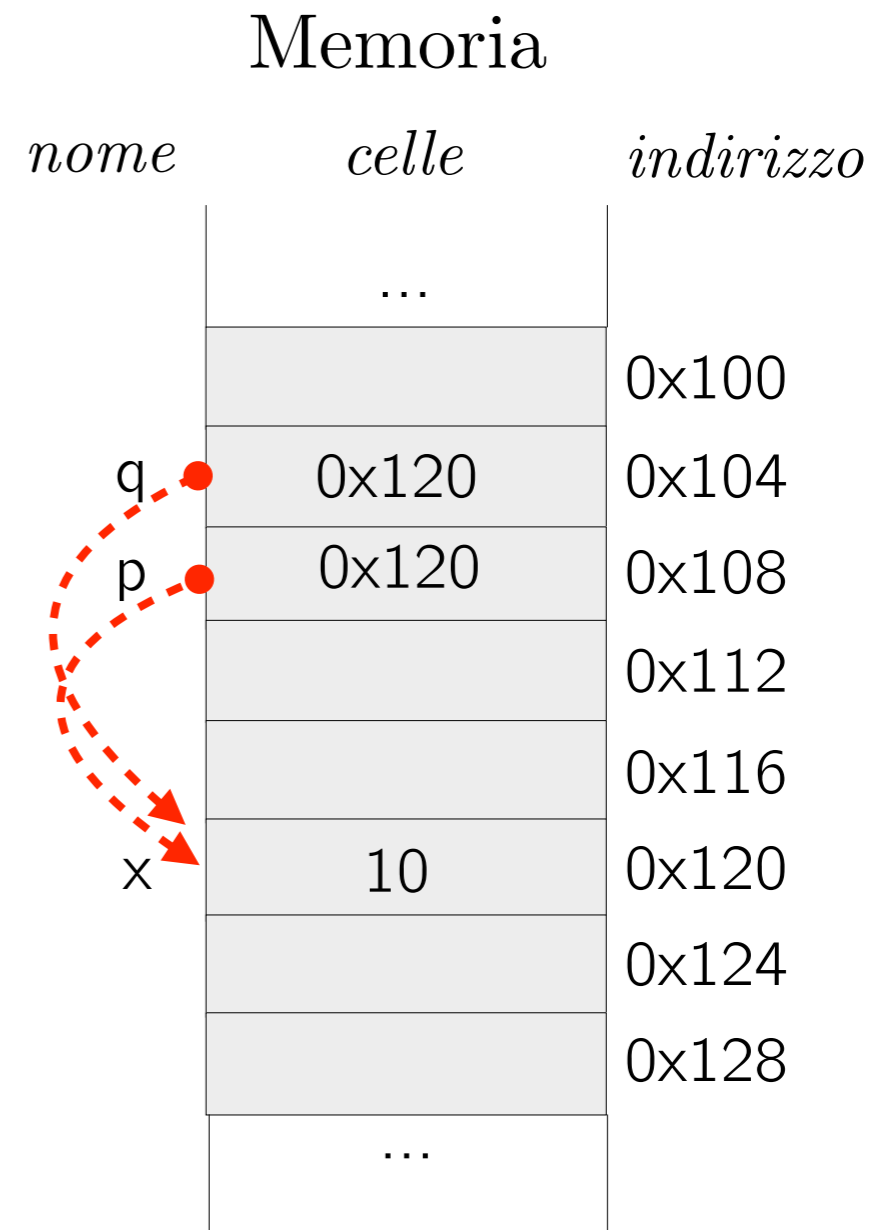
NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

Errore di tipo

Equivalente al precedente



Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
q = NULL;
*q = 5;
q = x;
q = &p;
q = &x;
q = p;
*q = 11;
```

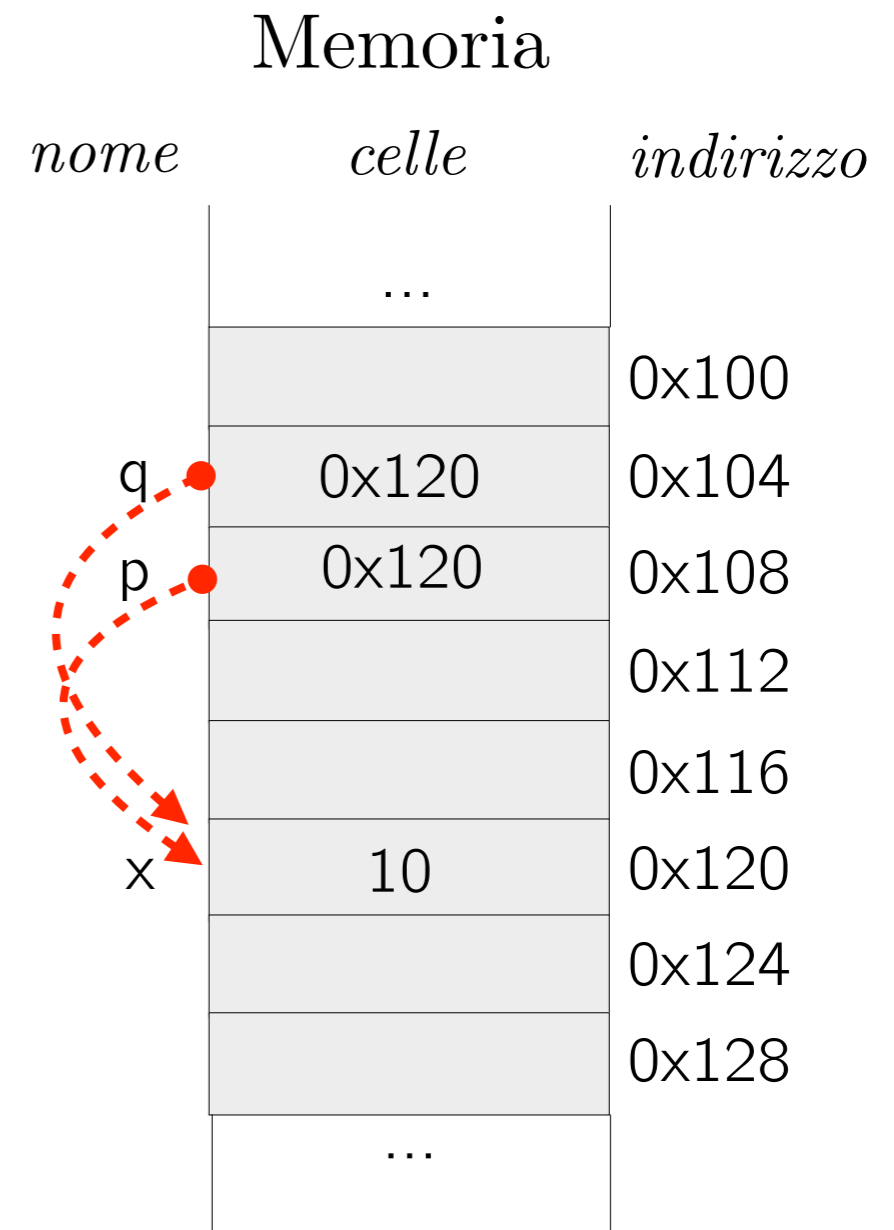
NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

Errore di tipo

Equivalente al precedente



Puntatori (2)

```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int *q;
*q = 5;
q = NULL;
*q = 5;
q = x;
q = &p;
q = &x;
q = p;
*q = 11;
```

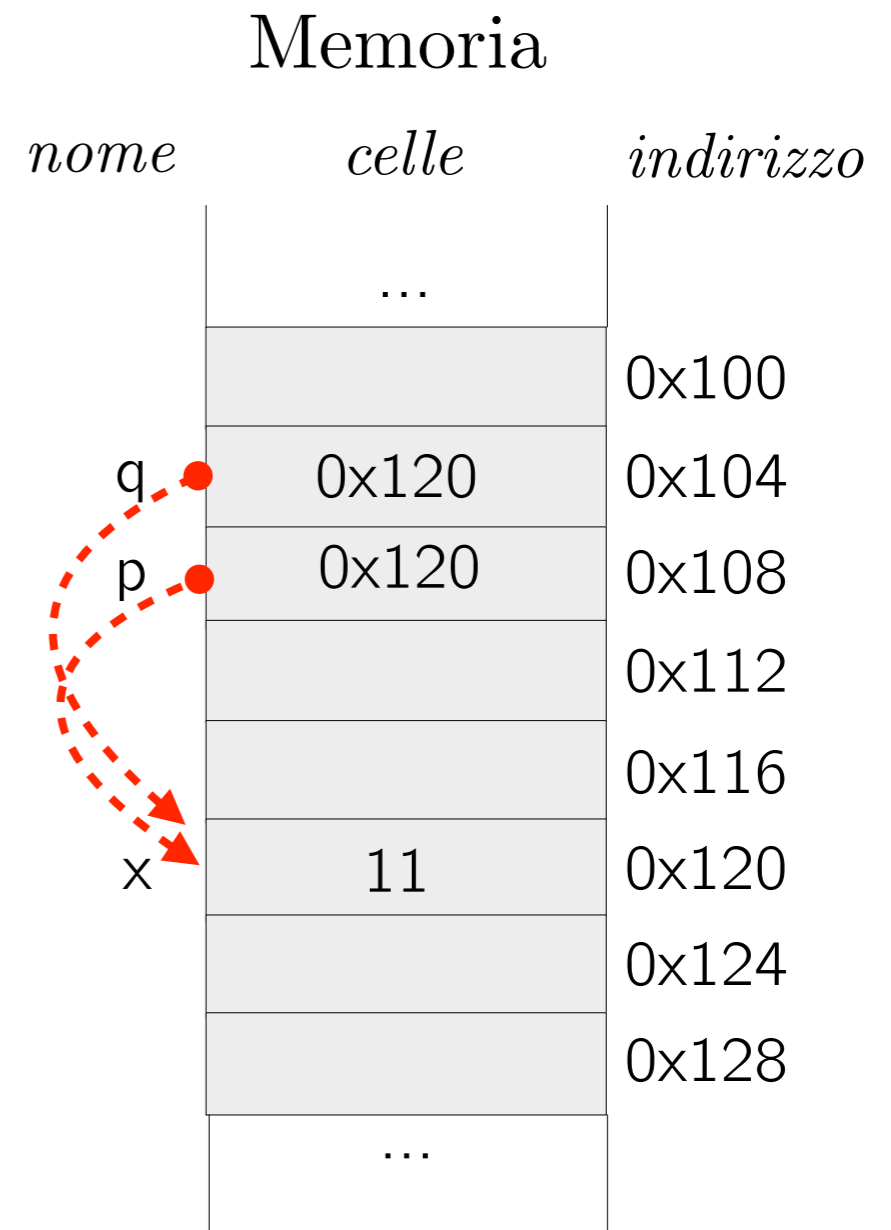
NO! q non è inizializzato.
Segmentation Fault!

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

Errore di tipo

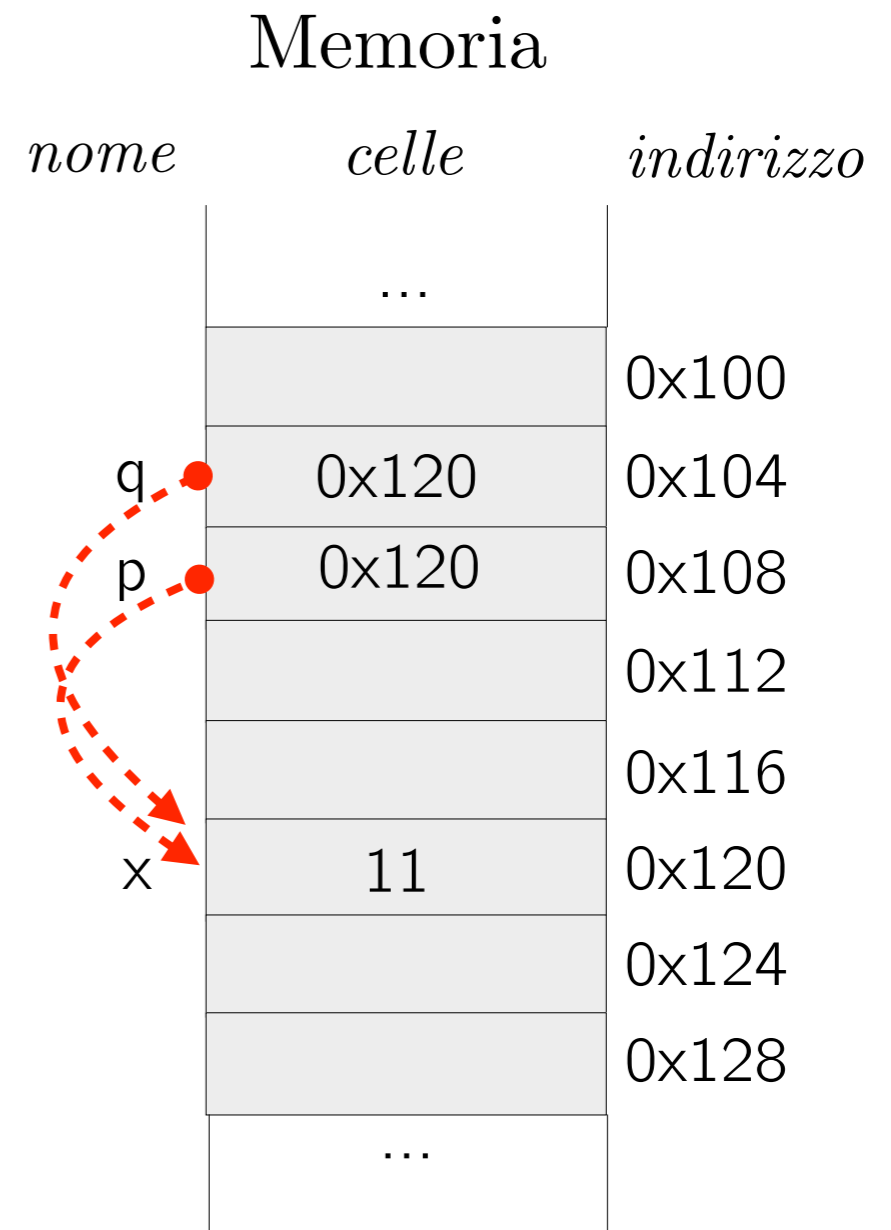
Errore di tipo

Equivalente al precedente



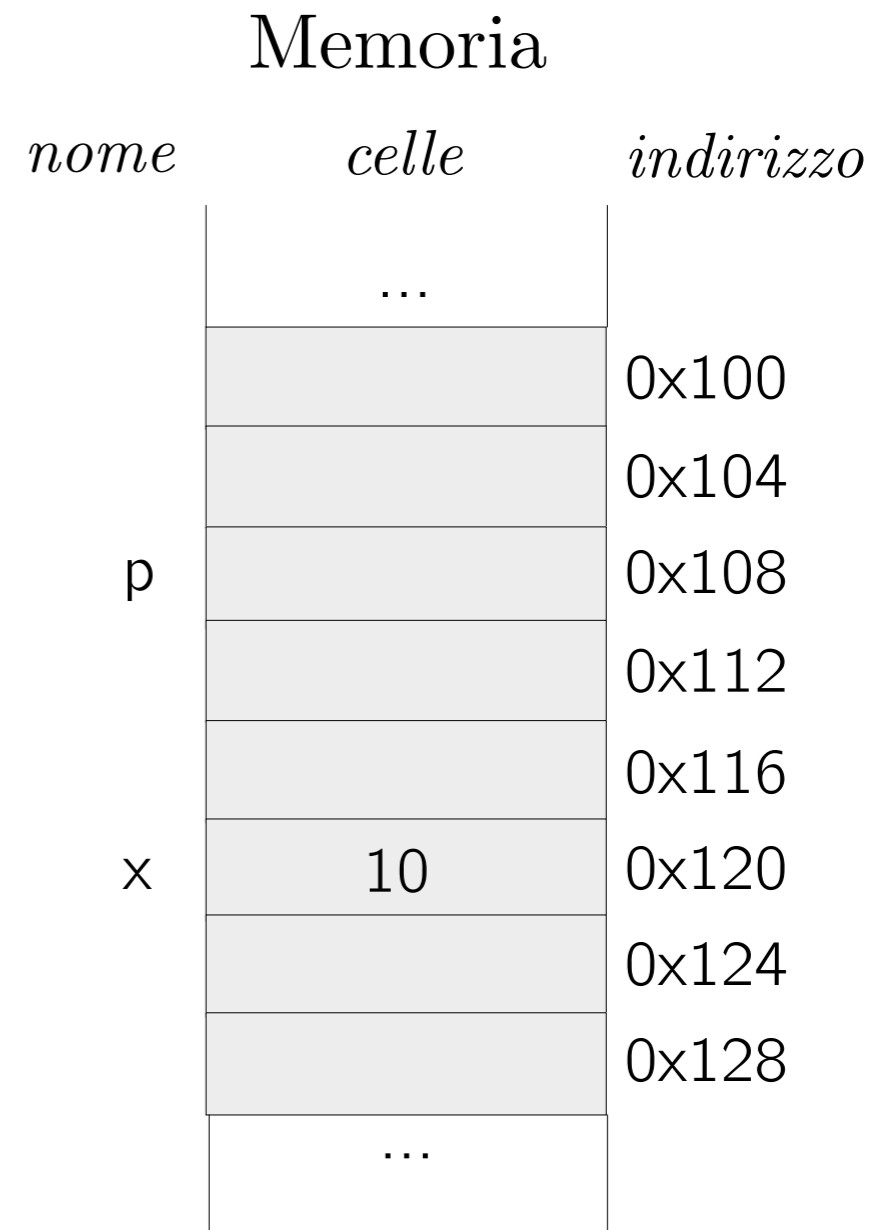
Puntatori (2)

```
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;
printf("%p %d %d", q, *q, *p);
```



Puntatori (3)

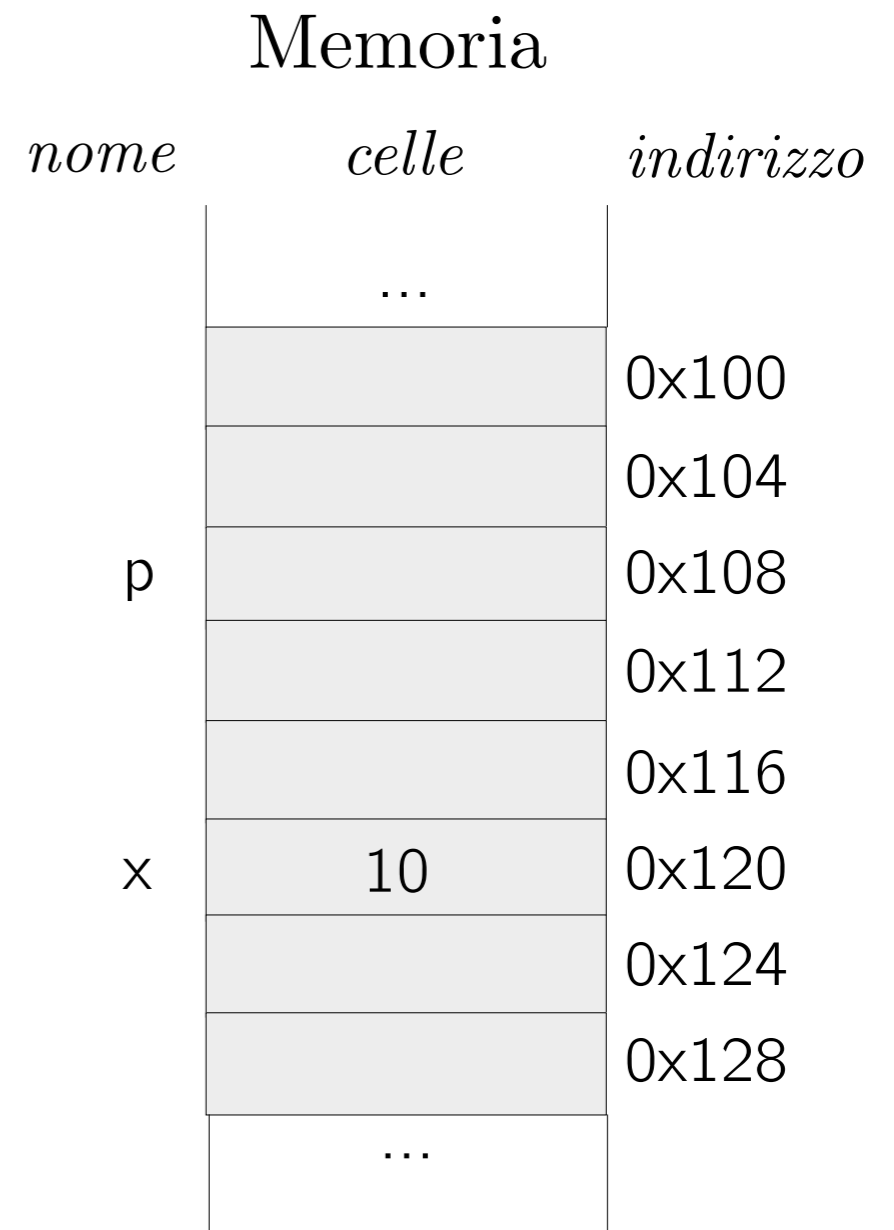
```
int x = 10;  
int *p; // dichiara un puntatore ad intero  
p = &x;
```



Puntatori (3)

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

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

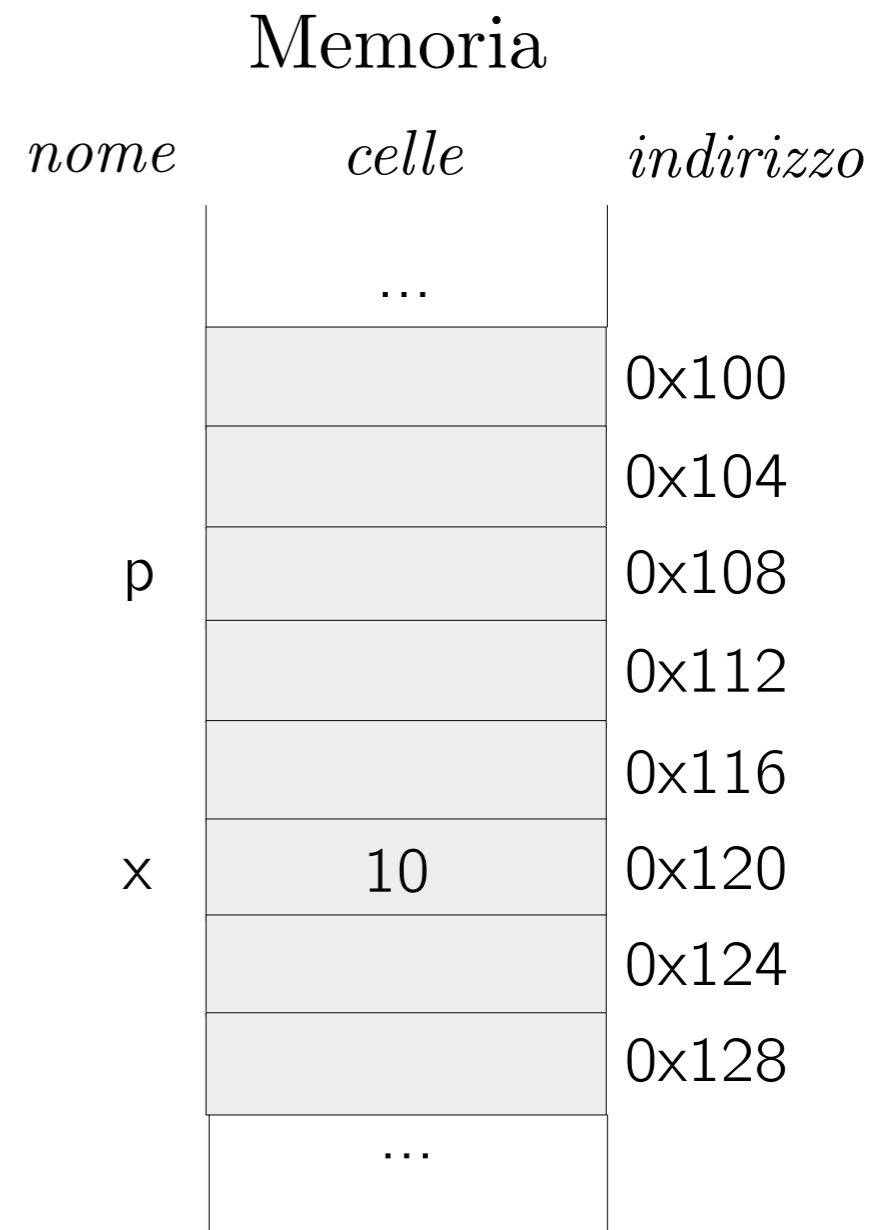


Puntatori (3)

```
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;
```



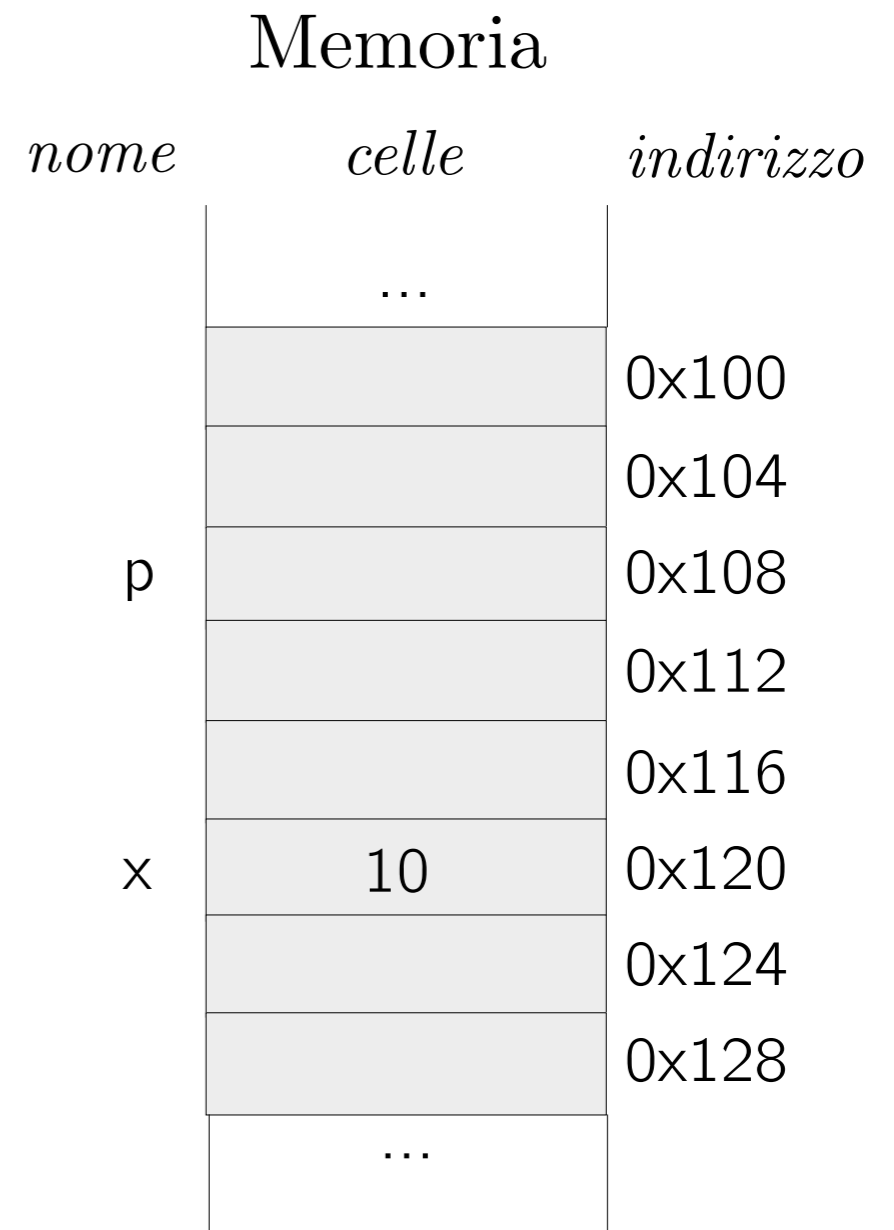
Puntatori (3)

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

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

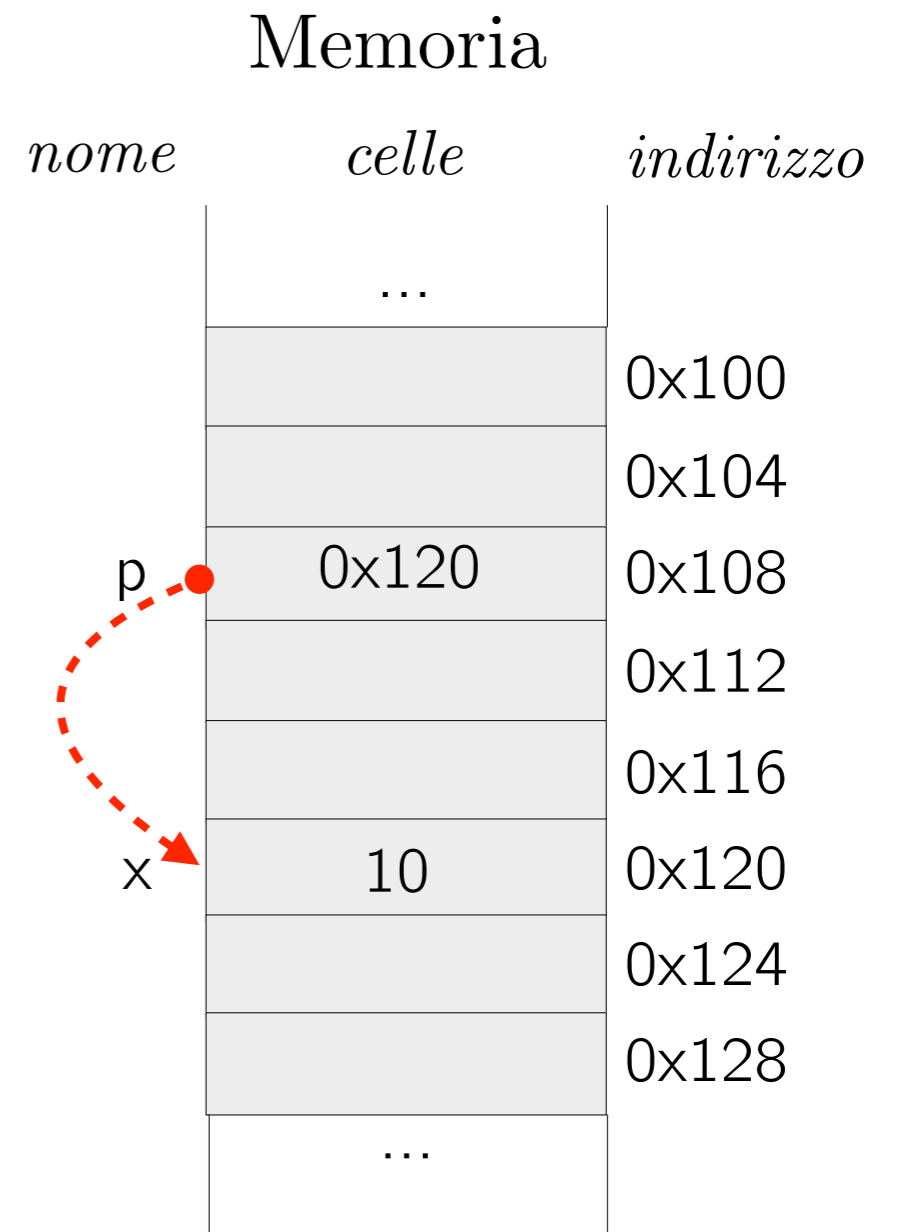
```
float x = 3.14f;  
float *p = &x;
```

```
double x = 3.141f;  
double *p = &x;
```



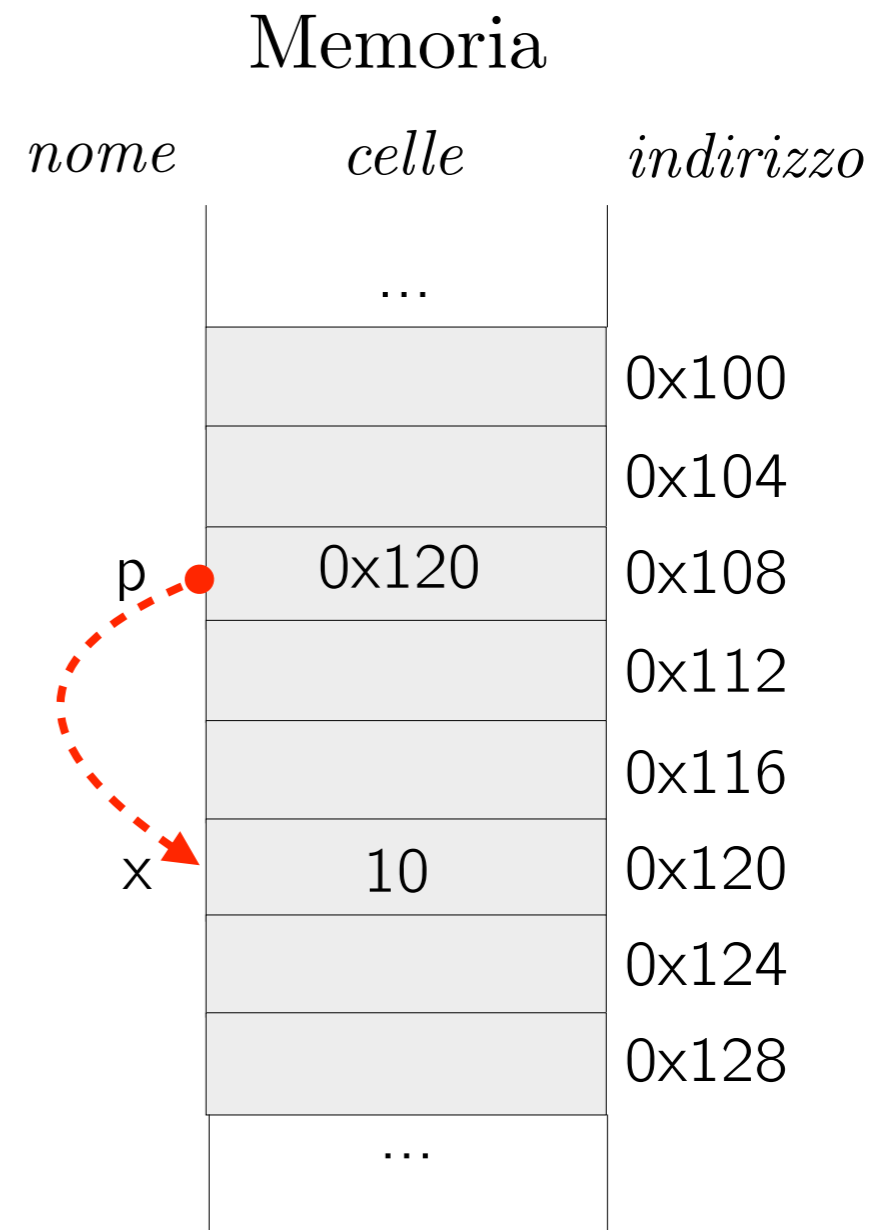
Puntatori (4)

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



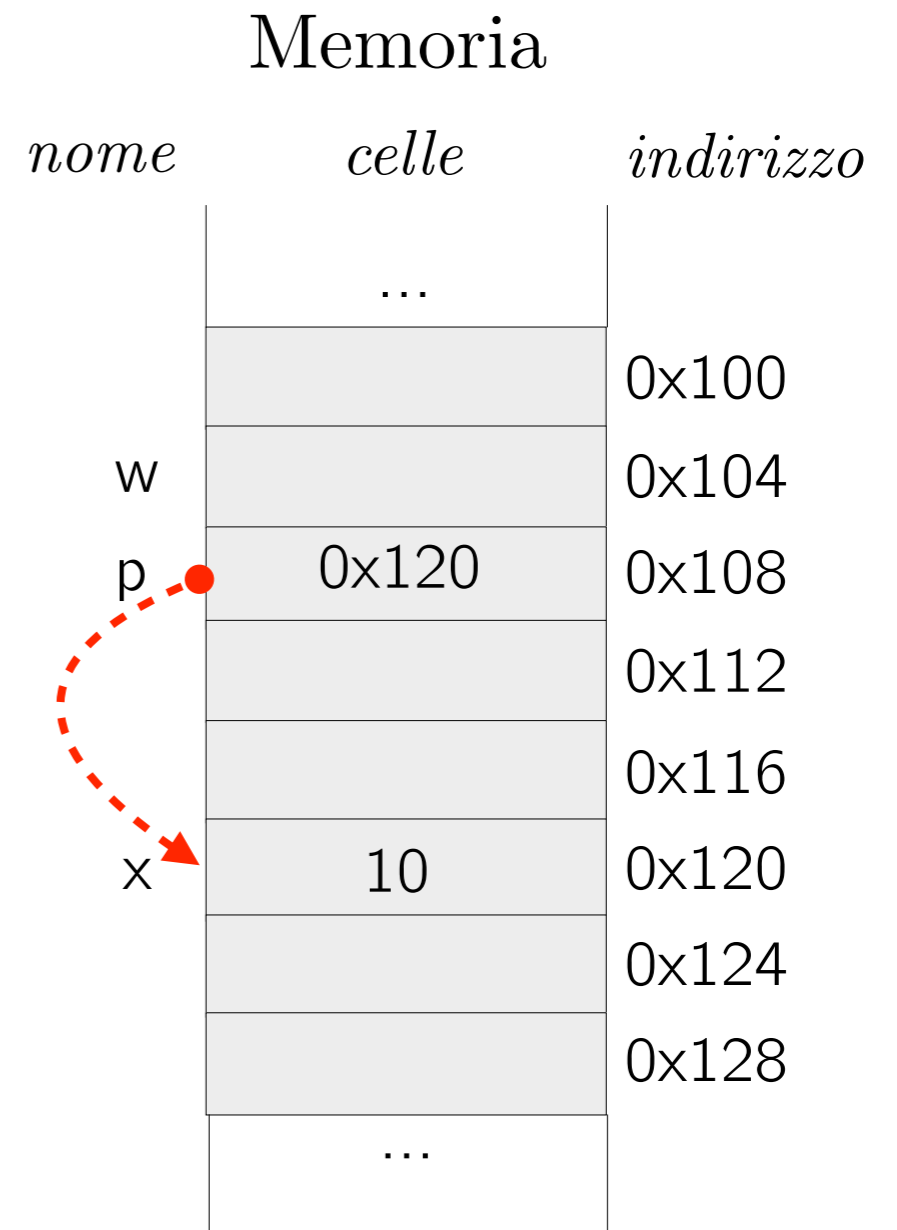
Puntatori (4)

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



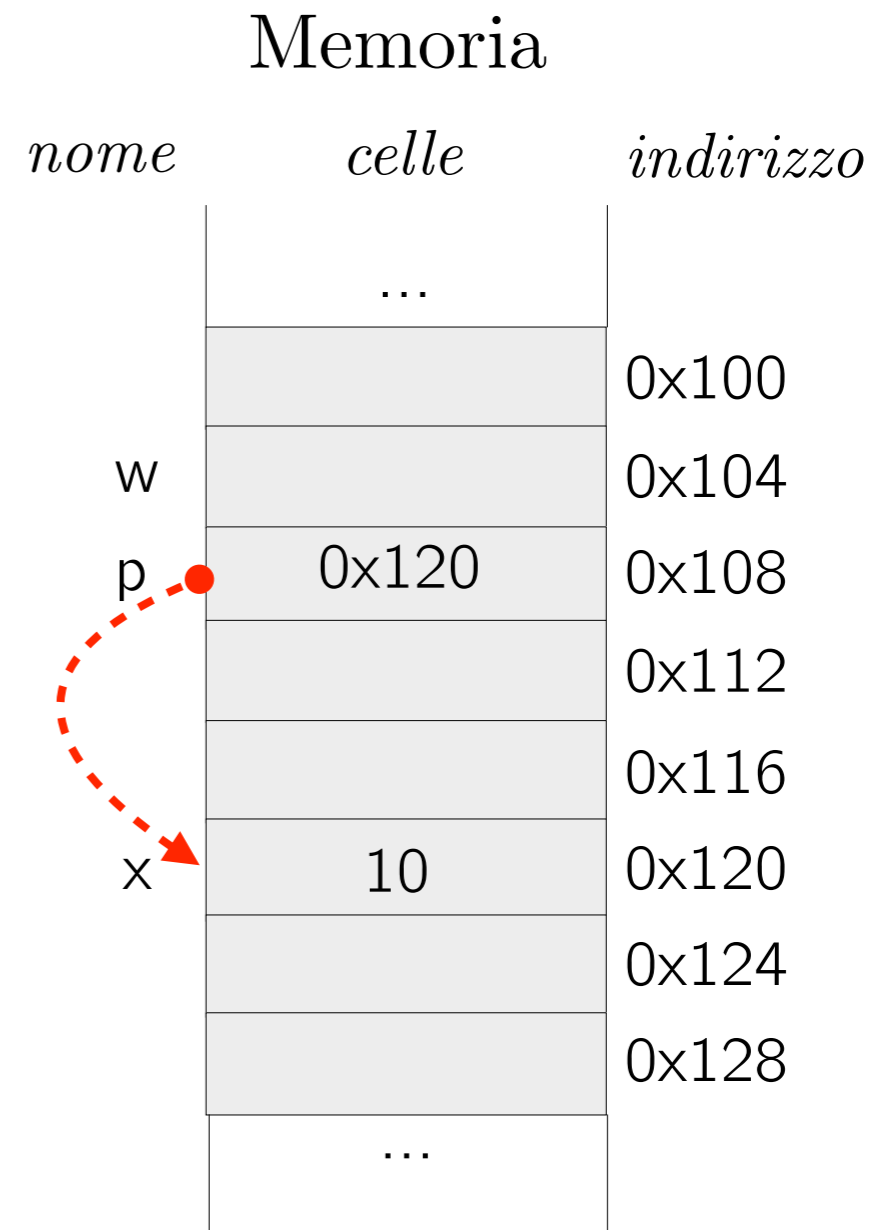
Puntatori (4)

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



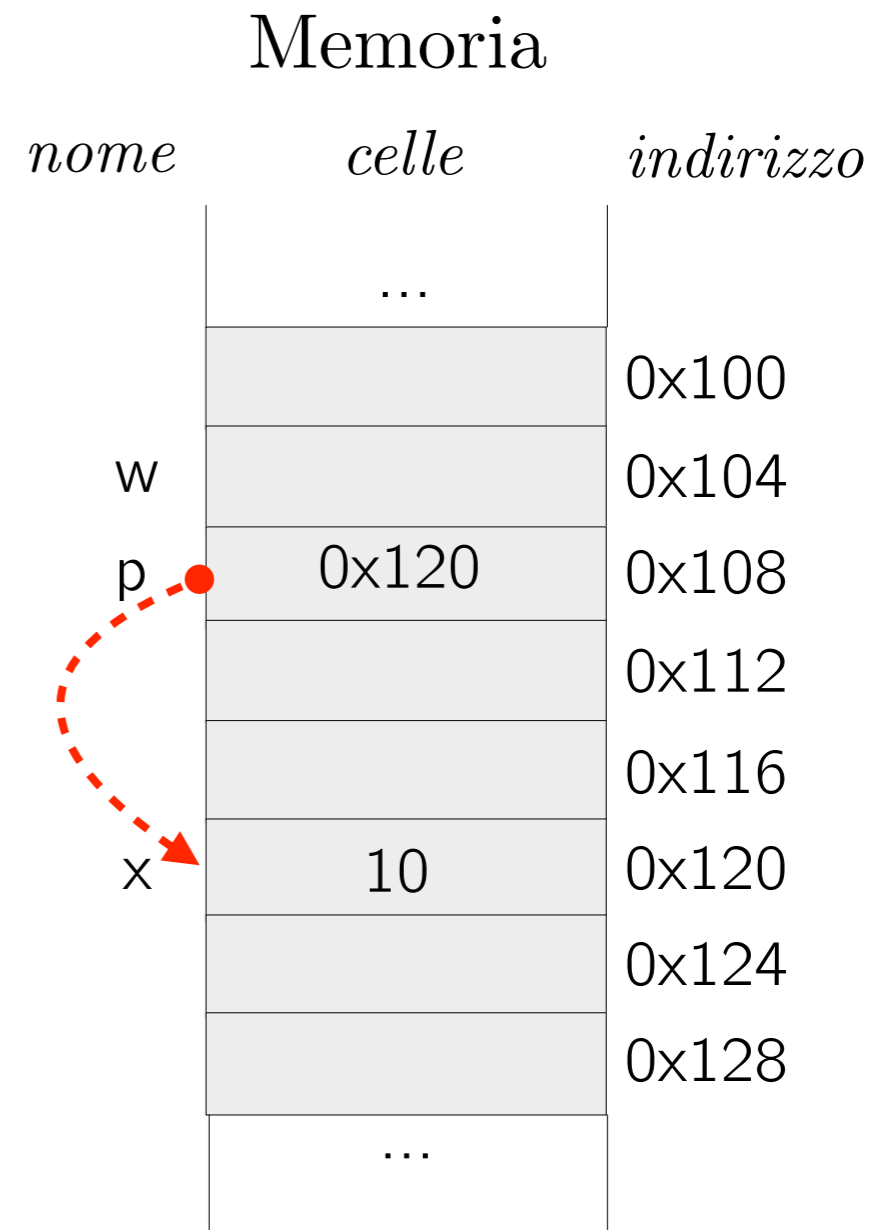
Puntatori (4)

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



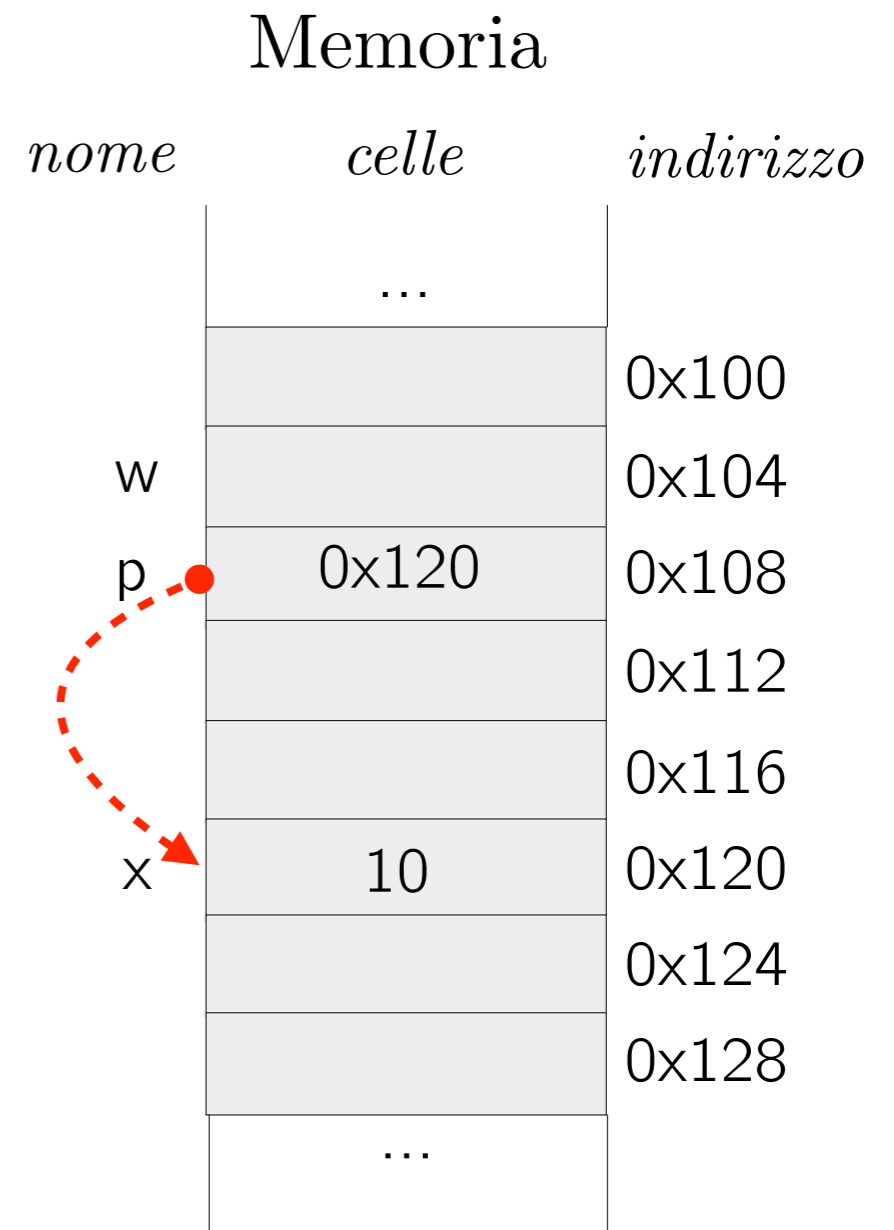
Puntatori (4)

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



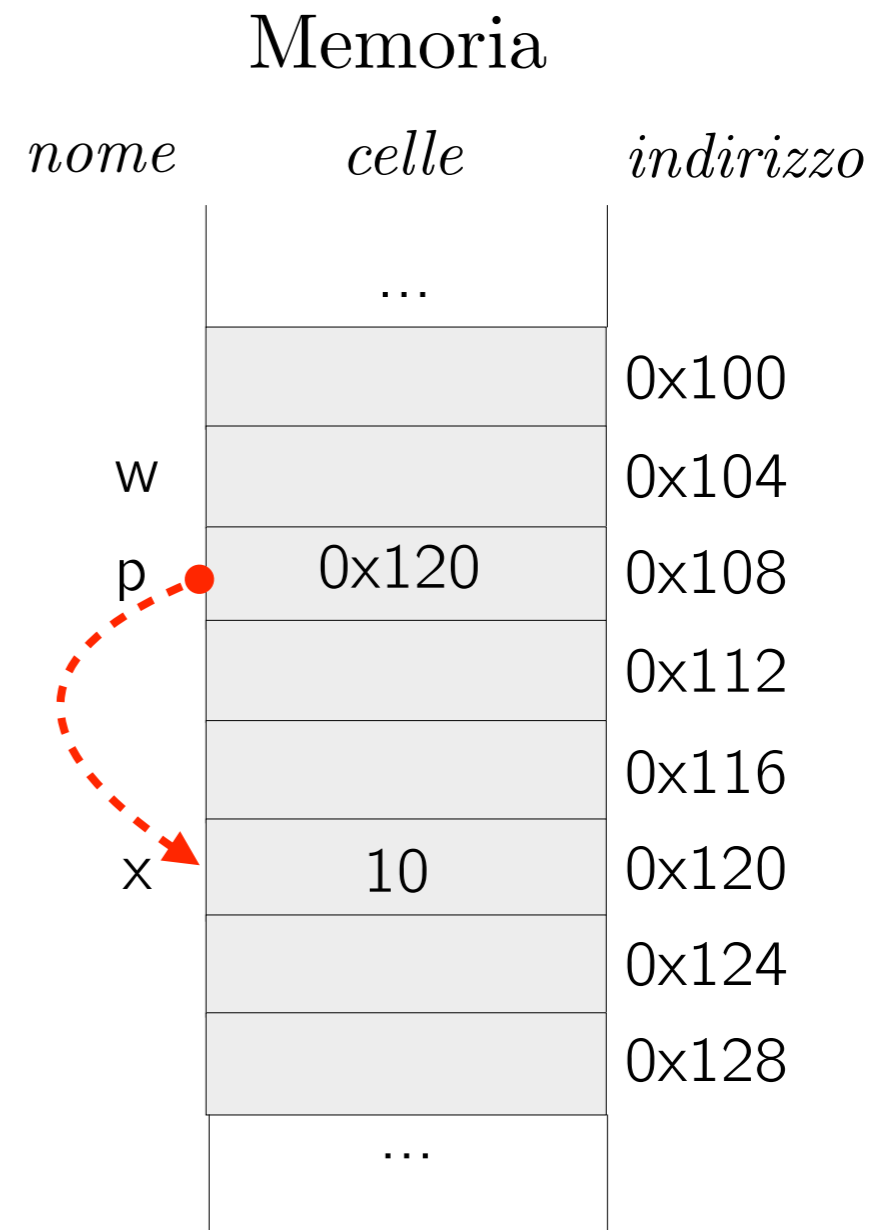
Puntatori (4)

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



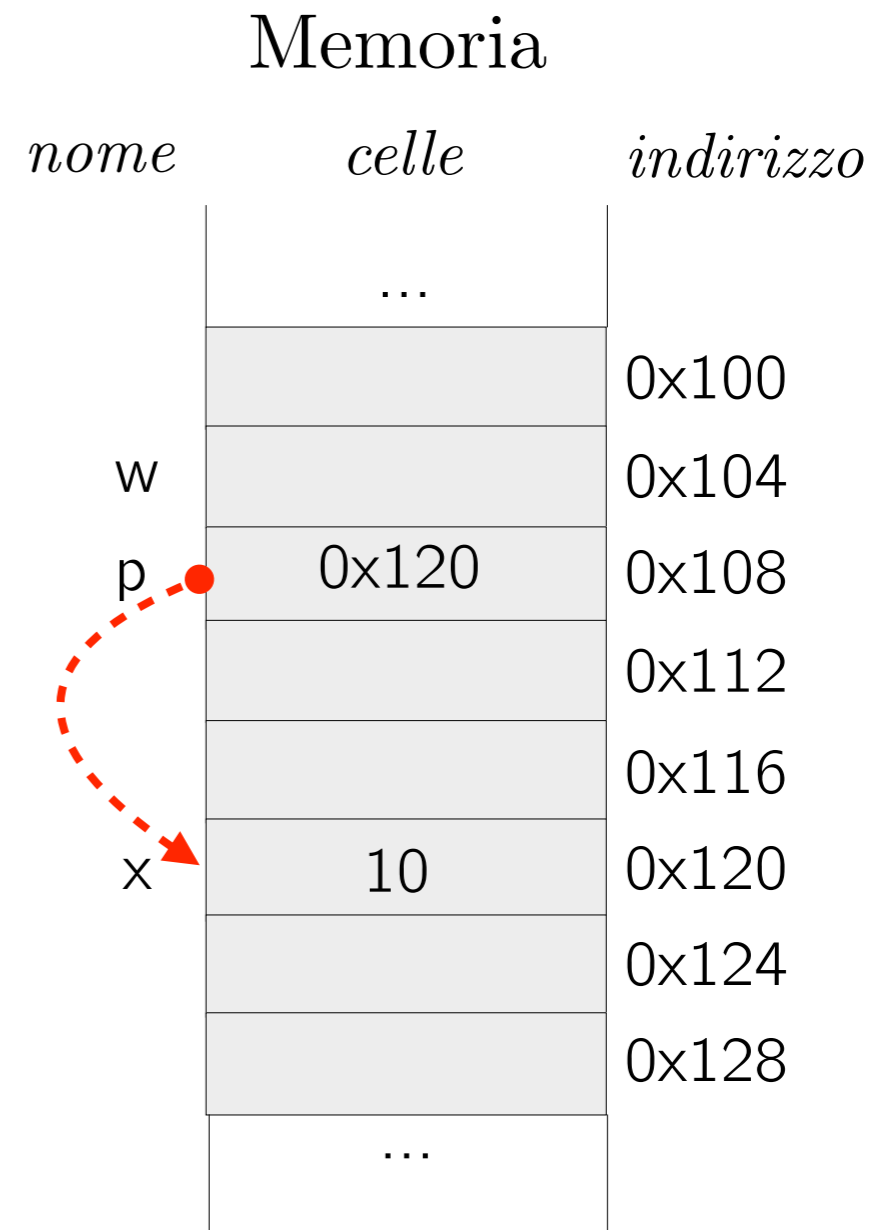
Puntatori (4)

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



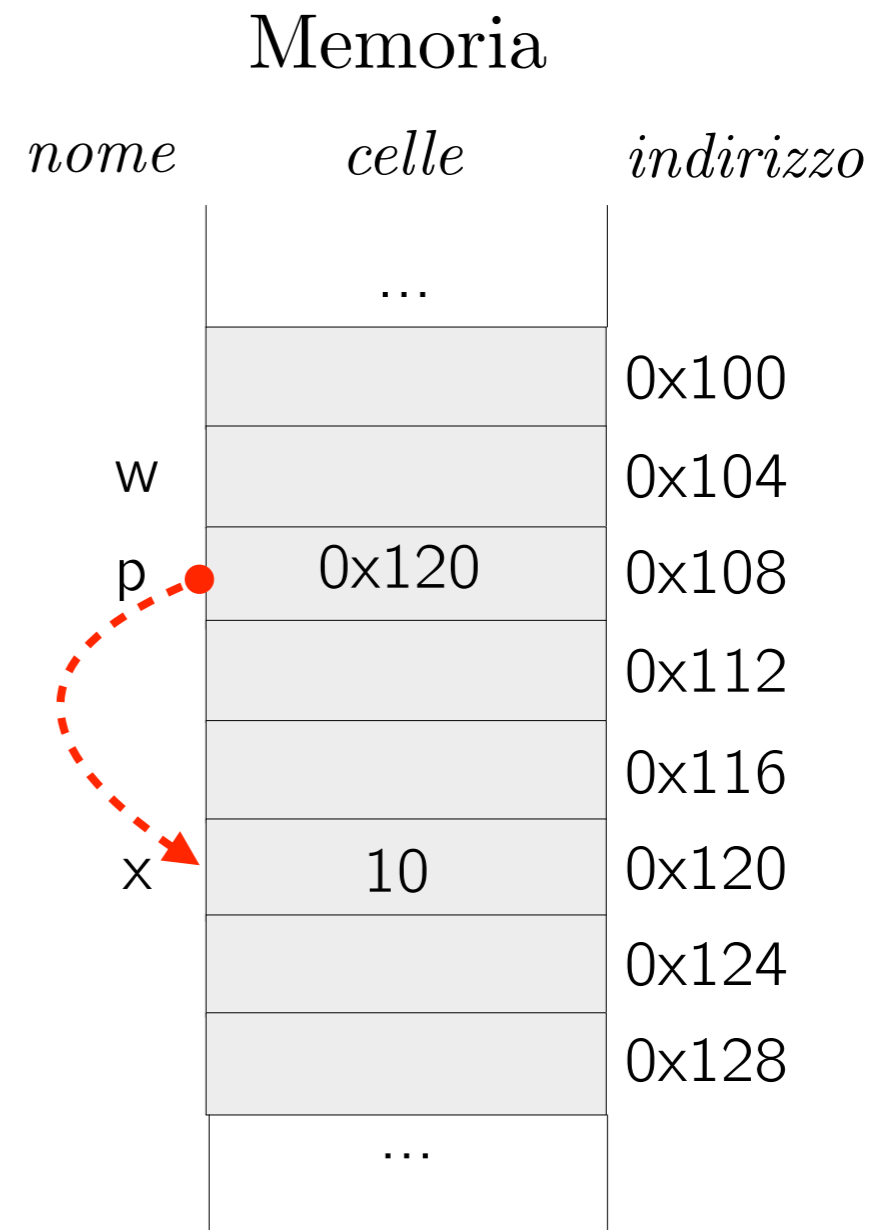
Puntatori (4)

```
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
```



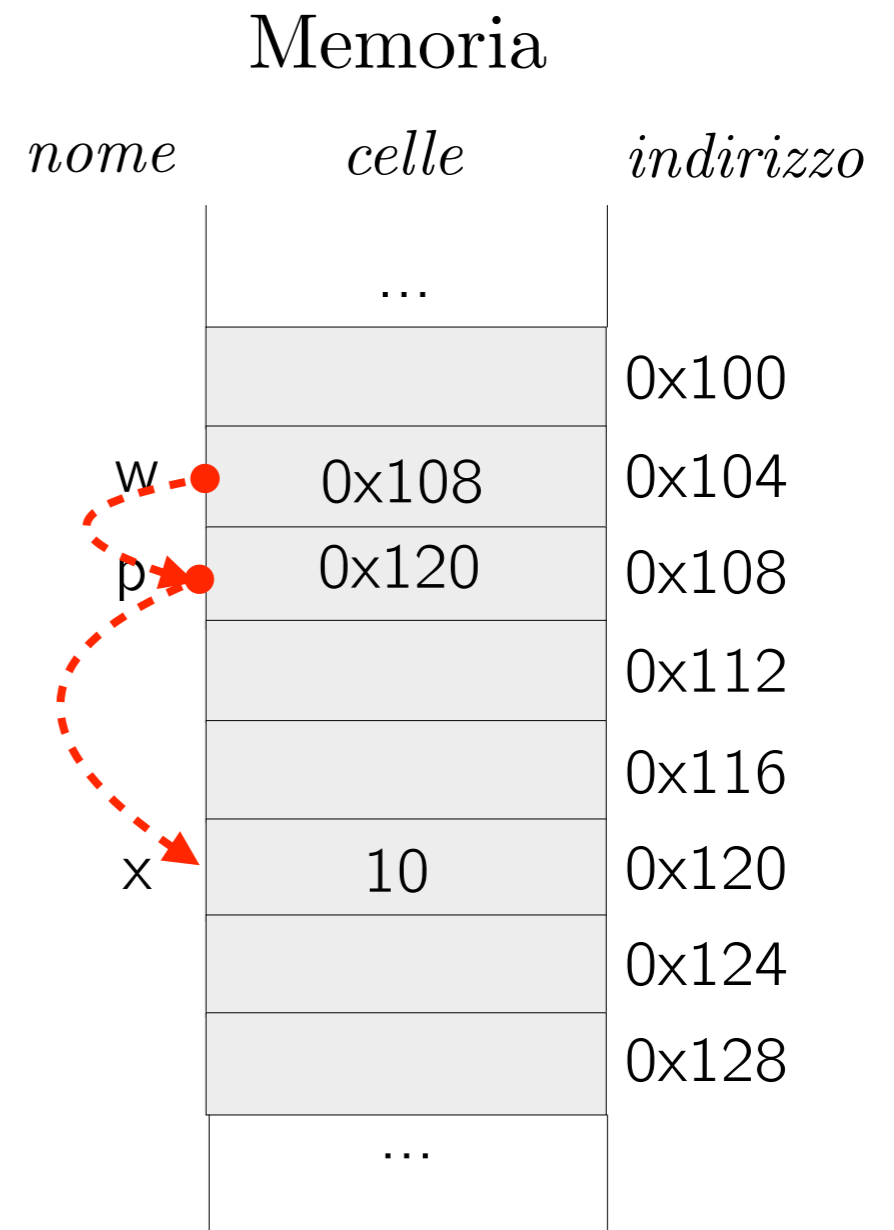
Puntatori (4)

```
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;
```



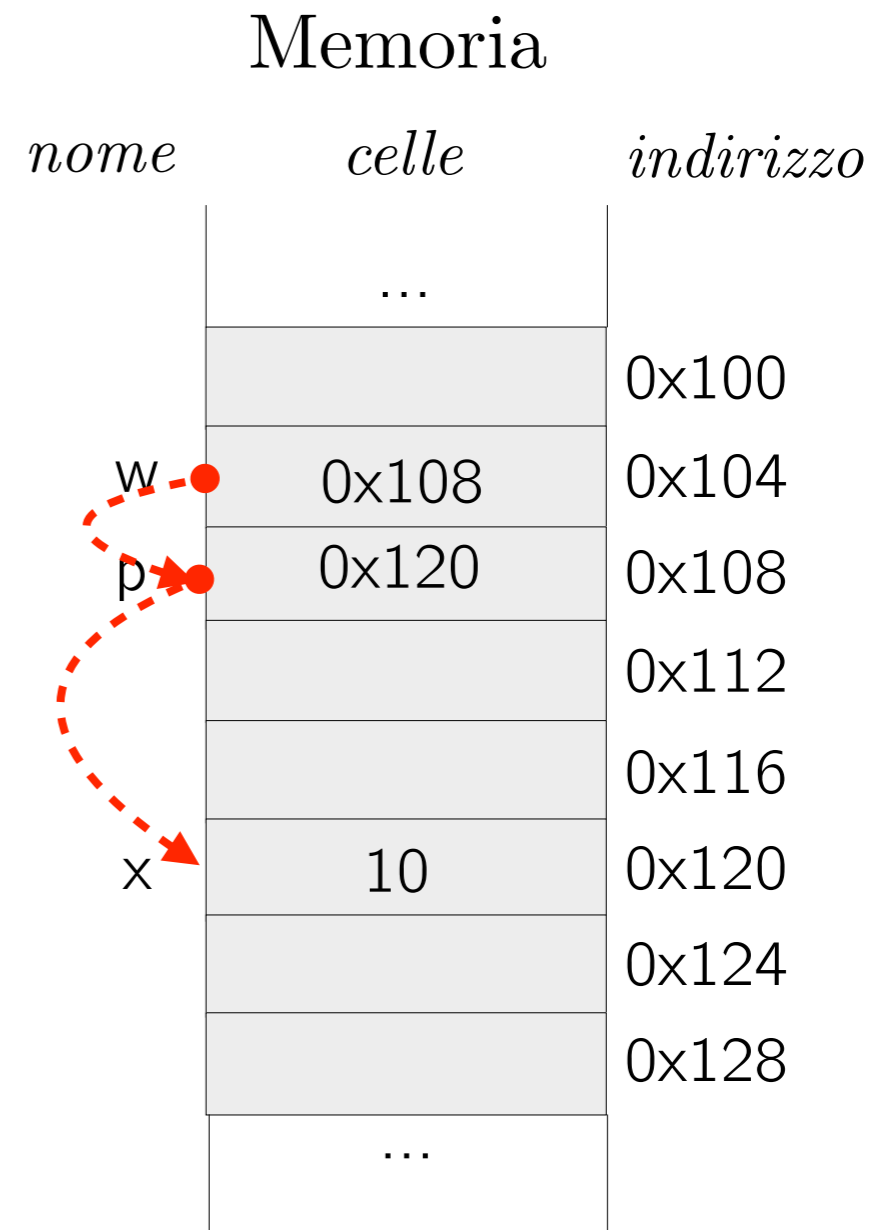
Puntatori (4)

```
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;
```



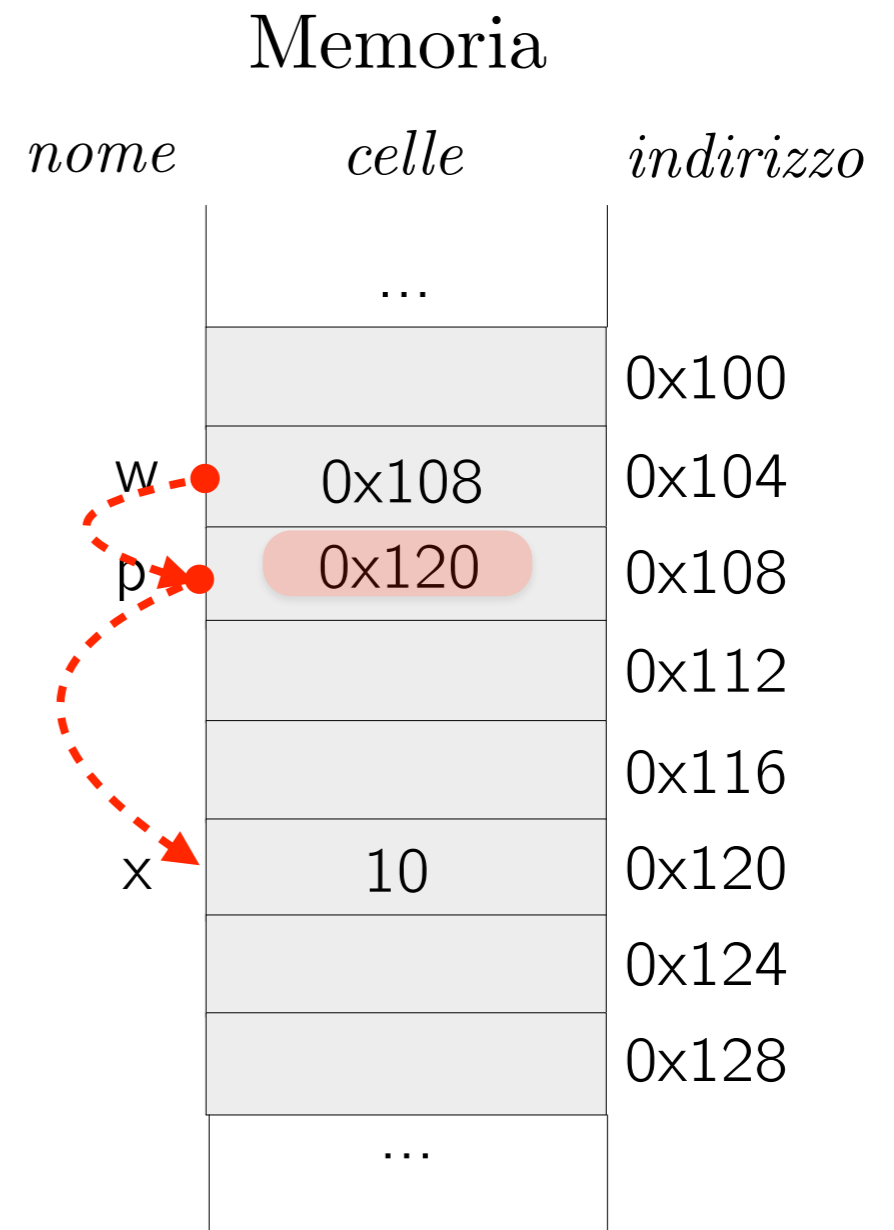
Puntatori (4)

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



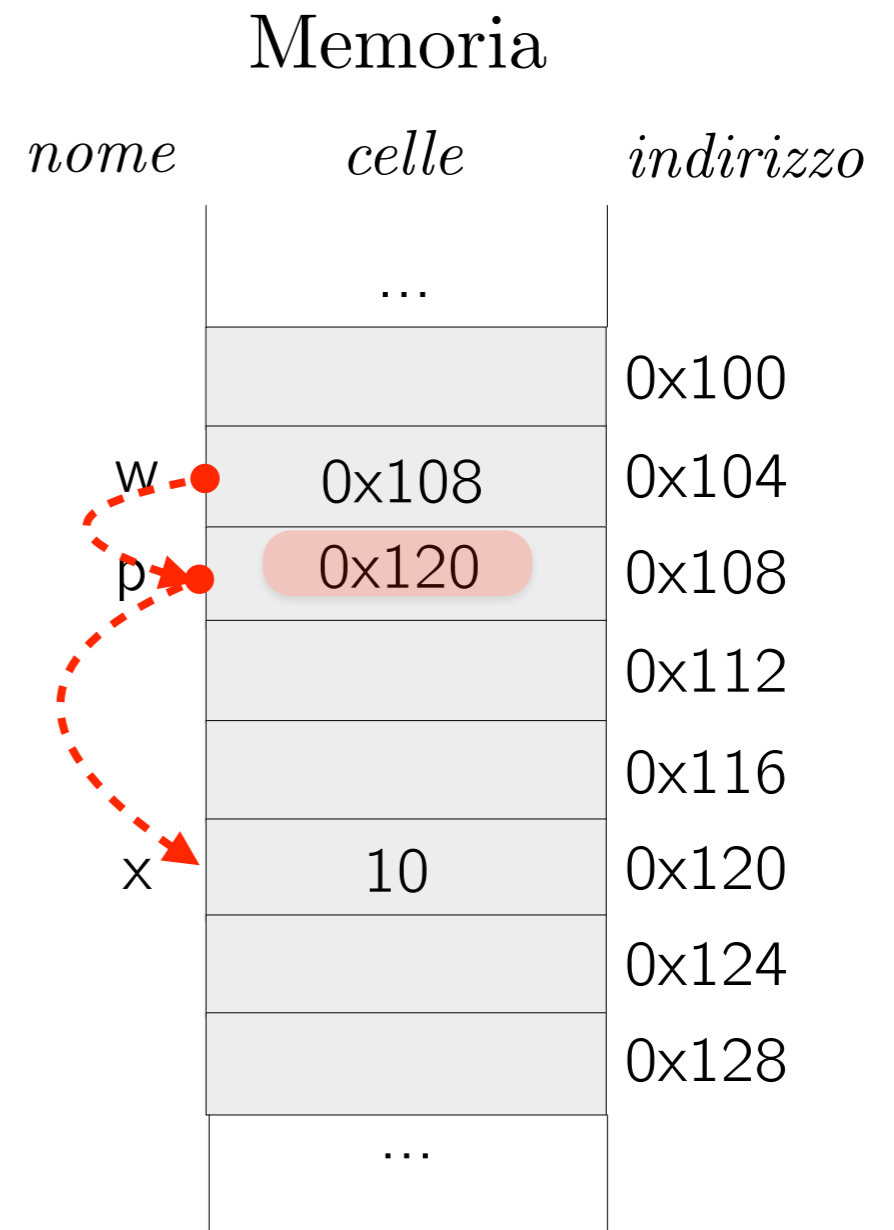
Puntatori (4)

```
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
```



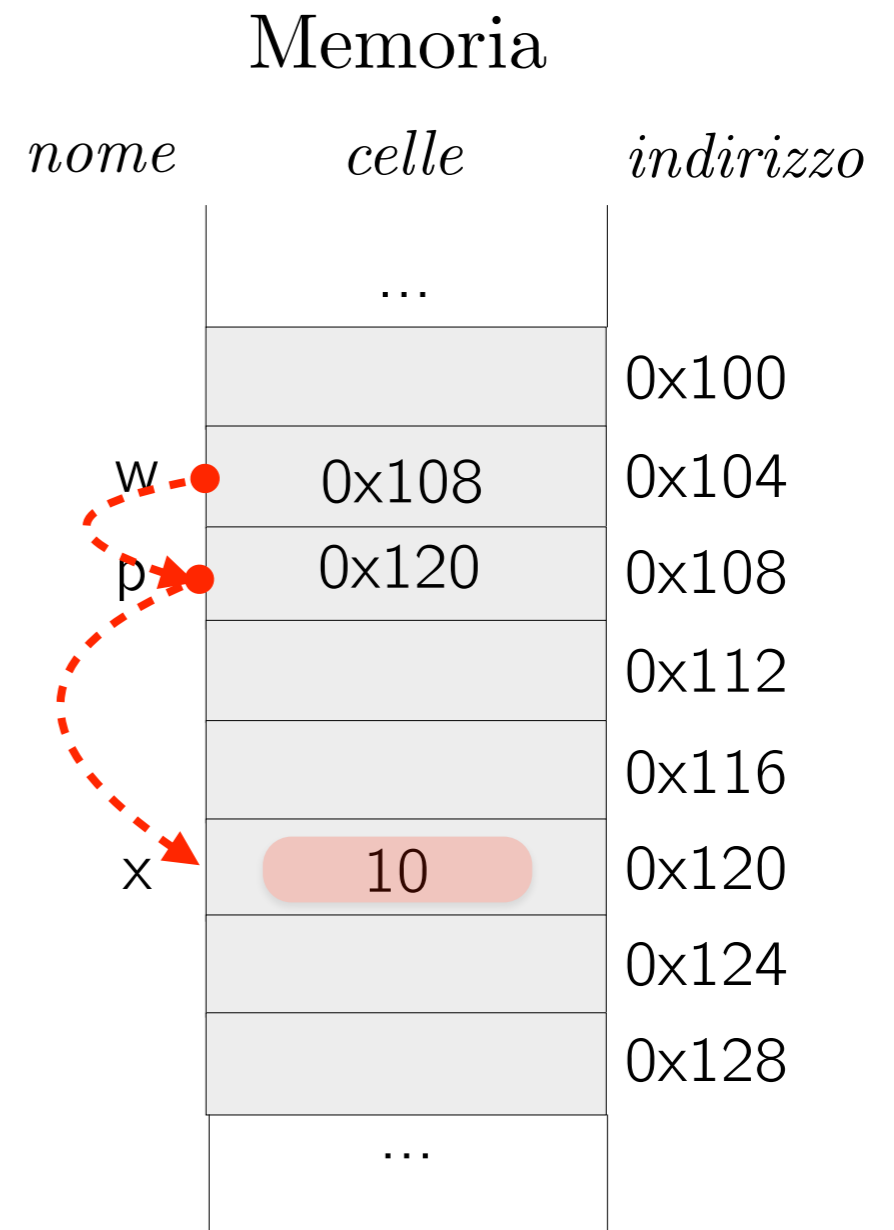
Puntatori (4)

```
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
```



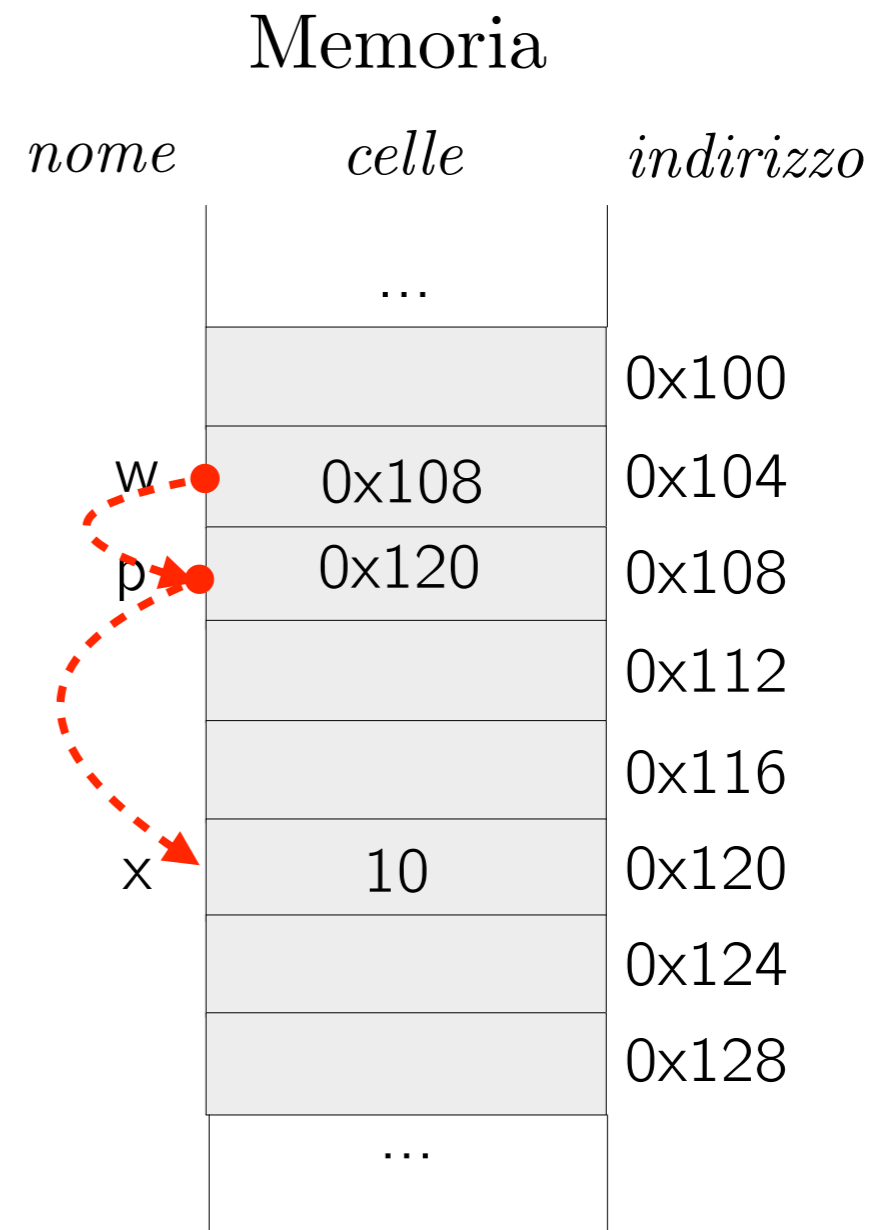
Puntatori (4)

```
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
```



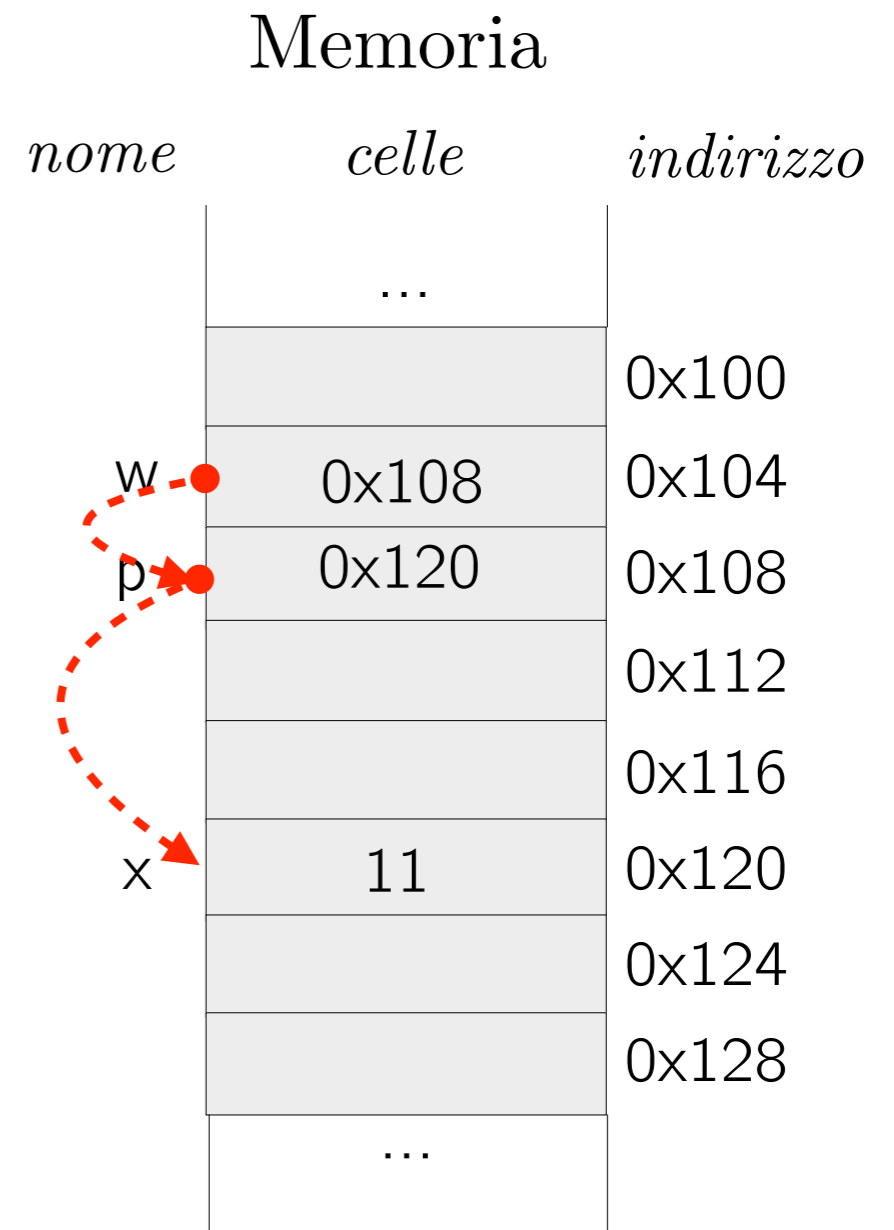
Puntatori (4)

```
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
```



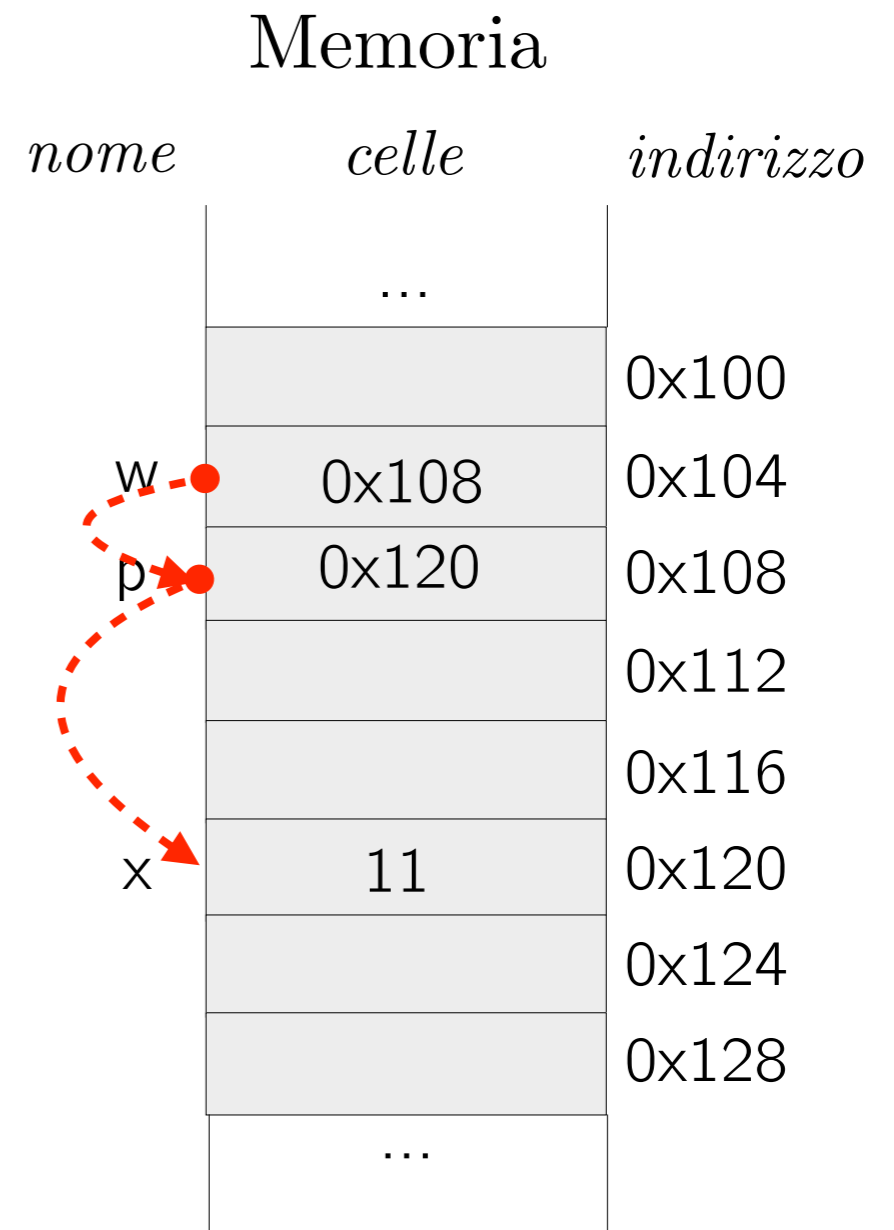
Puntatori (4)

```
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
```



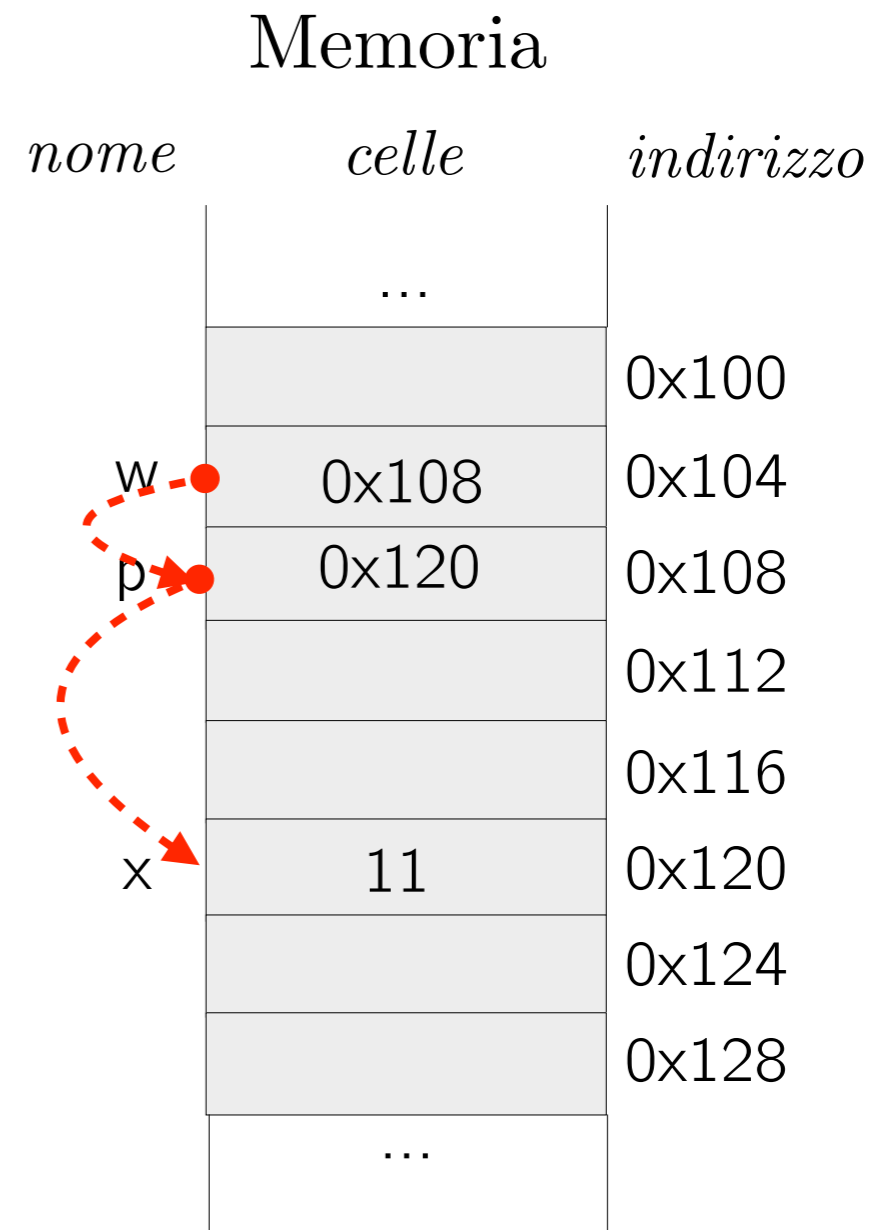
Puntatori (4)

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



Puntatori (4)

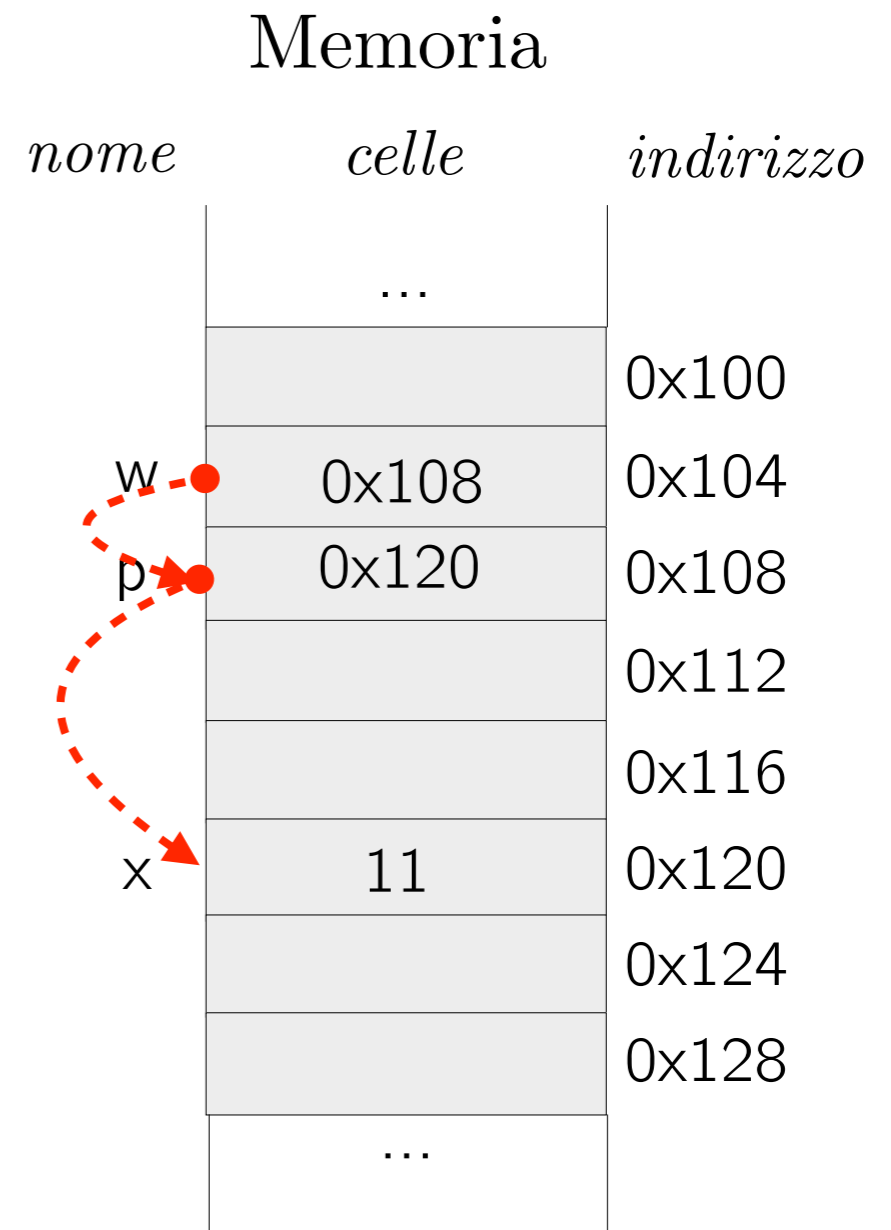
```
int x = 10;
int *p; // dichiara un puntatore ad intero
p = &x;
int **w; Puntatore a puntatore ad intero
w = &x; Errore di tipo
w = p; Errore di tipo
w = &p;
*w
**w
**w = 11
int ***z; Puntatore a puntatore
a puntatore ad intero
```



Puntatori (4)

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

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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotata di variabili 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;
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotata di un proprio spazio di memoria per le variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

la 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;  
}
```

<i>nome</i>	<i>celle</i>	<i>indirizzo</i>
	...	
		0x100
		0x104
		0x108
		0x112
		0x116
		0x120
		0x124
		0x128
	...	

Funzioni (5)

Ogni funzione è dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri
La funzione riceve
modifiche *non* si propagano

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

cellule
indirizzo

```
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 dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri
La funzione riceve
modifiche *non* si propagano

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

cella
indirizzo

```
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 dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri
La funzione riceve
modifiche *non* si propagano

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

cellule
indirizzo

```
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 dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dall'esterno
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri
La funzione riceve
modifiche *non* si propagano

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

cellule
indirizzo

```
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 dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri
La funzione riceve
modifiche *non* si propagano

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

cellule
indirizzo

```
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 dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri alla funzione avviene attraverso le celle di memoria. La funzione riceve copie locali di questi parametri. Le modifiche *non* si propagano al chiamante.

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamante 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	x	0x120	0x100
y	y	0x124	0x104
			0x108
			0x112
			0x116
x	x	10	0x120
y	y	5	0x124
			0x128
		...	

Funzioni (5)

Ogni funzione è dotata di un proprio ambiente di esecuzione (e variabili locali (e parametri))

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dall'ambiente chiamante
- allocate/deallocate all'interno della funzione

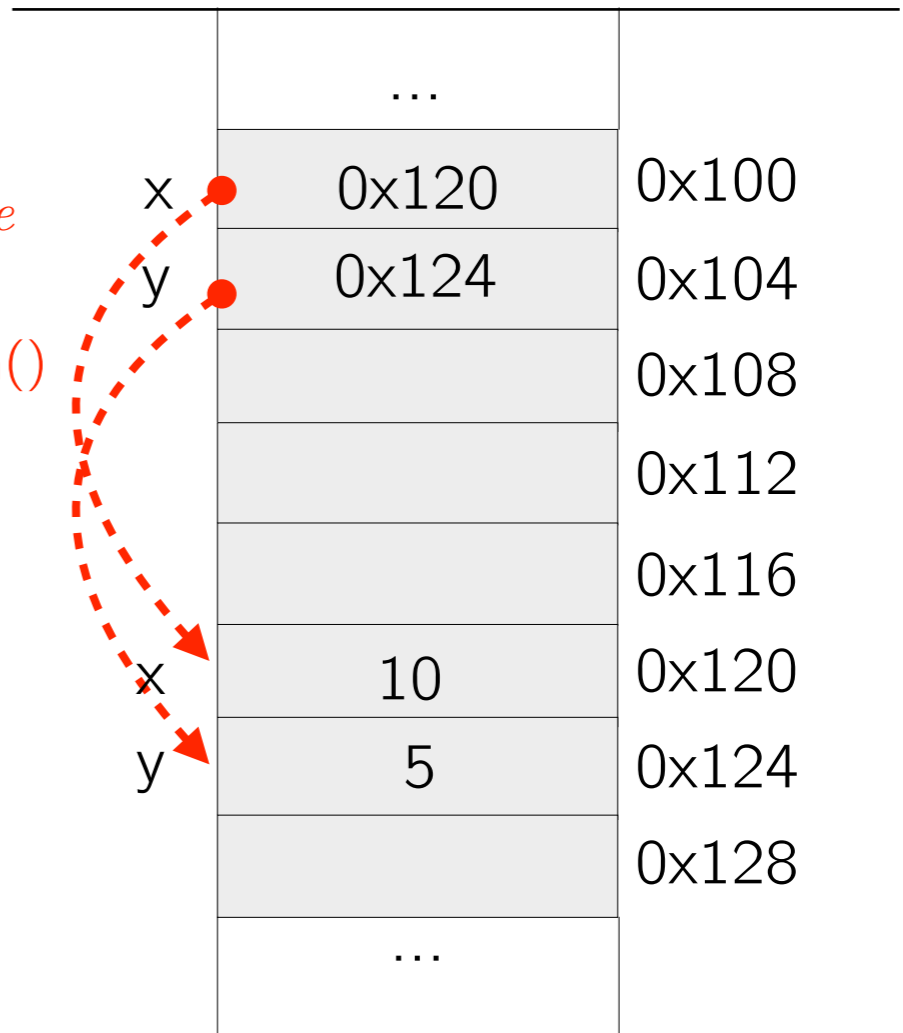
Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri avviene attraverso le celle di memoria. La funzione riceve copie locali delle celle di memoria. Le modifiche *non* si propagano all'ambiente chiamante.

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamante 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()



Funzioni (5)

Ogni funzione è dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri alla funzione avviene attraverso le celle di memoria. La funzione riceve copie locali delle variabili. Le modifiche *non* si propagano al chiamante.

Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamante 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	10	0x120
y	5	0x124
		0x128
...		

Funzioni (5)

Ogni funzione è dotata di un proprio ambiente di esecuzione (e variabili locali (e parametri))

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dall'ambiente chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri avviene per valore. La funzione riceve copie locali delle modifiche *non* si propagano

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

alla funzione

ntuali

celle

indirizzo

```
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	10	0x120
y	5	0x124
		0x128
...		

Funzioni (5)

Ogni funzione è dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dall'esterno
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri avviene attraverso le celle di memoria. La funzione riceve copie locali delle celle. Le modifiche *non* si propagano al chiamante.

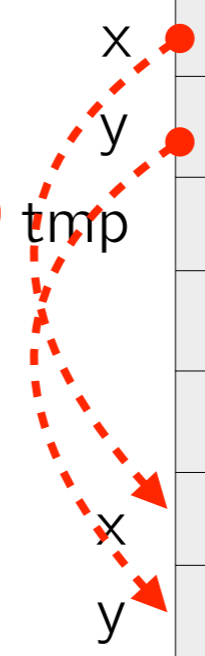
Si passa un puntatore alla variabile anziché il suo valore. In questo modo il chiamante può modificarne il contenuto.

```
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 dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dal chiamante
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri

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

La funzione riceve le modifiche *non* si propagano

alla funzione

realtà

celle

indirizzo

```
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;  
}
```

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

Funzioni (5)

Ogni funzione è dotata di variabili locali (e parametri)

Come può una funzione propagare le modifiche al chiamante?

- *non* visibili dall'esterno
- allocate/deallocate alla funzione

Simulando il passaggio per riferimento con l'uso dei puntatori

Il passaggio dei parametri
La funzione riceve
modifiche *non* si propagano

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

...
ntuali
celle
indirizzo

```
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
		...	

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
```

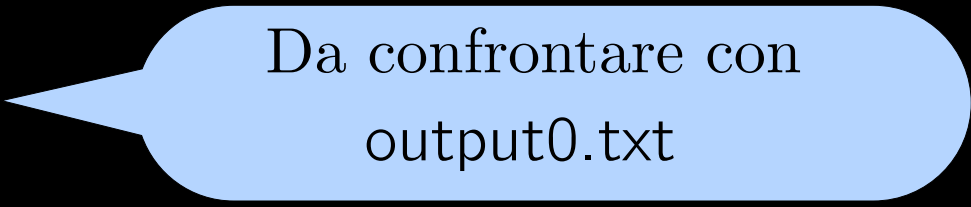
```
2
```

```
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
```

Da confrontare con
output0.txt

```
$ ./inverti < input0.txt
```

Testing

Compilazione, esecuzione e testing

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

```
$ ./inverti
```

```
2
```

```
3
```

```
4
```

```
3
```

```
4
```

Da confrontare con
output0.txt

```
$ ./inverti < input0.txt
```

```
3
```

```
4
```

Testing

Compilazione, esecuzione e testing

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

```
$ ./inverti
```

```
2
```

```
3
```

```
4
```

```
3
```

```
4
```

Da confrontare con
output0.txt

```
$ ./inverti < input0.txt
```

```
3
```

```
4
```

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


Testing

Compilazione, esecuzione e testing

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

```
$ ./inverti
```

```
2
```

```
3
```

```
4
```

```
3
```

```
4
```

Da confrontare con
output0.txt

```
$ ./inverti < input0.txt
```

```
3
```

```
4
```

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

```
1d0
```

```
< 3
```

```
2a2
```

```
> 3
```

Testing

Compilazione, esecuzione e testing

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

```
$ ./inverti
```

```
2
```

```
3
```

```
4
```

```
3
```

```
4
```

Da confrontare con
output0.txt

```
$ ./inverti < input0.txt
```

```
3
```

```
4
```

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

```
1d0
```

```
< 3
```

```
2a2
```

```
> 3
```

Niente in output se
l'output è corretto

Esercizio 1

Ciao Mondo!

Esercizio

Scrivere un programma che stampi `Ciao Mondo!` sul terminale.

Esempio

Input

Output

Ciao Mondo!

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 contenente l'intero x .

Il programma stampa in output 1 se x è primo, 0 altrimenti.

Esempi

Input

226

Output

0

Input

13

Output

1

Esercizio 3

Somma

Esercizio

Scrivere un programma che legga da tastiera una sequenza di numeri interi terminata da 0 e ne stampi la somma.

L'input una sequenza di numeri interi terminata da 0, un intero per riga.

Il programma stampa in output la somma degli interi.

Esempi

Input

3
9
1
2
3
0

Output

18

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 5

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

6

Output

36

Esercizio 6

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 7

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 8

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.