

Puntatori

Marco Alberti



Dipartimento
di Matematica
e Informatica



Università
degli Studi
di Ferrara

Programmazione e Laboratorio, A.A. 2020-2021

Ultima modifica: 30 novembre 2020

Attenzione! Questo materiale didattico è per uso personale dello studente ed è coperto da copyright.
Ne sono vietati la riproduzione e il riutilizzo anche parziale, ai sensi e per gli effetti della legge sul diritto d'autore.

Sommario

1 Puntatori

2 Passaggio per riferimento

3 Array e puntatori

4 Aritmetica dei puntatori

Accesso alla memoria

La memoria è una sequenza di celle identificate da un numero (indirizzo).

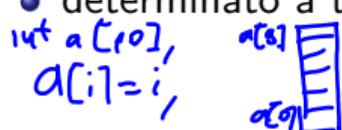
Sappiamo che abbiamo accesso alle celle che abbiamo riservato: variabili globali, variabili locali, parametri formali.

Gli indirizzi di memoria accessibili sono determinati una volta per tutte dal compilatore (per le variabili globali) o alla creazione del record di attivazione (per variabili locali e parametri).

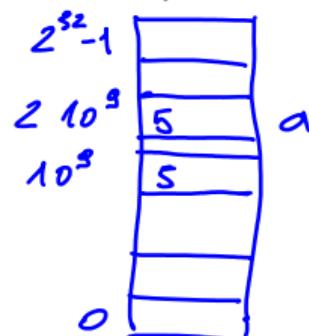
Possiamo decidere il valore di una variabile (tramite assegnamento), ma non il suo indirizzo¹.

E se volessimo aver accesso

- a un qualsiasi indirizzo della memoria, $a = 5;$
- determinato a tempo di esecuzione?



$\text{int } f()\{$
 $\text{int } a;$
 $a = 5;$
 $\}$
 $\text{int } b,$



¹Nel caso degli array, lo stesso l-value (ad esempio $a[i]$) può indicare indirizzi diversi variando l'indice, ma sempre limitatamente allo spazio riservato per variabile array.

Sommario

1 Puntatori

2 Passaggio per riferimento

3 Array e puntatori

4 Aritmetica dei puntatori

Puntatori
`int * pa;`

`int * pa;`

`pa = 1500`

Definizione di puntatore .

`int pa;`

$\langle \text{definizione} \rangle ::= \langle \text{tipo} \rangle * \langle \text{identificatore} \rangle ;$

definisce una variabile $\langle \text{identificatore} \rangle$ che contiene l'indirizzo di un'area di memoria di tipo $\langle \text{tipo} \rangle$.

- Si dice che $\langle \text{identificatore} \rangle$ è un **puntatore a** $\langle \text{tipo} \rangle$.
- $*\langle \text{identificatore} \rangle$ denota l'area di memoria il cui indirizzo è contenuto in $\langle \text{identificatore} \rangle$.
(qui * è detto operatore di **dereferenziazione**)
- $*\langle \text{identificatore} \rangle$ è quindi una pseudo-variabile di cui possiamo determinare il valore (assegnando a $*\langle \text{identificatore} \rangle$ un'espressione di tipo $\langle \text{tipo} \rangle$) ma anche l'indirizzo (assegnando a $\langle \text{identificatore} \rangle$ un valore).
- I puntatori ci permettono quindi di accedere
 - a un qualsiasi indirizzo della memoria,
 - determinato a tempo di esecuzione.



`printf("%d", *pa);`

$$*\overset{(1500)}{\underset{1}{pa}} = 7$$

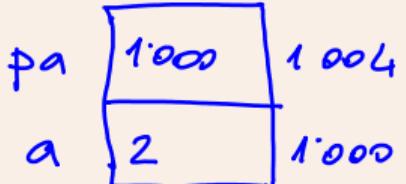
Esempio

Che cosa ha senso assegnare ad un puntatore a $\langle \text{tipo} \rangle$? Ad esempio, l'indirizzo di una variabile di tipo $\langle \text{tipo} \rangle$, ricavato tramite l'operatore indirizzo ($\&$).

110_puntatori/puntatore.c

```

1 #include <stdio.h>
2
3 int main() {
4     int a = 2;
5     int *pa;
6
7     pa = &a;
8     printf("%d\n", *pa);
9 }
```



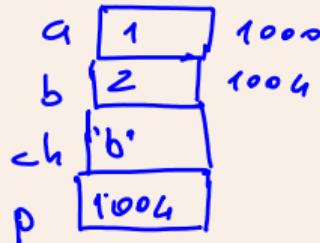
Area puntata come variabile configurabile

*p

110_puntatori/variabile-configurabile.c

```

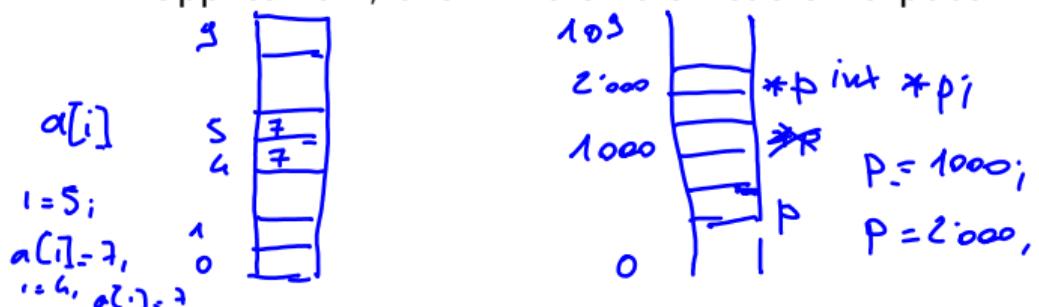
1 #include <stdio.h>
2
3 int main(void) {
4     int a, b;
5     char ch;
6     int* p;
7     printf("a o b?\n"); scanf("%c", &ch);
8     if (ch == 'a')
9         p = &a;
10    else
11        p = &b;
12    a = 1; b = 2;
13    // *p (stessa espressione) coincide con a oppure con b;
14    // si decide a tempo di esecuzione
15    printf("%d\n", *p); 2
16    return 0;
17 }
```



$$*p \equiv b$$

Area puntata come variabile configurabile

- Data la definizione di array `int a[10];`, la stessa espressione `a[i]` identifica aree di memoria a indirizzi diversi, a seconda del valore di `i`: il primo, il secondo, ..., il decimo elemento.
 - Lo stesso vale per i puntatori, ma in modo più generale. Data la definizione `int *p;`, l'espressione `*p` identifica una qualsiasi area di memoria di tipo intero, a un qualunque indirizzo, memorizzato in `p`.
 - E' come se della "variabile" `*p` potessimo impostare anche l'indirizzo, cosa che non è possibile con le variabili che abbiamo visto finora.
 - Questo meccanismo è estremamente potente (anche pericoloso), e ha varie applicazioni, che inizieremo a vedere fra poco.



Terminologia e notazione grafica

int α ;
int $*p$;

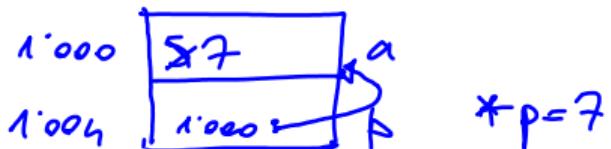
1'000 5
1'004 1'000 p

- Se p , puntatore a T , contiene l'indirizzo dell'area di memoria A di tipo T , si dice che p **punta a A** .
 $\boxed{0 \text{ or } \text{NULL}} \uparrow$
- Convenzionalmente, se un puntatore vale 0 si intende che non punta nessun'area di memoria.
- In questo contesto, 0 è solitamente indicato con la costante **NULL**, definita nell'header **stdlib.h**.

Nelle rappresentazioni grafiche della memoria, per chiarezza,

- il fatto che un puntatore p a un tipo T contenga l'indirizzo di un'area di memoria A di tipo T si rappresenta con una freccia dal centro di p al confine di A .
- il fatto che un puntatore abbia valore 0 o **NULL** si indica scrivendo dentro il puntatore 0, **NULL**, o anche altri simboli grafici.

$\boxed{0 \text{ or } \text{NULL}}$ \uparrow

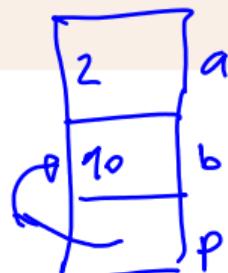
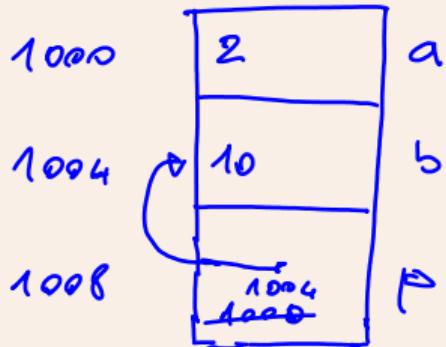


Esempio

int * p; i

110_puntatori/rappresentazione-puntatori.c

```
1 #include <stdio.h>
2
3 int main() {
4     int a = 2, b = 10, *p;
5     p = &a;
6     printf("%d\n", *p); 2
7     p = &b;
8     printf("%d\n", *p); 10
9     return 0;
10 }
```



Sommario

1 Puntatori

2 Passaggio " per riferimento "

3 Array e puntatori

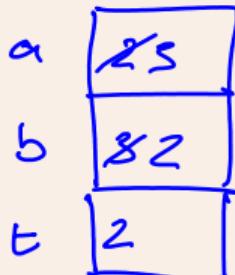
4 Aritmetica dei puntatori

Scambio di valori fra due variabili

Sappiamo come scambiare i valori di due variabili:

110_puntatori/swap.c

```
1 #include <stdio.h>
2
3 int main() {
4     int a = 2, b = 3, t;
5
6     t = a;
7     a = b; } swap
8     b = t;
9     printf("%d %d\n", a, b); // stampa 3 2
10 }
```



E' un procedimento che si usa spesso, quindi ha senso astrarrelo in una procedura.

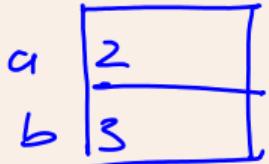
Procedura che scambia due valori

Definiamo una procedura **swap** che scambia i valori dei suoi due parametri.

110_puntatori/swap-procedura.c

```

1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```



Che cosa stampa il programma?

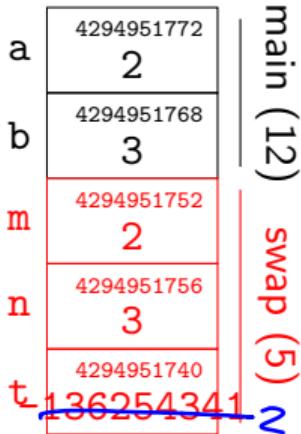
swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```

main (12)	
a	4294951772 2
b	4294951768 3

swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```



swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```

main (12)	
a	4294951772 2
b	4294951768 3
m	4294951752 2 3
n	4294951756 3
t	4294951740 2

swap (6)

swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```

main (12)	
a	4294951772 2
b	4294951768 3
m	4294951752 3
n	4294951756 3 2
t	4294951740 2

swap (7)

swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```

main (12) | swap (8)

a	4294951772
b	4294951768
m	4294951752
n	4294951756
t	4294951740

swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```

a	4294951772
	2
b	4294951768
	3

main (13)

swap-procedura

```
1 #include <stdio.h>
2
3 void swap(int m, int n) {
4     int t;
5     t = m;
6     m = n;
7     n = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(a, b);
13     printf("%d %d\n", a, b);
14 }
```

a	4294951772
	2
b	4294951768
	3

main (14)

↓

'2'	,	'3'	'\n'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'
-----	---	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

stdout

Passaggio parametri per valore

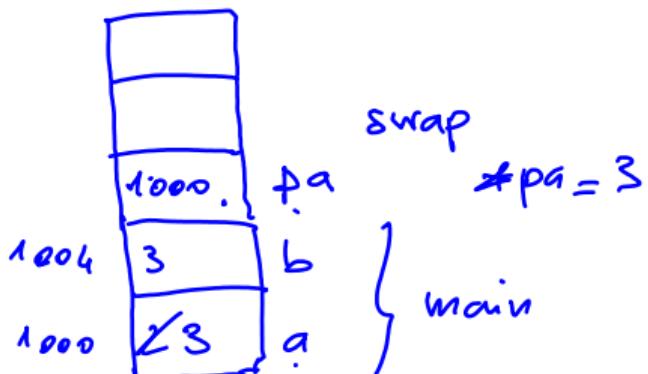
`swap (int m, int n) {
 main() { swap(a, b); } .`

- Alla chiamata di una funzione, la macchina astratta valuta le espressioni passate come parametri attuali e ne assegna il **valore** ai parametri formali della funzione.
- Quindi se i parametri attuali sono variabili (come **a** e **b** passati a **swap**) i parametri formali sono una **copia** dei parametri attuali e le modifiche fatte ai parametri formali non si riflettono sui parametri attuali.
- Questo meccanismo, detto **passaggio per valore**, è l'unico disponibile in C e solitamente è preferibile (ad esempio garantisce che una funzione non possa fare modifiche non desiderate alle variabili passate come parametri).
- Tuttavia a volte (come nel caso di **swap**) sarebbe utile che la funzione chiamata potesse accedere ai parametri attuali. Alcuni linguaggi lo consentono affiancando al passaggio per valore il **passaggio per riferimento**.

`main(){
 float radq(float x),
 b = radq(a);
}`

Passaggio per riferimento (o quasi) in C

- Supponiamo di voler scrivere una procedura che scambi i due valori delle variabili **a** e **b** del chiamante.
- L'unico modo è poter accedere alle aree di memoria identificate da **a** e **b**, ma tramite passaggio parametri (che in C è sempre per valore) questo non è possibile.
- Se solo potessimo creare due "variabili" di cui far coincidere gli indirizzi con quelli di **a** e **b**...
- ... in C si può **emulare** il passaggio per riferimento usando i puntatori.

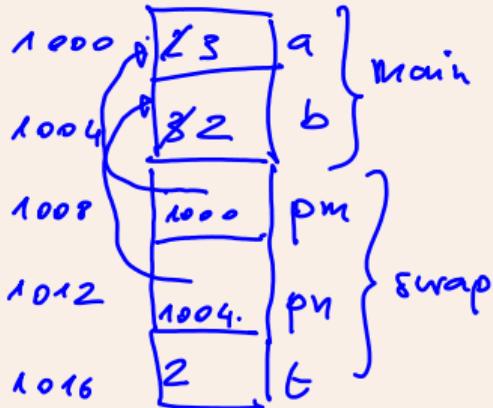


Scambio con puntatori

110_puntatori/swap-puntatori.c

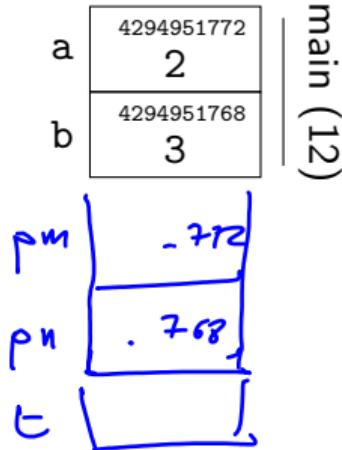
```

1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```



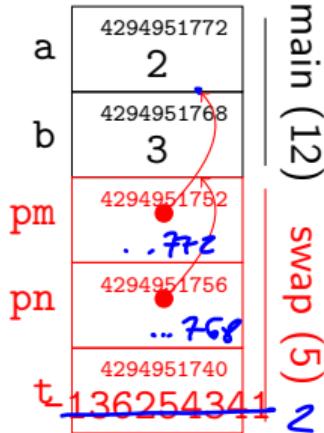
swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```



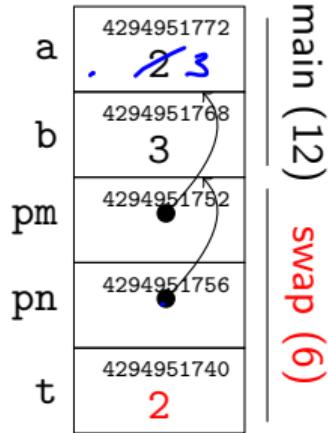
swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```



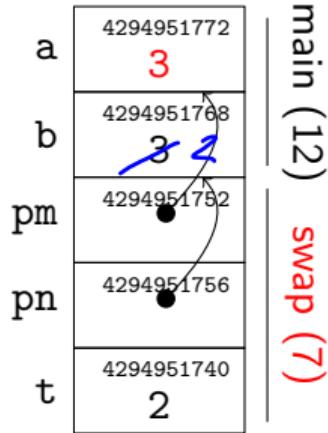
swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```



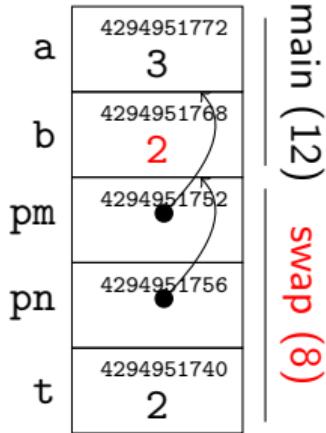
swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```



swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```



swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```

a	4294951772
	3
b	4294951768
	2

main (13)

swap-puntatori

```
1 #include <stdio.h>
2
3 void swap(int *pm, int *pn) {
4     int t;
5     t = *pm;
6     *pm = *pn;
7     *pn = t;
8 }
9
10 int main() {
11     int a = 2, b = 3;
12     swap(&a, &b);
13     printf("%d %d\n", a, b); // stampa 3 2
14 }
```

a	4294951772
	3
b	4294951768
	2

main (14)

↓

'3'	,	'2'	'\n'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'
-----	---	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

stdout

Simulazione del passaggio per riferimento in C

Supponiamo di voler scrivere una funzione **f**, da chiamare da una funzione **g**, in grado di modificare una variabile **a** di tipo **T** locale a **g**.

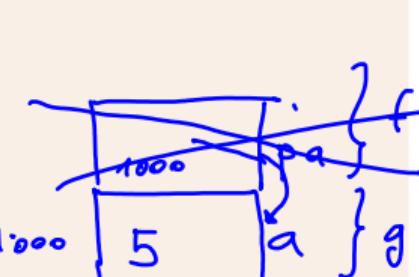
- ➊ Nel prototipo di **f** mettiamo un parametro **pa** di tipo **T***
- ➋ Nel corpo di **f** riferiamoci all'area di memoria a cui vogliamo accedere indicandola con ***pa**
- ➌ Nel corpo di **g**, nella chiamata di **f** passiamo come parametro l'indirizzo di **a**, cioè **&a**.

110_puntatori/riferimento.c

```

1 ... f(..., T *pa, ...)
2 ...
3     *pa = ...
4 ...
5 }
6 ...
7 ... g(... {
8     T a;
9 ...
10    ... f(..., &a, ...)
11 ...
12 }

```



Esercizio

Azzera

Scrivere una procedura di nome **azzera** che azzeri una variabile intera locale al chiamante.

Testarla chiamandola in modo opportuno.

```
Void azzera( ... ) {  
    .  
}
```

```
int main() {  
    int a=2;  
    azzera(..);  
    printf("%d", a), // stampa 0  
}
```

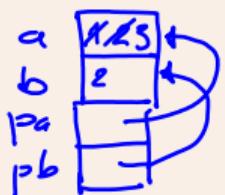
Assegnamenti fra puntatori e fra aree puntate

Se **pa** e **pb** sono puntatori, gli assegnamenti ***pa = *pb** e **pa = pb** sono equivalenti?
 Che cosa stampano questi programmi?

110_puntatori/assegnamento-1.c

```

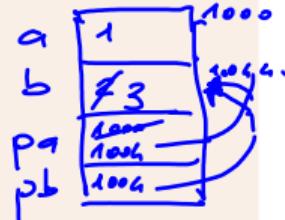
1 #include <stdio.h>
2
3 int main() {
4     int a = 1, b = 2;
5     int *pa = &a, *pb = &b;
6     *pa = *pb;           2   2
7     printf("%d %d\n", *pa, *pb);
8     *pa = 3;            3   2
9     printf("%d %d\n", *pa, *pb);
10    return 0;
11 }
```



110_puntatori/assegnamento-2.c

```

1 #include <stdio.h>
2
3 int main() {
4     int a = 1, b = 2;
5     int *pa = &a, *pb = &b;
6     pa = pb;           2   2
7     printf("%d %d\n", *pa, *pb);
8     *pa = 3;            3   3
9     printf("%d %d\n", *pa, *pb);
10    return 0;
11 }
```



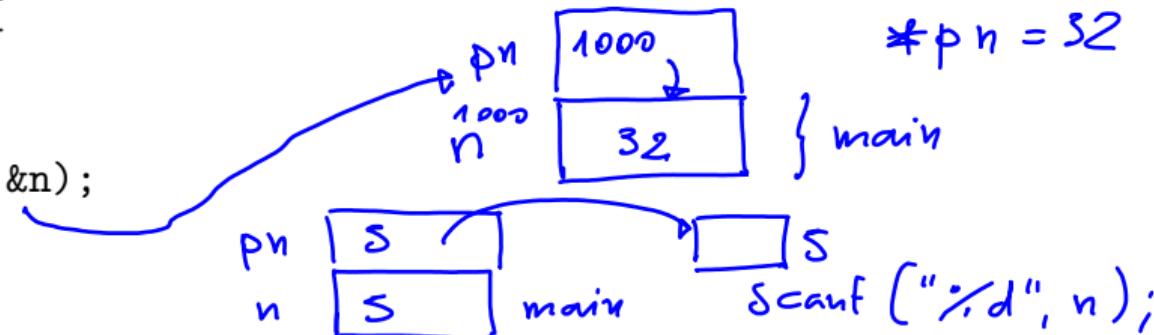
$$*\text{pa} \neq *\text{pb}$$

$$*\text{pa} \equiv *\text{pb}$$

Osservazione: operatore indirizzo nei parametri della `scanf`

Per leggere un intero `n`, ad esempio variabile locale del `main`, da tastiera, usiamo l'espressione `scanf("%d", &n)`.

```
int main() {
    int n;
    ...
    scanf("%d", &n);
    ...
}
```

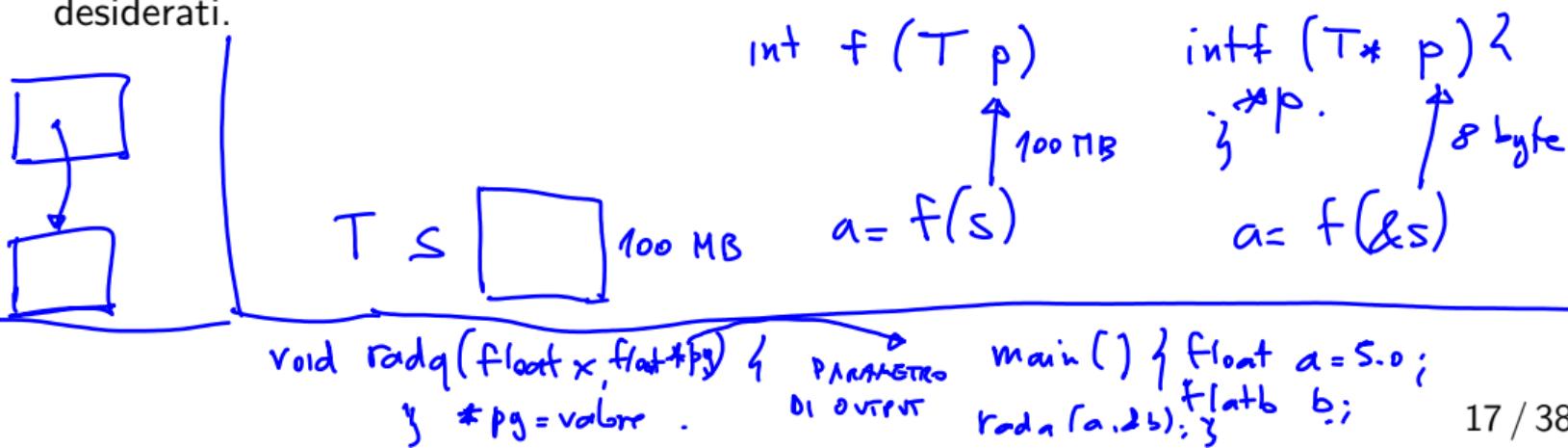


Ora sappiamo perché è necessario premettere l'operatore indirizzo (`&`) a `n`: la `scanf` ha bisogno di accedere all'area di memoria identificata da `n` nel record di attivazione della funzione `main`, per modificarla.

Nel record di attivazione della `scanf` c'è un parametro formale a cui, alla chiamata, viene assegnato l'indirizzo di `n`, permettendo alla `scanf` di scrivere in `n`.

Quando usare il passaggio per riferimento?

- Già visto: quando vogliamo modificare il valore di una variabile locale al chiamante
- Quando non vogliamo far modifiche, ma i parametri sono grandi (ad esempio strutture di 100MB) e copiarli sarebbe costoso. In questo caso è compito del programmatore non modificare le variabili del chiamante.
- Quando non è possibile o non è pratico restituire un valore al chiamante tramite **return**: in questo caso si usano una o più variabili locali al chiamante come **parametri di output** e il chiamante, terminata la chiamata, trova in essi i valori desiderati.



Esercizio

Divisione con procedura

Scrivere e testare una procedura di nome **divisione** che calcoli quoziente e resto di due espressioni intere.

INPUT	QUOZ	RESTO		
15 7	2	1	$q \begin{array}{ c } \hline 2 \\ \hline 1 \\ \hline \end{array}$	$\begin{array}{ c c } \hline 2 & 1 \\ \hline 0 & 1 \\ \hline \end{array}$

```
void divisione (int dividendo, int divisore, int * pquoz, int * presto) {
    *pquoz = dividendo / divisore;
    int quoz, resto;
    divisione(15, 7, &quoz, &resto);
}
```

Esercizio

Divisione con controllo errore

Modificare la soluzione all'esercizio della slide 18 in modo che, se il divisore è 0, la procedura o funzione `divisione` non esegua i calcoli (che comporterebbero *floating point exception*).

Esercizio

Ordina due

Si scriva una procedura che ordini in senso crescente due variabili locali al chiamante; se la seconda è minore della prima, ne scambi i valori usando la procedura **swap** definita precedentemente.

```
int main() {  
    int a=3, b=2,  
        ordina2(..a,...b);  
    printf("%d %d\n",a,b), // stampa 2 3  
}
```

```
void ordina2(..) {  
    swap(...);  
}
```

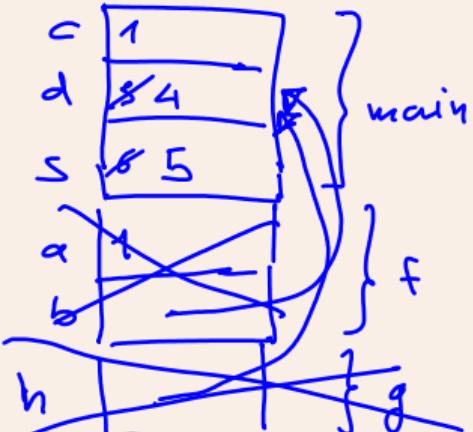
Esercizio di analisi

Si analizzi, disegnando i record di attivazione, l'esecuzione di questo programma.

110_puntatori/analisi.c

```

1 void g(int* h) {
2     (*h)++;
3 }
4 int f(int a, int* b) {
5     g(b);
6     return a + (*b);
7 }
8 main() {
9     int c = 1, d = 3, s = 6;
10    s = f(c, &d);
11 }
```



Sommario

1 Puntatori

2 Passaggio per riferimento

3 Array e puntatori

4 Aritmetica dei puntatori

Che cosa denota l'identificatore di un array

Supponiamo che la definizione `int a[5] = {4, 1, 0, 8, 5};` crei un array che inizia all'indirizzo **1000**, come in figura.

- Finora abbiamo sempre usato la notazione con indici per denotare i singoli elementi di un array.
- Ad esempio, `a[3]` denota l'elemento di indice **3** (cioè il quarto, all'indirizzo **1012**).

$$\text{int } 1000 + 4 * 3$$

<code>a[4]</code>	5	1016
<code>a[3]</code>	8	1012
<code>a[2]</code>	0	1008
<code>a[1]</code>	1	1004
<code>a[0]</code>	4	1000

a

Ma che cosa denota **a** cioè l'identificatore dell'array, senza indici?

L'identificatore di un array è l'indirizzo del suo primo elemento.

Nell'esempio in figura, **a** → **1000**, esattamente come `&a[0]`.

$$a \equiv \&a[0]$$

Accesso a un elemento di un array

Supponiamo che un `int` occupi 4 byte e di avere

```
int a[5] = {4,1,0,8,5};  
printf("%d\n", a[3]);
```

$$1000 + 4 + 4 + 4$$

$$1000 + 3 * 4$$

a indice dim singolo elemento

La macchina astratta deve accedere all'area di memoria all'indirizzo del quarto elemento (quello di indice 3) e stamparla come intero. Come calcola l'indirizzo?

- ① Valuta `a`, cioè l'indirizzo del primo elemento: 1000
- ② Per accedere al quarto elemento deve saltarne tre.
Poiché ogni elemento occupa 4 byte, all'indirizzo del primo elemento (1000) vanno aggiunti $3 \times 4 = 12$ byte, ottenendo 1012.

a[4]	5	1016
a[3]	8	1012
a[2]	0	1008
a[1]	1	1004
a[0]	4	1000

In generale, se `a` è un array di tipo `T`, l'elemento `a[i]` si trova all'indirizzo $a + d * i$, dove d è la dimensione in byte del tipo `T`.

Array e puntatori

Quindi l'identificatore di un array denota un indirizzo di memoria... esattamente come un puntatore. Questo fa sì che, in alcuni casi, array (senza indici) e puntatori siano intercambiabili.

- Definiamo l'array in figura:

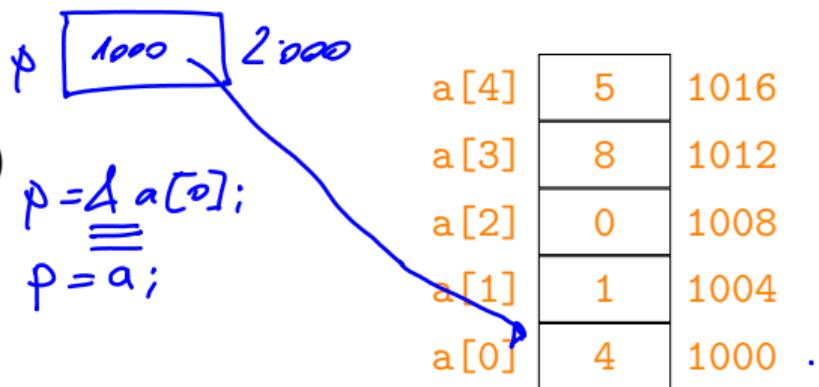
```
int a[5] = {4, 1, 0, 8, 5};
```

(supponiamo che inizi all'indirizzo 1000)

- e un puntatore allo stesso tipo:

```
int *p;
```

$a[0]$
 $\text{printf}("%d", *p);$



Ora l'assegnamento $p = a;$ è lecito: p assume il valore 1000, esattamente come a , e usare p in un'espressione o in un L-Value è come usare a .

Ad esempio:

- $*p$, $p[0]$, $a[0]$ e $*a$ denotano tutti l'area di memoria di 4 byte all'indirizzo 1000.
- $p[3]$ e $a[3]$ denotano entrambi il quarto elemento dell'array, all'indirizzo 1012.

$\underbrace{}_{1000}$

Differenze fra array e puntatori

Fra array e puntatori esistono anche differenze. Le più evidenti:

- A un array non si può assegnare. Date le definizioni:

`int a[5], b[5], *p;`

`p = b` è lecito, `a = b` no.

$a[3] = b[3]$
 $\text{strcpy}(s2, s1)$ anziché $s2 = s1$

- La definizione di un array riserva anche la memoria per gli elementi; quella di un puntatore no.

- `int a[5];`

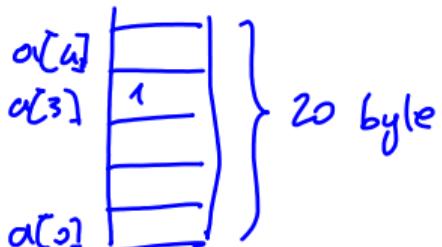
`a[3] = 1;`

è corretto.

- `int *p;`

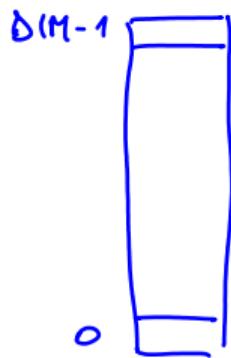
`p[3] = 1;`

è sbagliato: per l'elemento a cui si sta cercando di assegnare non è stata allocata memoria.



Sommario

1 Puntatori



2 Passaggio per riferimento

3 Array e puntatori

4 Aritmetica dei puntatori

sizeof

T vi

L'operatore **sizeof**, chiamato con il nome di un tipo **T** come argomento, restituisce la dimensione (in numero di byte) occupata in memoria dal tipo **T**.

Esempio

`printf("%d", sizeof(int)) stampa 4.`

L'operatore **sizeof** si può chiamare anche con il nome di una variabile come argomento: in tal caso restituisce la dimensione del tipo della variabile.

Esempio

Anche `int i; printf("%d", sizeof(i)) stampa 4.`

Invece `int i; printf("%d", sizeof((char)i)) stampa 1.`

1

Tipi primitivi

- `sizeof(char) = 1`
- `sizeof(short) = 2`
- `sizeof(int) = 4`
- `sizeof(long) = 8`
- `sizeof(float) = 4`
- `sizeof(double) = 8`

possono
varicare
IEEE

Tipi composti

Date le seguenti dichiarazioni di tipo:

```
typedef struct { int x, y; } punto;
```

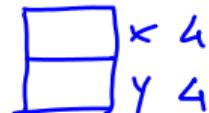
```
typedef char stringa[81];
```

```
typedef punto linea[10];
```

```
typedef struct { stringa nome; int eta; } persona;
```

- **sizeof(punto)**

punto



Tipi composti

Date le seguenti dichiarazioni di tipo:

```
typedef struct { int x, y; } punto;
```

```
typedef char stringa[81];
```



```
typedef punto linea[10];
```

```
typedef struct { stringa nome; int eta; } persona;
```

- `sizeof(punto)` → 8
- `sizeof(stringa)`

Tipi composti

Date le seguenti dichiarazioni di tipo:

```
typedef struct { int x, y; } punto; 8
```

```
typedef char stringa[81];
```

```
typedef punto linea[10];
```



```
typedef struct { stringa nome; int eta; } persona;
```

- `sizeof(punto)` → 8
- `sizeof(stringa)` → 81
- `sizeof(linea)`

Tipi composti

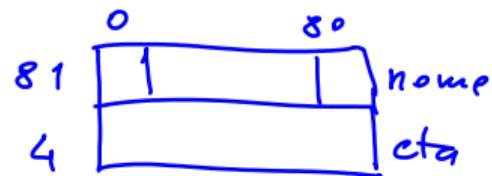
Date le seguenti dichiarazioni di tipo:

```
typedef struct { int x, y; } punto;
```

```
typedef char stringa[81];
```

```
typedef punto linea[10];
```

```
typedef struct { stringa nome; int eta; } persona;
```



- `sizeof(punto)` → 8
- `sizeof(stringa)` → 81
- `sizeof(linea)` → 80
- `sizeof(persona)`

Tipi composti

Date le seguenti dichiarazioni di tipo:

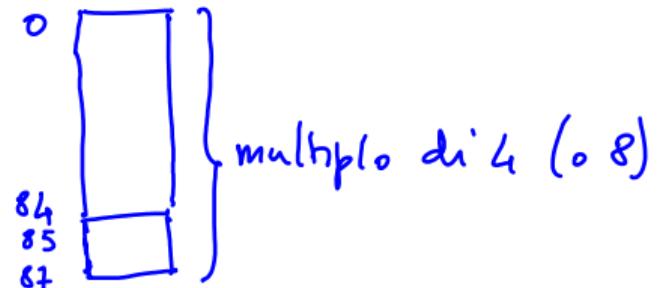
```
typedef struct { int x, y; } punto;
```

```
typedef char stringa[81];
```

```
typedef punto linea[10];
```

```
typedef struct { stringa nome; int eta; } persona;
```

- `sizeof(punto)` → 8
- `sizeof(stringa)` → 81
- `sizeof(linea)` → 80
- `sizeof(persona)` → 88



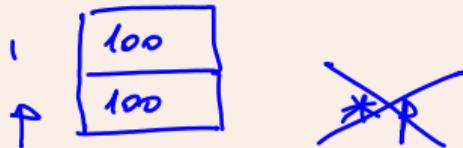
Perché non `sizeof(persona)` → 85? gcc allinea le strutture a multipli di 4 (o 8) byte.

Puntatori e aritmetica

Il seguente programma

110_puntatori/intero-vs-puntatore.c

```
1 #include <stdio.h>
2 main() {
3     int i = 100;
4     int* p = (int*)100;
5     printf("i vale %d; i + 1 vale %d\n", i, i + 1);
6     printf("p vale %d; p + 1 vale %d\n", p, p + 1);
7 }
```



100 101
100 101

stampa

i vale 100; i + 1 vale 101

Puntatori e aritmetica

Il seguente programma

110_puntatori/intero-vs-puntatore.c

```
1 #include <stdio.h>
2 main() {
3     int i = 100;
4     int* p = (int*)100;
5     printf("i vale %d; i + 1 vale %d\n", i, i + 1);
6     printf("p vale %d; p + 1 vale %d\n", p, p + 1);
7 }
```

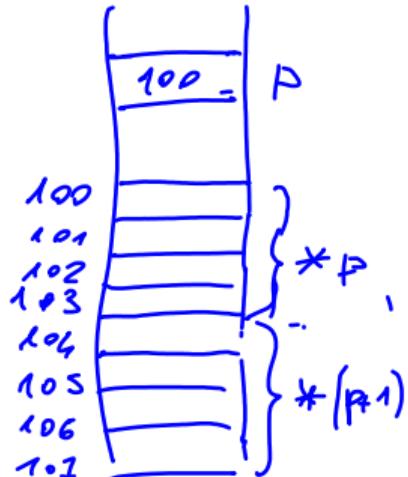
stampa

i vale 100; i + 1 vale 101
p vale 100; p + 1 vale 104

Aritmetica dei puntatori

double *q;
 $q = \begin{matrix} 100 \\ 100+1+8 \end{matrix}$
 $q+1 \rightarrow \begin{matrix} 100+1+8 \\ 108 \end{matrix}$ $q+2 \rightarrow \begin{matrix} 100+1+8 \\ 116 \end{matrix}$

- Il comportamento del programma precedente non è un errore: per motivi pratici, l'aritmetica dei puntatori è diversa da quella degli interi.
- Data la definizione `int *p;`, `p` contiene l'indirizzo di un intero; ad esempio, un elemento di un array di interi. Supponiamo che `sizeof(int)` sia 4.
- Se l'aritmetica dei puntatori seguisse le regole di quella degli interi `*(p + 1)` sarebbe l'intero che inizia al secondo dei quattro byte dell'intero `*p` e si estende nel byte successivo, il che non ha significato.
- E' più intuitivo che (in aritmetica dei puntatori) `p + 1` punti al successivo intero in memoria, cioè al byte (in aritmetica intera) `p + sizeof(int)`, ossia `p + 4`.



$\begin{matrix} 100 \\ p+1 \end{matrix} \rightarrow \begin{matrix} 100+1+4 \\ 104 \end{matrix}$

Aritmetica dei puntatori

sizeof(T) = 8

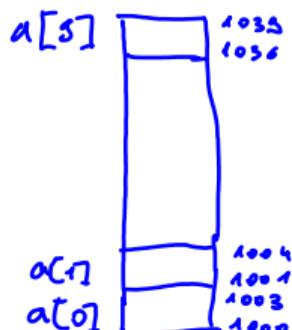
Se T è un tipo, data la definizione $T *p$; e se inizialmente p vale p_0 : $\uparrow 100$

$p = p + n$

Espressione	Valore	Effetto: nuovo valore di p
$p + n$	$p_0 + n * \text{sizeof}(T)$	p_0
$\cdot p++$	p_0	$p_0 + \text{sizeof}(T)$
$++p$	$p_0 + \text{sizeof}(T)$	$p_0 + \text{sizeof}(T)$
$p += n$	$p_0 + n * \text{sizeof}(T)$	$p_0 + n * \text{sizeof}(T)$
$p - n$	$p_0 - n * \text{sizeof}(T)$	p_0
$p--$	p_0	$p_0 - \text{sizeof}(T)$
$--p$	$p_0 - \text{sizeof}(T)$	$p_0 - \text{sizeof}(T)$
$p -= n$	$p_0 - n * \text{sizeof}(T)$	$p_0 - n * \text{sizeof}(T)$

Array e aritmetica dei puntatori

- Supponiamo che `sizeof(int)` → 4.
- Data la definizione `int a[10]`, sappiamo che `a` contiene l'indirizzo del primo elemento dell'array `a`; supponiamo che l'indirizzo sia 1000.
- Qual è l'indirizzo del terzo elemento di `a`, cioè `a[2]`?
- Poiché l'identificatore di un array è anche un puntatore, ad esso si applica l'aritmetica dei puntatori; quanto vale `a + 2`?
- In generale, se `T` è un tipo, data la definizione `T v[DIM];`, le espressioni `v[i]` e `*(v+i)` sono equivalenti.



$$*a \quad T * p_i$$

$1000 + 2 * 4 = 1008$

$a + i * \text{sizeof}(T)$ *intervall*

$$*(a+i) = a[i] \quad \text{puntatore}$$

Esercizio

Che cosa stampa questo programma?

110_puntatori/array-e-puntatori.c

```

1 #include <stdio.h>
2
3 int main() {
4     int a[10], i;
5     for (i = 0; i < 10; i++)
6         *(a + i) = i; // equivalente ad a[i] = i;
7     for (i = 0; i < 10; i++)
8         printf("%d\n", *(a + i));
9     return 0;
10 }
```

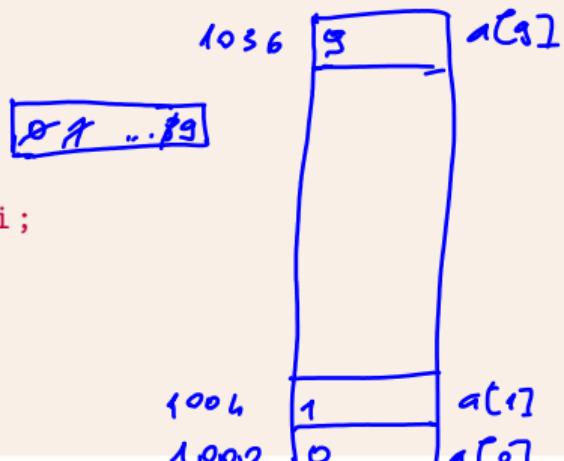
0

1

2

.

3



Array e aritmetica dei puntatori

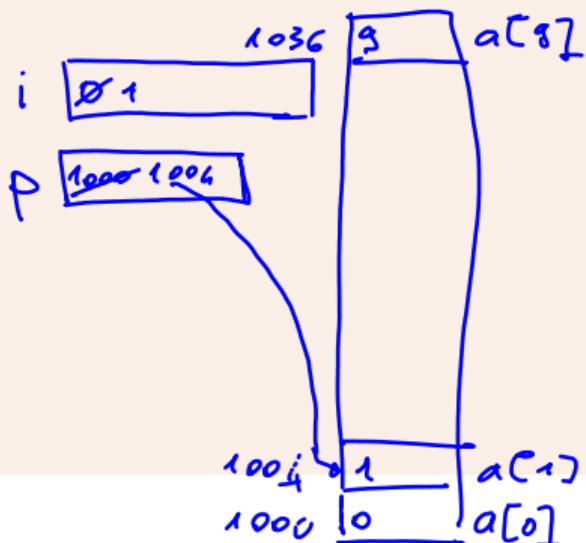
Che cosa stampa questo programma?

110_puntatori/array-e-puntatori-2.c

```

1 #include <stdio.h>
2
3 int main() {
4     int a[10], *p, i;
5     for (i = 0, p = a; i < 10; i++, p++)
6         *p = i;
7     for (i = 0, p = a; i < 10; i++, p++)
8         printf("%d\n", *p);
9     return 0;
10 }
```

0
1



3

Stringhe e aritmetica dei puntatori

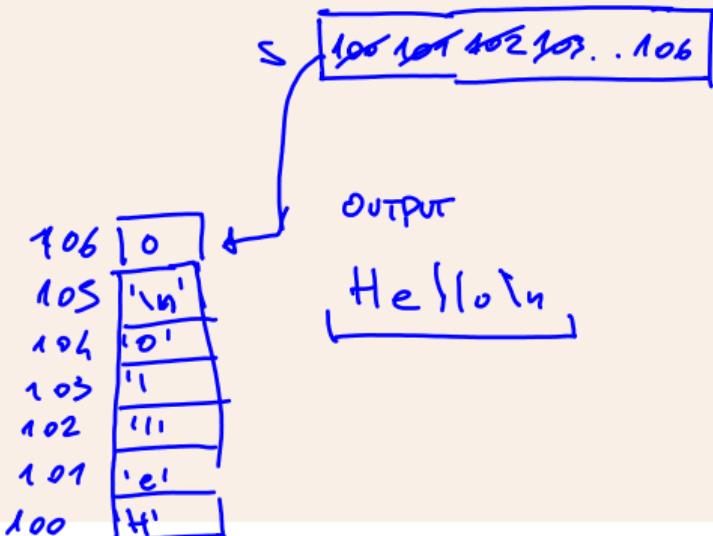
Che cosa stampa questo programma?

110_puntatori/stampa-stringa.c

```

1 #include <stdio.h>
2
3 void f(char* s) {
4     while (*s)
5         printf("%c", *s++);
6 }
7
8 int main() {
9     f("Hello\n");
10    return 0;
11 }
```

char s[10] = "Hello\n",
f(s);



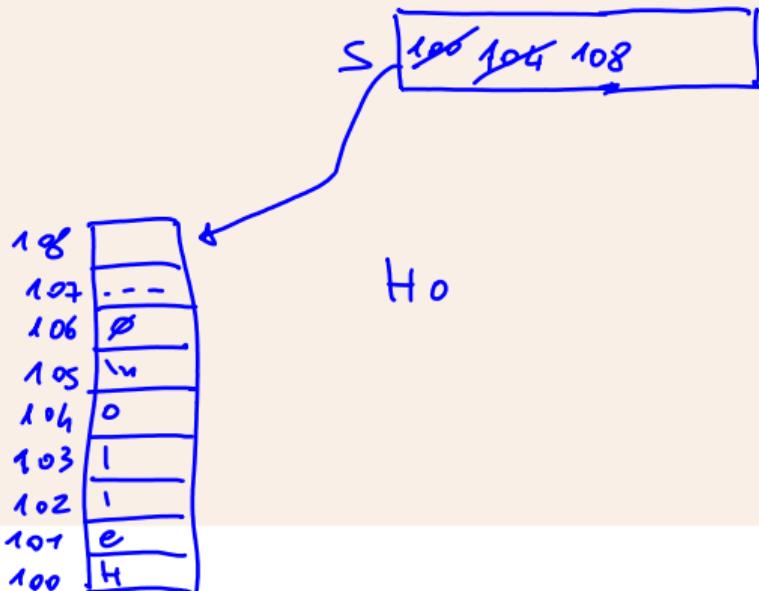
Stringhe e aritmetica dei puntatori

Che cosa stampa questo programma?

110_puntatori/stampa-stringa-2.c

```

1 #include <stdio.h>
2
3 void f(int* s) {      SBAGLIATO
4     while (*s)
5         printf("%c", *s++);
6 }
7
8 int main() {
9     f("Hello\n");
10    return 0;
11 }
```



Esercizio (dalla prova teorica del 6/2/2019)

Detta m la rappresentazione come intero decimale del proprio numero di matricola, la funzione di prototipo `int f(int d)` restituisce

- la d -esima cifra di m a partire da destra se d è compreso fra 1 e il numero di cifre di m ;
 - 0 altrimenti.
- 23273 $f(1) \rightarrow 3$ $f(10) \rightarrow 0$ $f(0) \rightarrow 0$
 $f(3) \rightarrow 2$

Per esempio, se il numero di matricola è `123456`, la chiamata `f(2)` restituisce `5`, mentre le chiamate `f(0)` e `f(7)` restituiscono `0`.

Che cosa scrive il programma costituito dal codice alla slide 38 e dalla definizione della funzione `f`? Motivare la risposta.

110_puntatori/20190206-t1.c

```

1 #include <stdio.h>
2
3 int f(int d);
4
5 int main(int argc, char* argv[]) {
6     char s[] = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
7     int i = 1;
8     while (i < 7) {
9         printf("%c", *(((char*)((int*)s + i))) );
10        i += f(i) ? f(i) : 1;
11    }
12    printf("\n");
13 }
```

i 147

OUTPUT EQ ↴

