Puntatori

Variabili di tipo puntatore

Esempio: `int a = 5;`

Proprietà della variabile `a`:

nome:	`a`
tipo:	`int`
valore:	5
indirizzo:	A010

A00E	···
A010	5
A012	···

Finora abbiamo usato solo le prime tre proprietà. Come si usa l'indirizzo?

`&a` . . . **operatore indirizzo** "`&`" applicato alla variabile `a`
$\Longrightarrow$ ha valore 0xA010 (ovvero, 61456 in decimale)

Gli indirizzi si utilizzano nelle variabili di tipo puntatore, dette anche **puntatori**.

Esempio: `int *pi;`

Proprietà della variabile `pi`:

nome:	`pi`
tipo:	**puntatore ad intero** (ovvero, indirizzo di un intero)
valore:	inizialmente casuale
indirizzo:	fissato una volta per tutte

Sintassi della dichiarazione di variabili puntatore

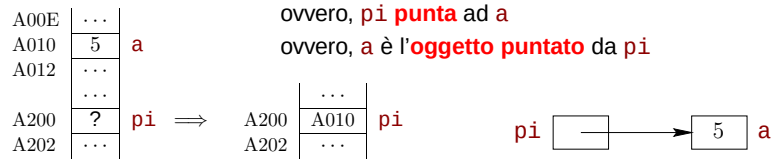
```
tipo *variabile, *variabile, ..., *variabile;
```

Esempio:
```
int *pi1, *pi2, i, *pi3, j;
float *pf1, f, *pf2;
```

`pi1`, `pi2`, `pi3`	sono di tipo puntatore ad `int`
`i`, `j`	sono di tipo `int`
`pf1`, `pf2`	sono di tipo puntatore a `float`
`f`	è di tipo `float`

Una variabile puntatore può essere inizializzata usando l'operatore di indirizzo.

Esempio: `pi = &a;` . . . il valore di `pi` viene posto pari all'indirizzo di `a`
ovvero, `pi` **punta** ad `a`
ovvero, `a` è l'**oggetto puntato** da `pi`



Operatore di dereferenziamento "*"

Applicato ad una variabile puntatore fa riferimento all'oggetto puntato.

Esempio:

```
int *pi;          /* dichiarazione di un puntatore ad intero */
int a = 5, b;     /* dichiarazione di variabili intere */

pi = &a;          /* pi punta ad a ==> *pi e' un altro modo di denotare a */
b = *pi;          /* assegna a b il valore della variabile puntata da pi,
                          ovvero il valore di a, ovvero 5 */
*pi = 9;          /* assegna 9 alla variabile puntata da pi, ovvero ad a */
```

N.B. Se `pi` è di tipo `int *`, allora `*pi` è di tipo `int`.

Non confondere le due occorrenze di "`*`":

- "`*`" in una dichiarazione serve per dichiarare una variabile di tipo puntatore
  Es.: `int *pi;`
- "`*`" in una espressione è l'operatore di dereferenziamento
  Es.: `b = *pi;`

Operatori di dereferenziamento "`*`" e di indirizzo "`&`":

- hanno priorità più elevata degli operatori binari
- "`*`" è associativo a destra
  Es.: `**p` è equivalente a `*(*p)`
- "`&`" può essere applicato solo ad una variabile;
  `&a` non è una variabile $\Longrightarrow$ "`&`" non è associativo
- "`*`" e "`&`" sono uno l'inverso dell'altro
  - data la dichiarazione `int a;`
    `*&a` è un alias per `a` (sono entrambi variabili)
  - data la dichiarazione `int *pi;`
    `&*pi` ha valore uguale a `pi`
    però: `pi` è una variabile
    `&*pi` non lo è (ad esempio, non può essere usato a sinistra di "`=`")

Stampa di puntatori

I puntatori si possono stampare con `printf` e specificatore di formato "`%p`" (stampa in formato esadecimale).

Esempio:

```
int a = 5;
int *pi;
```

A00E	···	
A010	5	a
A012	A010	pi
	···	

```
pi = &a;
printf("indirizzo di a = %p\n", &a);        /* stampa 0xA010 */
printf("valore di pi   = %p\n", pi);        /* stampa 0xA010 */
printf("valore di &*pi = %p\n", &*pi);      /* stampa 0xA010 */

printf("valore di a   = %d\n", a);          /* stampa 5      */
printf("valore di *pi = %d\n", *pi);        /* stampa 5      */
printf("valore di *&a = %d\n", *&a);        /* stampa 5      */
```

Si può usare `%p` anche con `scanf`, ma ha poco senso leggere un indirizzo.

Esempio: Scambio del valore di due variabili.

```
int a = 10, b = 20, temp;
temp = a;
a = b;
b = temp;
```

Tramite puntatori:

```
int a = 10, b = 20, temp;
int *pa, *pb;

pa = &a;         /* *pa diventa un alias per a */
pb = &b;         /* *pb diventa un alias per b */

temp = *pa;
*pa = *pb;
*pb = temp;
```

Inizializzazione di variabili puntatore

I puntatori (come tutte le altre variabili) devono venire inizializzati prima di poter essere usati.

$\Longrightarrow$ È un errore dereferenziare una variabile puntatore non inizializzata.

Esempio:

```
int a;
int *pi;
```

A00E	···	
A010	?	a
A012	F802	pi
	···	
F802	412	
F804	···	

`a = *pi;` $\Longrightarrow$ ad `a` viene assegnato il valore 412

`*pi = 500;` $\Longrightarrow$ scrive 500 nella cella di memoria di indirizzo $\mathrm{F802}$

Non sappiamo a cosa corrisponde questa cella di memoria!!!

$\Longrightarrow$ la memoria può venire corrotta

Tipo di variabili puntatore

Il tipo di una variabile puntatore è "puntatore a *`tipo`*". Il suo valore è un **indirizzo**.

I tipi puntatore sono **indirizzi** e **non interi**.

Esempio:

```
int a;
int *pi;
a = pi;
```

compilando si ottiene un warning: "assignment makes integer from pointer without a cast"

Due variabili di tipo **puntatore a tipi diversi non sono compatibili** tra loro.

Esempio:

```
int x;
int *pi;
float *pf;
```

`x = pi;` assegnazione `int*` a `int`
$\Longrightarrow$ warning: "assignment makes integer from pointer without a cast"

`pf = x;` assegnazione `int` a `float*`
$\Longrightarrow$ warning: "assignment makes pointer from integer without a cast"

`pi = pf;` assegnazione `float*` a `int*`
$\Longrightarrow$ warning: "assignment from incompatible pointer type"

Perché il C distingue tra puntatori di tipo diverso?

Se tutti i tipi puntatore fossero identici (ad es. puntatore a `void`), non sarebbe possibile determinare a tempo di compilazione il tipo di `*p`.

Esempio:

```
void *p;
int i;  char c;  float f;
```

Potrei scrivere:

```
p = &c;
p = &i;
p = &f;
```

Il tipo di `*p` verrebbe a dipendere dall'ultima assegnazione che è stata fatta!!!

Quale è il significato di `i/*p` (divisione intera oppure divisione reale)?

Il C permette di definire un puntatore a `void` (tipo `void*`)

- è compatibile con tutti i tipi puntatore
- **non** può essere dereferenziato (bisogna prima fare un cast esplicito)

Funzione `sizeof` con puntatori

La funzione `sizeof` restituisce l'occupazione in memoria in byte di una variabile. Può anche essere applicata anche ad un tipo.

Tutti i puntatori sono indirizzi $\Longrightarrow$ occupano lo spazio di memoria di un indirizzo.

L'oggetto puntato ha dimensione del tipo puntato.

Esempio: file `puntator/puntsize.c`

```
char *pc;
int *pi;
double *pd;

printf("%d %d %d  ", sizeof(pc),     sizeof(pi),    sizeof(pd));
printf("%d %d %d\n", sizeof(char *), sizeof(int *), sizeof(double *));

printf("%d %d %d  ", sizeof(*pc),  sizeof(*pi), sizeof(*pd));
printf("%d %d %d\n", sizeof(char), sizeof(int), sizeof(double));
```

```
4 4 4  4 4 4
1 2 8  1 2 8
```


Passaggio di parametri per indirizzo

Differenza tra copia del valore e copia dell'indirizzo di una variabile:

Esempio:

```
int b, x, *p;

b = 15;                         /* b vale 15 */
x = b;                          /* il valore di b viene copiato in x */
p = &b;                         /* l'indirizzo di b viene messo in p */

x = 23;
printf("b vale %d\n", b);       /* b non e' cambiato */
*p = 47;
printf("b vale %d\n", b);       /* b e' cambiato */
```

Se si ha una **copia dell'indirizzo** di una variabile questa copia **può essere usata per modificare la variabile** (il puntatore dereferenziato è un modo alternativo di denotare la variabile).

Sfruttando questa idea è possibile fare in modo che una **funzione modifichi una variabile della funzione chiamante**.

In C i **parametri** delle funzioni sono **passati per valore**:

- il parametro formale è una nuova variabile locale alla funzione
- al momento dell'attivazione il valore del parametro attuale viene copiato nel parametro formale

$\Longrightarrow$ le modifiche fatte sul parametro formale **non** si riflettono sul parametro attuale (come `b` e `x` dell'esempio precedente).

Però, se **passiamo** alla funzione **un puntatore ad una variabile**, la funzione può usare il puntatore per modificare la variabile.

Esempio: file `puntator/parametr.c`

Si tratta di un **passaggio per indirizzo**:

- la funzione chiamante passa l'indirizzo della variabile come parametro attuale
- la funzione chiamata usa l'operatore "`*`" per riferirsi alla variabile passata

(simula il **passaggio per riferimento** che esiste in molti linguaggi)

Esempio: Per passere un intero `i` per indirizzo:

il parametro formale si dichiara come: `int *pi` (di tipo: `int*`)
il parametro attuale è l'indirizzo di `i`: `&i`
nel corpo della funzione si usa: `*pi`

Il passaggio per indirizzo viene usato ogni volta che una funzione deve restituire più di un valore alla funzione chiamante.

Esempio: Funzione per lo scambio dei valori di due variabili, e funzione che stampa due valori in ordine crescente.
Implementazione: file `puntator/scambio.c` e file `puntator/ordina2.c`

Esempio: Scrivere una funzione che riceve come parametri giorno, mese, ed anno di una data, e li aggiorna ai valori per la data del giorno dopo.
Implementazione: file `puntator/datasuN1.c`

Esercizio: Utilizzare la funzione appena sviluppata per calcolare la data dopo *n* giorni:

- iterando *n* volte il calcolo della data del giorno successivo
  Soluzione: `puntator/datasuN1.c`
- versione ottimizzata, che passa direttamente al primo del mese successivo
  Soluzione: `puntator/datasuN2.c`

Allocazione dinamica della memoria

Un puntatore deve puntare ad una zona di memoria

- a cui il sistema operativo permette di accedere
- che non viene modificata inaspettatamente

Finora abbiamo visto un modo per soddisfare questi requisiti: assegnare ad un puntatore l'indirizzo di una delle variabili del programma.

Metodo alternativo: **allocazione dinamica della memoria**, attraverso una chiamata di funzione che crea una nuova zona di memoria e ne restituisce l'indirizzo iniziale.

- la zona di memoria è accessibile al programma
- la zona di memoria non viene usata per altri scopi (ad esempio variabili in altre funzioni)
- ad ogni chiamata della funzione viene allocata una nuova zona di memoria

Funzione `malloc`

La funzione `malloc` è dichiarata in `<stdlib.h>` con prototipo:

```
void * malloc(size_t);
```

- prende come parametro la dimensione (numero di byte) della zona da allocare (`size_t` è il tipo restituito da `sizeof` e usato per le dimensioni in byte delle variabili — ad esempio potrebbe essere `unsigned long`)
- alloca (riserva) la zona di memoria
- restituisce il puntatore iniziale alla zona allocata (è una funzione che restituisce un puntatore)

N.B. La funzione `malloc` restituisce un puntatore di tipo `void*`, che è compatibile con tutti i tipi puntatore.

Esempio:
```
float *p;
p = malloc(4);
```

Uso tipico di `malloc` è con `sizeof(`*`tipo`*`)` come parametro.

Esempio:

```
#include <stdlib.h>

int *p;
p = malloc(sizeof(int));
*p = 12;
(*p)++;                          /* N.B. servono le parentesi */
printf("*p vale %d\n", *p);
```

- attivando `malloc(sizeof(int))` viene allocata una zona di memoria adatta a contenere un intero; ovvero viene creata una nuova variabile intera
- il puntatore restituito da `malloc` viene assegnato a `p`
  $\Longrightarrow$ `*p` si riferisce alla nuova variabile appena creata

Lo heap (o memoria dinamica)

La zona di memoria allocata attraverso `malloc` si trova in un'area di memoria speciale, detta **heap** (o **memoria dinamica**).

$\Longrightarrow$ abbiamo **4 aree di memoria**:

- zona programma: contiene il codice macchina
- stack: contiene la pila dei RDA
- statica: contiene le variabili statiche
- heap: contiene dati allocati dinamicamente

Funzionamento dello heap:

- gestito dal sistema operativo
- le zone di memoria sono **marcate libere o occupate**
  - marcata libera: può venire utilizzata per la prossima `malloc`
  - marcata occupata: non si tocca

Potrebbe **mancare la memoria** per allocare la zona richiesta. In questo caso `malloc` restituisce il puntatore `NULL`.

$\Longrightarrow$ Bisogna sempre verificare cosa restituisce `malloc`.

Esempio:

```
p = malloc(sizeof(int));
if (p == NULL) {
  printf("Non ho abbastanza memoria per l'allocazione\n");
  exit(1);
}
...
```

La costante `NULL`

- è una costante di tipo `void*` (quindi compatibile con tutti i tipi puntatore)
- indica un puntatore che non punta a nulla $\Longrightarrow$ non può essere dereferenziato
- ha tipicamente valore 0
- definita in `<stdlib.h>` (ed in altri file header)

Deallocazione della memoria dinamica

Le celle di memoria allocate dinamicamente devono essere **deallocate** (o rilasciate) quando non servono più.

Si utilizza la funzione `free`, che è dichiarata in `<stdlib.h>`:

```
void * free(void *);
```

Esempio:

```
int *p;
...
p = malloc(sizeof(int));
...
free(p);
```

- il parametro `p` **deve** essere un puntatore ad una zona di memoria allocata precedentemente con `malloc` (altrimenti il risultato non è determinato)
- la zona di memoria **viene resa disponibile** (viene marcata libera)
- `p` non punta più ad una locazione significativa (ha **valore arbitrario**)

Prima del termine del programma **bisogna deallocare tutte le zone** allocate dinamicamente.

$\Longrightarrow$ Per ogni `malloc` deve essere eseguita una `free` corrispondente (sulla stessa zona di memoria, non necessariamente usando lo stesso puntatore).

Esempio: file `puntator/puntator.c`

```
int *pi;
int *pj;

pi = malloc(sizeof(int));    /* allocazione dinamica di memoria */
*pi = 150;                   /* ora *pi ha un valore significativo */
pj = pi;                     /* pi e pj PUNTANO ALLA STESSA CELLA DI MEMORIA */
free(pj);                    /* deallocazione di *pj, E QUINDI ANCHE DI *pi */
pi = malloc(sizeof(int));    /* allocazione dinamica di memoria */
*pi = 4000;                  /* ora *pi ha di nuovo un valore significativo */
pj = malloc(sizeof(int));    /* allocazione dinamica di memoria */
*pj = *pi;                   /* le celle contengono lo stesso valore */
pj = malloc(sizeof(int));    /* ERRORE METODOLOGICO:
                                HO PERSO UNA CELLA DI MEMORIA */
```


Tempo di vita di una variabile allocata dinamicamente

- dalla chiamata a `malloc` che la alloca fino alla chiamata a `free` che la dealloca
- indipendente dal tempo di attivazione della funzione che ha chiamato `malloc`

Attenzione: il puntatore ha tempo di vita come tutte le variabili locali

Esempio: file `puntator/vitadin.c`