Appunti di C - Puntatori 1

1 Introduzione

Una variabile in C è caratterizzata da:

- un TIPO
- un NOME
- un VALORE
- un INDIRIZZO

Esempi:
`int a;`
dichiara una variabile

- di TIPO `int` (quindi intero),
- di NOME `a`,
- di VALORE indefinito (solitamente è `0`, ma non bisogna farci affidamento, è sempre meglio *inizializzare* le variabili una volta dichiarate, e nel caso di un puntatore è praticamente d'obbligo, come vedremo in seguito.),
- ad un INDIRIZZO di memoria ben definito, ma che non conosciamo a priori.

`float b = 10.0;`
dichiara una variabile

- di TIPO `float`(quindi reale, con virgola),
- di NOME `b`,
- con il VALORE `10.0` (questa è un'*inizializzazione* a tutti gli effetti),
- ad un INDIRIZZO di memoria preciso, ma anche in questo caso sconosciuto al momento.

2 Come scrivere una dichiarazione di variabile in C

Ragioniamo su come viene interpretata una dichiarazione di variabile dal compilatore, entriamo per un attimo nella "testa" del PC.

```
int a;
```

Con questa riga chiediamo al compilatore: "Riserva un'area di memoria adatta a contenere una variabile *di tipo* `int`, chiama questa cella `a`, e non scriverci dentro nessun valore preciso (cioè non effettuare alcuna inizializzazione)".

```
float b = 10.0;
```

In questo caso invece chiediamo: "Riserva un'area di memoria adatta a contenere una variabile *di tipo* `float`, chiama questa cella `b`, e scrivici dentro il *valore* `10.0` (cioè inizializzala)".

Praticamente la composizione tipo di una dichiarazione di variabile in C è questa:

```
<TIPO della variabile> <NOME della variabile> = <VALORE di inizializzazione>;
```

La parte di inizializzazione come abbiamo visto prima è facoltativa.

3 Dove sono le variabili in memoria? Indirizzi

Ora chiediamoci: *DOVE* sono in memoria queste variabili? Per rispondere alla domanda dobbiamo introdurre il discorso *INDIRIZZO*.
Una rappresentazione molto semplificata della memoria può essere questa:

Indirizzo	Valore
1010	0000 0000 0000 0000 0000 0000 0000 0000
1009	0000 0000 0000 0000 0000 0000 0000 0000
1008	0000 0000 0000 0000 0000 0000 0000 0000
1007	0000 0000 0000 0000 0000 0000 0000 0000
1006	0000 0000 0000 0000 0000 0000 0000 0000
1005	0000 0000 0000 0000 0000 0000 0000 0000
1004	0000 0000 0000 0000 0000 0000 0000 0000
1003	0000 0000 0000 0000 0000 0000 0000 0000
1002	0000 0000 0000 0000 0000 0000 0000 0000
1001	0000 0000 0000 0000 0000 0000 0000 0000
1000	0000 0000 0000 0000 0000 0000 0000 0000

In questa rappresentazione sono state fatte alcune semplificazioni: ogni cella di memoria è di 32 bit (questo dipende dall'architettura del calcolatore, possiamo pensare nei nostri esempi una dimensione di 32 o 64 bit), e gli indirizzi sono stati scritti nel sistema decimale (che è più vicino alla nostra comprensione rispetto al sistema esadecimale che si usa solitamente per indicare gli indirizzi di memoria).
Gli indirizzi crescono verso l'alto (per convenzione), quindi se leggiamo la tabella nell'esempio qui sopra, dal basso verso l'alto abbiamo 11 locazioni di memoria, i cui indirizzi vanno da `1000` a `1010`. Quali valori vi siano contenuti non lo sappiamo al momento (per semplicità ho riempito la tabella con degli zeri, ma come dicevo prima, non bisogna fare affidamento su questo fatto). Proprio per questo motivo è buona prassi *inizializzare* una variabile prima di usarla, e nel caso dei puntatori questo è un **obbligo**, come vedremo tra poco.

Se inizializziamo una variabile in questo modo:

```
int c = 51;
```

e supponiamo (è un esempio puramente teorico, dato che non possiamo controllare l'indirizzo in cui viene allocata una variabile) che sia stata riservata la locazione di memoria all'indirizzo `1004`, ora abbiamo questa situazione (concedetemi di scrivere i valori in decimale, e non in binario come prima):

Indirizzo	Valore	Nome
1010	0	
1009	0	
1008	0	
1007	0	
1006	0	
1005	0	
1004	51	`c`
1003	0	
1002	0	
1001	0	
1000	0	

Cioè la locazione `1004` viene identificata come "variabile di tipo `int` chiamata `c`" e al suo interno è memorizzato il valore `51`.

4 Puntatori

Ora introduciamo i famigerati (non spaventatevi!) **puntatori**.

4.1 Cosa è un puntatore?

Un puntatore è *un tipo di variabile* che contiene *un indirizzo*.
Non ci sarebbe altro da dire, è un tipo di variabile come un altro, che invece di contenere un `char`, un `int`, un `float`, o un `double`, contiene un indirizzo. Ma in C si possono fare tante cose interessanti con i puntatori, si potrebbe dire che sono *croce e delizia di ogni programmatore C*. Senza di essi il C perderebbe gran parte della sua efficacia, i puntatori sono il suo punto di forza, e una volta che si padroneggiano si è in grado di fare qualsiasi tipo di programma.

4.2 Come si dichiara un puntatore?

La dichiarazione tipo di un puntatore è questa:

```
int *p;
```

che va letta così: "Riservami una locazione di memoria adatta a contenere un **puntatore a** `int`, chiamala `p`, e non inizializzarla.
Quindi l'espressione **int *** la leggiamo "`puntatore a int`".
Quale particolare notiamo? Un puntatore è un indirizzo, ma bisogna dire al compilatore il tipo della variabile di cui vogliamo memorizzare l'indirizzo (in questo caso p è un puntatore a `int`, e non possiamo copiarci dentro il valore dell'indirizzo di una variabile `float`→ **darebbe luogo ad un ERRORE!**).
Ma ho ripetuto più volte che i puntatori **devono** essere inizializzati, quindi sorge spontanea la prossima domanda:

4.3 Come si inizializza un puntatore?

Abbiamo bisogno di un altro operatore, e dopo aver introdotto * introduciamo &:
Quando scrivo:

```
int a = 10;
int *p;
p = &a;
```

cosa chiedo al compilatore? Andiamo con ordine: le prime due righe non hanno più segreti, la novità è rappresentata dalla terza.
1. Riserva una locazione di memoria per una variabile *di tipo* ***`int`***, chiamala `a`, e inizializzala col valore `10`.
2. Riserva una locazione di memoria per una variabile *di tipo* ***`puntatore a int`***, chiamala `p`, e non inizializzarla.
3. **N.B.:** Scrivi nella variabile `p` il valore dell'indirizzo della variabile `a`.
Analizziamo la cosa: la terza riga è un assegnamento, quindi bisogna che a sinistra dell'uguale ci sia un qualcosa dello stesso tipo di ciò che è a destra dell'uguale (non posso uguagliare le mele e le pere, no?).
A sinistra abbiamo `p`, di che tipo è? `p` è un ***`puntatore a int`***, cioè è **l'indirizzo di un** `int`. E a destra cosa abbiamo? `&a` restituisce proprio **l'indirizzo di** `a` (che è una variabile di tipo `int`). Quindi sia a sinistra sia a destra dell'uguale abbiamo l'indirizzo di un `int`.
Ecco cosa succede: nella variabile `p` viene copiato l'indirizzo della variabile `a`, e si dice che `p` **punta ad** `a`. Diamo uno sguardo alla memoria, assumendo che la variabile `p` sia all'indirizzo `1004`, e la variabile `a` all'indirizzo `1008`:

Indirizzo	Valore	Nome
1010	0	
1009	0	
1008	10	a
1007	0	
1006	0	
1005	0	
1004	1008	p
1003	0	
1002	0	
1001	0	
1000	0	

Prima dell'istruzione di inizializzazione `p = &a;` il valore di `p` era indefinito, ma **dopo** quell'istruzione `p` contiene l'indirizzo di `a`, cioè `1008`. Se la cosa finisse qui i puntatori servirebbero a ben poco... Una delle cose che possiamo fare ora è accedere al valore di `a` "passando per `p`", senza conoscere il valore effettivo di `p` (cioè senza conoscere l'indirizzo di `a`). Questo lo facciamo con l'operatore `*` in questo modo:

```
*p = 50;
```

Questa istruzione si legge "Scrivi il valore `50` nella locazione di memoria puntata da `p`" (cioè in questo caso "Scrivi `50` nella variabile `a`").

ERRORE DA EVITARE:

```
int main (int argc, char *argv[])
{
    int a;
    int *p;
    *p = 31;     //Errore!  Tentiamo di accedere ad una zona di memoria non definita!!
}
```

In questo caso chiediamo di scrivere `31` nella locazione di memoria puntata da `p`. Ma *dove punta* `p`? In altri termini, che valore c'è scritto in `p`? Esatto, `p` è **un puntatore non inizializzato** (questa è una delle cause principali di **Segmentation Fault**). Se avessimo scritto `p = &a;` **prima** di `*p = 31;` non ci sarebbero stati né errori, né problemi, il valore di `a` sarebbe diventato `31`, perché `p` avrebbe puntato ad una locazione di memoria valida, cioè all'indirizzo di `a`.

Cos'altro possiamo fare con i puntatori? Ad esempio possiamo accedere ai valori contenuti in un array (vettore) in maniera molto rapida. In C un array si dichiara così:

```
int vet[10];
```

questo è un array di 10 `int`, e gli indici vanno da 0 a 9. Quindi il primo valore nell'array sarà contenuto nella locazione `vet[0]` e l'ultimo nella locazione `vet[9]`. Attenzione, dato che gli indici partono da 0, 10 è il numero *totale* di celle dell'array, e 9 (cioè 10-1) è l'indice relativo all'ultima cella. Tentare di accedere a `vet[10]` darà errore in compilazione.
Questo perché un array è una struttura dati *statica*, definita in maniera precisa e non modificabile, che non può allungarsi o accorciarsi a runtime (cioè durante l'esecuzione del programma). L'array `vet` sarà grande abbastanza per contenere 10 `int` dall'inizio alla fine del programma e il compilatore si accorgerà se tenteremo di accedere a zone di memoria al di fuori dei limiti (cioè se tenteremo di accedere alla cella `vet[-1]` ad esempio, o a `vet[10]`), fermando la compilazione e dando errore.
Non dovrebbero esserci problemi ormai a comprendere questo:

```
vet = &(vet[0]);
```

Questo è un punto **molto** importante: in C il nome del vettore equivale **sempre** all'indirizzo della sua prima cella. Cioè il nome del vettore è il puntatore alla sua prima cella (che è quello che abbiamo scritto qui sopra).

Introduciamo ora la cosiddetta *aritmetica dei puntatori*. Abbiamo detto prima che un puntatore deve puntare a "un tipo" di variabili. Perché questo? Potreste obiettare che un indirizzo è pur sempre un indirizzo, sia che si tratti dell'indirizzo di una variabile di tipo `int` che l'indirizzo di una variabile di tipo `float`, ad esempio. Facciamo ora questa supposizione:

```
double vetdbl[10];
int vetint[10];
```

Cosa sono questi? `vetdbl` è un array di 10 `double`, e `vetint` è un array di 10 `int`. Quanto sono grandi le rispettive celle dei due array? Sul mio sistema (Debian GNU/Linux 32bit, su AMD64X2, compilatore gcc 4.3.2-1.1) la risposta è questa: un `int` occupa 32 bit, mentre un `double` occupa 64 bit. Però non si può fare affidamento su queste dimensioni, dato che possono cambiare a seconda dell'implementazione delle librerie, dell'architettura, del compilatore, ecc... Abbiamo detto prima che scrivere il nome dell'array senza le [] equivale a scrivere l'indirizzo della prima cella. Quindi: `vetdbl` e `&vetdbl[0]` sono scritture equivalenti, allo stesso modo scrivere `vetint` equivale a scrivere `&vetint[0]`.

E se scriviamo `(vetdbl + 1)`? Questo chiede al compilatore "l'indirizzo di memoria della locazione seguente a quella puntata da `vetdbl`". L'aritmetica dei puntatori è intelligente abbastanza da capire di quanti bit effettuare "il salto". Se il tipo è `int`, farà un salto di 32 bit, se il tipo è `double`, farà un salto

di 64 bit.
Se scriviamo `*(vetint + 1) = 50;` quello che chiediamo è: "Scrivi il valore `50` nella locazione di memoria seguente a quella puntata da `vetint`", cioè: "Scrivi il valore `50` nella cella `vetint[1]`".
Completando la tabella con valori random, questa potrebbe essere la situazione in memoria:

Indirizzo	Valore	Array	Puntatore
1009	21	`vetint[9]`	`vetint + 9`
1008	45	`vetint[8]`	`vetint + 8`
1007	457	`vetint[7]`	`vetint + 7`
1006	1073	`vetint[6]`	`vetint + 6`
1005	99	`vetint[5]`	`vetint + 5`
1004	0	`vetint[4]`	`vetint + 4`
1003	1000	`vetint[3]`	`vetint + 3`
1002	72	`vetint[2]`	`vetint + 2`
1001	50	`vetint[1]`	`vetint + 1`
1000	3701	`vetint[0]`	`vetint`

Ora possiamo "giocare" come vogliamo con questo esempio. Possiamo scrivere il valore `1` nella cella `vetint[6]` sia scrivendo `vetint[6] = 1;` sia scrivendo `*(vetint + 6) = 1;` nel primo caso modifichiamo il valore usando l'indirizzamento classico degli array, nel secondo invece usiamo l'aritmetica dei puntatori (le parentesi tonde in questi casi sono **necessarie e obbligatorie** per forzare la precedenza dell'operatore `+` sull'indirizzo rispetto all'operatore `*`).

NB: `vetint` è una **COSTANTE**, non si può modificare, perché l'array è una struttura statica, e cambiare il valore di `vetint` equivarrebbe a spostare il vettore in memoria (cioè portarlo su un altro indirizzo), cosa **IMPOSSIBILE**. E' altrettanto un'operazione **non valida** *l'incremento* di `vetint`, cioè ad esempio non si può scrivere `*(vetint++);` per fare riferimento al contenuto della cella `vetint[1]`.

Usando funzioni di allocazione dinamica della memoria (come `malloc()` e simili) si può ovviare al difetto principale degli array, cioè alla loro staticità. Con `malloc()` si richiede una quantità arbitraria di memoria, e la funzione ritorna un puntatore alla memoria richiesta. In questo modo si può fare qualcosa del tipo chiedere all'utente a runtime il numero di elementi che vuole trattare, e in base a quello allocare la memoria necessaria, che sarà poi utilizzata mediante puntatore. In questo modo non si ha né spreco di memoria (se si volesse usare un array occorrerebbe sovradimensionarlo in rapporto al numero di elementi che ci si aspetta) e nemmeno dare un limite massimo all'utente (con un array di 1000 elementi ad esempio sarebbe impossibile indicizzare 2000 elementi, e si dovrebbe limitare la richiesta dell'utente).

Ma questo sarà argomento (forse... chi lo sa...) di un prossimo pdf :-)
Se avete richieste, correzioni, consigli, ecc... scrivetemi!