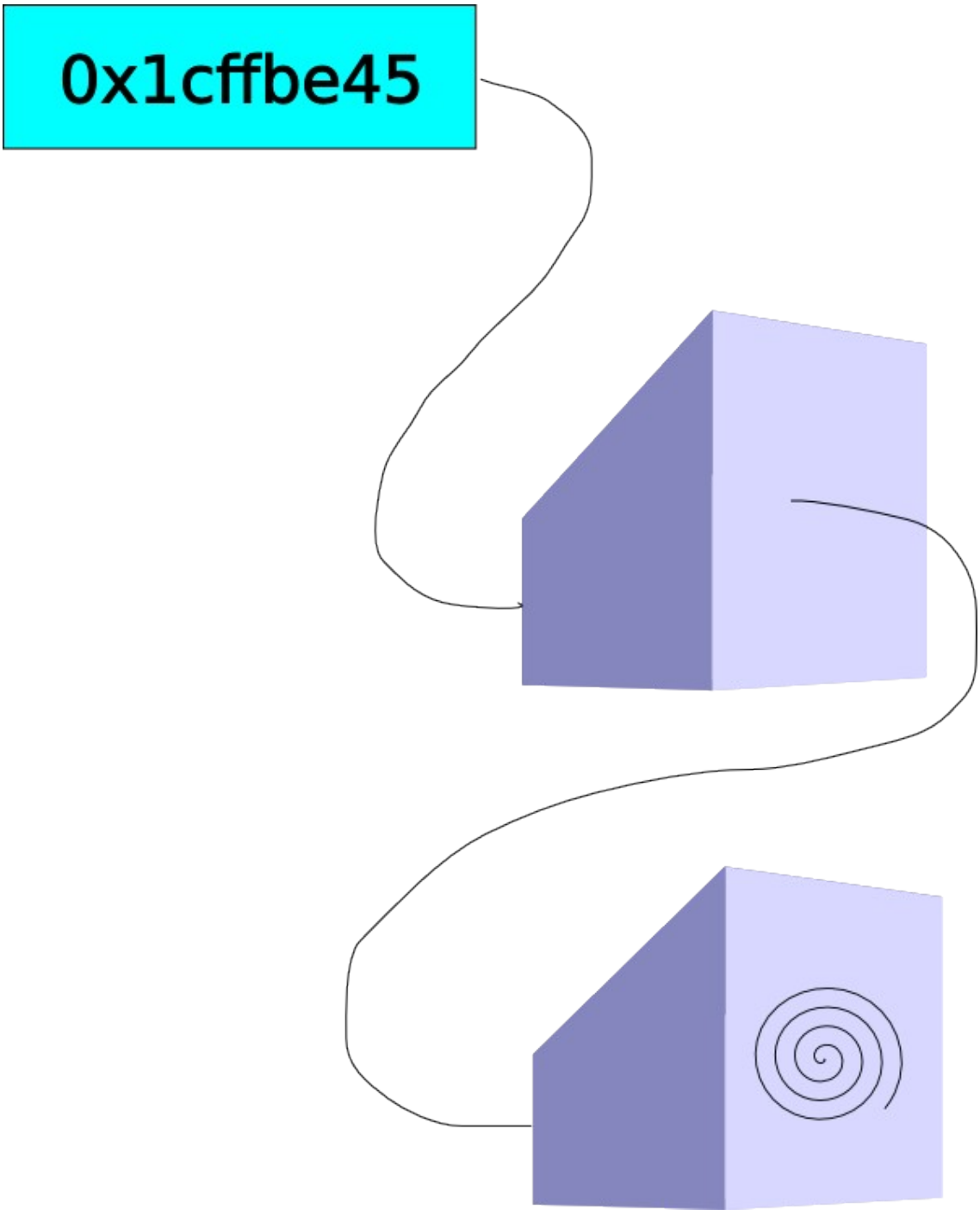
I Puntatori In C



Copyleft

Quest'opera è rilasciata sotto licenza GNU Free Documentation License, pertanto essa può essere copiata, modificata e distribuita senza problemi.

Questo tutorial rappresenta il lavoro di uno studente di informatica che cerca di far sì che i maggiori ostacoli di chi studia per la prima volta il linguaggio C siano ridotti al minimo, in modo che lo sviluppo di applicazioni e di software risulti un piacere e non un obbligo o una noia mortale da fare in un linguaggio "difficile" come il C.

Il tutorial non presuppone grosse pre-conoscenze del linguaggio; basta infatti conoscere la sintassi di base del C e sapere cosa siano le procedure (o funzioni).

Al termine del tutorial sono disponibili alcune domande per applicare ciò che avete appreso.

Le relative risposte, sono riportate sul seguente sito:

http://markon.netsons.org/Guide/C

N.B.:

Tutti gli esempi riportati all'interno del manuale sono stati compilati (con GCC 4.3.3) ed eseguiti in un sistema operativo GNU/Linux, in particolare Debian Squeeze.

Per maggiori informazioni:

http://www.debian.org/index.it.html

http://gcc.gnu.org/

Perchè i puntatori sono importanti?

Il primo problema che un nuovo programmatore deve affrontare quando si trova a leggere il capitolo dei puntatori in un libro sul C è quello di capire sostanzialmente a cosa servano.

Praticamente un puntatore altro non è che una variabile che contiene un indirizzo di memoria. Perchè allora in giro tutti ne parlano come "una cosa difficile da capire e da studiare"?

Perchè un puntatore presuppone una conoscenza - per lo più basilare - della memoria e di come funzionano gli indirizzi di memoria in C.

I puntatori sono utilizzati per creare tipi di dati dinamici (come liste, code, alberi...), per allocare dati dinamicamente in modo che rimangano in memoria per un tempo o per tutta l'esecuzione del programma (grazie all'allocazione dinamica), per semplificare l'astrazione dei dati e delle funzioni e infine per diminuire l'overhead nel passaggio degli argomenti nelle chiamate a funzione.

Potreste pensare ora: ok, chiudo questo tutorial, preferisco il libro di quindicimila pagine.

Sappiate però che da questo punto in poi le cose si faranno interessanti, soprattutto dal lato pratico.

Cos'è un puntatore?

I puntatori sono uno dei dati più importanti che il C mette a disposizione del programmatore.

Ma per capire a fondo cos'è un puntatore c'è bisogno di avere chiaro in mente cosa accade quando lo dichiariamo e definiamo[1].

Vediamo dunque come fare e cosa accade quando dichiariamo un puntatore:

```
int *ptr;
```

1 Non confondete dichiarazione con definizione: mentre la dichiarazione fa sapere al compilatore che quella variabile esiste da qualche parte, specificando il tipo a cui appartiene, la definizione alloca realmente lo spazio in memoria.

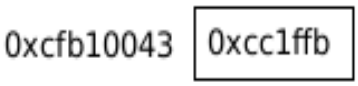
Come potete vedere, il simbolo *****, usato per il prodotto, viene usato anche per dichiarare un puntatore.

In questo modo stiamo dicendo al compilatore che abbiamo bisogno di una variabile che *punti* a un intero, già allocato da qualche altra parte nella memoria.

Ricordiamoci però che un puntatore è una variabile, quindi consuma spazio che, seppur minimo, rimane pur sempre spazio. La dimensione della variabile dipende dal sistema operativo, ma solitamente è la stessa degli interi (2 o 4 byte).

N.B.: da adesso in poi useremo indirizzi e interi di 4 byte.

Occupiamo dunque 4 byte di memoria quando definiamo un puntatore.



Cosa contiene la variabile dopo la definizione? Un numero intero casuale.

La variabile *ptr* è definita all'indirizzo 0xcfb10043 e il suo contenuto è 0xcc1ffb.

Come possiamo quindi sovrascrivere il contenuto del puntatore?

In questo modo:

```
int *ptr;
ptr = 5;
```

Un lettore più attento potrebbe subito intuire che si tratta di un errore.

E infatti è così perchè stiamo dicendo alla variabile *ptr:* "punta all'indirizzo di memoria 5". Come facciamo a sapere se 5 è un indirizzo "buono", ovvero l'indirizzo di una variabile definita precedentemente nel resto del codice? Non siamo mica dei maghi, per cui non possiamo sapere a priori l'indirizzo di una variabile.

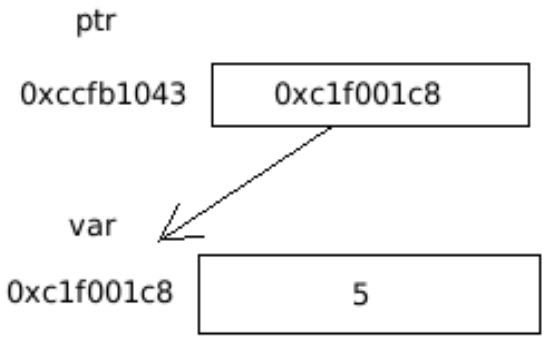
Per fortuna che in C non si procede per tentativi ed è per questo che è stato creato l'operatore di indirizzo: **&**.

```
int var;
int *ptr;
var = 5;
ptr = &var;
```

Ma analizziamo i vari passaggi:

prima dichiariamo due variabili, una di tipo *intero*, un'altra *puntatore a intero*.

var avrà il suo indirizzo, così come *ptr,* solo che la differenza è che *var* ha un valore proprio, indipendente da altre variabili, mentre il puntatore dipende da un'altra variabile; un puntatore non serve a nulla senza uno spazio già allocato in memoria.



Potrebbe sorgere spontanea una domanda: com'è possibile riferirsi al valore a cui punta la variabile *ptr*?

Con l' operatore di dereferenziazione: *****.

In pratica in questo modo potremmo stampare il valore di *var*, passando per *ptr*:

```
printf("%d\n", *ptr);
```

Con questa sintassi stiamo effettuando questo passaggio:

va' all'indirizzo di *ptr*, prendi il contenuto (indirizzo di *var*) e ottieni il valore assegnato a tale indirizzo(5).

E se provassimo a modificare il valore del contenuto (ovvero l'indirizzo all'interno) del puntatore quando non è stato ancora inizializzato?

In poche parole, così:

```
int *ptr;
*ptr = 5;
```

Riceveremmo un bel Segmentation Fault[1]. Infatti stiamo cercando di assegnare il valore 5 a un indirizzo di memoria che magari esiste, ma in cui non abbiamo i permessi per accedere o scrivere.

Vi renderete conto che da solo un puntatore non serve a nulla. Ovvero, non è possibile usare un puntatore come una variabile semplice a cui è possibile

1 Gli errori di seg. fault vengono intercettati dal sistema operativo e capitano perchè stiamo cercando di accedere a un indirizzo di memoria a cui non c'è permesso di accesso oppure stiamo cercando di accedere in un modo che non ci è permesso (scrittura in una locazione in sola lettura).

assegnare un valore qualunque. Ma in un programma C i puntatori sono il cavallo di battaglia.

Per poter utilizzare senza problemi un puntatore, è comunque consigliato inizializzare il puntatore a NULL.

In questo modo non esistono ambiguità e si riducono gli errori in fase di debug[1].

```
int *ptr = NULL;
```

In questo modo, se proviamo a fare un controllo:

```
if (ptr == NULL)
{
     printf("Il puntatore non  punta a nulla.\n");
}
```

esso risulterà positivo e stamperebbe la scritta.

Al contrario, se non inizializziamo il puntatore a NULL, il controllo risulterebbe negativo e non stamperebbe nulla, dato che il puntatore contiene un indirizzo casuale, il che è diverso da NULL.

Applicazioni dei puntatori

Dopo aver visto cosa sono i puntatori e come si dichiarano, passiamo ora al loro uso in esempi pratici.

1. Come argomento delle funzioni

Tra i vari vantaggi nell'uso dei puntatori, come abbiamo già visto nel primo paragrafo, c'è quello della diminuzione dell'overhead[2] nel passaggio degli argomenti.

Vediamo più da vicino cosa vuol dire.

Nel nostro esempio abbiamo una funzione quadrato che deve effettuare il quadrato del numero passato come argomento.

In C ci sono principalmente due metodi per farlo:

1) definendo una funzione che ritorna il risultato desiderato e che accetta come argomento il numero di cui trovare il quadrato. Esempio:

1 La fase di debug è il momento in cui proviamo a scovare e a risolvere gli errori che rendono il nostro programma inutilizzabile. Ovviamente per esempi banali è possibile usare delle semplici stampe a video (printf).

2 L'overhead è lo spreco di risorse rispetto all'algoritmo "ideale".

```
int main(void)
{
	int x;
	printf("x ora vale %d\n", x);
	x = quadrato(x);
	printf("Dopo la funzione x vale %d\n", x);
}
```

La funzione quadrato è definita in questo modo:

```
int quadrato(int n)
{
	return n*n;
}
```

In questo caso l'overhead è quasi nullo, visto che i dati da gestire sono piccoli e pochi, ma più avanti vedremo un metodo che può dare effettivamente migliori performance.

2) Andando a modificare direttamente il valore della variabile passata come argomento, visto che questa rappresenta essa stessa il numero di cui trovare il quadrato. Esempio:

```
int main(void)
{
	int x;
	x = 5;
	printf("x ora vale %d\n", x);
	quadrato(&x);
	printf("Dopo la funzione x vale %d\n", x);
}
```

In questo caso la funzione quadrato non ritorna nulla, ed è definita in questo modo:

```
void quadrato(int *n)
{
	*n = (*n) * (*n); /* le parentesi non sono necessarie */
}
```

Le differenze sembrano minime, ma sono davvero importanti sotto ogni aspetto.

Innanzitutto in quest'ultimo caso l'overhead potrebbe diminuire vertiginosamente dato che non sarà salvato sullo stack il valore proprio di x

(ovvero 5). Come ho già detto però, in questo esempio non ne troviamo un grosso vantaggio, dato che un intero è 4 byte e un puntatore è 4 byte, quindi non risparmiamo nulla, se non un'istruzione per il return.

Spiego meglio cosa significa "copia sullo stack".

Copia sullo stack

Se avete già studiato le funzioni, probabilmente saprete che senza il return il valore dell'argomento passato alla funzione **non** può essere modificato (se non con i puntatori).

```
int main(void)
{
	x = 5;
	printf("Valore di x: %d\n", x) /* stamperà 5*/
	cambia(x);
	printf("Valore di x: %d\n", x) /* stamperà ancora 5*/
}
```

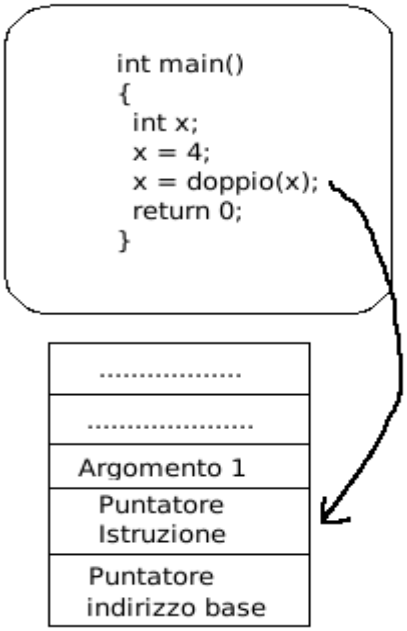
Dove *cambia* è definita in questo modo:

```
void cambia(int n)
{
	n = 10;
}
```

Ciò avviene perchè quando viene chiamata una funzione, i relativi argomenti vengono salvati in un'area di memoria chiamata stack.

Cosa ancora più importante è che quando chiamiamo una funzione, il puntatore dell'istruzione (ovvero il puntatore che segna in che parte di codice ci troviamo) viene salvato sullo stack.

In pratica avviene una cosa del genere:



Il puntatore all'indirizzo base sarà, all'interno della funzione chiamata, l'indirizzo di partenza da cui ottenere tutti i vari valori presenti sullo stack. Tale puntatore viene aggiunto allo stack all'interno della funzione, in modo da facilitarne l'utilizzo.

Dunque se il puntatore all'indirizzo base è a 0, il primo argomento sarà a +8 (dato che ogni indirizzo è di 4 byte), mentre il puntatore all'istruzione è a +4. Se vogliamo aggiungere cose al nostro stack, basterà riferirci ad essi con numeri negativi, proprio perchè cresce verso il basso: -4 è la prima cosa aggiunta nella funzione, -8 la seconda...e così via.

A questo punto abbiamo quindi sullo stack i vari argomenti passati alla funzione seguiti dal puntatore all'istruzione di codice (della funzione chiamata).

Poiché ogni funzione utilizza una parte di stack, partendo dalla prima locazione libera dello stack (ovvero quella successiva all'indirizzo di base), quando siamo al termine della funzione, il puntatore allo stack, che nel frattempo è cambiato (visto che abbiamo usato variabili interne alla funzione), dovrà ritornare allo stato iniziale, ovvero al puntatore all'istruzione, in modo che il programma saprà da dove continuare.

Come fa a ritornare allo stato iniziale? Semplice, incrementando il puntatore allo stack. Se insomma per la nostra funzione abbiamo avuto bisogno di 4 variabili intere, dopo aver concesso 16 byte per lo stack, dovremo sottrarne esattamente 16 al momento dell'uscita dalla funzione.

E le variabili all'interno della funzione? Semplice, portando il puntatore allo stack al punto di partenza, le abbiamo rese *locali*, rendendole invisibili all'esterno.

In questo modo abbiamo dunque visto cosa accade quando chiamiamo una funzione.

Questo ci è stato utile per capire perchè un puntatore permette la modifica dall'interno della funzione.

Tornando a noi, poiché la copia sullo stack può essere stata complessa per qualcuno, facciamo qualche esempio pratico per vedere cosa significa ciò che abbiamo visto.

Nel seguente esempio vedremo come vengono interpretati da un compilatore gli indirizzi di memoria e perchè non è possibile cambiare il valore di una variabile all'interno di una funzione senza utilizzare i puntatori. Insomma, vediamo in linea generale ciò che riguarda lo stack e le funzioni.

Esempio 1:

```
void func_called(int n)
{
     /* stampa dell'indirizzo e del valore della variabile */
     printf("All'interno di func_called: %p - %d\n", &n, n);
     n = 10;
     printf("Ancora all'interno di func_called: %p - %d\n", &n, n);
}
```

Mentre il main:

```
int main(void)
{
     int x;
     x = 5;
     printf("Valore di x prima della funzione: %p - %d\n", &x,  x);
     func_called(x);
     printf("Valore di x dopo la funzione: %p - %d\n", &x,  x);
}
```

Se proviamo a compilare e ad eseguirlo, ci troveremo con un risultato simile:

```
./a.out
Valore di x prima della funzione: 0xbfd02950 - 5
All'interno di func_called: 0xbfd02930 - 5
Ancora all'interno di func_called: 0xbfd02930 - 10
Valore di x dopo la funzione: 0xbfd02950 - 5
```

Come potete intuire, all'interno del *main*, la variabile *x* ha un indirizzo proprio e un valore che le abbiamo assegnato successivamente (= 5).

Nel mio caso l'indirizzo di *x* è 0xbfd02950, ma molto probabilmente sul vostro computer sarà diverso. Quando andiamo a chiamare la funzione *func_called* passandole il valore di *x*, nello stack sarà salvato il numero 5 e non l'indirizzo di *x*. Tale indirizzo sarà del tutto estraneo alla nostra funzione chiamata, per cui i rapporti con l'esterno sono stati rotti del tutto.

E' per questo motivo che quando assegnate un valore a *n*, state modificando solo *n*! E ciò avviene perchè *n* è una variabile a sé stante. Infatti il suo indirizzo è 0xbfd02930; dunque assegneremo 10 a tale indirizzo.

Potreste chiedere allora: perchè *n* all'inizio vale 5? Ebbene, perchè *n* avrà il valore dell'argomento che abbiamo passato alla funzione. Se passiamo il numero 20 come argomento, l'effetto sortito sarà sempre lo stesso. Cambierà solo il valore iniziale di *n*.

Questo discorso vale anche con i puntatori, solo che bisogna avere chiaro in mente cosa accade quando passiamo un puntatore come argomento di una funzione.

In un esempio precedente abbiamo visto:

```
void quadrato(int *n)
{
   printf("Prima dell'assegnamento: %p - %p - %d\n", &n, n, *n);
   *n = (*n) * (*n); /* le parentesi non sono necessarie */
   printf("Dopo l'assegnamento: %p - %p - %d\n", &n, n, *n);
}
```

E il main:

```
int main(void)
{
     int x;
     x = 5;
     printf("x ora vale %p - %d\n", x);
     quadrato(&x);
     printf("Dopo la funzione x vale 0xbfbee030 - %d\n", x);
}
```

Dopo averlo compilato ed eseguito:

```
./a.out
x ora vale 0xbfbee030 - 5
Prima dell'assegnamento: 0xbfbee010 - 0xbfbee030 - 5
Dopo l'assegnamento: 0xbfbee010 - 0xbfbee030 - 25
Dopo la funzione x vale 0xbfbee030 - 25
```

Come potete vedere dall'output del programma, prima di chiamare la funzione, *x* vale 5 e si trova all'indirizzo 0xbfbee030.

Chiamata la funzione, con l'argomento *x*, succede qualcosa di questo tipo:

0xbfbee030 viene salvato sullo stack e proprio per questo motivo possiamo riferirci al suo contenuto (=5) e modificarlo.

Se invece avessimo voluto modificare tale indirizzo, innanzitutto era una cosa che il C non ci permette di fare (modificare l'indirizzo di una variabile?!) e in secondo luogo tale modifica non avrebbe avuto effetto, perchè stavamo semplicemente andando sullo stack all'indirizzo 0xbfbee010 e cambiando il suo contenuto, ovvero 0xbfbee030, in un altro (magari 0xbfbee050).

Graficamente ci troviamo in una situazione come questa:

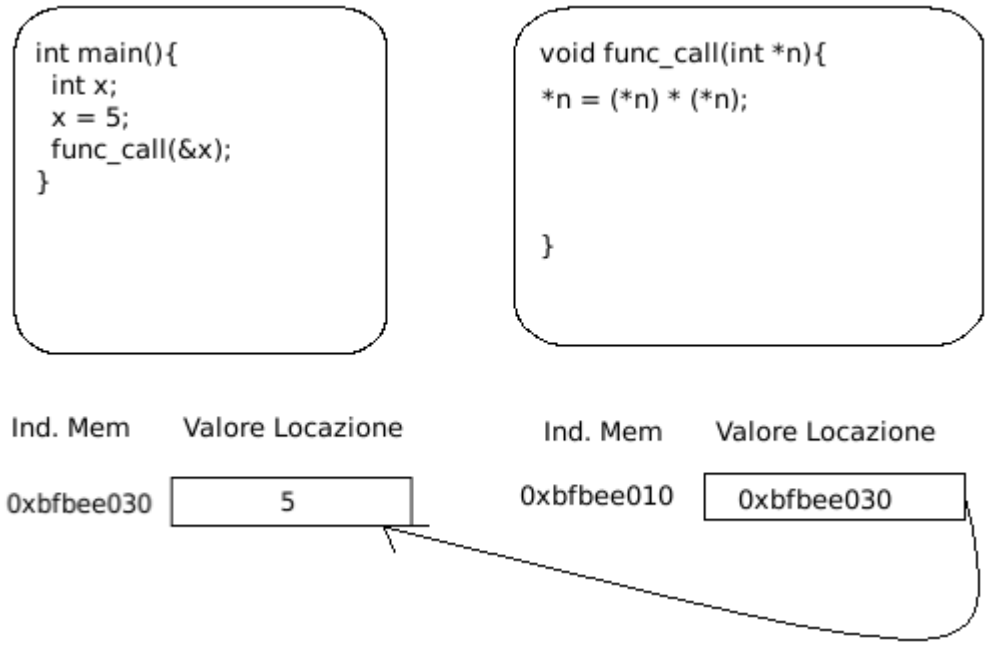


Fig. 1: Nella funzione func_call, dereferenziando n con l'operatore *, possiamo far sì che x cambi, visto che prima calcoliamo n*n e poi lo andiamo a salvare nell'indirizzo 0xbfbee030

Insomma, lo stesso discorso fatto per gli argomenti passati senza riferirci all'indirizzo, vale anche per gli indirizzi!

E' per questo motivo che quando in giro trovate nomi come "passaggio per valore" e "passaggio per riferimento", l'autore del manuale sta rendendo ambigua la definizione delle funzioni in C. Questo perchè fa pensare al lettore che il valore delle variabili non può essere modificato e quello dei puntatori sì. In realtà non è così, dato che vale lo stesso discorso!

Ma vediamo più da vicino:

```
int main(void)
{
	int *p;
	int arr[10] = {0,1,2,3,4,5,6,7,8,9};
	p = arr;
	printf("Primo valore di p: %p - %p - %d\n", &p, p, *p);
	func_call(p);
	printf("Valore di p dopo func_call: %p - %p - %d\n", &p, p,
	                                                      *p);
}
```

Vediamo func_call stavolta com'è definita:

```
void func_call(int *n)
{
	int new_arr[5] = {1,2,3,4,5};
	printf("Valore di n: %p - %p - %d\n", &n, n, *n);
	n = new_arr;
	printf("Nuovo valore di n: %p - %p - %d\n", &n, n, *n);
}
```

Eseguendo il programma:

```
./a.out
Primo valore di p: 0xbfeeab30 - 0xbfeeab08 - 0
Valore di n: 0xbfeeaaf0 - 0xbfeeab08 - 0
Nuovo valore di n: 0xbfeeaaf0 - 0xbfeeaad4 - 1
Valore di p dopo func_call: 0xbfeeab30 - 0xbfeeab08 - 0
```

Come potete notare il risultato non è affatto quello che desideravamo avere!

Esercizio 1:
Provate a spiegare perchè non otteniamo il risultato desiderato.

2. Creazione di dati dinamici

I puntatori sono il tipo di dato per la creazione di strutture di dati allocate dinamicamente per antonomasia.

Ma cosa vuol dire questo esattamente?

Significa una struttura di dati che può essere allocata a seconda delle necessità in ogni punto del programma e non obbligatoriamente all'inizio di una funzione.

Facciamo un po' di chiarezza. Sostanzialmente esistono due metodi di allocazione: statico e dinamico.

Il metodo statico viene utilizzato per definire al momento della dichiarazione quanto spazio va allocato per quel tipo di dato, che rimarrà costante fino al termine della funzione in cui si trova (o a seconda delle regole di scope[1] in cui si trova).

Esempio:

```
int main()
{
     int array[10];
}
```

In questo modo abbiamo definito un array di 10 interi e tale rimarrà fino alla fine dell'esecuzione della funzione *main*. Questo perchè sullo stack vengono concessi 10 spazi per interi, che al termine della funzione vengono utilizzati per altri valori (di altre funzioni).

Infatti in questa situazione possiamo capire meglio cosa accade:

```
/* ret_array ritorna l'indirizzo base di un array allocato
staticamente nella funzione*/
int * ret_array(int n)
{
   int arr[n];
   int i;
   for (i = 0; i < n; i++)
   {
      arr[i] = i*2; /* associamo il doppio di i ad ogni indice */
   }
   printf("Indirizzo base dell'array: %p\n", arr);
   return arr;
}
```

1 Regole di scope: in sostanza sono quelle regole che dettano gli attributi delle variabili create: una variabile può essere globale, statica, esterna e infine locale. Per approfondimenti, vedere i riferimenti esterni.

```
/*
Come prima, anche qui ritorniamo l'indirizzo base dell'array
allocato staticamente nella funzione.
*/

int * ret_oth_array(int n)
{
   int arr[n];
   int i;
   for (i = 0; i < n; i++)
   {
      arr[i] = i+n; /* solo che qui facciamo partire il conteggio da
                       n, per poi aggiungerlo ad i.
                    */
   }
   printf("Indirizzo base dell'array: %p\n", arr);
   return arr;
}
```

Con un main del genere:

```
int main(void)
{
   int *p = NULL;
   int *x = NULL;
   p = ret_array(5);
   x = ret_oth_array(6);

   printf("Indirizzo contenuto in p: %p\nValore *p: %d\n", p, *p);
}
```

Compilandolo otterremmo due warning:

```
funzioni4.c: In function ‘ret_array’:
funzioni4.c:12: warning: function returns address of local
variable
funzioni4.c: In function ‘ret_oth_array’:
funzioni4.c:24: warning: function returns address of local
variable
```

Tralasciando i warning, che talvolta – come in questo caso - sono più importanti degli errori, vediamo cosa accade eseguendo il programma.

```
Indirizzo base dell'array: 0xbf8a4c70
Indirizzo base dell'array: 0xbf8a4c70
Indirizzo contenuto in p: 0xbf8a4c70
Valore *p: 6
```

Sul mio computer gli indirizzi coincidono, ma il valore di *p è diverso da quello che che desideravamo. E' "un caso"?

E' molto probabile di no, dato che come ho già detto il puntatore allo stack deve ritornare sempre al punto di partenza, altrimenti incorriamo in problemi.

Analizziamo cosa ha fatto il compilatore per noi:

Righe 1,2 del main: inizializziamo i due puntatori a NULL, riservando 8 byte per queste due variabili sullo stack.

Incrementiamo dunque il puntatore allo stack di 8 byte.

Supponendo che il puntatore allo stack all'inizio fosse 0, ora si troverà a -8 (perchè decresce).

Fig. 2: All'interno del main definiamo due variabili, che occupano 8 byte in tutto. Incrementiamo il puntatore a -8

*Riga 3 del main: o*ra dobbiamo passare un argomento alla funzione, quindi riserviamo altri 4 byte per lo stack - e il puntatore passa a -12 - e altri 4 byte per l'indirizzo di ritorno - quindi il puntatore è a -16.

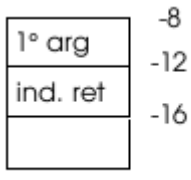


Fig. 3: Prima della chiamata a funzione, salviamo sullo stack l'argomento da passare e l'indirizzo di ritorno.

All'interno della funzione *ret_array* ci troviamo in una situazione del genere:

aggiungi N locazioni di memoria usando il puntatore allo stack, partendo da -16, e assegna ad ognuna di esse il valore di **i*2**.

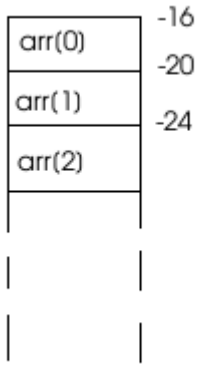


Fig. 4: Siamo all'interno della funzione e assegniamo valori, salvandoli sullo stack.

Al termine della funzione, fai ritornare lo stack pointer esattamente a -12 (indirizzo di ritorno).

Questo poi diventerà -12 (dove si trova ancora il primo argomento) e infine -8 (dove si trova la variabile int *x).

A questo punto cosa conterrà la variabile *p*?

Se abbiamo fatto bene i conti, dovrebbe contenere l'indirizzo *-16* che a sua volta avrà valore *i*2* (in questo caso = 0, perchè *i* all'inizio è 0).

Siamo ora alla *riga 4 del main*: il puntatore allo stack lo stack pointer è a -8, quindi salviamo l'argomento da passare alla funzione e chiamiamola.

Stesso procedimento di prima: a -8 si trova l'argomento, a -12 l'indirizzo di ritorno della funzione.

A -16 si trova l'indirizzo base del nuovo array. Stavolta però assegneremo all'array dei valori diversi: **i+n**. Il valore all'indirizzo -16 quindi sarà +6, perchè n vale 6. Quello all'indirizzo -20 (ovvero il secondo elemento dell'array) avrà valore 1+6, e così via.

Uscendo dalla funzione, ci troviamo quindi con questa situazione:

p ed *x* contengono entrambe -16.

Provando a stampare il valore di p però vediamo che non risulta ciò che ci aspettavamo.

**p* infatti è uguale a +6, ovvero il valore che attualmente si trova all'indirizzo -16 dello stack.

A questo punto quindi qualcuno potrebbe chiedersi: "come posso far sì che *p* punti a ciò che dico io e non allo stack, che a quanto pare cresce e decresce?"

La risposta è il titolo del paragrafo: con l' allocazione dinamica.

In pratica quando creiamo un processo, questo dispone di alcune aree di memoria (più precisamente "segmenti"). Tra queste c'è lo stack, che è un'area il cui scopo è proprio quello che abbiamo visto sinora: contenere variabili locali, gestire le chiamate a funzioni e così via.

Tuttavia esiste anche un'altra area: lo spazio di heap.

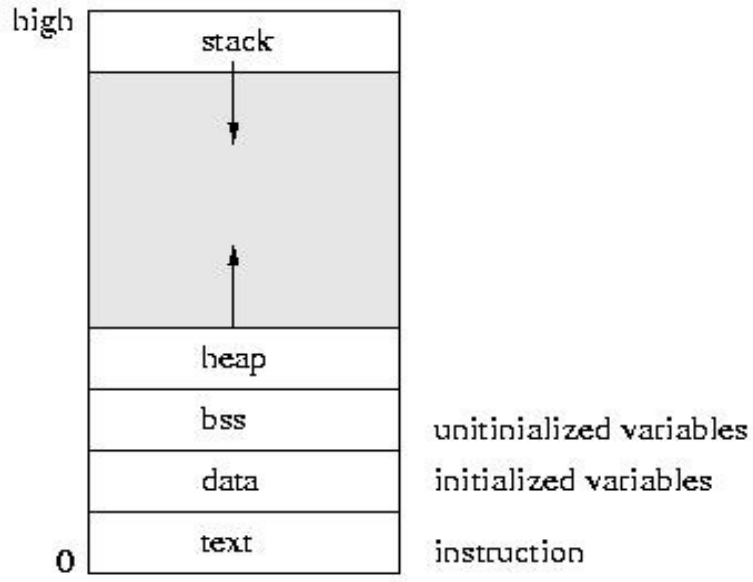


Fig. 5: Tratto da http://www.andrew.cmu.edu/course/15-412/ln/proccontext.jpg

Lo spazio di heap è sostanzialmente la regione di memoria in cui allochiamo dei dati "dinamici", in modo che chiamando una funzione o uscendo da un'altra i dati non scompaiano o vengano sovrascritti come avviene per lo stack.

Se per le funzioni c'era l'idea dello stack che cresceva e decresceva a seconda delle esigenze, per i dati allocati dinamicamente sarà stato ideato qualcosa sicuramente.

E infatti è così.

Attraverso l'uso di chiamate si sistema (*system call*) sono state implementate delle funzioni nella libreria C che ci semplificano il lavoro.

E queste sono sostanzialmente quattro:

- malloc : alloca una quantità di byte per il nostro tipo di dato. Tale quantità viene passata come argomento.
- calloc : fa sostanzialmente ciò che fa la malloc, solo che alloca memoria in modo contiguo e azzera i byte automaticamente.
- free : libera la memoria utilizzata dal tipo di dato specificato
- realloc: riassegna una nuova porzione di memoria, nel caso che la quantità precedente fosse più piccola o grande di quella da usare.

L'allocazione della memoria è uno degli argomenti più fraintesi e complicati nello sviluppo di un programma.

Molti linguaggi di programmazione di alto livello infatti fanno sì che quando un dato viene creato, esso verrà distrutto automaticamente al termine dell' esecuzione o quando non è più utilizzato[1].

In C invece il programmatore deve prestare attenzione a quando alloca memoria per un tipo di dato, perchè quando non ne ha più bisogno dovrà ricordarsi di eliminarlo manualmente, con una chiamata alla *free*.

Altrimenti, se il programma continua a richiedere memoria, rischia di incappare in un memory leak[2]. Inoltre, se il programmatore non libera la memoria, questa potrà essere disponibile solo al riavvio del sistema operativo.

Proviamo quindi a correggere il programma precedente utilizzando l'allocazione dinamica.

```
int main(void)
{
   int *p = NULL;
   int *x = NULL;
   p = ret_array(5);
   x = ret_oth_array(6);

   printf("Indirizzo contenuto in p: %p\nValore *p: %d\n", p, *p);
}
```

Passiamo ora alle due funzioni:

```
/* ret_array ritorna l'indirizzo base di un array allocato
dinamicamente nella funzione*/
int * ret_array(int n)
{
   int * arr = (int *) malloc(10 * sizeof(int));
   int i;
```

1 Questo meccanismo viene chiamato Garbage Collection. Per maggiori informazioni, vedi i riferimenti esterni.
2 Un memory leak si ha quando il programmatore lascia allocare memoria senza però liberarla quando non ne ha bisogno. In questo modo il bisogno di memoria del programma crescerà a dismisura.

```
   for (i = 0; i < n; i++)
   {
      arr[i] = i*2; /* associamo il doppio di i ad ogni indice */
   }
   printf("Indirizzo base dell'array: %p\n", arr);
   return arr;
}
```

Riferimenti Esterni

Di seguito sono elencati alcuni siti da cui potete trarre maggiori informazioni:

- Puntatori in C++
- Regole di Scope (Wikipedia)
- Regole di Scope (in C)
- Garbage Collection (Wiikpedia)
- Memory Leak (Wikipedia)

Bibliografia

Di seguito potete trovare alcuni dei libri più interessanti e importanti sul C: