# Parli del Diavolo... e Spuntano i Puntatori!

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2016-17

In questa dispensa verranno introdotti due concetti fondamentali della programmazione C: puntatori e i passaggi di parametro per indirizzo. Verranno inoltre analizzati due fenomeni che originano da questi costrutti linguistici: *alias* e *side effects*.

## 1 Puntatori

I *puntatori* sono uno degli aspetti più caratterizzanti del linguaggio C. Un puntatore altro non è che un indirizzo di memoria. Mentre il puntatore è praticamente l'unico modo disponibile per accedere alla memoria in linguaggio macchina (o *assembler*), molti linguaggi di programmazione moderni (ad esempio Java o i linguaggi funzionali) nascondono al programmatore, per amore di astrazione, questo dettaglio di basso livello[1]. In C, viceversa, capire a fondo i puntatori e avere dimestichezza con essi è fondamentale. Nel corso di queste note (e in quelle future) dovremmo fare i conti con i puntatori relativamente ad almeno tre problemi: il passaggio dei parametri, la gestione dei vettori e l'allocazione dinamica di memoria.

Cominciamo con il descrivere brevemente la dichiarazione di un puntatore e gli operatori base sui puntatori. Una variabile puntatore viene dichiarata nel seguente modo: `T* x`, dove `T` è un qualsiasi tipo. Ad esempio data la dichiarazione `int* x`, `x` sarà una variabile il cui contenuto è un puntatore (o riferimento) ad una cella di memoria che contiene un intero. Un particolare tipo puntatore che vedremo in dettaglio nel prossimo futuro è il `void *`.

Ci sono due fondamentali operatori legati ai puntatori: l'operatore di *dereferenziazione* `&` e l'operatore di *referenziazione* `*`. Data una variabile `x` di tipo `T`, `&x` restituisce *l'indirizzo di memoria* in cui è memorizzata `x` (attenzione quindi che l'espressione `&x` ha tipo `T*`). Viceversa, data una variabile `x` di tipo `T*`, `*x` restituisce il contenuto della cella puntata da `x` (attenzione quindi che l'espressione `*x` ha tipo `T`). Gli operatori `&` e `*` sono in un certo qualsenso l'uno l'inverso dell'altro, in quanto

[1] viene tradizionalmente detto di *basso livello* un linguaggio in cui gli algoritmi sono espressi in un modo "vicino" alla logica della macchina, mentre viene detto di *alto livello* un linguaggio che cerca di essere il più vicino possibile alla logica della soluzione dei problemi.

```
void provaPuntatori(){
   int m=3;
   int n=1;
   int p;
   int* x= &p;
   int* y=NULL;
   int* z= &m;  /* Punto 1 */

   (*z)++;
   y=&m;
   z=&p;
   p=7;
   (*z)++;
   x++;     /* Punto 2 */
   }
```

Figura 1: Funzione per sperimentare sui puntatori

le espressioni `* (& x)` e `& (* x)` restituiscono entrambe il *valore* della variabile `x`. Osservate che, nel primo caso `x` deve avere tipo `T`, mentre nel secondo `T *`. Osserviamo inoltre che l'operatore di referenziazione `*` non può essere applicato se `T` è il tipo `void`, cioè se `x` è stata dichiarata di tipo `void *`. Tra tutti i puntatori, va subito ricordata la costante `NULL` che indica il puntatore a niente e che viene interpretato come `false` nelle condizioni logiche. Tutti gli altri puntatori vengono interpretati come `true`. `NULL` è il tipico valore a cui inizializzare una variabile puntatore.

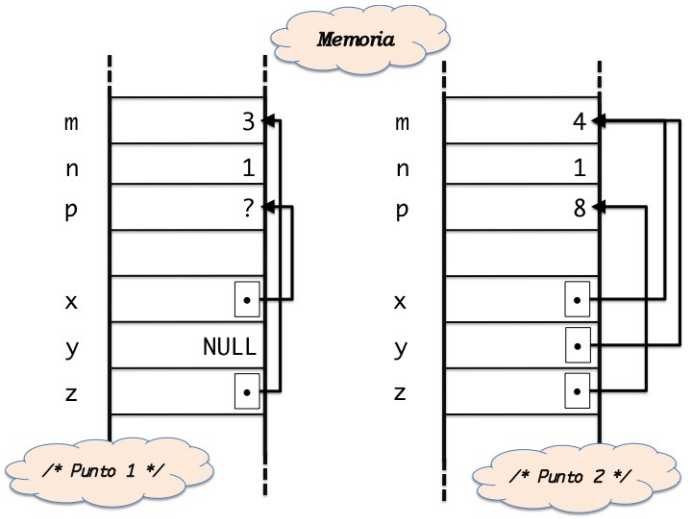


Figura 2: Stato della memoria nella funzione `provaPuntatori` in Fig. 1

Considerate la funzione in Fig. 1. Lo stato della memoria nei due punti se-

gnalati con i commenti `/* Punto 1 */` e `/* Punto 2 */` è mostrato in Fig. 2. In `provaPuntatori()` sono definite tre variabili intere e tre variabili puntatore. Osservate subito che la definizione di `x`, `y` e `z` non *alloca* spazio per memorizzare interi. Le variabili puntatore contengono un puntatore a celle di memoria delle variabili intere oppure `NULL`. L'istruzione `(*z)++;` in realtà incrementa il *contenuto* della variabile `m`, perché `z` contiene un puntatore a `m`. Dopo l'istruzione `y=&m;`, anche `y` contiene un puntatore a `m`. Stampando il valore di `*z` e `*y` otterremo comunque 4. L'istruzione `z=&p;` invece sposta il puntatore di `z` all'area di memoria di `p` che al momento ancora non è inizializzata. Dopo i comandi `p=7; (*z)++;` avremmo che `p` vale 8. L'ultima e più misteriosa istruzione è: `x++;` che, badate bene, *non incrementa* la variabile `p` a cui `x` punta. Incrementa il pointer contenuto in `x`. Facciamo la conoscenza con la cosidetta *aritmetica dei puntatori*, croce e delizia dei veri programmatori `C`. Incrementare un puntatore `x` di tipo `T*` significa *spostare in avanti* il puntatore `x` di tante caselle quante ne occupa un dato di tipo `T` (questo valore si può conoscere invocando la funzione `sizeof(T)`). Tranne nei casi in cui sia nota la strategia di allocazione (ad esempio vedremo che gli elementi di un vettore occupano celle di memoria contigue) *è bene evitare di speculare* su come il programma eseguibile organizzerà la memoria[2]. Ad esempio, con mia sorpresa, sul mio calcolatore, dopo `x++;` ho scoperto che `x` punta a `m` e non a `p` come mi sarei aspettato. Inoltre, cambiando l'ordine testuale in cui si sono dichiarate le variabili (o aggiungendo altre dichiarazioni), potrebbe cambiare il comportamento del programma. Di conseguenza, suggerisco di lasciare ai veri programmatori `C` il piacere perverso di scrivere programmi che fanno un uso spregiudicato dell'aritmetica dei puntatori.

## 2 Passaggio di Parametri

Poniamoci ora il problema di scrivere una funzione che scambia il contenuto di due celle di memoria. In Fig. 3 vediamo una prima versione. Per inciso, osserviamo che per scambiare il contenuto di due variabili è necessario fare uso di una variabile di appoggio (`h` nel nostro caso). Ci troviamo nella stessa situazione dell'oste disonesto che vuole scambiare il contenuto di una bottiglia di vino pregiato con il contenuto di una bottiglia di vino dozzinale (al fine ingannare il cliente). Per fare questa operazione avrà bisogno di una terza bottiglia. Infatti, l'operazione di assegnazione è un'operazione distruttiva, e se cominciassimo lo scambio per esempio con `a=b;` perderemo irrimediabilmente il valore memorizzato in `a`. La funzione `scambia1`, tuttavia non dà i risultati attesi.

Come già detto, infatti, funzioni diverse hanno spazi di nomi diversi, e i parametri formali sono variabili *locali alla funzione* dentro cui viene *ricopiato il valore*

[2]Tale strategia infatti non è specificata dal linguaggio e differenti compilatori (e addirittura differenti versioni dello stesso compilatore), possono comportarsi in modo diverso, causando errori molto subdoli e quindi difficili da scoprire.

```
void scambia1(int b, int a){
  int h;

  h=a;
  a=b;
  b=h;
 }
```

Figura 3: `scambia` (scorretta)

```
void scambiaGlobal(){
  int h;

  h=a;
  a=b;
  b=h;
 }
```

Figura 4: `scambia` (sconveniente)

dell'espressione contenuta nei parametri attuali. Chiamando la funzione `scambia1` ad esempio dal `main`, otterremo una memoria che astrattamente si può rappresentare come in Fig. 5. L'esecuzione di una funzione avviene dopo aver *allocato* sullo *stack* (o pila di sistema) un *activation record* (o record di attivazione). Il record di attivazione contiene una serie di informazioni necessarie all'esecuzione della funzione e al corretto rientro quando la funzione finisce di eseguire. Sostanzialmente contiene variabili per memorizzare le variabili locali, i parametri e il *punto di ritorno*, cioè l'indirizzo dell'istruzione da eseguire *dopo* la fine dell'esecuzione della funzione stessa. Questa informazione è necessaria in quanto la funzione potrebbe essere chiamata da molti punti diversi durante l'esecuzione di un programma.

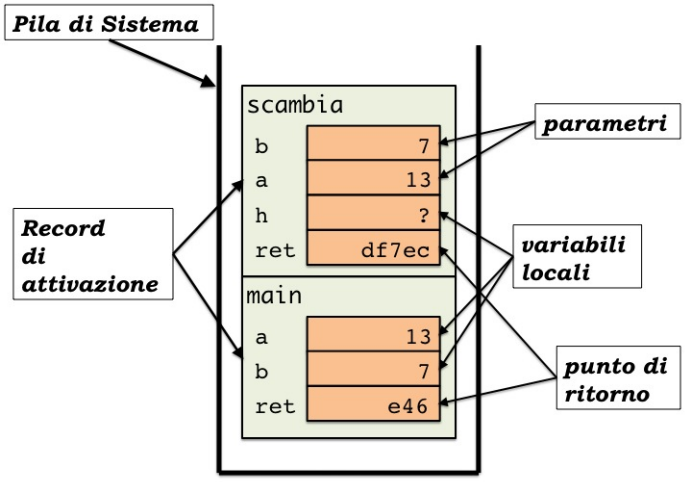


Figura 5: Stack di attivazione delle funzioni

In Fig. 6 viene riassunta l'evoluzione della memoria per effetto della funzione `scambia1`. Come vedete, lo scambio avviene correttamente, ma tra i valori dei parametri della funzione `scambia1` senza nessun effetto sulle variabili che effettivamente vorremmo scambiare.

Una soluzione molto naïf sarebbe dichiarare le variabili `a` e `b` come globali e definire una funzione `scambiaGlobal` come in Fig. 4. Questa volta otterremo gli

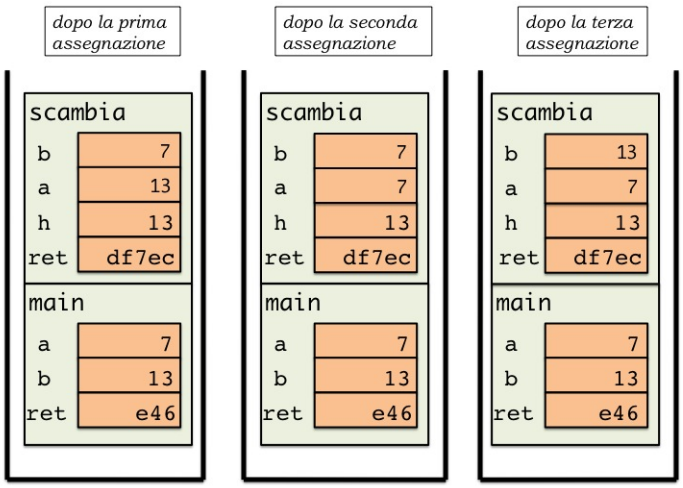


Figura 6: Esecuzione della funzione `scambia1` in Fig. 3

effetti desiderati, ma a caro prezzo! Infatti, saremmo obbligati a definire le variabili `a` e `b` come globali e inoltre la funzione svolgerebbe correttamente il suo compito *solo* per due variabili globali che si chiamino `a` e `b`. Inutile dire che tale soluzione è insoddisfacente e del tutto contraria al principio di scrivere funzioni riusabili il più possibile, attraverso una attenta scelta dei parametri. Il lettore attivo, starà già pensando che nella sezione precedente abbiamo visto come sia possibile, attraverso i puntatori, creare "legami" tra variabili diverse, o se preferite, far condividere la *stessa cella di memoria* a più variabili. O se preferite ancora, dare due nomi alla stessa cella di memoria (*alias*).

In Fig. 7 vediamo finalmente una soluzione soddisfacente al problema di scrivere una funzione che scambia il contenuto di due variabili. Il trucco consiste nel passare alla funzione parametri che sono *l'indirizzo di memoria* (o il riferimento) delle variabili di cui si vuole scambiare il contenuto. In Fig. 9 è riassunta l'evoluzione della memoria per effetto di una chiamata alla funzione `scambia2` in Fig. 7. Facciamo notare che questa volta i parametri `a` e `b` sono *puntatori a interi*. Di conseguenza, dovremmo fare attenzione che per usare o modificare il loro valore è necessario usare l'operatore di referenziazione `*`. Inoltre, dovremo fare attenzione anche nella chiamata della funzione, ed invocare `scambia2` con la chiamata `scambia2(&a, &b);` in quanto non dobbiamo tanto passarle i valori delle variabili `a` e `b`, quanto piuttosto i loro *indirizzi di memoria*.

Tale tecnica di passaggio di parametro viene detta *passaggio di parametri per riferimento* (o per *indirizzo*). A differenza di altri linguaggi, il C non ha bisogno di una distinzione netta tra passaggi per valore e passaggi per indirizzo: in C tutti i passaggi di parametro sono per valore, ma siccome il linguaggio permette di trattare in modo esplicito i puntatori, è possibile passare per valore puntatori come parametri, e ottenere quindi l'effetto dei passaggi di parametro per riferimento. Aldilà di

```
void scambia2(int* b, int *a){
  int h;

  h=*a;
  *a=*b;
  *b=h;
 }
```

Figura 7: Funzione `scambia` con passaggio per riferimento

```
void scambia3(int* x, int *y){

  *x=*x-*y;
  *y=*y+*x;
  *x=*y-*x;
 }
```

Figura 8: Funzione `scambia` senza variabile di appoggio

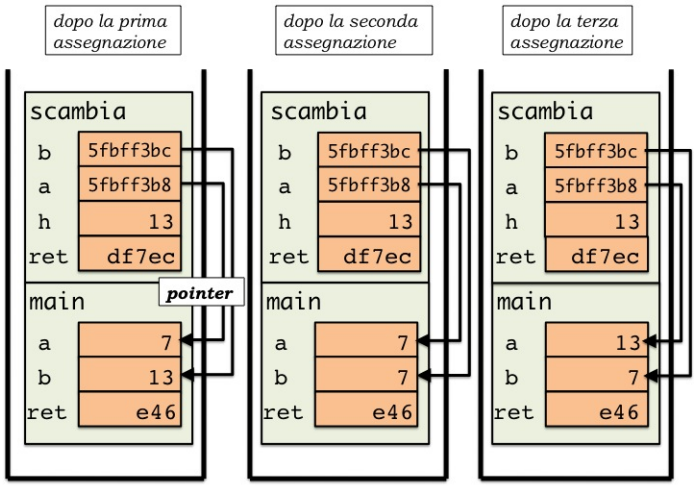


Figura 9: Esecuzione della funzione `scambia2` in Fig. 7

quest'ultima discussione metafisica che probabilmente appassionerà solo l'aspirante studioso della teoria dei Linguaggi di Programmazione, è bene ricordare che i passaggi di parametro per indirizzo sono *necessari* ogni qualvolta si desidera che una funzioni modifichi lo stato (cioè il valore delle variabili) del chiamante. Rispetto all'uso di variabili globali, questa tecnica presenta almeno due grossi vantaggi: a) permette la parametrizzazione e b) gli effetti sullo stato del chiamante (cioè quali variabili del chiamante possono essere modificate) sono chiari semplicemente guardando la chiamata della funzione (sono le variabili precedute dal simbolo `&`). Infine, possono essere molto utili anche nel caso in cui una funzione debba restituire più di un risultato (vedi Esercizio 3.1(5) qui sotto). L'idea è che una funzione, oltre al valore ritornato, può comunicare con il chiamante, modificando il valore di una variabile passata per riferimento.

## 3 Side Effects ed Alias

Chiudiamo il nostro primo incontro con i puntatori, osservando che i puntatori (e i passaggi di parametri per indirizzo) introducono dei fenomeni che possono rendere la leggibilità e la verifica di correttezza dei programmi problematica. Consideriamo una nuova funzione `scambia3` che effettua lo scambio del contenuto di due variabili (vedi Fig. 8). Si tratta di una procedura alquanto arguta, che permette di scambiare il valore di due variabili intere (o comunque numeriche) senza l'uso di una variabile d'appoggio. Si fonda sull'osservazione che per ricostruire correttamente i valori del risultato, è sufficiente mantenere in memoria uno dei due valori e la loro differenza. Infatti, dati due numeri $x$ e $y$, avremo che $x = y + (x - y)$ e $y = x - (x - y)$. Analizziamo in dettaglio questa procedura, aldilà delle convenienze pratiche di usare questo algoritmo (risparmia una variabile ausiliaria al prezzo di eseguire 3 operazioni aritmetiche, e di avere una soluzione che non si può generalizzare a tipi di dato che non si comportino come un gruppo commutativo rispetto a una qualche operazione).

Probabilmente, ciascuno di voi, eseguendo manualmente una traccia di esecuzione (cioè scrivendo il valore di ogni variabile dopo ogni istruzione) concluderà che la funzione è corretta, cioè invocata con `scambia3(&a,&b);` scambia correttamente i contenuti delle variabili `a` e `b`. Solo qualcuno particolarmente malizioso, si metterà ad analizzare il caso limite in cui `x` e `y` sono riferimenti alla *stessa cella di memoria*. Qualche lettore potrebbe spazientirsi: chi mai potrebbe essere tanto sciocco da scrivere una chiamata `scambia3(&a,&a);`?

La realtà, purtroppo, non è mai così semplice. Abbiamo visto infatti, che l'uso dei puntatori introduce nei programmi degli *alias*: cioè nomi diversi per lo stesso oggetto. Per esempio `x` e `y` *potrebbero* in una qualche esecuzione puntare alla stessa cella di memoria. La cosa potrebbe non essere evidente neanche al chiamante: ad esempio, se al termine della funzione `provaPuntatori` in Fig. 1 aggiungessimo la chiamata `scambia3(x,y)` non avremmo immediata evidenza che `x` e `y` siano alias, in quanto entrambi sono puntatori alla cella di memoria in origine allocata per la variabile `m`. Analogamente, vedremo con gli array, che scritture come `a[i]` e `a[j]` potrebbero essere alias, perché a dispetto della loro rappresentazione sintattica, ad un certo punto dell'esecuzione, `i` e `j` potrebbero avere lo stesso valore.

Gli alias sono pericolosi perché usualmente facciamo l'ipotesi (implicita) che nomi diversi indichino cose diverse. Fatta questa premessa, se `x` ed `y` sono puntatori alla stessa cella di memoria, nella funzione `scambia3`, la prima assegnazione `*x=*x-*y;` azzera immediatamente tale cella di memoria, perdendo irrimediabilmente il valore memorizzato in quella cella all'ingresso della funzione. Tutte le operazioni successive, infatti, continueranno a scrivere 0 su tale cella di memoria, indipendentemente dal valore iniziale.

L'altro fenomeno pericoloso, già accennato, è quello dei *side effects*, cioè gli effetti collaterali che l'esecuzione di una funzione ha sullo stato del chiamante o sullo stato globale. Lo stato del chiamante si può modificare solo attraverso i parametri passati

per indirizzo, e quindi è quantomeno chiaro dal prototipo della funzione o dalla sua chiamata (vedremo che questo fenomeno può inasprirsi quando parleremo di memoria allocata dinamicamente). Lo stato globale (ossia il valore delle variabili globali) viceversa potrebbe essere modificato senza che ciò sia chiaro dal prototipo o dalla chiamata della funzione. L'uso di side effects va quindi limitato ai casi di estrema necessità, mentre va lasciato ai veri programmatori C il malcostume di usare e modificare variabili globali. Loro, se lo fanno, avranno dei buoni motivi!

# 4 Esercizi e Spunti di Riflessione

1. ▸ Nella funzione `printf`, `%p` è il formato di stampa dei puntatori. Provate a stampare gli indirizzi di alcune variabili.

2. ▸ Provate a stampare una variabile puntatore con un formato intero. O ad assegnare un intero a un puntatore o viceversa. Come risponde il compilatore?

3. ▸ Provate a stampare il nome di una funzione. Ad esempio `printf("%p\n", main);` Riuscite a dare un'interpretazione al risultato?

4. ▸ Definite due variabili `int m;` e `long int n;`. Definite poi due pointer `int * x = &m;` e `long int* y== &m;`. Poi, sfruttando l'aritmetica dei puntatori eseguite i comandi `x++;` e `y++;`. Stampate i valori di `x` e `y` prima e dopo l'incremento. Cosa notate? (se non siete abituati a contare in esadecimale, sfidate le ire del compilatore e stampate i puntatori col formato `%u`).

5. Scrivere una funzione `void div(int m, int n, int* q, int* r)` che calcoli la divisione intera tra `m` ed `n`, restituendo i risultati nei parametri `q` ed `r`. (Osservate che una chiamata avrà la forma `div(m, n, &q, &r);`). Modificate la funzione `mcdDellaMaestra`, usando questa funzione invece delle funzioni `div` e `resto` come fatto nelle precedenti note. Quali benefici ne otteniamo?

6. Scrivere una funzione `int scomponi(int m, int *p, int* q)` che restituisca 1 se `n` si può scrivere come il prodotto di due interi (diversi da 1) e 0 altrimenti. Se il risultato ritornato è 1, i parametri passati per indirizzo verranno caricati due numeri il cui prodotto vale $m$.

7. Usare la funzione `scomponi` dell'esercizio precedente per scrivere una funzione che stampa tutti i fattori primi di un numero.

8. Scrivere una funzione `int goldbach(int m, int *p1, int* p2, int* p3)` che restituisca 1 se `n` si può scrivere come la somma di tre primi e 0 altrimenti. Se il risultato ritornato è 1, i parametri passati per indirizzo verranno caricati tre numeri primi la cui somma è `n`. Trovare un numero per cui la funzione `goldbach` ritorna 0 ☺.