

Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Puntatori e Passaggi di Parametri

Corso di Laurea in **Informatica**, III anno



SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **20**, 6 maggio 2021



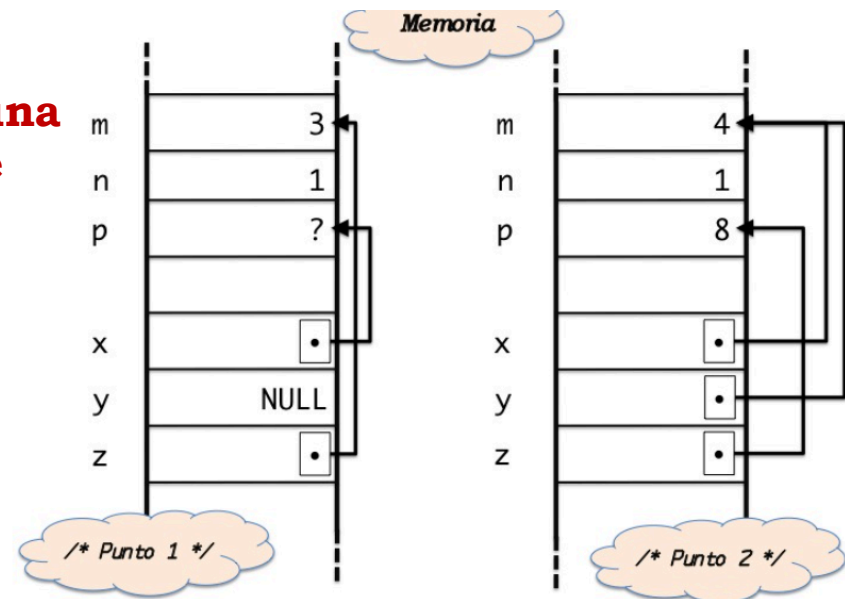
Lezione 20a:

*Parli del Diavolo ...
... e spuntano i puntatori*

Parli del diavolo, spuntano puntatori

Una variabile puntatore è una variabile che **contiene un indirizzo di memoria**, ad esempio di un'altra variabile.

```
void provaPuntatori(){
  int m=3;      & è l'operatore di
  int n=1;      referenziamento: data una
  int p;        variabile ne restituisce
  int* x= &p;   l'indirizzo
  int* y=NULL; /* NULL è la costante
               * "puntatore a niente" */
  int* z= &m;
               /* Punto 1 */
  (*z)++;
  y=&m;
  z=&p;
  p=7;
  (*z)++;
  x++;        /* Punto 2 */
}
```



Attenzione alla notazione * sui tipi e all'operatore di dereferenziazione *. E all'operatore di referenziamento &.

Puntatori e passaggi di parametri

In C i passaggi di parametri **sono tutti per valore**: ma siccome ci sono i puntatori, posso passare puntatori a una funzione.

```
void scambia(int a, int b){  
    int h = a;  
    a = b;  
    b = h;  
}
```

Questa funzione **scambia** correttamente i valori delle variabili **a** e **b** **LOCALI** alla funzione **scambia**.

Ma non ha **nessun effetto** sullo stato del chiamante.

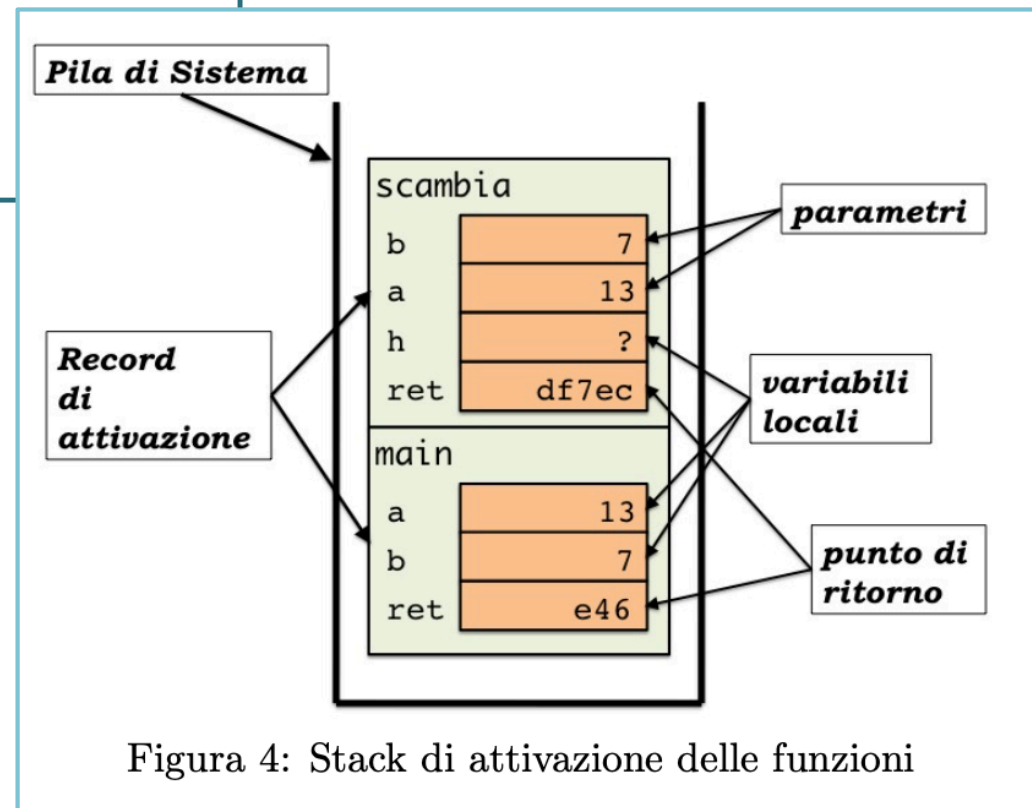


Figura 4: Stack di attivazione delle funzioni

Esempio di Esecuzione

Infatti la funzione `scambia1` agisce sul suo stato locale, mentre le variabili del chiamante rimangono non coinvolte.

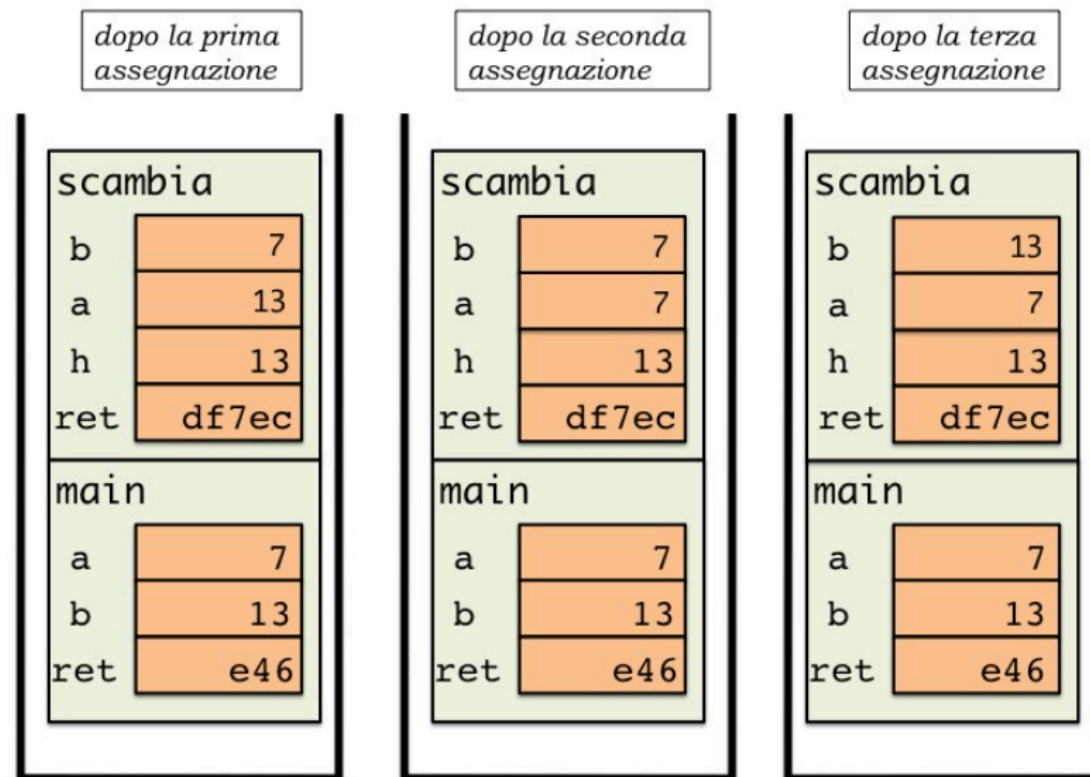


Figura 5: Esecuzione della funzione `scambia1` in Fig. 3

Puntatori e passaggi di parametri 2

Per modificare i valori delle variabili del chiamante dobbiamo **usare i puntatori**. Ecco la versione corretta della funzione `scambia`.

```
void scambia(int *a, int *b){  
    int h = *a;  
    *a = *b;  
    *b = h;  
}  
scambia(&b, &a)
```

Nella funzione `scambia` le variabili `a` e `b` sono pointer alle variabili del chiamante.

Le modifiche a `*a` ed a `*b` modificano direttamente le variabili `a` e `b` del `main`.

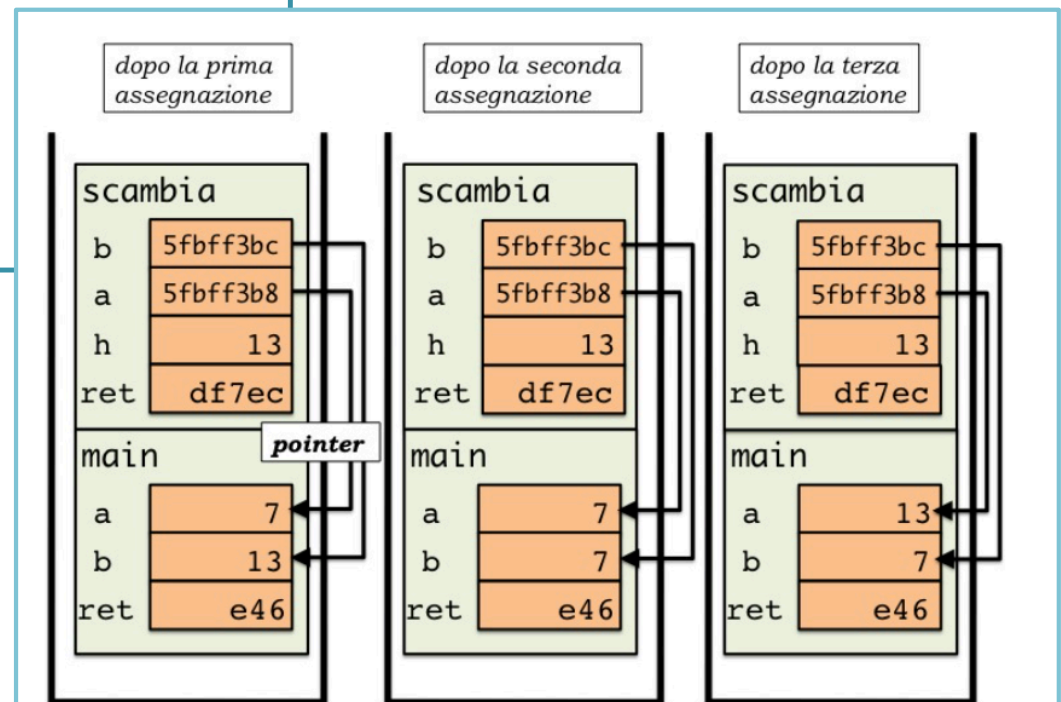


Figura 7: Esecuzione della funzione `scambia2` in Fig. 6

Alias & side-effects

Vediamo alcuni curiosi fenomeni generati dal passaggio di parametri, specie nel caso di passaggi per indirizzo. Li vedremo usando un'altra versione della funzione **scambia**.

```
void scambia(int *a, int *b){  
    *a = *a - *b;  
    *b = *b + *a;  
    *a = *b - *a;  
}
```

Questa funzione sembra corretta. Per ricostruire il valore di due numeri è sufficiente conoscerne uno dei 2 e la loro differenza. Immaginiamo $a=3$ e $b=5$.

- Dopo la prima assegnazione, a diventa $a - b = -2$ e b rimane 5;
- Dopo la seconda, b diventa $a + b = -2 + 5 = 3$.
- Dopo la terza, a diventa $b - a = 3 - (-2) = 5$, e b rimane 3.

Cosa c'è che non va?

Alias & side-effects

```
void scambia(int *a, int *b){  
    *a = *a - *b;  
    *b = *b + *a;  
    *a = *b - *a;  
}
```

Una chiamata `scambia(&a, &a)` dopo l'esecuzione della prima istruzione, siccome in tal caso `a` e `b` denotano **la stessa cella di memoria**, **entrambe le variabili si azzerano**, perdendo ogni possibilità di ricostruire i valori iniziali.

Ma chi è così scemo da scambiare i contenuti della stessa cella di memoria? **Noi programmatori!**

La chiamata potrebbe essere `scambia(&a[i], &a[j])` in cui non è immediatamente evidente che in qualche esecuzione `i` e `j` hanno lo stesso valore.

Esempio: Selection Sort

```
int minimo(int a[], int inf, int sup){
/* REQ: inf < sup
 * ENS: ritorna m, forall j.inf<=j<sup
        a[m]<=a[j] */

void selectionSort(int a[], int n){
    for (int i=0; i<n; i++){
        int m = minimo(a, i, n);
        scambia(&a[i], &a[m]);
    }
}
```

In questa funzione, non si va a verificare $m \neq i$ prima di eseguire lo scambio.

Non è immediatamente evidente dal testo che $\&a[i]$ e $\&a[m]$ possono denotare la **stessa cella di memoria!**

Attenzione: noi facciamo implicitamente l'ipotesi mentale che **nomi diversi indicano cose diverse.**

Soluzioni (1)

```
void scambia(int *a, int *b){  
    /* REQ: a != b */  
    *a = *a - *b;  
    *b = *b + *a;  
    *a = *b - *a;  
}  
  
void selectionSort(int a[], int n){  
    for (int i=0; i<n; i++){  
        int m = minimo(a, i, n);  
        if (i!=m) scambia(&a[i], &a[m]);  
    }  
}
```

*metto una
precondizione alla
funzione*

*In selectionSort
dobbiamo evitare di
scambiare celle uguali per
rispettare la REQ di
scambia*

Soluzioni (2)

```
void scambia(int *a, int *b){  
    if (*a != *b){  
        *a = *a - *b;  
        *b = *b + *a;  
        *a = *b - *a;  
    }  
}
```

*scambia solo quando
necessario: $*a \neq *b$ che
implica $a \neq b$*

```
void selectionSort(int a[], int n){  
    for (int i=0; i<n; i++)  
        scambia(&a[i], &a[minimo(a, i, n)]);  
}
```

*In selectionSort
posso ignorare se $i==m$*



Lezione 20b

*Tipici usi dei
passaggi per indirizzo*

div con risultati sui parametri

Abbiamo visto la divisione intera la scorsa lezione.
Probabilmente un Vero Programmatore C **scriverebbe un'altra funzione.**

Siccome **div calcola simultaneamente quoziente e resto** e spesso è utile sapere se un numero divide un altro...

```
int div(int m, int n, int* q, int *r){
/* REQ:n > 0
 * ENS:ritorna 1 se n divide m
 * MOD:*q(quoziente),*r(resto) di m/n
 */
    int *q = 0;
    int *r = n;
    while (*r >= n){
/* INV: m=q*n+r */
        *r -= n;
        *q++;
    }
    return (*r==0); /* return !*r */
}
```

MCD della Maestra

Filastrocca della Maestra:

Il massimo comun divisore di due numeri è il prodotto di tutti i fattori (primi) comuni, presi con il loro minimo esponente.

Letta questa filastrocca, per scrivere una funzione che calcola l'MCD seguendo questa strategia, **sembrerebbe necessario memorizzare i fattori primi** dei due numeri e poi scegliere opportunamente quelli da moltiplicare per calcolare l'MCD.

Ma è veramente necessario?

La scomposizione di 24 la vedo come $2*2*2*3$

La scomposizione di 36 la vedo come $2*2*3*3$

Posso scegliere quelli comuni producendoli in ordine, quando necessario!

MCD della Maestra

```
int mcdMaestra(int m, int n){
/* REQ: m, n > 0
 * ENS:ritorna MCD(m,n)
 */
    int mcd = 1;
    int p = 2;
/* occorre allocare queste variabili: */
    int q1, q2, r1, r2;
        (m)>(n)?(m) : (n)
    while (p < min(m, n)) {
/* INV: mcd*MCD(m,n)=MCD(m0,n0) */
        if (div(m,p,&q1,&r1)) m=q1;
        if (div(n,p,&q2,&r2)) n=q2;
        if (r1 && r2) p++;
        if (!r1 && !r2) mcd *= p;
    }
    return mcd;
}
```

*entrambi non sono
divisibili, incremento p*

*entrambi divisibili per
p, aggiorno mcd*

```
/* macro espansioni (pre-processor)
 * espressioni condizionali */
#define min(A, B) (A)>(B) ? (B) : (A)
```


Assegnamento Parallelo

Un'interessante istruzione propagandata da Dijkstra già negli anni '60 e raramente implementata nei LdP (eccetto Python!) è il cosiddetto assegnamento parallelo:

$$v[i], i = v[j], i+1 \neq i, v[i] = i+1, v[j]$$

$$x_1, x_2, \dots, x_n = exp_1, exp_2, \dots, exp_n$$

dove $exp_1, exp_2, \dots, exp_n$ sono valutate **nell'ambiente prima delle assegnazioni**.

Ad esempio **scambiare due variabili** x e y si può scrivere facilmente con: $x, y = u, x$

```
int assPar(int* x, int* y, int u, int v){
  /* MOD: *x e *y prenderanno i valori
  * di u e v
  */
  *x = u;
  *y = v;
}
```

Fibonacci con AssPar

Abbiamo visto la divisione intera la scorsa lezione.
Probabilmente un Vero Programmatore C scriverebbe la seguente funzione:

```
int fib(int n){
/* REQ: n > 1
 * ENS: ritorna fib(n)
 */
    int i = 1;
    int f1 = 1;
    int f2 = 1;

    if (n < 2) return n;
    while (i++ < n)
        assPar(&f1, &f2, f1+f2, f1);
/* INV: f1 = fib(i) */
    return f1;
}
```

Il problema del Massimo Primo

Esercizio 2 *Si supponga di avere a disposizione un esecutore la cui unica abilità aritmetica sia la seguente funzione:*

```
int scomponi(int n, int *f1, int *f2)
```

*che ha il seguente comportamento: sotto la precondizione che il parametro **n** sia un numero intero strettamente positivo restituisce come risultato 0 se **n** contiene un numero primo, e 1 se **n** contiene un numero composto. In tal caso, carica nei parametri **f1** ed **f2** due fattori (maggiori strettamente di 1) il cui prodotto è il valore passato nel parametro **n**.*

*Ad esempio, siano **x** ed **y** due variabili intere. La chiamata `scomponi(17, &x, &y)` restituisce 0. Viceversa la chiamata `scomponi(24, &x, &y)` restituisce 1 e carica **x** ed **y** con due fattori di 24, per esempio 4 e 6, oppure 8 e 3, oppure 12 e 2 (ma non possiamo fare ipotesi su quali essi siano).*

1. *Scrivere una funzione ricorsiva `maxPrimoRec(int m)` che usa `scomponi` per calcolare il massimo fattore primo di un numero positivo **m**;*

Il problema del Massimo Primo

Ovviamente il massimo divisore primo di un numero $n=f_1 * f_2$ sarà il massimo tra i massimi fattori primi di f_1 e f_2 .

Provare a scrivere una versione iterativa...

```
int maxPrimo(int n){
  /* REQ: n > 1
   * ENS: ritorna massimo fattore primo di n
  */
  int f1, f2;
  if (scomponi(n, &f1, &f2))
    return max(maxPrimo(f1), maxPrimo(f2))
  return n; /* n è primo */
}
```

Il problema del Massimo Primo

Come spesso accade, è interessante un errore che si potrebbe fare: dichiarare `f1` e `f2` come `int*`.

Il problema è che queste dichiarazioni **non allocano memoria per memorizzare un intero**. Il risultato più probabile è un errore di Segmentation Fault.

```
int maxPrimo(int n){
  /* REQ: n > 1
   * ENS: ritorna massimo fattore primo di n */
  int *f1, *f2;
  if (scomponi(n, f1, f2))
    return max(maxPrimo(*f1), maxPrimo(*f2))
  return n; /* n è primo */
}
```

Passaggio Parametri: C vs C++

In C++ vengono considerate le **reference** (simile anche in Pascal e altri linguaggi).

Personalmente preferisco la versione old-fashioned C che **esplicita anche nella chiamata i possibili side-effects!**

```
void scambia(int *a, int* b){  
    int h = *a;  
    *a = *b;  
    *b = h;  
}
```

```
/* chiamata */  
scambia(&x, &y);
```

*dereferenziazione
automatica*

*nessuna
informazione
nel chiamante*

```
void scambia(int &a, int &b){  
    int h = a;  
    a = b;  
    b = h;  
}
```

```
/* chiamata */  
scambia(x, y);
```

C

C++

if, ||, && e side-effects

Ovviamente, l'esistenza di side-effects rende ad esempio i consueti operatori aritmetici o booleani non-commutativi!

In quasi tutti i linguaggi, ciò dipende quasi esclusivamente da **side-effects** dovuti a **chiamate di funzione**, ma in **C** **anche comandi come il post-incremento** possono avere un ruolo.

```
... ; x=0 ;  
if (x++ && x==0) ...  
if (!n && div(m,n)) ...
```

In Haskell (vedi discussione su leftOr etc.) questa differenza riguarda solo il **rischio di non-terminazione**.

*void **

* è un costruttore di tipo a tutti gli effetti: dato un tipo T, permette di costruire il tipo T*.

Vedremo che è ad esempio interessante il tipo T** (ad esempio quando voglio **passare per indirizzo** una **variabile puntatore!**) e, occasionalmente può accadere di usare tipi come T***!

Tra tutti i tipi pointer, merita menzione **void***: void è la tupla vuota, un tipo inutile (ad esempio è il tipo di ritorno delle funzioni che non tornano alcun risultato).

void* è piuttosto interpretato come il **puntatore a un dato di cui non conosco il tipo**. È un puntatore universale **compatibile per assegnazione con tutti**, cioè con tutti i tipi puntatore T* per ogni tipo T.

Ovviamente, triangolando con void*, posso scrivere programmi multipati.

Non è possibile applicare *x se x è di tipo void*. Perché?



Lezione 20

That's all Folks!

Grazie per l'attenzione...

...Domande?