Informatica Generale Programmazione C

Ivano Salvo

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **3**, 12 marzo 2020

Condividiamo le esperienze…

Lezione 3a:

Un paio di esercizi

Moltiplicazione Egiziana

Problema 1: *Scrivere una funzione che calcoli la moltiplicazione sfruttando le seguenti uguaglianze:*

$$\begin{aligned} m \times 2n &= (m+m) \times n \quad (n > 1) \\ m \times (2n+1) &= m \times 2n + m \\ m \times 0 &= 0 \end{aligned}$$

Idea: somme e moltiplicazioni ×2 sono operazioni più facili di fare prodotti (**provare per credere**).

Si tratta di una definizione induttiva alternativa del prodotto, che definisce il prodotto per **induzione** sul secondo parametro: specifica quanto vale il prodotto in 3 casi: 0 (**caso base**) e numero pari o dispari, sempre in termini di prodotti con un secondo fattore **minore**.

Soluzione: Ragionamento

Soluzione: Per comodità, indicheremo con m_0 ed n_0 i valori iniziali da moltiplicare, e con m, n e p i valori delle variabili `m`, `n` e `p`, e con m', n' e p' i loro valori dopo l'esecuzione del corpo del ciclo.

L'idea di fare solo le operazioni "semplici" (cioè somme e predecessori) accumulando i risultati in una variabile `p`, in modo che venga mantenuta la proprietà invariante $m_0 n_0 = mn + p$. Questa relazione è banalmente soddisfatta all'ingreso del ciclo inizializzando `p` a 0. Quando n è pari, raddoppiando m e dividendo n per 2, la relazione si mantiene perchè $m'n' + p' = 2m\frac{n}{2} + p = mn + p$. Quando n è dispari, toglieremo 1 a n e aggiungeremo m a p, mantenendo la relazione in quanto in questo caso $m'n' + p' = m(n-1) + (p+m) = mn - m + p + m = mn + p$. Usciremo dal ciclo quando raggiungiamo il caso base delle equazioni induttive scritte sopra, cioè quando $n = 0$. In tale situazion $mn + p = m_0 n_0$ implica proprio che $p = m_0 n_0$, che è quanto volevamo ottenere. Il caso base viene raggiunto, perchè la guardia $(n \neq 0)$ implica che $n' < n$. Fatte queste osservazioni, il programma si scrive da solo, come in Fig. 12.

Soluzione: funzione C

```
int multiplyingLikeAnEgiptian(int m, int n){
/* PREC: m,n>= 0 */
    int p=0;
    while (n!=0){
    /* INV: m0 * n0 = p+m*n  */
       if (resto(n,2) == 0)
             { m = somma(m,m);
               n = div(n,2);
               }
         else
             { p = somma(p,m);
               n = pred(n);
               }
      }
    return p;
}
```

Massimo Comun Divisore

Problema 2: *Scrivere una funzione che calcola il massimo comun divisore tra due numeri.*

Soluzione 1: Per risolvere questo problema seguiremo diverse strategie. Cominciamo con un algoritmo "ingenuo". Siano m ed n i numeri di cui vogliamo calcolare $\mathsf{mcd}(m, n)$. Prendiamo il minore dei due (diciamo sia $p = \mathsf{min}(m, n)$) e proviamo tutti i numeri $p, p-1, p-2 \ldots 2, 1$. Il primo valore che divide sia m che n è il massimo comun divisore di m ed n. Ecco il programma corrispondente:

```
int mcdIngenuo(int m, int n){
/* PREC: m,n> 0 */
    int p;
    if (minore(m,n)==1) p=m; else p=n;
    while (p!=0){
    /* INV: forall. p'>p p' non divide n o p' non divide m*/
       if (resto(n,p) == 0)
             if (resto(m,p)=0) return p;
       p=pred(p);
      }
    return p;
}
```

Lezione 3b:

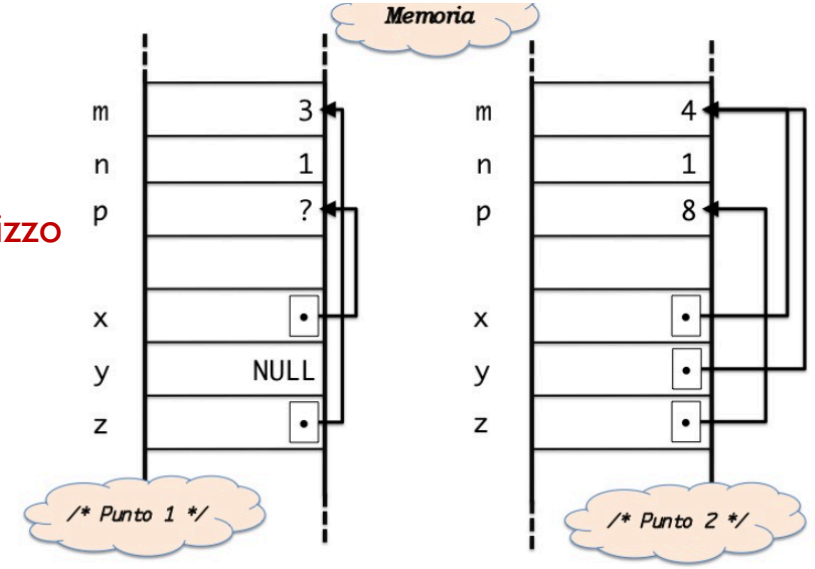
Parli del Diavolo…
…e spuntano i puntatori

Parli del diavolo, spuntano puntatori

Una variabile puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile.

```
void provaPuntatori(){
   int m=3;
   int n=1;
   int p;
   int* x= &p;
   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 */
   }
```

& è l'operatore di dereferenziazione data una variabile restituisce il suo indirizzo



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

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 scambia1(int b, int a){
     int h;
h=a;
a=b;
b=h;
}
```

Questa funzione `scambia1` correttamente I valori delle variabili a e b LOCALI alla funzione scambia.

Ma non ha nessun effetto sullo stato del chiamante.

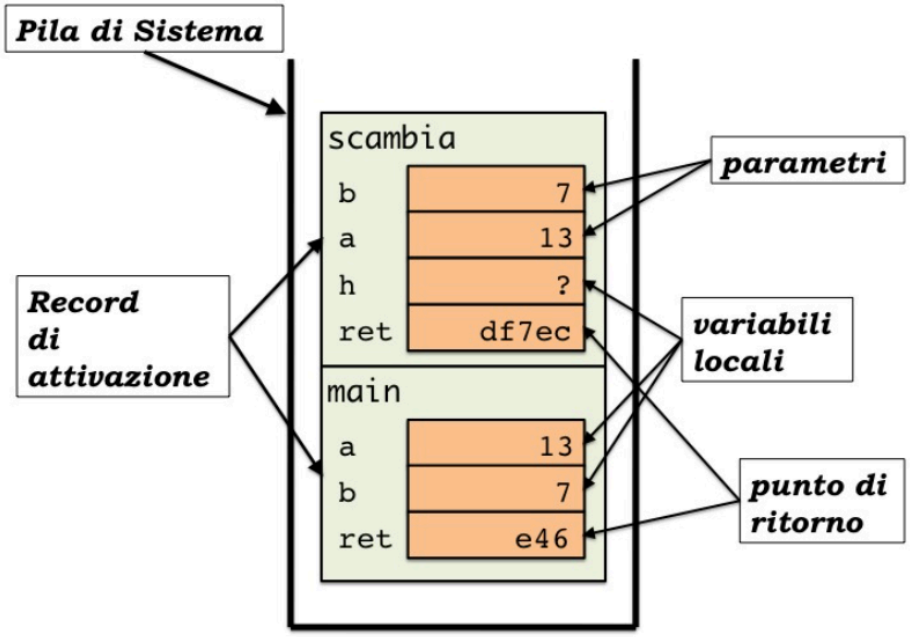


Figura 4: Stack di attivazione delle funzioni

Esempio di Esecuzione

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

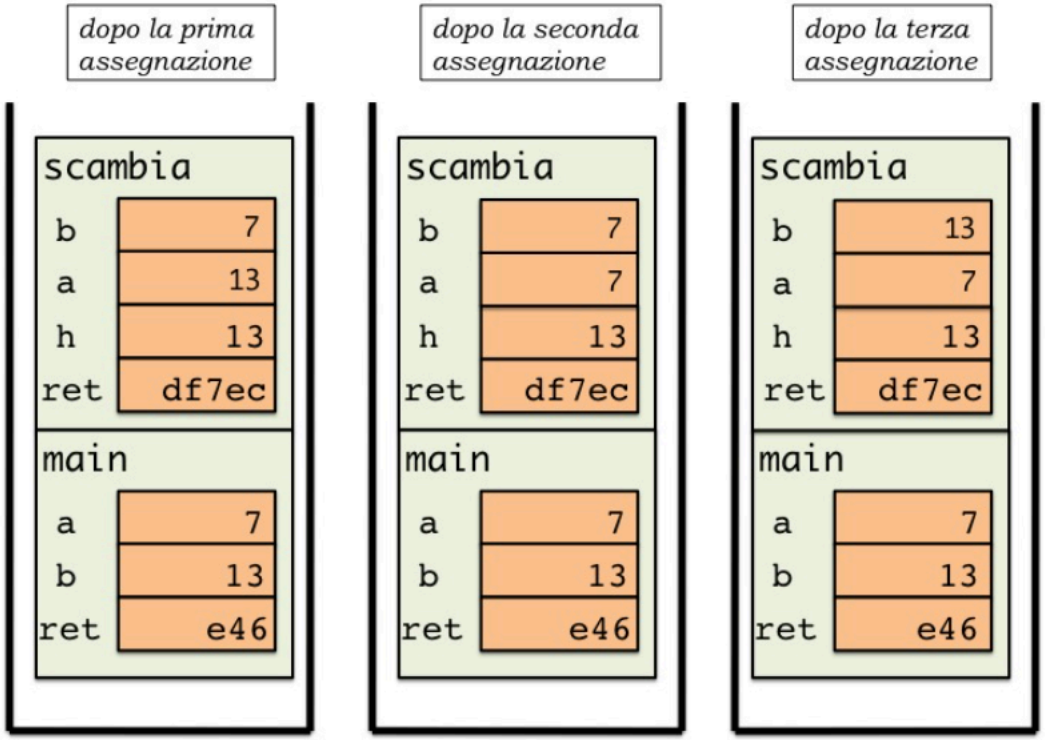


Figura 5: Esecuzione della funzione `scambia1` in Fig. 3

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

Nella funzione `scambia2` 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.

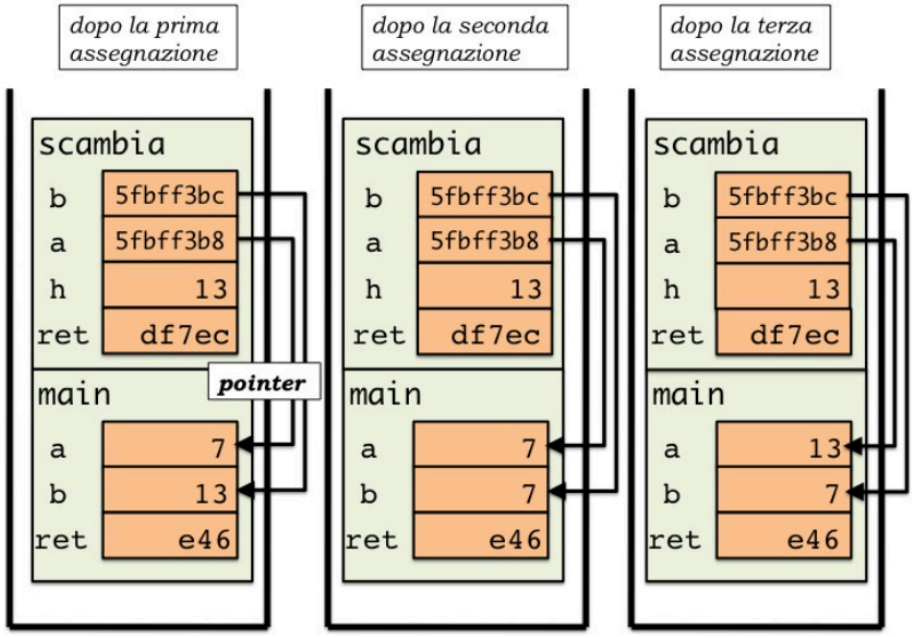


Figura 7: Esecuzione della funzione `scambia2` in Fig. 6

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 scambia3(int* x, int *y){
    *x=*x-*y;
    *y=*y+*x;
    *x=*y-*x;
}
```

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

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

Cosa c'è che non va?

Alias & side-effects

```
void scambia3(int* x, int *y){
    *x=*x-*y;
    *y=*y+*x;
    *x=*y-*x;
}
```

Una chiamata **`scambia(&a, &a)`** dopo l'esecuzione della prima istruzione, siccome in tal caso x e y 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.

Oppure la chiamata potrebbe essere **`scambia(&a, &b)`** in cui le variabili **`a`** e **`b`** sono alias per qualche altro motivo.

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

Lezione 3c:

Altri esercizi con puntatori

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

MCD della Maestra - 2

Risolviamo un problema più semplice: scriviamo una funzione che stampa tutti i fattori primi di un numero (perché funziona? Perché non stampa **fattori non primi**?):

```
void stampaDivisoriPrimi(int m){
/* PREC: n>0 */
    int p=2;

    while (p*p<=m){
    /* INV: */
       if (resto(m,p)) p++;
          else { printf(" %d",p);
                 m = div(m,p);
                 }
       }
    if (m>1) printf(" %d",m);
    printf("\n");
    }
```

Notare che non serve calcolare la radice quadrata!

Morale della favola: non serve memorizzare i fattori!

MCD della Maestra - 3

Possiamo **accumulare i prodotti dei fattori comuni** generando i fattori primi in parallelo dei due numeri, come nella funzione di stampa.

```
int mcdDellaMaestra(int m, int n){
/* PREC: m,n>0 */
    int p=2;
    int mcd=1;

    while (minore(mult(p,p),max(m,n))=1 && minore(p,min(m,n))){
    /* INV: mcd(m0,n0)=mcd*mcd(m,n) */
       if (resto(m,p) && resto(n,p)) p++;
          /* entrambi non sono divisibili:
             passo al prossimo potenziale divisore */
       if (!resto(m,p)) && !resto(n,p)) mcd=mult(mcd,p);
          /* entrambi sono divisibili:
             accumulo il risultato con il nuovo
             fattore comune scoperto */
       if (!resto(m,p)) m=div(m,p);
       if (!resto(n,p)) n=div(n,p);
       }
       if (m==n) mcd=mult(mcd,n);
       return mcd;
    }
```

MCD della Maestra - 4

Siccome le funzioni **`div`** e **`resto`** fanno le stesse computazioni, sembra uno spreco chiamarle sulle stesse coppie di valori.
Per risolvere questo problema, dobbiamo risolvere il problema di **ritornare 2 valori al chiamante**.

```
int divRef(int m, int n, int* q, int* r){
    /* PREC: m>=0,n>0.
     * POST: carica *q e *r . q*n+r=m & r<n
     *       torna 1 se m è divisibile per n, 0 altrimenti
     */
    *q=0;
    *r=m;
    while (minUguale(n,*r)) {
        /* INV: m = q*n + r */
        *r=diff(*r,n);
        (*q)++;
    }
    if (*r==0) return 1;
    else return 0;
}
```

return !*r

Questo è una funzione che segue un tipico schema in C: si torna un valore, ma il vero risultato passa sui parametri.
Anche **`printf`** torna un valore! (il numero di caratteri stampati)

MCD della Maestra - 5

Usando divRef, possiamo scrivere una funzione che calcola l'MCD seguendo l'algoritmo della maestra in modo più elegante ed efficiente al tempo stesso.

```
int mcdMaestra(int m, int n){
/* PREC:m,n>0 */
    int mcd=1;
    int p=2;
    int q1,q2,r1,r2;
        /* queste variabili VANNO ALLOCATE come INTERI */

    while (minUguale(p,min(m,n))){
        /* INV: mcd*MCD(m,n)=MCD(m0,n0) */
        if (divRef(m,p,&q1,&r1)) m=q1;
        if (divRef(n,p,&q2,&r2)) n=q2;
        if (r1 && r2) p++;
        if (!r1 && !r2) mcd=multEgypt(mcd,p);
    }
    return mcd;
}
```

Può essere che siate abituati ad un'altra notazione per il passaggio dei parametri per indirizzo tipica del C++.

Lezione 3

That's all Folks…

…Domande?