Informatica per Statistica
Riassunto della lezione del 24/10/2012

Igor Melatti

Le funzioni ricorsive (in C)

- Si consideri la funzione C alla Figura 1

```
1 unsigned long fattoriale(unsigned n)
2 {
3   unsigned long res = 1;
4   unsigned i;
5
6   for (i = 1; i <= n; i++)
7     res *= i; /* esattamente come scrivere res = res*i */
8   return res;
9 }
```

Figure 1: Fattoriale iterativo

  - è una definizione di una funzione, `fattoriale`
  - non è un vero e proprio programma, perché manca il `main`
    - un `main` potrebbe leggere da tastiera un numero positivo e poi stamparne il fattoriale, provare per **esercizio**
  - nel seguito, verranno spesso scritte (definizioni di) funzioni senza `main`, più o meno come si fa negli algoritmi
  - questo per spostare l'attenzione sugli specifici problemi da affrontare

- In generale, il fattoriale di un numero viene definito (*iterativamente*) così:

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

  - la funzione `fattoriale` di Figura 1 effettua esattamente questo calcolo

  - in realtà, è possibile fare subito una facile ottimizzazione che permette di fare un ciclo in meno: quale? (per **esercizio**)

- Notare che il risultato è un `unsigned long`, ovvero ci si aspetta che il risultato potrebbe essere troppo grande per essere memorizzato su un `unsigned` "normale"

  - su questo argomento si ritornerà nella prossima lezione

- Il fattoriale può anche essere definito (*ricorsivamente*) così:

$$n! = \begin{cases} 1 & \text{se } n = 1 \\ n(n-1)! & \text{altrimenti} \end{cases}$$

- Nel secondo caso, per definire una funzione si usa la funzione stessa

  - da notare che c'è sempre un *caso base*
  - permette di calcolare effettivamente la funzione
  - è il caso in cui non c'è ricorsione, ovvero il caso in cui non viene usata la funzione stessa
  - nell'esempio del fattoriale, si ha per $n = 1$: senza chiamare un'altra volta il fattoriale, la definizione dice che il risultato è semplicemente 1
  - quindi, se si vuole conoscere il valore di 5!, si procede come segue:

$$\begin{aligned} 5! &= 5 \cdot 4! \\ &= 5 \cdot 4 \cdot 3! \\ &= 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 120 \end{aligned} \tag{1}$$

  - l'idea è che, se si vuole conoscere il valore della funzione in un punto, si applica la definizione fino a raggiungere il caso base

- Negli algoritmi e nei linguaggi di programmazione si può fare altrettanto

- La versione ricorsiva del fattoriale è in Figura 2

- Ma per capire come funziona il tutto, occorre andare un po' più (ma non troppo) nel dettaglio di come si effettuano le chiamate a funzione

- Ovvero, occorre conoscere un po' di dettagli implementativi che il C nasconde sul meccanismo delle chiamate a funzione

```
1 unsigned long fattoriale(unsigned n)
2 {
3   if (n == 1)
4     return 1;
5   else /* questa riga e' in realta' inutile: perche'? */
6     return n*fattoriale(n - 1);
7 }
```

Figure 2: Fattoriale ricorsivo

- per prima cosa, occorre ricordare la semantica della *chiamata a funzione*
- l'effetto di una chiamata a funzione è quello di sospendere l'esecuzione della funzione in cui ci si trova attualmente (funzione chiamante), eseguire la funzione chiamata (dove gli input hanno il valore specificato dagli argomenti della chiamata) e poi ritornare il controllo alla funzione in cui ci si trovava in precedenza, a partire dal punto immediatamente seguente quello della chiamata
  - in alcuni casi, occorre ritornare il controllo nel punto esatto della chiamata, anziché al seguente
  - ad esempio, se la chiamata di funzione è contenuta in un'espressione a destra di un assegnamento
- detto così sembra semplice ma nasconde delle difficoltà
- si considerino le 3 definizioni di funzioni in Figura 3
- quello che effettivamente fanno queste 3 funzioni è irrilevante, vengono usate proprio per illustrare degli aspetti sintattico-semantici generali del C
- si supponga che il `main` chiami `f1`
- al suo interno, `f1` dichiara 3 variabili intere (dette per questo motivo *locali*)
- dopo un assegnamento ad `a`, chiama `f2`
- che a sua volta ha una variabile locale `d`, e chiama `f3`
- infine, `f3` ha pure una variabile locale `a`, ci assegna un valore, e poi ritorna una certa espressione dei suoi argomenti e di `a`
- al che, il controllo deve tornare ad `f2`
- *non in punto qualsiasi: esattamente da subito dopo che la chiamata ad* `f3` *era stata effettuata*
- quindi, nell'esempio attuale, occorre che il valore appena ritornato da `f3` sia assegnato alla variabile `d` e poi si prosegua ritornando `d + 1`

```
1  int f1(int arg1)
2  {
3    int a, b, c;
4
5    a = arg1;
6    b = f2(a);
7    c = f3(a + b, arg1);
8    return c;
9  }
10
11 int f2(int arg1)
12 {
13   int d;
14
15   d = f3(2*arg1, arg1);
16   return d + 1;
17 }
18
19 int f3(int arg1, int arg2)
20 {
21   int a;
22
23   a = 4;
24   return arg1*arg2 + a;
25 }
26
27 int main()
28 {
29   return f1(3);
30 }
```

Figure 3: Esempio per le chiamate a funzione

– con ciò si torna ad `f1`, nuovamente non in un punto qualsiasi, ma esattamente da dove era stata chiamata `f2`

– ovverosia, occorre assegnare a `b` il valore appena tornato da `f2`

– dopodiché, occorre di nuovo chiamare `f3`, passandogli tra le altre cose anche il valore di `a`

– ma da dove lo si prende il valore di `a`? Se non si organizzano le cose per bene, si corre il rischio di non ritrovare più tale valore

– si potrebbe pensare che tutte le variabili locali dichiarate in tutte le funzioni che si definiscono in un programma C abbiano la loro locazione di memoria definita in partenza, quindi separatamente le une dalle altre

– analogo trattamento andrebbe quindi riservato anche agli *argomenti* di tali funzioni

– quindi, in questo esempio, il compilatore dovrebbe organizzare 9 aree di memoria RAM predefinite, ciascuna grande a sufficienza per un `int`, che il compilatore stesso potrebbe chiamare `arg1_f1`, `a_f1`, `b_f1`, `c_f1`, `arg1_f2`, `d_f2`, `arg1_f3`, `arg2_f3`, `a_f3`, e poi risolvere opportunamente i vari riferimenti

– questa soluzione, apparentemente lapalissiana, è estremamente inefficiente

– di più: per le funzioni ricorsive è anche praticamente impossibile (vedere più avanti)

– per quanto riguarda l'inefficienza, essa discende dal fatto che spesso i programmi usano, di volta in volta, poche funzioni contemporaneamente

– è pertanto molto più efficiente, anziché mantenere in memoria tutte le variabili locali e tutti gli argomenti di tutte le funzioni *contemporaneamente*, mantenere solo quelle che interessano davvero

  * anche perché nei programmi C "veri" le funzioni possono essere nell'ordine delle decine di migliaia, con possibilmente migliaia di variabili (ad esempio sotto forma di vettori): a volerle mantenere tutte in RAM contemporaneamente (dall'inizio alla fine del programma), potrebbe non bastare neanche tutta la RAM
  * quando invece la RAM potrebbe bastare se venissero considerate solo le funzioni effettivamente chiamate, e solo nel momento in cui vengono chiamate

– come si fa a decidere quali sono le funzioni (e i relativi argomenti e variabili) che interessano davvero?

– non è difficile: basta considerare solo la funzione attualmente in esecuzione, più la *catena di chiamate* (iniziante dal `main`) che ha portato fino ad essa

- ad esempio, nel caso ora discusso, quando viene eseguita `f2` la catena di chiamate è `main-f1-f2` (considerando anche quella attuale)
  - quando viene eseguita `f3`, la catena di chiamate può essere o `main-f1-f2-f3` (quando `f3` viene chiamata da `f2`) o `main-f1-f3` (quando `f3` viene chiamata da `f1`)
  - si consideri il seguente esempio: un programma con 2 definizioni di funzioni, chiamate `funzione1` e `funzione2`, più il `main` che chiama prima l'una e poi l'altra
  - si supponga altresì che tanto `funzione1` quanto `funzione2` dichiarino al loro interno un array di 1 miliardo di `double`, e che non chiamino altre funzioni
  - con la soluzione lapalissiana di mantenere tutto in RAM, sono necessari circa 16 GB di RAM
    - un `double` richiede 8 bytes, quindi 1 miliardo di `double` richiede 8 GB, e 2 miliardi richiedono 16 GB
  - quindi su un computer da 8 GB di RAM non si riuscirebbe a mantenere l'intero programma in RAM
  - ma con la soluzione che mantiene solo le chiamate effettive, 8 GB di RAM sono sufficienti, perché le due funzioni non sono mai attive contemporaneamente in questo esempio
- quindi quello quindi il compilatore fa (o meglio: fa sì che succeda) è di memorizzare opportunamente (e separatamente) le variabili locali e gli argomenti per ciascuna funzione nella catena di chiamate
- per capire come il tutto viene organizzato, occorre tornare all'analogia tra la macchina di Von Neumann e la scrivania per l'esecuzione di un algoritmo
- sulla scrivania c'è un foglio per memorizzare i risultati di calcoli temporanei (che altro non sono che le variabili locali) e un foglio con le istruzioni (*programma*)
- l'analogia per le chiamate a funzioni parte dal presupposto che ogni chiamata a funzione usi un foglio a parte
- su ognuno di questi fogli vengono scritti i valori delle variabili locali (aggiornati man mano che cambiano) e degli argomenti, più un'ulteriore informazione: in che punto della funzione ci si trova
  - cioè qual è la prossima istruzione da eseguire
  - tecnicamente, si chiama *program counter* (PC)
- per ogni chiamata di funzione, un nuovo foglio viene aggiunto *sopra* quelli esistenti, e per prima cosa, sul nuovo foglio, si scrivono i valori degli argomenti che sono stati passati
- tuttavia, *prima* di fare quest'aggiunta, il PC della funzione (che indica il punto in cui sta avvenendo la chiamata a funzione) viene scritto sull'attuale foglio)

    - così, quando si ritornerà a questo foglio, si ritroverà la giusta informazione
  - per ogni `return` (o raggiungimento del } finale della funzione), il foglio in cima viene tolto, e il valore ritornato (se c'è) viene copiato nell'opportuna variabile presente nel foglio sottostante
    - per esempio, quando `f2` finisce a ritorna, sempre ad esempio, 3, allora nel foglio sottostante si scrive 3 come valore della variabile locale `b`
  - quindi si lavora sempre e solo sulla cima (inizialmente, c'è solo il `main`)
  - questa è una *struttura dati* chiamata *pila* o *stack*
  - la zona di stack necessaria per i valori delle variabili locali + argomenti + PC di ogni funzione è chiamata *stack frame*
  - l'evoluzione dello *stack delle chiamate* per il programma in Figura 3 è quella illustrata nelle Figure 4–12
    - il PC, o program counter, in queste figure è relativo ad ogni funzione, quindi quando vale ad esempio 1 si intende che sta per eseguire la prima istruzione della funzione corrispondente, se vale 2 la seconda etc.
  - ovviamente, una funzione con molte variabili locali e argomenti avrà uno stack frame più grande di una con poche variabili locali e argomenti
    - qui "molte" e "poche" sono termini che vanno ovviamente pesati: se una funzione dichiara `int A[1000]` e un'altra dichiara `int a, b, c, d, e, f`, è ovviamente la prima ad avere lo stack frame più grande

- Messa così, diventa facile capire come funzionano le chiamate ricorsive: esattamente con lo stesso meccanismo
  - tornando al discorso di sopra, con le funzioni ricorsive è praticamente impossibile non usare uno stack: ogni chiamata ricorsiva ha le stesse identiche variabili locali e argomenti!
    - le variabili locali potrebbero anche mancare (come in `fattoriale`), ma gli argomenti difficilmente mancano nelle funzioni ricorsive
  - ad esempio, se per `fattoriale` si usasse una sola zona di memoria per il suo parametro `n`, come prevederebbe la soluzione "lapalissiana" discussa sopra, semplicemente il calcolo non andrebbe a buon fine
    - supponiamo infatti che ci sia una chiamata `fattoriale(2)` (per esempio da dentro un `main`)

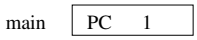

Figure 4: Situazione dello stack delle chiamate dopo la chiamata del `main` (da parte del sistema operativo)

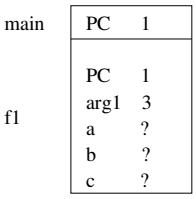

Figure 5: Situazione dello stack delle chiamate dopo la chiamata a `f1` (riga 29 di Figura 3)

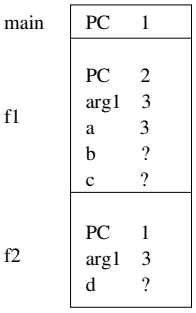

Figure 6: Situazione dello stack delle chiamate dopo la chiamata a `f2` (riga 6 di Figura 3)

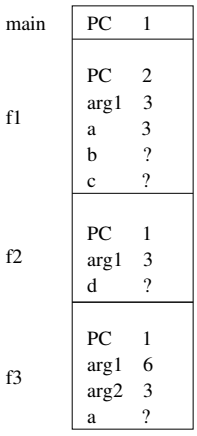

Figure 7: Situazione dello stack delle chiamate dopo la chiamata a `f3` (riga 15 di Figura 3)

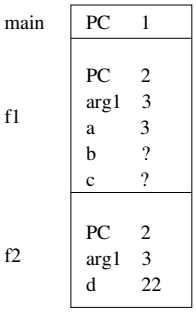

Figure 8: Situazione dello stack delle chiamate dopo l'esecuzione della `return` di `f3` (riga 24 di Figura 3)

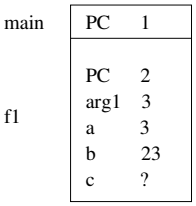

Figure 9: Situazione dello stack delle chiamate dopo l'esecuzione della `return` di `f2` (riga 16 di Figura 3)

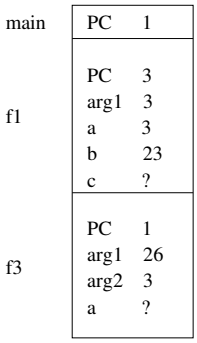

Figure 10: Situazione dello stack delle chiamate dopo la chiamata a `f3` (riga 7 di Figura 3)

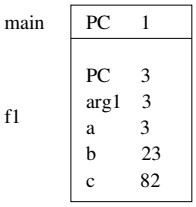

Figure 11: Situazione dello stack delle chiamate dopo l'esecuzione della `return` di `f3` (riga 24 di Figura 3)

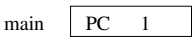

Figure 12: Situazione dello stack delle chiamate dopo l'esecuzione della `return` di `f1` (riga 8 di Figura 3)

    - ∗ dato che 2 è diverso da 1, verrebbe effettuata la chiamata ricorsiva a `fattoriale`, che ha l'effetto di sovrascrivere quell'unica zona di memoria contenente `n`, sovrascrivendo quindi il valore precedente (ovvero 2) con il valore attuale (1)
    - ∗ questa nuova chiamata terminerebbe subito tornando 1
    - ∗ e qui viene il problema: tornando alla chiamata precedente (supponendo che lo si riesca a fare) si avrebbe che ora il valore di `n` è rimasto 1 (non l'ha cambiato più nessuno...) e quindi il valore ritornato sarebbe `1*1` (il primo 1 è il valore di `n`, il secondo 1 è il valore appena ritornato da `fattoriale`), mentre la risposta corretta sarebbe stata 2 perché $2! = 2$
  - in generale, come si può vedere, il problema è che si sovrascriverebbero i valori delle variabili locali e degli argomenti, cosa che non deve avvenire: concettualmente, variabili locali e argomenti devono avere una "vita" ristretta all'esecuzione della funzione all'interno della quale si trovano
  - è quello che succede grazie allo stack delle chiamate: la chiamata `fattoriale(2)` sarebbe eseguita come illustrato in Figure 13–16

- Le funzioni ricorsive sono solitamente più inefficienti di quelle iterative
  - soprattutto per quanto riguarda la memoria usata
  - in una funzione iterativa, ci sono solo le variabili effettivamente dichiarate
  - in una funzione ricorsiva, ce ne possono essere più copie, a seconda di quante chiamate ricorsive si fanno

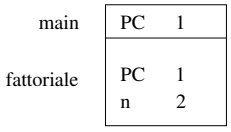

Figure 13: Situazione dello stack delle chiamate dopo la chiamata a `fattoriale`(2)

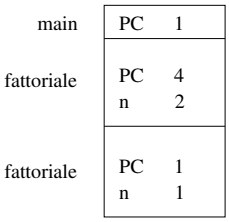

Figure 14: Situazione dello stack delle chiamate dopo la chiamata a `fattoriale`(1)

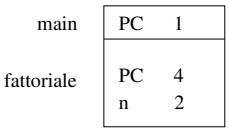

Figure 15: Situazione dello stack delle chiamate dopo l'esecuzione della `return` di `fattoriale`(1);

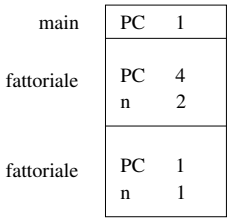

Figure 16: Situazione dello stack delle chiamate dopo della `return` di `fattoriale`(2)

- Tuttavia, per molti problemi scrivere una funzione ricorsiva porta a soluzioni più chiare e concise