Esercizi su visibilità delle variabili e Record di Attivazione

Stefano Cherubin*

02/12/2016

[**Informatica A**] Esercitazione #11

corso per Ing. Gestionale a.a. 2016/17

*<nome>.<cognome>@polimi.it

Indice

1 Understand fixes

Simulare l'esecuzione di ciascuna delle chiamate a sottoprogrammi, limitandosi alla descrizione dei valori stampati tramite `printf`.

Listato 1: Understanding fixes

```
1  #include <stdio.h>
2
3  int varA = 12, varB = 25;
4
5  int fix(int* par) {
6    *par = *par - 7;
7    return(23);
8  }
9
10 void veryImportantFix(int* varX, int varY) {
11   int varB;
12   *varX = varY;
13   varB = *varX + varY;
14 }
15
16 int thisReallyFix(int varA) {
17   int varZ = 17;
18   varA = varA + varZ;
19   return(varA);
20 }
21
22 int main() {
23   /* initially... */
24   int varC = 63;
25   printf("0) varA: %d, varB: %d,  varC: %d\n",
         varA, varB, varC);
26
27   /* step 1 */
28   varC = fix(&varA);
29   printf("1) varA: %d, varB: %d,  varC: %d\n",
         varA, varB, varC);
30
31   /* step 2 */
32   veryImportantFix(&varA, varC);
33   printf("2) varA: %d, varB: %d,  varC: %d\n",
         varA, varB, varC);
34
35   /* step 3 */
36   varB = thisReallyFix(varC);
```

```
37    printf("3) varA: %d, varB: %d,  varC: %d\n",
          varA, varB, varC);
38
39    /* step 4 */
40    varB = fix(&varB);
41    printf("4) varA: %d, varB: %d,  varC: %d\n",
          varA, varB, varC);
42    return 0;
43  }
```

1.1 Soluzione

	varA	varB	varC
initially	12	25	63
step 1	5	25	23
step 2	23	25	23
step 3	23	40	23
step 4	23	23	23

2 The Shadow

Sia dato il seguente programma.

Listato 2: The Shadow

```
1  #include <stdio.h>
2  #include <string.h>
3  #define MAXL 30
4  #define TURN_TO_APOCALYPSE 2
5
6  typedef struct {
7    char playerClass;
8    char name[MAXL];
9    int hp;
10   int mp;
11   int canCastSpells;
12 } Player;
13
14 int attack(Player *attacker);
15
16 int _attack(Player *attacker);
17
18 Player TheShadow;
19
20 int main() {
21   Player warrior, drunkCitizen, mage, bard,
        orc;
22
23   /* Initialize party */
24   warrior.playerClass = 'W';
25   strcpy(warrior.name, "Gerardo");
26   warrior.hp = 60;
27   warrior.mp = 15;
28   warrior.canCastSpells = 0;
29
30   drunkCitizen.playerClass = 'C';
31   strcpy(drunkCitizen.name, "Giovanni");
32   drunkCitizen.hp = 20;
33   drunkCitizen.mp = 5; /* magic burp! */
34   drunkCitizen.canCastSpells = 0;
35
36   mage.playerClass = 'M';
37   strcpy(mage.name, "Alex");
38   mage.hp = 45;
39   mage.mp = 55;
40   mage.canCastSpells = 1;
```

```
41
42    bard.playerClass = 'B';
43    strcpy(bard.name, "Stefano");
44    bard.hp = 40;
45    bard.mp = 25;
46    bard.canCastSpells = 1;
47
48    orc.playerClass = 'O';
49    strcpy(orc.name, "Nicholas");
50    orc.hp = 75;
51    orc.mp = 0;
52    orc.canCastSpells = 0;
53
54    /* Initialize enemy */
55    TheShadow.playerClass = 'S';
56    strcpy(TheShadow.name, "Venom");
57    TheShadow.hp = 100;
58    TheShadow.mp = 100;
59    TheShadow.canCastSpells = 1;
60
61    /* let the game begin! */
62    int turn = TURN_TO_APOCALYPSE;
63    while (turn > 0) {
64      attack(&warrior);
65      attack(&drunkCitizen);
66      attack(&mage);
67      attack(&bard);
68      attack(&orc);
69      turn--;
70    }
71    if (TheShadow.hp > 0) {
72      printf("The world is lost!\n");
73    } else {
74      printf("The world is safe!\n");
75    }
76    return 0;
77  }
78
79  int attack(Player *attacker) {
80    int damage;
81    if (attacker->canCastSpells && attacker->mp
          >= 20) {
82      damage = attacker->mp / 2;
83      attacker->mp -= 20;
84      TheShadow.hp -= damage;
85    } else {
```

```
86      damage = _attack(attacker);
87    }
88    return damage;
89  }
90
91  int _attack(Player *attacker) {
92    int damage;
93    Player theShadow;
94    if (attacker->mp > 10) {
95      Player citizen;
96      damage = attacker->hp / 2;
97      attacker->mp -= 5;
98      TheShadow.hp -= damage;
99      citizen.hp -= damage;
100   } else if(attacker->canCastSpells){
101     damage = attacker->hp / 2;
102     attacker->mp -= 5;
103     theShadow.hp -= damage;
104   } else {
105     Player TheShadow;
106     damage = attacker->hp / 2;
107     attacker->hp -= 10;
108     TheShadow.hp -= damage;
109   }
110   return damage;
111 }
```

1. Si dica quali giocatori, se ne esistono, sono effettivamente in grado di recare danno almeno una volta al malefico `"Venom"`.
2. Si dica se con queste impostazioni il mondo si salverà.
3. Si trovi un valore (se esiste) per la `#define TURN_TO_APOCALYPSE` in grado di cambiare le sorti del mondo.

2.1 Soluzione

1. Infligono danno a `"Venom"` almeno una volta tutti i giocatori che iniziano con `mp` > 10, cioè:

   - warrior «Gerardo»
   - mage «Alex»
   - bard «Stefano»

2. Il mondo è destinato a non salvarsi.

3. Aumentando il valore assegnato alla `#define TURN_TO_APOCALYPSE` il mondo si salverà. Un valore di 3 permetterebbe al party di distruggere `"Venom"` prima che sia troppo tardi. È quindi sufficiente trovare il modo di distrarre `TheShadow` per un turno.

3 Il record di attivazione attraverso un esempio

3.1 Calcolo della media

Dato il seguente codice sorgente espresso in linguaggio C relativo ad una funzione, si disegni il record di attivazione che la sua esecuzione genera in memoria. Si valuti la memoria del programma all'istante prima dell'esecuzione dell'istruzione **`return`**.

```
float media(int v[], int N) {
  int i;
  float sum;
  sum = 0;
  for (i = 0; i < N, i++)
    sum += v[i];
  sum /= N;
  return sum;
}
```

3.2 Assunzioni sull'architettura

Il testo non contiene dati relativi all'architettura del calcolatore che eseguirà questo codice. Si fanno quindi assunzioni relative alle dimenisoni in memoria di ciascun tipo di dato e alla struttura della memoria stessa.

3.2.1 Struttura della memoria

Si assume di lavorare con una memoria indirizzabile alla parola (unità minima allocabile), con parole di 32 bit.

Si assume inoltre di avere indirizzi (sia per i dati sia per le istruzioni) di lunghezza 32 bit.

L'area di memoria della funzione si assume divisa tra stack e heap. Lo stack cresce da indirizzi più alti verso indirizzi più bassi, lo heap cresce in verso opposto.

3.2.2 Dimensione dei tipi

int 4 Byte

char 1 Byte

float 4 Byte

double 8 Byte

(pointers) 4 Byte

3.2.3 Vettori in memoria

Si ricorda che un vettore altro non è che un puntatore ad un'area di memoria. Passare come parametro un vettore equivale a passare alla funzione l'indirizzo dell'area di memoria dove il vettore è locato e non avviene una copia di tutto il vettore completo.

Ogni volta che avviene il passaggio di un indirizzo come parametro, un puntatore viene chiamato in causa. Nel caso di un vettore, si avrà un puntatore al tipo del vettore. In questo esempio, un vettore di interi passato come parametro sarà equivalente a un puntatore a intero `int*`.

3.3 La struttura della memoria

L'immagine Stack: calcolo della media mostra la rappresentazione dell'area di memoria del programma mettendo in evidenza (in verde) il record di attivazione della funzione `media` all'istante di tempo richiesto.

La rappresentazione dell'area di memoria del programma è coerente con una chiamata alla funzione `media` da parte del `main`. A indirizzi più alti in valore, contigui al record di attivazione di `media`, si trova infatti il record di attivazione della funzione `main`.

A sinistra dello stack sono indicati i valori contenuti nei registri Stack Pointer e Frame Pointer. Questi indicano rispettivamente la prima parola di memoria non allocata sullo stack e l'indirizzo base del record di attivazione corrente. Le variabili e i parametri, a livello macchina, saranno espressi come spiazzamenti positivi e negativi relativi al FP.

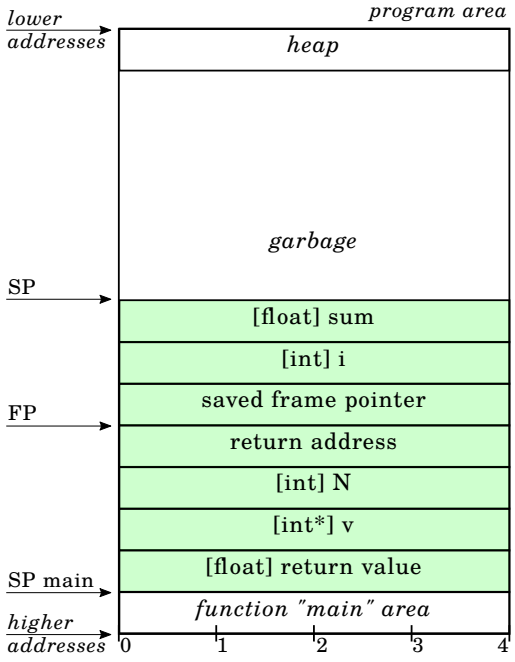

Figura 1: Stack: calcolo della media

Si noti che la memoria è rappresentata a blocchi di 1 parola (= 32 bit = 4 Byte) e che ciascun elemento sullo stack occupa esattamente una parola di memoria, coerentemente con le assunzioni discusse in precedenza.

Al puro scopo di migliorare la leggibilità dello schema, per ogni parametro e variabile viene indicato all'interno della porzione di memoria che occupa, il tipo e il nome assegnatogli in linguaggio C.

4 Messaggi offuscati

Sia dato il seguente programma.

Listato 3: Meggaggi offuscati

```
1  void f(char *c, char **p) {
2    while (*c != '\0') {
3      *(p++) = c;
4      c += 2;
5    }
6    return;
7  }
8
9  int main() {
10   char c[15] = "ESPRITCOWLIOND";
11   char *p[7], **q;
12   int i = 0;
13   f(c, p);
14   while (*(p[i%7]) != '\0')
15     printf("%c", **( p+((i++)%7)));
16   return 0;
17 }
```

4.1 Ispezionare la memoria

Si disegni lo stack dei record di attivazione di questo programma all'esecuzione dell'istruzione alla riga 6.

Si rappresentino tutti i record di attivazione delle funzioni attive, incluso quello relativo alla funzione main. Si mostrino **tutte** le variabili dichiarate.

Si adottino le seguenti convenzioni:

vettori blocchi contigui

puntatori frecce

valori indefiniti ?

4.2 Ispezionare l'output

Si dica qual è l'output del programma

4.3 Approccio alla soluzione

4.3.1 Deoffuscare il codice

Il testo dell'esercizio presenta del codice sintatticamente corretto ma alquanto poco leggibile. Si procede quindi ad effettuare delle riscritture che pur mante-

nendo invariati i valori delle variabili, contribuiscano a rendere più comprensibile il programma.

Scorrimento di un array con post-incremento dei puntatori Si ricorda che il post-incremento si applica dopo la valutazione dell'espressione. Separare l'incremento dalla valutazione dell'espressione migliora la leggibilita dell'espressione ed evita una errata interpretazione dell'incremento.

```
*(p++) = c;
/* it's equivalent to */
*p = c;
p++;
```

Accesso ad un array con notazione implicita Si ricorda che `*(p + i)` si può riscrivere in notazione implicita (e per alcuni più intuitiva) con le parentesi quadre come `p[i]`.

```
**(p+((i++)%7))
/* it's equivalent to */
*p[i%7];
i++;
```

Risultato delle trasformazioni Applicando queste trasformazioni al codice dato, si ottiene il seguente codice. Si osservi che l'istruzione da considerare ora non si trova più alla riga 6 ma viene spostata alla riga 7.

```
1  void f(char *c, char **p) {
2    while (*c != '\0') {
3      *p = c;
4      p++;
5      c += 2;
6    }
7    return;
8  }
9
10 int main() {
11   char c[15] = "ESPRITCOWLIOND";
12   char *p[7], **q;
13   int i = 0;
14   f(c, p);
15   while (*(p[i%7]) != '\0') {
16     printf("%c", *p[i%7]);
17     i++;
18   }
19   return 0;
20 }
```

4.4 Soluzione

4.4.1 Schema di memoria

La granularità ritenuta sufficiente per rispondere a questo quesito è quella mostrata in Figura 2 che distingue i diversi record di attivazione e colloca i blocchi di memoria relativi alle variabili nel corretto record di attivazione a cui appartiene.

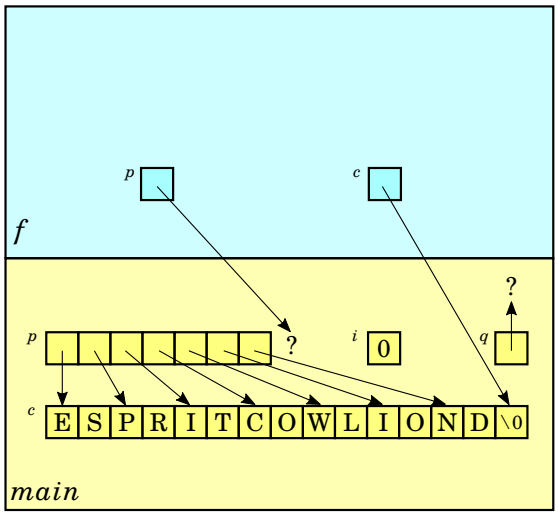

Figura 2: Soluzione al quesito 4.1

4.4.2 Output del programma

Il programma stampa la stringa `EPICWINEPICWIN` ...

4.4.3 Approfondimento

La stampa si ripete molte volte ma non prosegue all'infinito nè fino alla saturazione della memoria.

La memoria non satura perché non vi sono istruzioni che allocano memoria tra una stampa e la successiva.

L'esecuzione non prosegue all'infinito perché la variabile **`int i`** non può rappresentare infiniti valori. Al superamento del massimo valore rappresentabile in una variabile intera, occorrerà *overflow* e `i` contterrà valori negativi.

Matematicamente parlando, questo non dovrebbe fermare l'esecuzione del programma tuttavia `i%7` invece di restituire sempre valori compresi tra 0 e 6, quando `i` è negativa restituisce valori tra -6 e 0. Al primo valore negativo usato per accedere ad un vettore, avremo un tentativo di accesso alla memoria `*(p+i)` non definita per valori negativi di `i`. Questo con elevata probabilità causerà un errore di violazione di memoria (accesso a memoria al di fuori dei limiti consentiti).

È possibile aggirare questo problema cambiando la dichiarazione della variabile in **`unsigned int i`**; un overflow su variabili unsigned riporta la variabile semplicemente a 0. Si avrà quindi una discontinuità dell'output (perché il massimo valore rappresentabile in una variabile **`unsigned int`** non è multiplo di 7) ma l'output non si fermerà.

5 Brescia-Napoli: ricorsione tropicale

Sia dato il seguente programma

Listato 4: Ricorsione tropicale

```
1  int f(char *nord, char *sud, int x) {
2    if(!x)
3      printf(" & ");
4    else {
5      printf("%c", sud[x % 2]);
6      x = f(nord, sud, x - 1 );
7      printf("%c", sud[x % 2]);
8    }
9    return x + 1;
10 }
11
12 int main() {
13   int x = 5;
14   char nord[] = "BS", sud[] = "NA";
15   printf("%c", nord[(++x) % 2]);
16   x = f(nord, sud, x - 1);
17   printf("%c", nord[(++x) % 2]);
18   return 0;
19 }
```

1. Si disegni lo stack dei record di attivazione nell'istante precedente al momento in cui viene eseguita per la prima volta l'istruzione alla riga 9. Si rappresentino tutte le variabili adottando le convenzioni precedentemente specificate al punto 4.1.
2. Si mostri la linea stampata dal programma sullo standard output.

5.1 Approccio alla soluzione

5.1.1 Funzioni ricorsive

La funzione `f` chiama sè stessa al suo interno, è quindi un esempio di funzione ricorsiva. In memoria, viene creato un record di attivazione per ogni chiamata ad una funzione. Anche nel caso di multiple chiamate alla stessa funzione, si ha un nuovo record di attivazione per ogni chiamata.

I record di attivazione ricorsivamente allocati si accumulano sullo stack e vengono deallocati, partendo dall'ultimo inserito, al raggiungimento della rispettiva istruzione **`return`**.

5.1.2 Valori e indirizzi

Si noti che la funzione f accetta come parametri due indirizzi e un valore. Quando viene passato come parametro un indirizzo, l'area di memoria a cui viene fat-

to riferimento (destinazione della freccia) viene stabilita dal chiamante; quando viene passato come parametro un valore, viene eseguita una copia del valore che risiederà nel record di attivazione della funzione chiamata.

5.2 Soluzione

5.2.1 Schema di memoria

Si osservi come lo schema di memoria in Figura 3 allochi diversi record di memoria per diverse chiamate ricorsive alla funzione `f`.

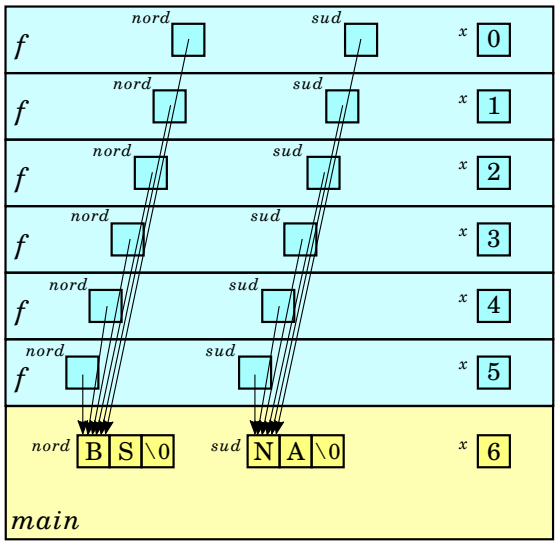

Figura 3: Ricorsione tropicale: schema di memoria

5.2.2 Output del programma

Il programma stampa su standard output la stringa `BANANA & ANANAS`.

Licenza e crediti

Crediti

Quest'opera contiene elementi tratti da materiale di Gerardo Pelosi.

Licenza beerware[1]

Quest'opera è stata redatta da Stefano Cherubin. Mantenendo questa nota, puoi fare quello che vuoi con quest'opera. Se ci dovessimo incontrare e tu ritenessi che quest'opera lo valga, in cambio puoi offrirmi una birra.

[1] http://people.freebsd.org/~phk/