Esame info 3 11 Febbraio 2009 Soluzioni

1 Passaggio parametri

Data questa procedura che (in psesudo codice) incrementa x di n:

```
procedure inc(x,n) { while (n>0) { x++; n—;}}
```

Considera i diversi passaggi di parametri sia per x sia per n (valore, rif, puntatore).

Per quali combinazioni non hai problemi, mentre per quali potresti avere problemi?

Delle seguenti chiamate per quali passaggi avresti quali problemi (con a e b variabili intere)?

inc(a,b)

inc(a,a)

inc(a,3)

inc(3,4)

Implementa in C la procedura in modo che almeno un parametro sia passato per puntatore. Come dovresti riscrivere le chiamate sopra (se possibile) con passaggio di puntatori?

---

	x	n	Commenti – va bene?
1	val	-	NO: indipendentemente da come passo n, x viene modificato sulo sullo stack, alla fine della procedura perdo le modifiche
2	ref/punt	ref/punt	Sì: x viene modificata in modo corretto, però viene modificato anche n (portato a zero) e questo protrebbe causare dei problemi. Inoltre in questo modo non posso passare n con costante o esressione ma solo con una variabile. Il passaggio per puntatori poi richiede una scrittura più attenta sia del codice della funzione sia delle chiamate.
3	ref/punt	val	Questo è il modo preferito: x viene modificato in modo corretto , mentre n no.

Chiamate

ERRORE ?	1	2	3
inc(a,b)	Ok, ma non modifica a	Ok	ok
inc(a,a)	Ok, ma non modifica a	Non c'è errore in compilazione però Problemi: ho un ciclo infinito perchè creo un alias su a e sia x che n puntano alla stessa variabile	OK
inc(a,3)	Ok, ma non modifica a	NO: non posso passare un numero per rif.	OK
inc(3,4)	Ok, ma non modifica nulla	NO: non posso passare un numero per rif.	No, non posso passare 3 per x per ref.

Codice:

```
void inc(int * x, int n) { while (n>0) { (*x)++; n--;}}
```

inc(&a,b) inc(&a,a) inc(&a,3) inc(3,4) NON è possibile

---

2 Record di attivazione - tail recursion

Scrivi una funzione ricorsiva in C che, dato un array di interi, restituisca la loro somma. Fai due versioni: una senza tail recursion e l'altra con tail recursion. Quale ottimizzazione può fare il compilatore con la versione tail recursion? Disegna il record di attivazione in entrambi i casi per una chiamata in un main che faccia la somma di un array con tre elementi. Usa anche variabili globali.

Senza tail recursion

```
int sum_ntr(int* a, int n){
   if (n== 0) return 0;
   return *a + sum_ntr(a+1,n-1);
}
```

```
con tail recursion:
```

```
int sum_tr(int* a, int sum, int n){
   if (n==0) return sum;
        return sum_tr(a + 1, sum + *a, n-1);
}

int x[] = {1,2,3};
void main() {
  printf("somma ntr %d",sum_ntr(x,3));
  printf("somma tr %d",sum_tr(x,0,3));
}
```

```
Alternativamente (anche se meno elegante) si poteva usare una variabile globale
sum per implementare la somma parziale, del tipo:

int sum;

int sum_tr(int* a, int n){
   if (n==0) return sum;
   sum += *a;
   return sum_tr(a + 1, n-1);
}


record di attivazione … TODO
```

---

3 Dynamic Binding in java

Date le seguenti dichiarazioni (ignora il corpo dei metodi che supponi corretto):

```
class A{ public boolean equals(A a){ … }}
```

```
class B extends A{
public boolean equals(B b){ … }
public boolean equals(A a){ … }}
```

per ognuna delle seguenti istruzioni dire se è corretta e spiega quale metodo viene eseguito (se viene eseguito un metodo)

`Object o = new Object();`	OK
`A a = new A();`	OK
`A ab = new B();`	OK, B è sottoclasse di A
`B b = new B();`	OK
`B ba = new A();`	NO, A non è sottoclasse di B
`o.equals(o);`	EB: equals(Object) di Object il quale non è ridefinito, LB: idem
`o.equals(a);`	EB: equals(Object) di Object il quale non è ridefinito, A promosso ad Object LB: idem
`ab.equals(a);`	EB: due segn. candidate equals(Object) e equals(A) di A. Il secondo non richiede promozioni e viene selezionato. LB: ab si riferisce ad un B, cerco un metodo con la segnatura selezionata in B !!. viene eseguito equals(A) di B
`a.equals(b);`	EB: due segn. candidate in A equals(Object) e equals(A). Il secondo richiede meno promozioni e viene selezionato. LB: a si riferisce ad un A, viene eseguito equals(A) di A
`b.equals(ab);`	EB: due segn. candidate in B equals(Object) e equals(A) di B. Il secondo non richiede promozioni e viene selezionato. LB: b si riferisce ad un B, viene eseguito equals(A) di B

4 C++ 1

Considera questo programma C++:

```
#include <iostream>
using namespace std;
class Animale {
   public:
   string name;
```

```
static Animale * giraffa(){
   Animale g;
   g.name="Giraffa";
   return &g;
 }
```

```
};                                          {
                                              Animale* p = Animale::giraffa();
                                              cout << p->name << endl;
                                              return 0;
                                            }
int main( int argc, char *argv[] )
```

E' corretto? Se no, come lo correggeresti? Quale è l'output?

Il programma non è corretto perchè il metodo giraffa restituisce un dangling pointer poichè g viene creato sullo stack e restituisco il puntatore a g. Forse l'output potrebbe anche essere corretto, perchè non rioccupo lo stack, ma non c'è garanzia. Lo correggo così (in C++ per creare oggetti nello heap devo usare new, non malloc !!):

```
#include <iostream>
using namespace std;


class Animale {
   public:
   string name;
   // sbagliato
   static Animale * giraffa(){
      Animale g;
      g.name="Giraffa";
      return &g;
    }

   static Animale * giraffa2(){
      Animale* g = new Animale;
      g->name="Giraffa";
      return g;
    }

};

int main( int argc, char * argv[] )
{
    Animale* p = Animale::giraffa2();
    cout << p->name << endl;
    return 0;
}
```

5 C++ 2

Fai un esempio di friend function e un esempio di friend class con le relative dichiarazioni e le chiamate.

6 Java

Date le seguenti dichiarazioni

```
class A{}
```

```
class B extends A{}
```

dire se le seguenti istruzioni sono corrette e spiegare perchè (spiega anche ventuali problemi):

```
A[] aa = new B[10];
List<A> la = new ArrayList<B>();
A[] bb = new A[10];
List<B> lb = new ArrayList<A>();
```

```
A[] aa = new B[10];
```

OK B è sottotipo di A, quindi B[] è sottotipo di A[] (covrianza)

```
List<A> la = new ArrayList<B>();
```

NO B è sottotipo di A, ma C<B> e C<A> non hanno alcuna relazione per ogni classe C

```
A[] bb = new A[10];
```

OK

```
List<B> la = new ArrayList<A>();
```

Questo è proprio sbagliato.

---

7. Cyclone

Considera questa funzione scritta in Cyclone:

```
void foo(char *s, int offset) {
     unsigned int len = strlen(s);
     for (unsigned int i = 0; offset+i < len; i++)
        s[offset+i] = 'a';
  }
```

E' corretta? E' efficiente? Come potresti migliorarla?

I char* in cyclone sono sempre zero terminated, quindi è permessa l'aritmetica e il programma è corretto. All'esecuzione dell'istruzione s[offset+i] verrà controllato che non ci sia uno zero tra s[0] e s[offset+i]. Questo viene fatto ogni volta che incremento i. In questo modo non ho mai buffer overflow. Però e' inefficiente per lo stesso motivo (e anche inutile visto che se ho conttrollato già da 0 a offset+i è inutile che ricontrolli tutto, mi basterebbe ricontrollare solo offset+i+1

Posso modificarla così:

```
  void foo(char *s, int offset) {
     unsigned int len = strlen(s);
     s = s + offset;
     for (unsigned int i = 0; offset+i < len; i++)
        *s = 'a';
        s++;
  }
```

In questo modo faccio il controllo solo che `*s` è non zero quando incremento e quando assegno *s = 'a'.

Oppure converto il puntatore (o l'intero metodo) a fat:

```
void foo(char *s, int offset) {
  char *@fat @nozeroterm fat_s = (char *@fat @nozeroterm)s;
  unsigned int len;
  fat_s += offset;
  len = numelts(fat_s);
  for (unsigned int i = 0; i < len; i++)
    fat_s[i] = 'a';
}
```

In questo modo il puntatore viene convertito a fat con dimensione fino al primo zero di s.

8. Verifica formale

Scrivi le pre e post condizioni del programma al punto 1 e prova la correttezza.

La precondizione è che n >= 0 (x invece può essere qualunque)

PRE:{n >=0}

La poscondizione è che alla fine x venga incrementata di n rispetto al valor iniziale. Come fare visto che x viene modificato (e anche n)? Introduco due valori per ospitare i valori iniziali di x e n: old_x (o x') e old_n (o n')

POST: {x = old_x + old_n}

Per la prova riporto qui il codice key hoare:

```
\functions {    int old_x;   int old_n; }
\programVariables {  int x, n; }
\hoare {
  { old_x = x  & old_n = n & n >= 0}
   \[{
     while( n >0 ){
          x = x + 1;
          n = n - 1;
       }
    }\]
  { x = old_x + old_n}
}
```

Come lo provo?
Essenzialmente mi serve solo l'invariante. Provo con qualche dato (ad esempio old_x = 3, old_n = 2)
Cosa osservi?

	I ingresso			
x	3	4	5	
n	2	1	0	

Cosa è invariante: la somma tra x e n è uguale proprio al valore finale voluto di x (old_x + old_n), cioè: x + n = old_x + old_n.
Nota che all'uscita del ciclo n <=0, mentre a me servirebbe n = 0. Posso aggiungere n >=0 all'invariante?
Prova con I:{x + n = old_x + old_n & n >=0}
Devo provare:
1) I valido inizialmente

```
|-    old_x = x & old_n = n & n >= 0
        ->    x + n = old_x + old_n & n >= 0
```

OK banale per l'aritemtica

2) I preserva l'invariante cioè:

```
{ x + n = old_x + old_n & n >= 0 & n >  0 }[]
  x=x+1;
  n=n-1;
{ x + n = old_x + old_n & n >= 0 }
```

Applico due volte la regola per l'assegnamento e poi l'exit:
|-
x + n = old_x + old_n & n >= 0 & n > 0
->
[](x+1 + n -1= old_x + old_n & n-1 >= 0)
OK vera

3. Uso l'invariante per provare la post condizione:

{ x + n = old_x + old_n & n >= 0 & !n > 0 }
[]
{ x = old_x + old_n }

da n >=0 e !n >0 segue n = 0, sostuituisco e ho provato.