ASSIGNMENT 1

Abhishek Gupta - agupta12@binghamton.edu
Dipankar Ghosh - dghosh1@binghamton.edu

Solution 1:

A) No, the program is not producing any dangling references.

If any pointer is pointing the memory address of any variable and after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as *dangling pointer*
In the problem below, variables are very much within the scope and no memory location has even been freed up. z=3 is not used up anywhere since the *y=4 overshadows the existing value of z. All three x,,y,z are linked to each other thus their values always keep on changing together simultaneously.
The allocation is done as follows:

Iteration 1:
X after x=&y, x points to address of y

Y after y=&z, y points to address of z

Z =3, value is defined

Iteration 2:
X after *y = 4, x points to address of y

Y after *y = 4, the value of z becomes 4

Z =4, the values becomes 4 because of above assignment by y, and **x = z will also mean the same.

Iteration 3:
printf("%d\n", *y); will print 4
z = 2;
printf("%d\n", *y); will print 2
**x = 1;
printf("%d\n", z); will print 1

B)
Yes garbage is produced, the heap allocation done at line 6 becomes garbage at line 8 when y pointed to reference of z. Line 6 allocates memory; line 8 leaks that memory by changing y to point to a local variable, **losing the only reference to the allocated memory. So, you do have a leak.**

Solution 2:

Call by value:
Input values are : i=1 and a[i]=1
procedure swap(i, a[i])

i ← i + a[i] // i holds sum of both. i=2
a[i]← i - a[i] // a[i] now holds value of i. a[i]=2-1=1;
i ← i - a[i] // i now holds value of a[i]. i=2-1=1;
end procedure
printf("%d %d %d %d\n", i, a[0], a[1], a[2]); ***values printed are : 1 2 1 0***

Similar for procedure swap(a[i],a[i]);
Input values are : a[i]=1 and a[i]=1
procedure swap(a[i], a[i])
a[i] ←a[i] + a[i] // a[i] holds sum of both. i=2
a[i]← a[i] - a[i] // a[i] now holds value of i. a[i]=2-1=1;
a[i] ←a[i] - a[i] // a[i] now holds value of a[i]. a[i]=2-1=1;
end procedure
printf("%d %d %d\n", a[0], a[1], a[2]); ***values printed are : 2 1 0***

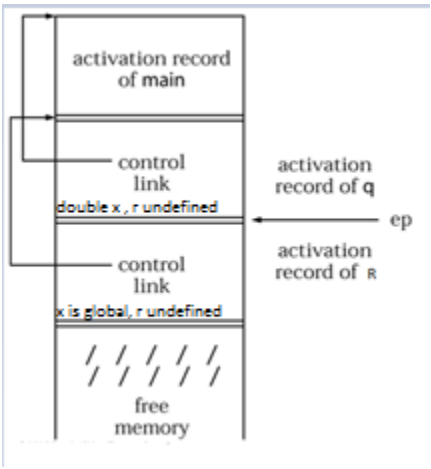
Call by reference
Input values are : i=1 and a[i]=1
procedure swap(&i, &a[i])
&i ← passed address of i from main function to swap function
&a[i]← passed address of a[i] from main function to swap function

void swap(int *x, int *y)
*x= Thus address of i will be collected in "**x**" pointer variable
*y= Thus address of i will be collected in "**y**" pointer variable
*x ← *x + *y // *x holds sum of both. *x=2
*y = *x-*y // *y now holds value of *y. *y=1;
*x = *x-*y; // *x now holds value of *x. x=2-1=1;
end procedure

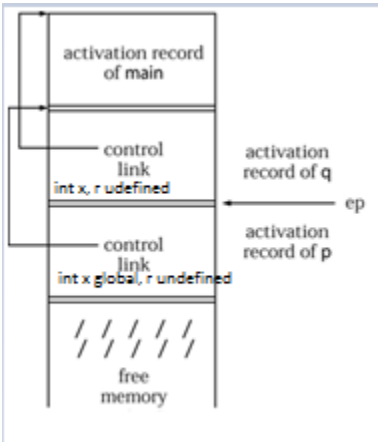
printf("%d %d %d %d\n", i, a[0], a[1], a[2]); ***values printed are : 1 2 1 0***
Similar for procedure swap(a[i],a[i]);
&a[i]← passed address of a[i] from main function to swap function
void swap(int *x, int *y)
*x= Thus address of i will be collected in "**x**" pointer variable
*y= Thus address of i will be collected in "**y**" pointer variable
*x ← *x + *y // *x holds sum of both. *x=2
*y = *x-*y // *y now holds value of *y. *y=0
*x = *x-*y; // *x now holds value of *x. x=0
printf("%d %d %d\n", a[0], a[1], a[2]); ***values printed are : 2 0 0***

Solution 3:

a) Here R produces garbage value and x prints 0 as it is defined globally.



b)



Here the r is still undeclared and can't be accessed.

c) When P() is called from main function.
R: defined locally within the function p
X: defined globally

When q() is called from main function-->r()--p()
R: defined locally within the function p
X: defined globally but takes value of function r

When q() is called from main function--p()
R: defined locally within the function p
X: defined globally but takes value of function

So after execution the result produced is 2 0 2 1 2 1.
All the above cases produces varied results if we change the printf statement for printing x and write %f and the result produced will be
2 2.00 2 2.00 2 2.00

Solution 4:

a) Stack is a very convenient way to represent recursion, if the language does not allow recursion the only 1 instance of any procedure may be active at any given moment and variables for that procedure can be statically allocated. Stack is not only used to perform recursion but also bunch of nested function calls. Recursion is a special case where all the nested function calls are to the same function, in a cyclic chain of function calls.
This is because the nature of nested function calling and returning is the same as that of a stack. The one that is called at top level is returned the last. The later a function is called in the nested hierarchy, the earliest it returns. So, the start/ return of a function call turns out to push/ pop operations of a stack and thus a stack can be used to replica this functionality. For eg. Calculating factorial of a given number.

```
public int factorial(int n)
{
if(n == 0 || n == 1)
      return 1;
      return n*factorial(n-1);
}
```

Now in the above program: we have base condition i.e. either n is 0 or 1 factorial will be 1. Once the program start let's say for n=4, this will call itself for n =3 then n =2 & finally n =1. In order to calculate the final solution we need all the number in same order (more visible in Fibonacci problem) i.e. last in first out(LIFO data structure hence stacks is the best choice.

b) Yes mostly it should, If the language does allow procedure recursion, but a recursive call can only happen at the end of a procedure then it depends whether to have the stacks or not. Logically we need to have multiple nested argument and that is a stack by normal definition. For instance let's take 2 functions p() & q(). If we call p() from q() then the function p() must have a return address stored which is available within Stacks thus stacks might be used. If the procedure doesn't use recursive calling methodology then stacks might not be used.

Solution 5:

(a) It creates only one continuous area with x, y, and color fields inside

(b) No, it won't compile because there is no space in pa to take values of pb
Compilation Error
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Type mismatch: cannot convert from Point to ColorPoint
at test.main (test.java:5)

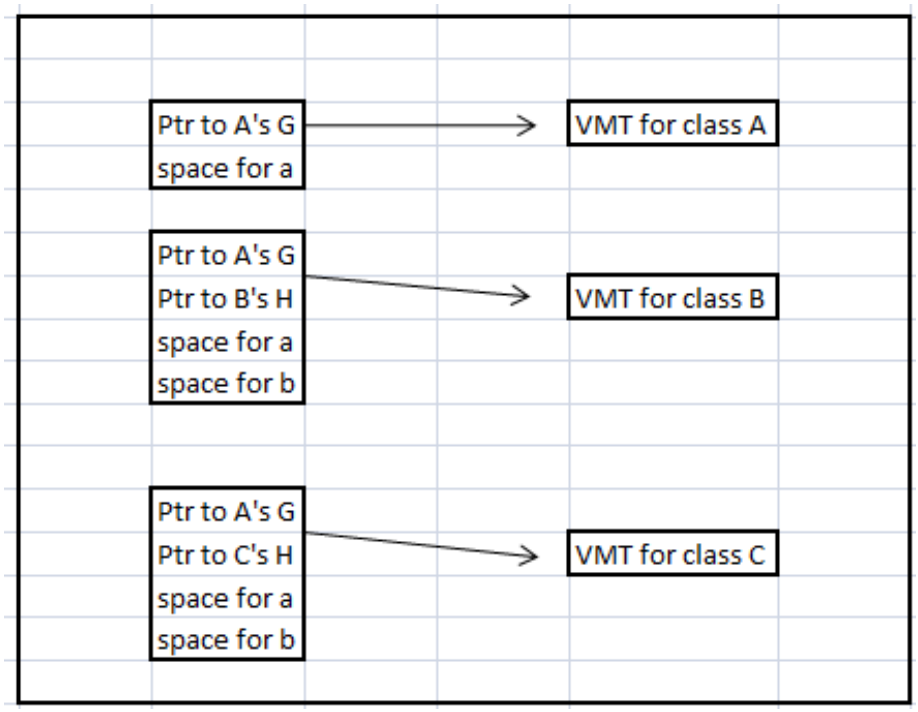
(c) Yes it will compile and value will be 1
Compilation and execution both take place value of pb.x=1

(d) Will compile, but will give classcast exception error at runtime Exception in thread "main" java.lang.ClassCastException: test.Point cannot be cast to test.ColorPoint

Exception in thread "main" java.lang.ClassCastException: Point cannot be cast to ColorPoint at test.main(test.java:7)

(e) Will compile, and will also execute at run time, values if any is there to be printed
Compilation occurs but values are not printed since no command line has been intialised for printing of values.

Solution 6:



Solution 7:

The argument is false, by casting we can use it and the value can be printed. Eg

```
public interface AInter {

	public void cat();
	public void dog();
}

public class Bclass implements AInter {
	@Override
	public void cat() {
		// TODO Auto-generated method stub
		System.out.println("In Cat");
	}
	@Override
	public void dog() {
		// TODO Auto-generated method stub
		System.out.println("In Dog");
	}
	public void m()
	{
		System.out.println("In M which you can't access until you use casting");
	}
}
public class Calling {
	public static void main(String[] args) {
		AInter a= new Bclass();
a.cat();
	a.dog();

//	a.m(); // error according to statement

((Bclass) a).m(); // casting done

	if(a instanceof AInter)
	{
		System.out.println("Instance of A");
	}
	else
		System.out.println("Instance of B");
	}
}
```

Reason: casting

According to the argument, in an interface holding as a reference to parent class subclass we cannot access any method function defined in it. In java by casting we can access hidden function in the object. When object 'a' was defined of 'A' java automatically updated it but lately we down casted so it was limited so is inaccessible.