Zachary Seymour
CS 571
Assignment 2
October 14, 2013

1 Given the C program....

1a Does the program produce dangling references? Why?

There is no call to `free` and all the code is running in the same scope, so no dangling references are produced.

1b Does the program produce garbage? Why?

The program does not produce garbage.

2 Give the output of the following program (in pseudo-code) using the two parameter passing methods (call by value/reference).

- Call-by-value:

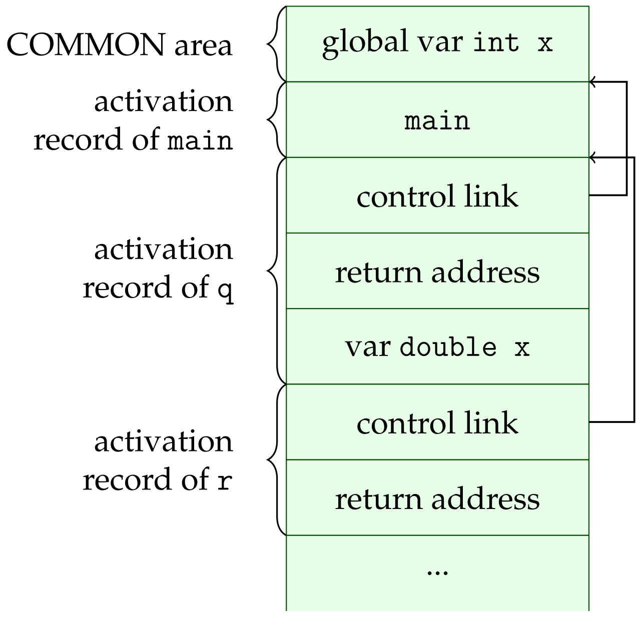
```
2 1 0
2 1 0
```

- Call-by-reference:

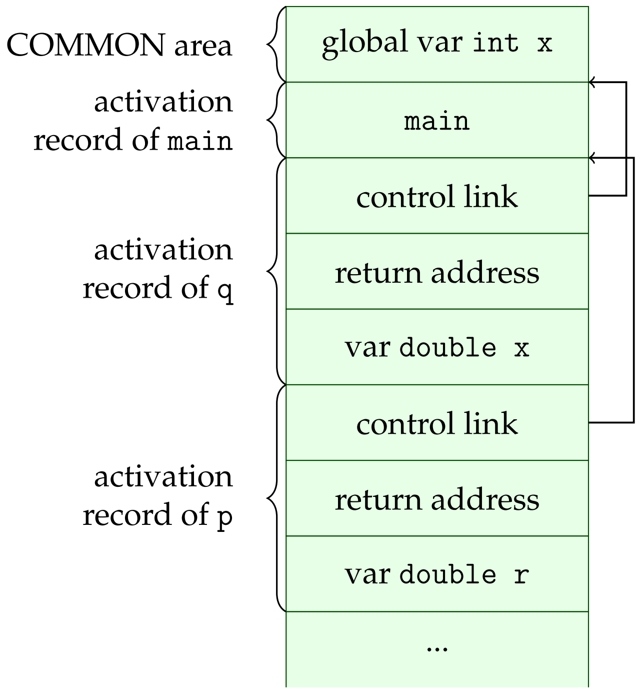
```
2 1 0
2 1 0
```

3 For the following C program....

3a Draw the stack activation records after the call to `r` on line 13.



3b Draw the stack activation records after the call to p on line 14.



3c Describe where variables r and x are located on lines 4 and 5 and how they are accessed during the execution of p.

The variable r is found with in the activation record from of p, so it is accessed by its offset within the frame. The variable x is global, so it is found and accessed in the COMMON area before the activation frame of main.

4 To model the run-time semantics of procedures, it is known that a stack is generally needed.

4a If the language does not allow procedure recursion, do we have to have stacks? Why?

If no function can be called inside another, there would not be a need for a stack. We would either have one big activation record, containing the whole program or, perhaps, a linked-list of records, functions called one after the other, sharing and mutating common data.

4b If the language does allow procedure recursion, but a recursive call can only happen at the end of a procedure, do we have to have stacks? Why?

This is in fact the benefit of tail-call recursion so common in functional languages. Since the recursive call is the last step in the function, nothing reaming in the activation frame is really needed. We can just replace the old activation record with the new one.

5 Read the following Java code:

5a After Line 0 is executed, does the heap create one continuous area with x, y, and color fields inside, or does the heap create two distinct areas, one with x, y and the other with color?

One continuous area is created with the entry for color immediately following x and y.

5b Suppose Line 1 is included in the program but Lines 2, 3, 4 are commented out, should the program compile? If not, why? If yes, what is the value of field x of pa?

The program will not compile as a Point cannot assigned to a ColorPoint without a cast declaration.

5c Suppose Line 2 is included in the program but Lines 1, 3, 4 are commented out, should the program compile? If not, why? If yes, what is the value of field x of pb?

Now, the program will compile, and we have `pb.x` equal to 1.

5d Suppose Line 3 is included in the program but Lines 1, 2, 4 are commented out, should the program compile? If not, why? If yes, what would be run-time behavior of the program?

The program does now compile, since the cast is made explicit; however, a `ClassCastException` is thrown at runtime.

5e Suppose Line 4 is included in the program but Lines 1, 2, 3 are commented out, should the program compile? If not, why? If yes, what would be run-time behavior of the program?

This compiles and behaves in the same way as part (c) because the cast is redundant.

6 Given the following C++ declarations, draw the VMT of each class and the layout of memory for a dynamically allocated object of *each class*.

- VMT for class `A`

space for `int a`
pointer to `g`

- VMT for class `B`

space for `int a`
pointer to `g`
pointer to `h`
space for `int b`

- VMT for class `C`

space for `int a`
pointer to `g`
space for `int b`

- Memory layout for object of type `A`

value of `a`
pointer to VMT of `A`

- Memory layout for object of type `B`

value of `a`
pointer to VMT of `A`
pointer to VMT of `B`
value of `b`

- Memory layout for object of type `C`

value of `a`
pointer to VMT of `A`
value of `b`

7 Many people think the Java "interface" mechanism is an encapsulation mechanism. Their arguments are as follows. Suppose we have an interface A, being implemented by class B. If programmers consistently use type A every time an instance of B is referred to, then any method, say m, defined in B but not specified in A is hidden and cannot be accessed by any code holding a reference of type A. There however is a fatal flaw in this argument. Explain how a piece of code referring to an instance of B (with type A) can in fact access m.

Provided the object is a true instance of B just having type A, the A object can be cast at runtime to a B object, and the method can validly be called.

8 Given the following Featherweight Java class definition:

```
class A extends Object
{
    A() { super(); }
}

class B extends Object
{
    B() { super(); }
}

class Pair extends Object
{
    Object fst;
    Object snd;
    Pair(Object fst, Object snd) { super(); this.fst=fst; this.snd=snd; }
    Pair setfst(Object newfst) { return new Pair(newfst, this.snd);}
}
```

8a Is it valid to write the following class:

```
class MyPair extends Pair
{
     MyPair(Object fst, Object snd) { super(fst, snd); }
     Pair setfst(Object newfst) { return new MyPair(newfst, this.snd); }
}
```

If so, write a type derivation for method declaration `Pair setfst(Object newfst) { return new MyPair(newfst, this.snd); }` If not, explain why.

We must make all of the checks for method typing to insure this method is OK in class C. First check is $\bar{x} : \bar{C}, \mathtt{this} : C \vdash e_0 : E_0$. So, given that all of the arguments are of the correct type and `this` is of type `MyPair`, then the return value has type `E0`, which should be a subtype of the return type, `Pair`. Since `MyPair` is a subtype of `Pair` and `newfst` is of type `Object`, this condition checks out. Since the `setfst` of `Pair` and `MyPair` take the same type of parameters and return the same type, the last criterion is met as well, so this method is valid and of type `Pair`.

8b How does the following expression reduce? Please write it down step by step until no more reduction is possible:

```
new Pair(
         (new Pair(new A(), new B()).setfst(new B())).snd,
         (new Pair(new A(), new B()).setfst(new B())).fst
        )
```

```
new Pair((new Pair(new A(), new B()).setfst(new B())).snd, (new Pair(new A(),
new B()).setfst(new B())).fst)
→ new Pair(new Pair(new A(), new B()).snd, (new Pair(new A(), new B()).setfst(new B()))
→ new Pair(new Pair(new A(), new B()).snd, new Pair(new A(), new B()).fst)
→ new Pair(new B(),  new Pair(new A(), new B()).fst)
→ new Pair(new B(), new A())
```