Assignment 1

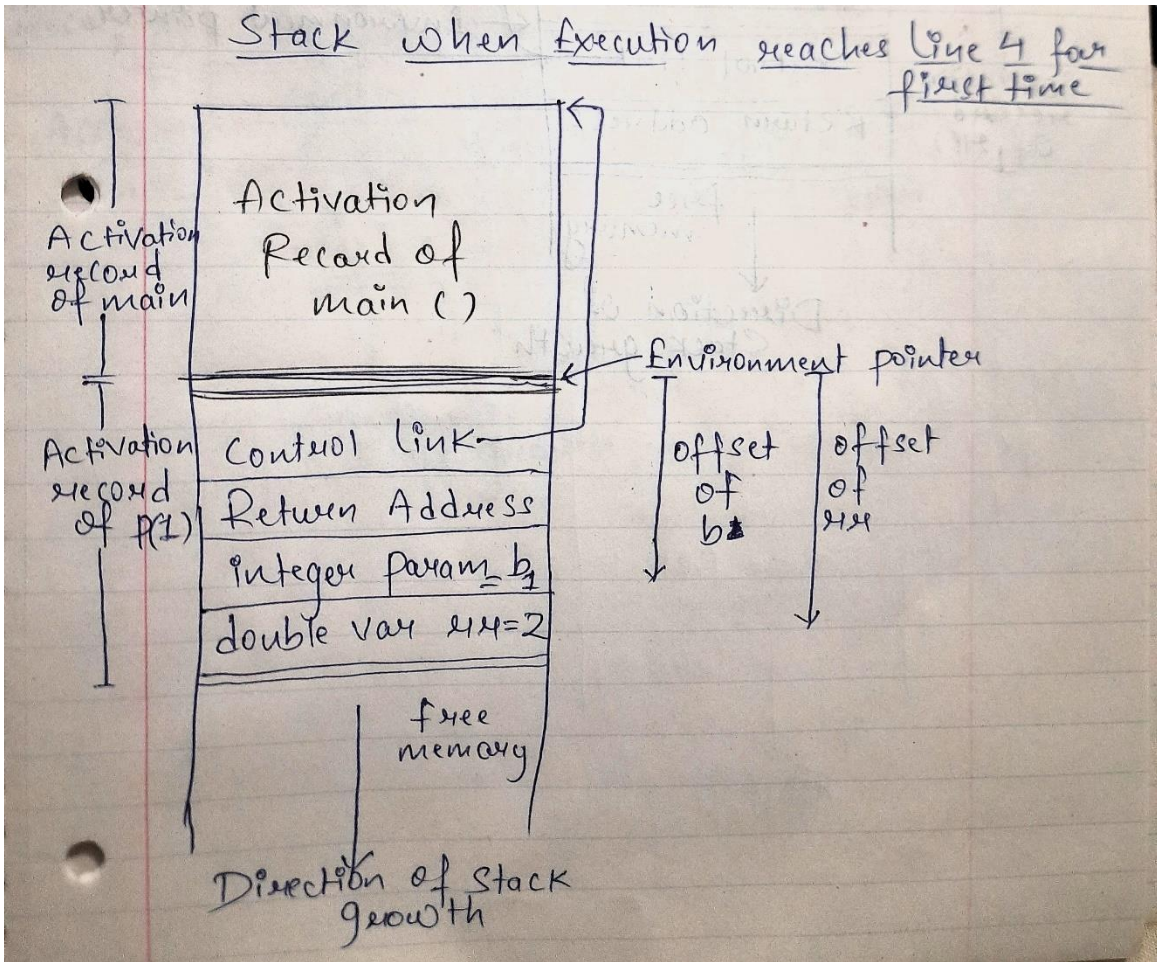
Due: 11:59PM EST, September 11, 2023

All questions must either be answered individually, or in a team of at most 2 members.

1. [48 points] For the following C program:

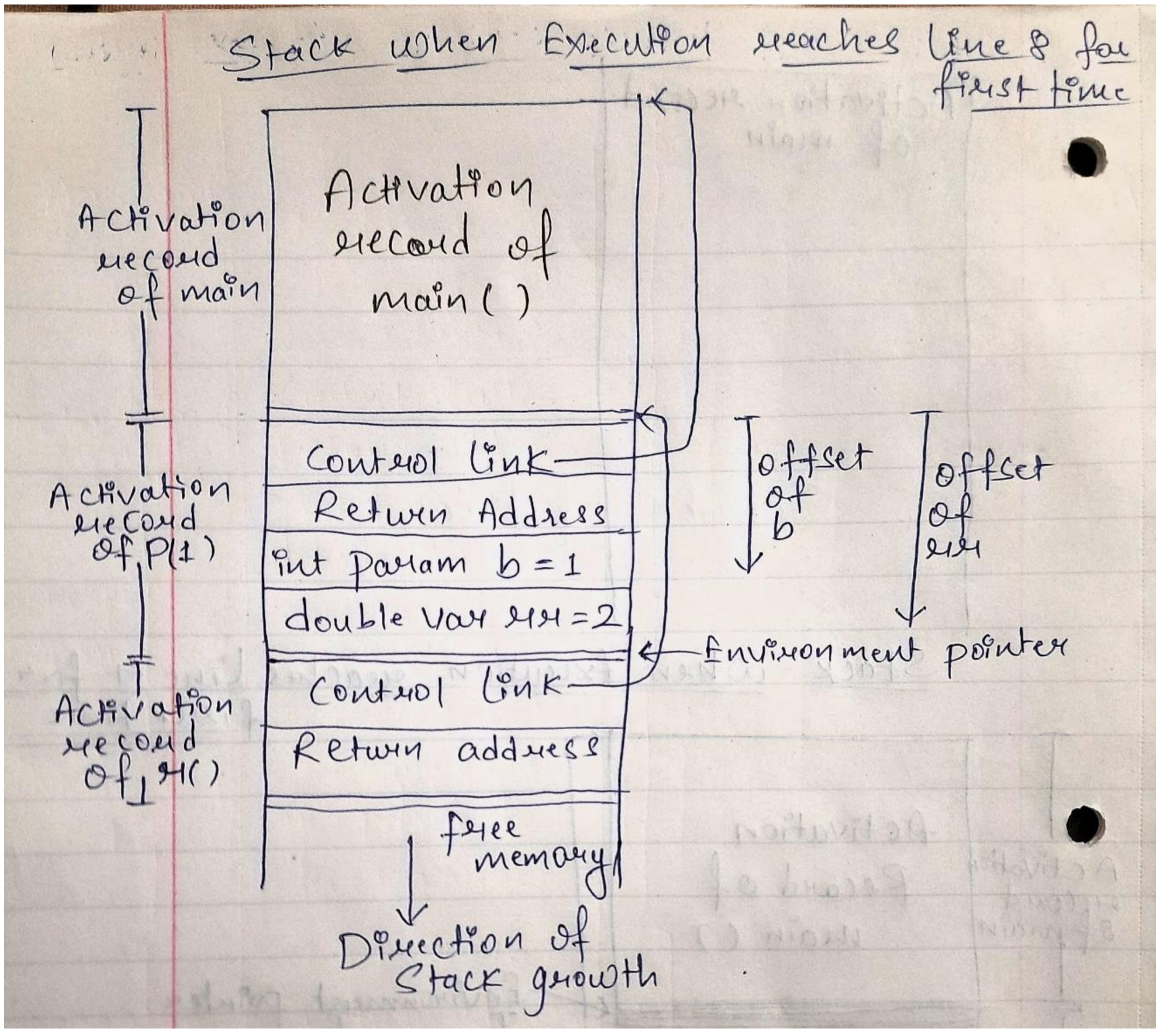
(a) **[4 points]** Draw the stack (with activation records) when the execution reaches line 4 for the first time.

Answer:


Stack when Execution reaches line 4 for first time
Activation record of main
Activation Record of main ( )
Environment pointer
Activation record of f(1)
Control link
Return Address
integer param b1
double var rr=2
offset of b
offset of rr
free memory
Direction of stack growth

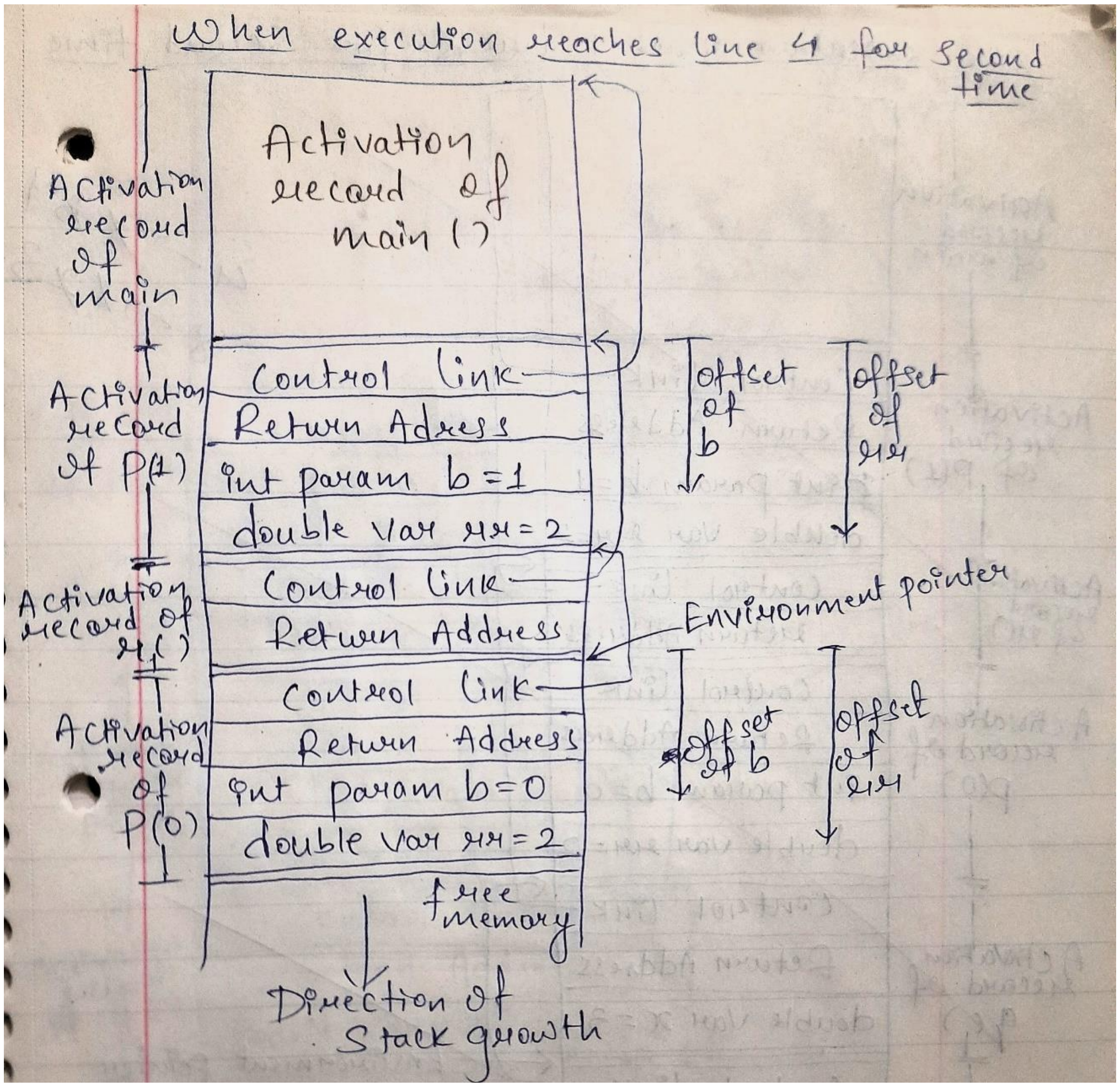
(b) **[4 points]** Draw the stack (with activation records) when the execution reaches line 8 for the first time.

Answer:


Stack when Execution reaches line 8 for first time
Activation record of main
Activation record of main ( )
Activation record of P(1)
Control link
Return Address
int param b = 1
double var rr = 2
Environment pointer
offset of b
offset of rr
Activation record of 1 r( )
Control link
Return address
free memory
Direction of Stack growth

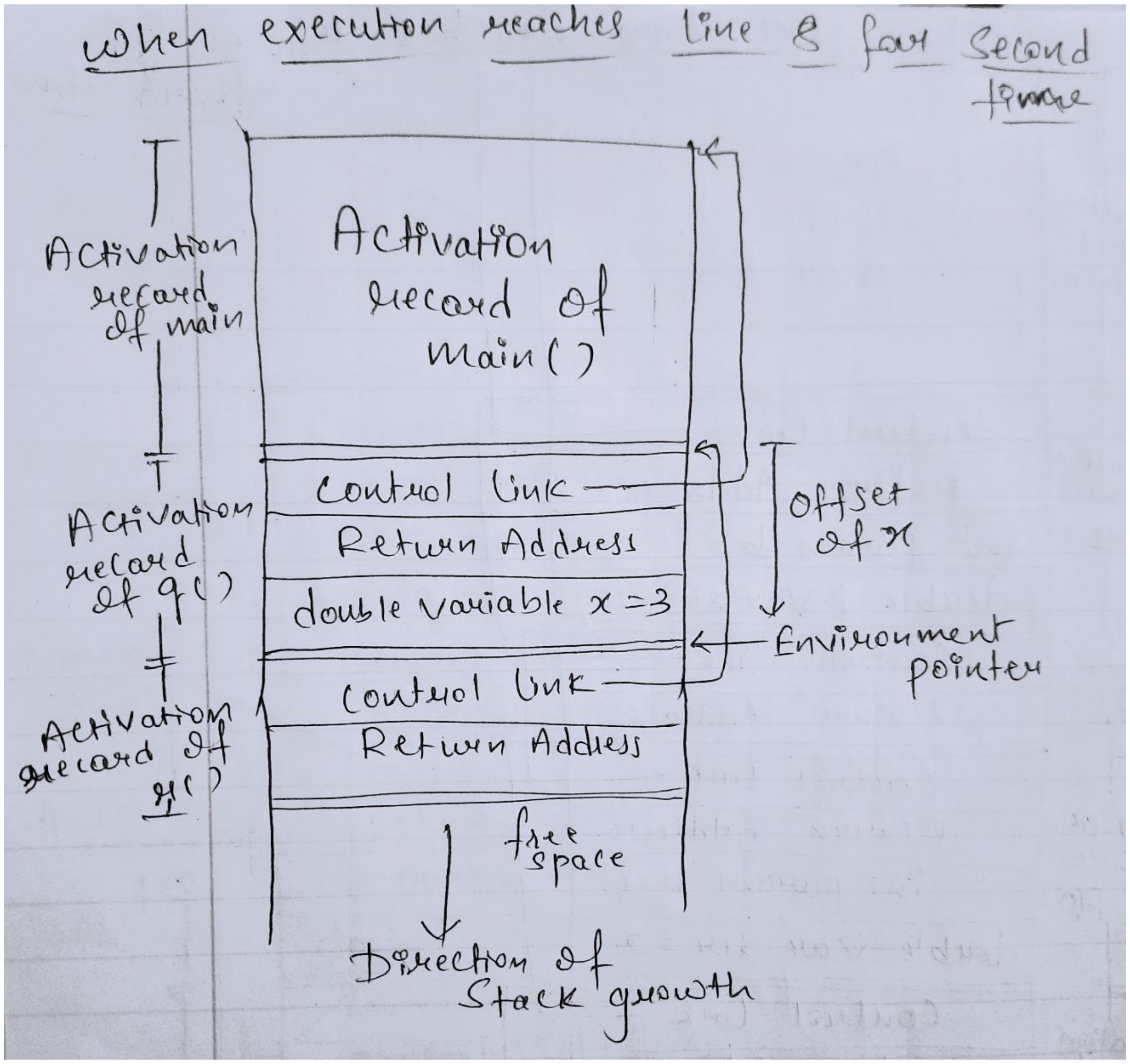
(c) **[4 points]** Draw the stack (with activation records) when the execution reaches line 4 for the second time.

Answer:



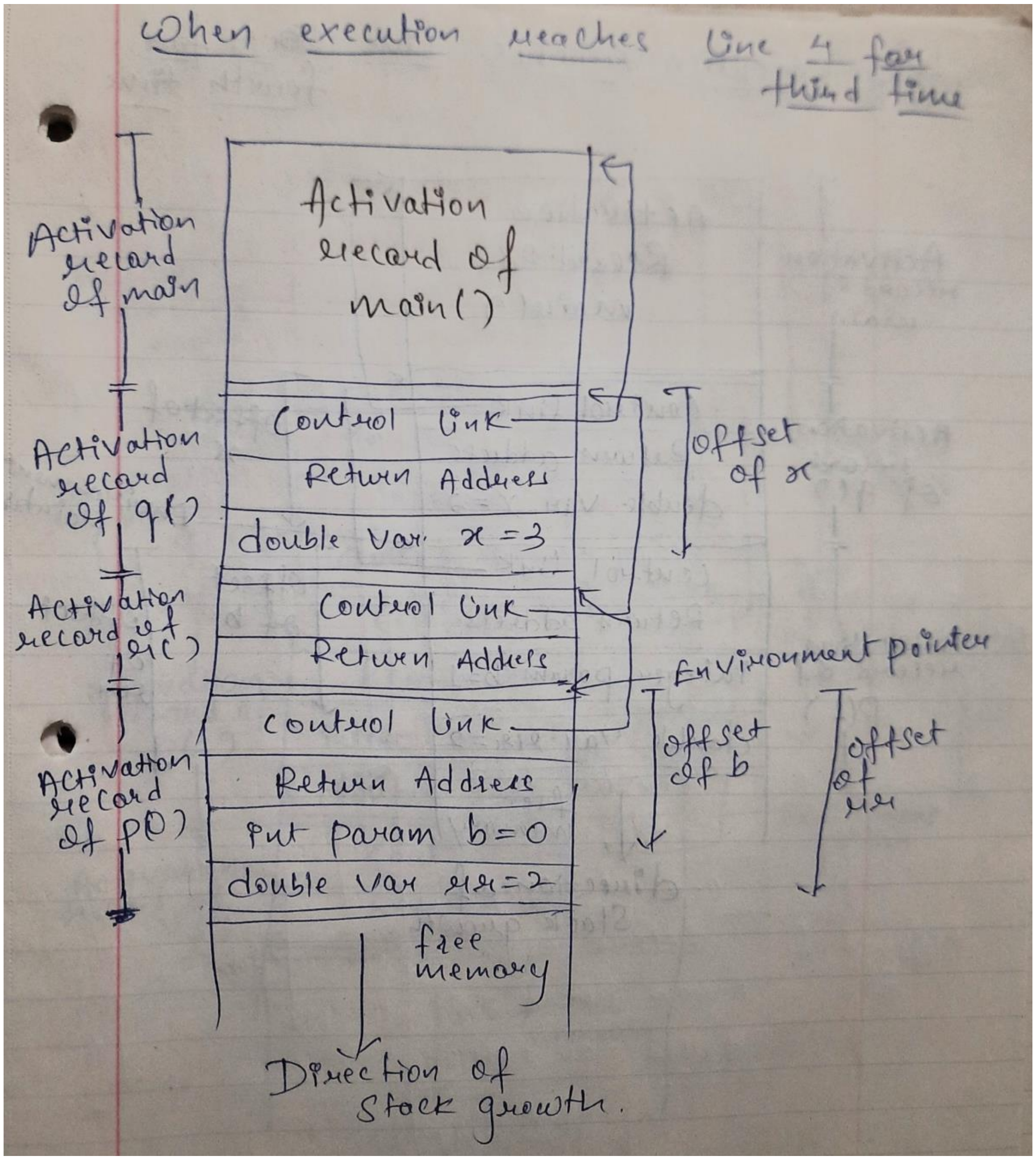
(d) **[4 points]** Draw the stack (with activation records) when the execution reaches line 8 for the second time.

Answer:



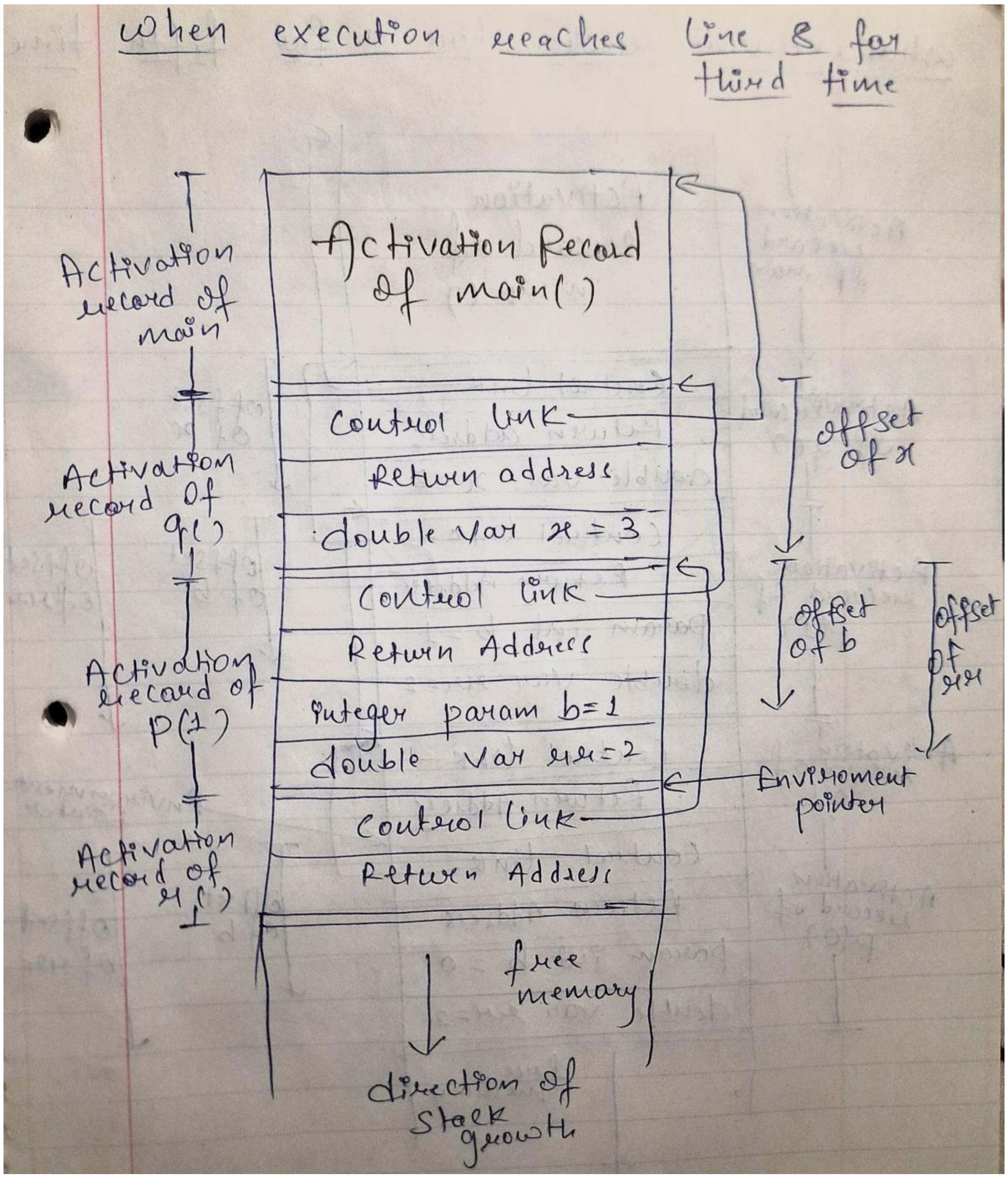
(e) **[4 points]** Draw the stack (with activation records) when the execution reaches line 4 for the third time.

Answer:



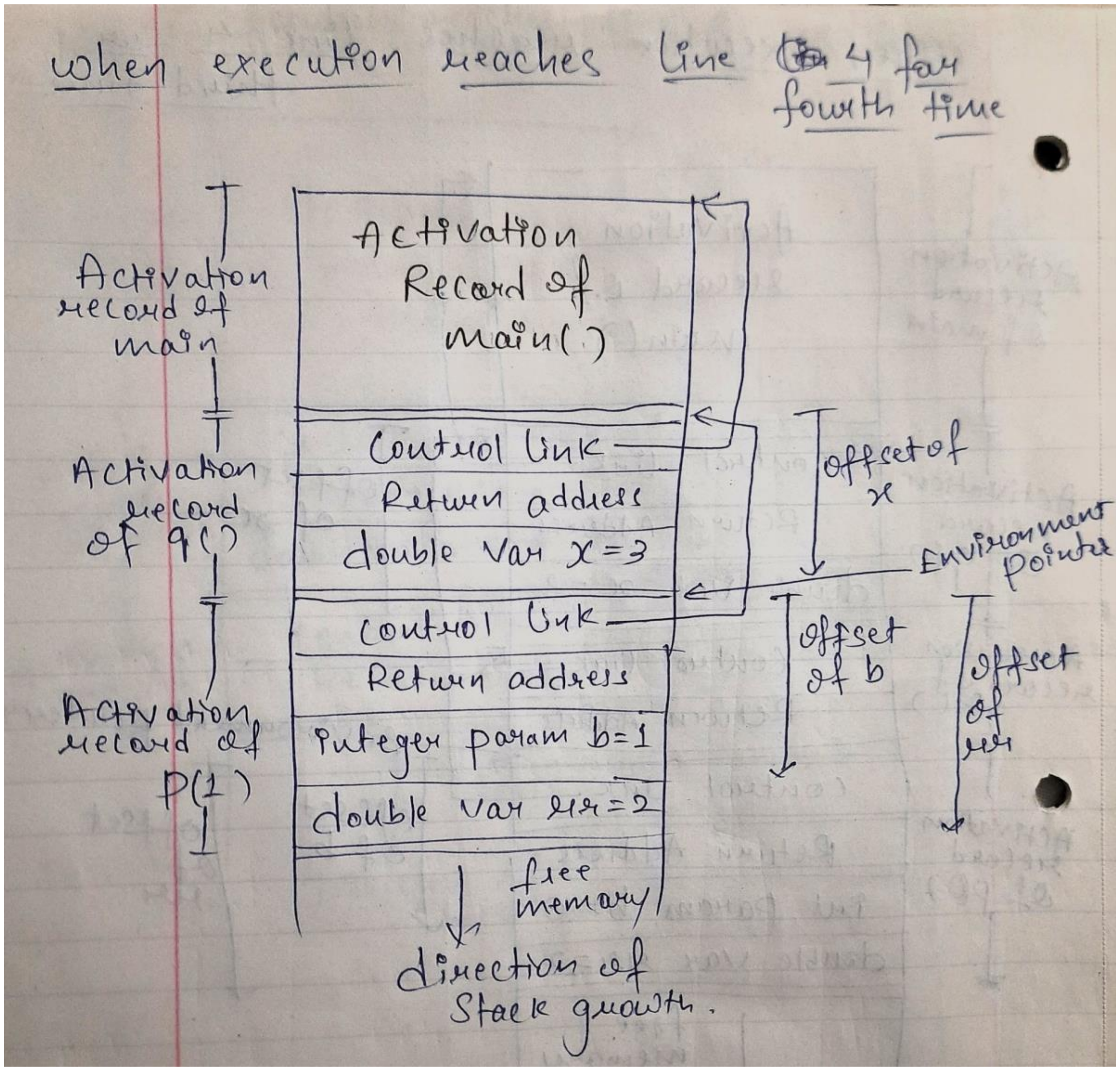
(f) **[4 points]** Draw the stack (with activation records) when the execution reaches line 8 for the third time.

Answer:


When execution reaches line 8 for third time
Activation Record of main( )
Activation record of main
Control link
Return address
double var x = 3
Activation record of q( )
offset of x
Control link
Return Address
integer param b=1
double var xx=2
Activation record of p(1)
offset of b
offset of xx
Enviroment pointer
Control link
Return Address
Activation record of r( )
free memory
direction of stack growth

(g) **[4 points]** Does the execution reaches line 4 for the fourth time? If so, draw the stack (with activation records) at the time. If not, explain why.

Answer: Yes, the execution reaches line 4 for the fourth time. Below is the stack at the time-

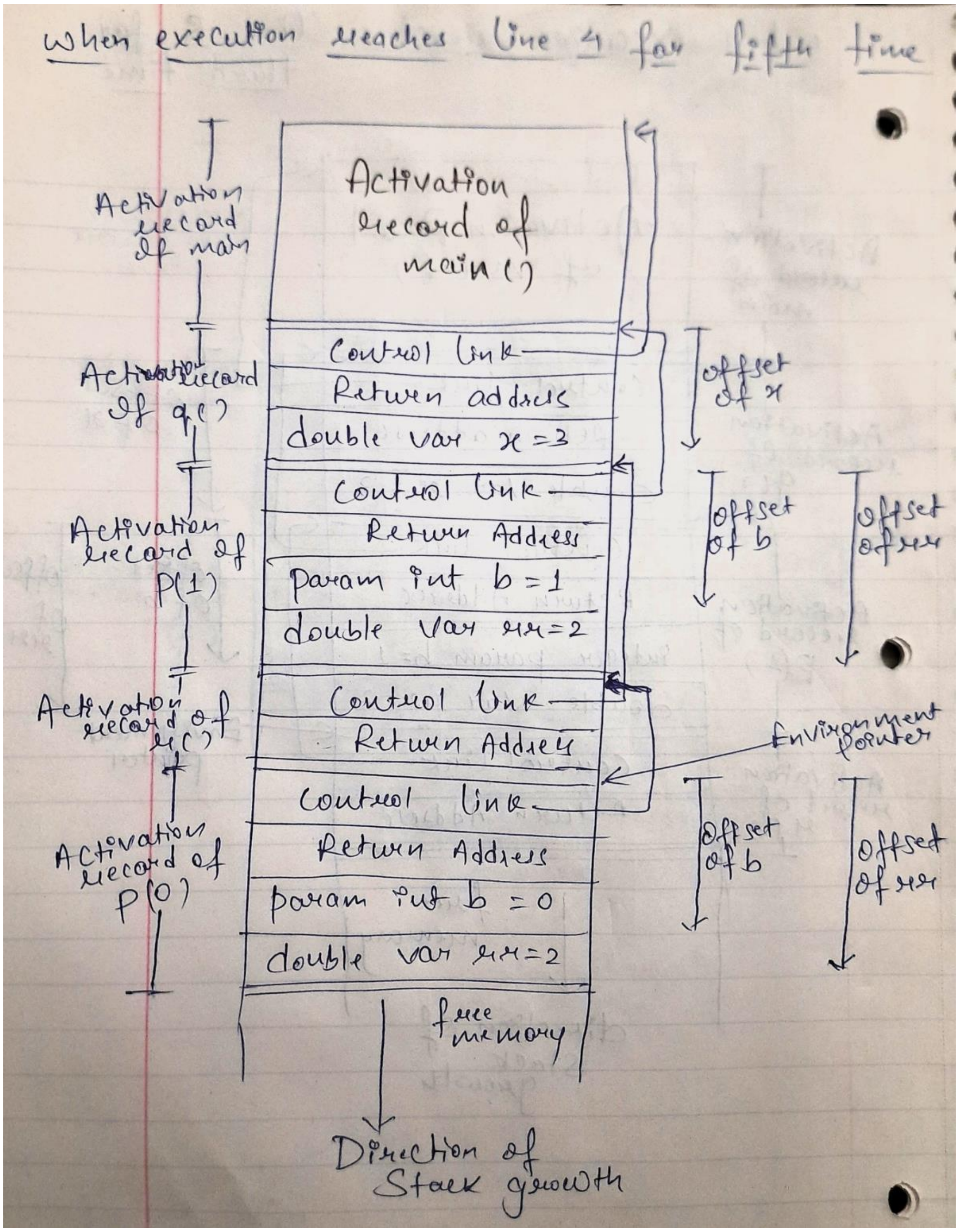


(h) **[4 points]** Does the execution reaches line 8 for the fourth time? If so, draw the stack (with activation records) at the time. If not, explain why.

Answer: No, the execution does not reach line 8 for the fourth time because after the execution reaches line 8 for the third time, function p(0) will be called and since the parameter passed to function p is 0, it will not call function r() again on line 5. Also, after call to p(0) is completed, activation records of p(0), r(), p(1) and q() will be dropped one by one as execution will reach to end of each function.

(i) **[4 points]** Does the execution reaches line 4 for the fifth time? If so, draw the stack (with activation records) at the time. If not, explain why.

Answer: Yes, the execution reaches line 4 for the fifth time. Below is the stack at the time-



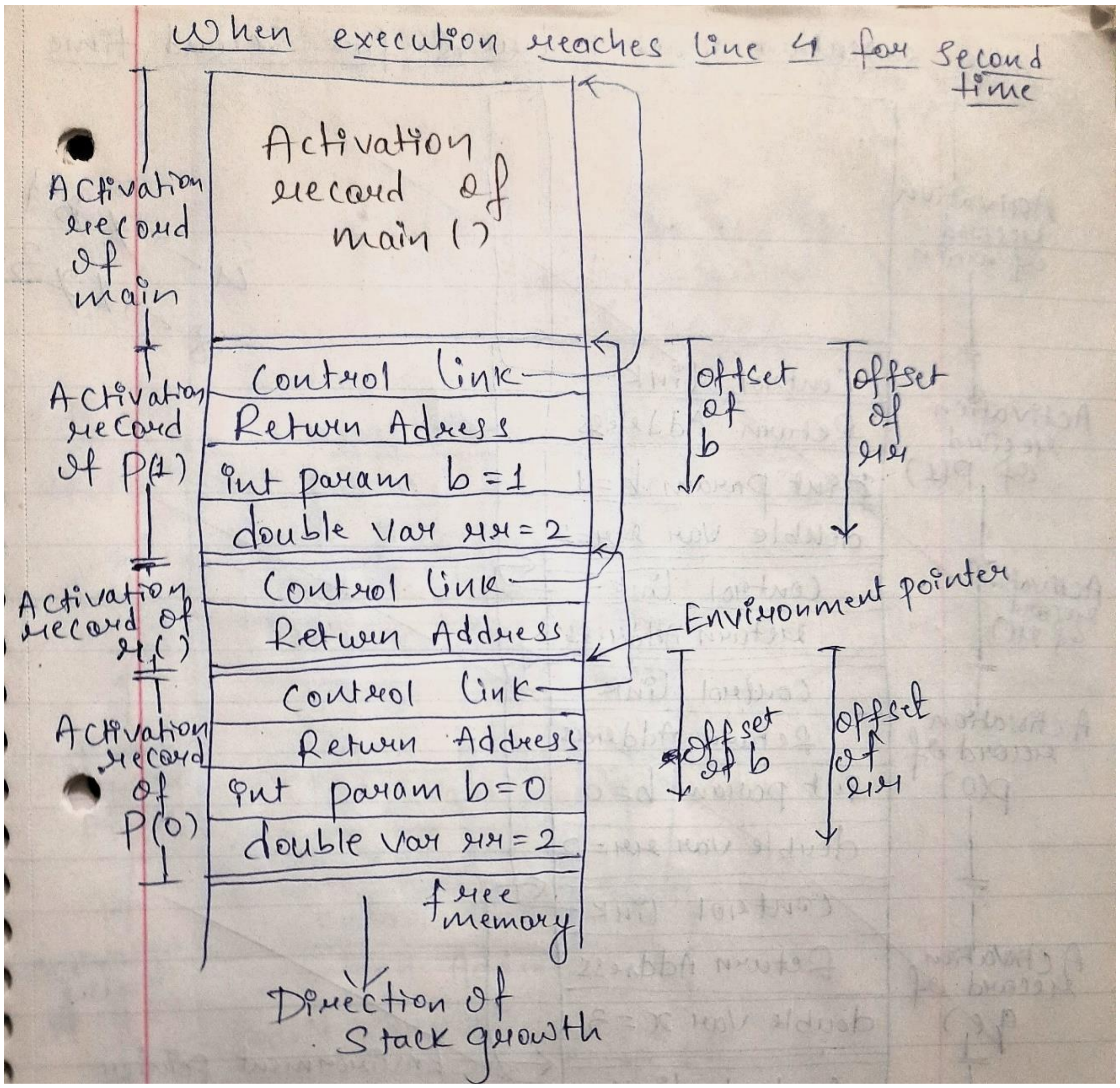
(j) **[4 points]** Does the execution reaches line 8 for the fifth time? If so, draw the stack (with activation records) at the time. If not, explain why.

Answer: No, the execution does not reach line 8 for the fifth time because it has not even reached line 8 for the fourth time.

(k) **[4 points]** Describe where variables rr and x are located when the program execution reaches line 4 for the second time.

Answer: When the program execution reaches line 4 for the second time, variable rr will be located in the activation records of p(0) and p(1) because it is a local member variable of function p. It will be located at a fixed offset from the start of activation record as shown in below diagram. Both the instance of rr will be different as variable rr is not statically declared.

Variable x will be located in the global/common data section accessible by all the functions because x is a global int variable.



When execution reaches line 4 for second time
Activation record of main ()
Activation record of main
Activation record of P(1)
Control link
Return Adress
int param b=1
double var rr=2
Activation record of r1()
Control link
Return Address
Environment pointer
Activation record of P(0)
Control link
Return Address
int param b=0
double var rr=2
offset of b
offset of rr
free memory
Direction of Stack growth

(l) **[4 points]** What are the print-out results of the program when it ends?

Answer:

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

(When you draw the activation records, you must show the control links and fill in the local names in each activation record.)

```
(1)     int x;
(2)     void p(int b)
(3)     { double rr = 2;
(4)        printf("%g\n",rr); printf("%d\n",x);
(5)        if (b) r();
(6)     }
(7)     void r(void)
(8)     { x=1;
(9)        p(0);
(10)    }
(11)     void q(void)
(12)    {  double x = 3;
(13)        r();
(14)        p(1);
(15)    }
(16)    main()
(17)    { p(1);
(18)      q();
(19)      return 0;
(20)    }
```

2. [12 points] Given the following C program

```
#include <stdio.h>
main()
{  int z;
   int** x;
   int* y;
```

```
    y = (int *) malloc(sizeof(int));
    x = &y;
    y = &z;
    z = 3;
   *y = 4;
   **x = z;
   printf("%d\n", *y);
   z = 2;
   printf("%d\n", *y);
  **x = 1;
   printf("%d\n", z);
}
```

(a) **[4 points]** Does the program produce dangling references? Why?

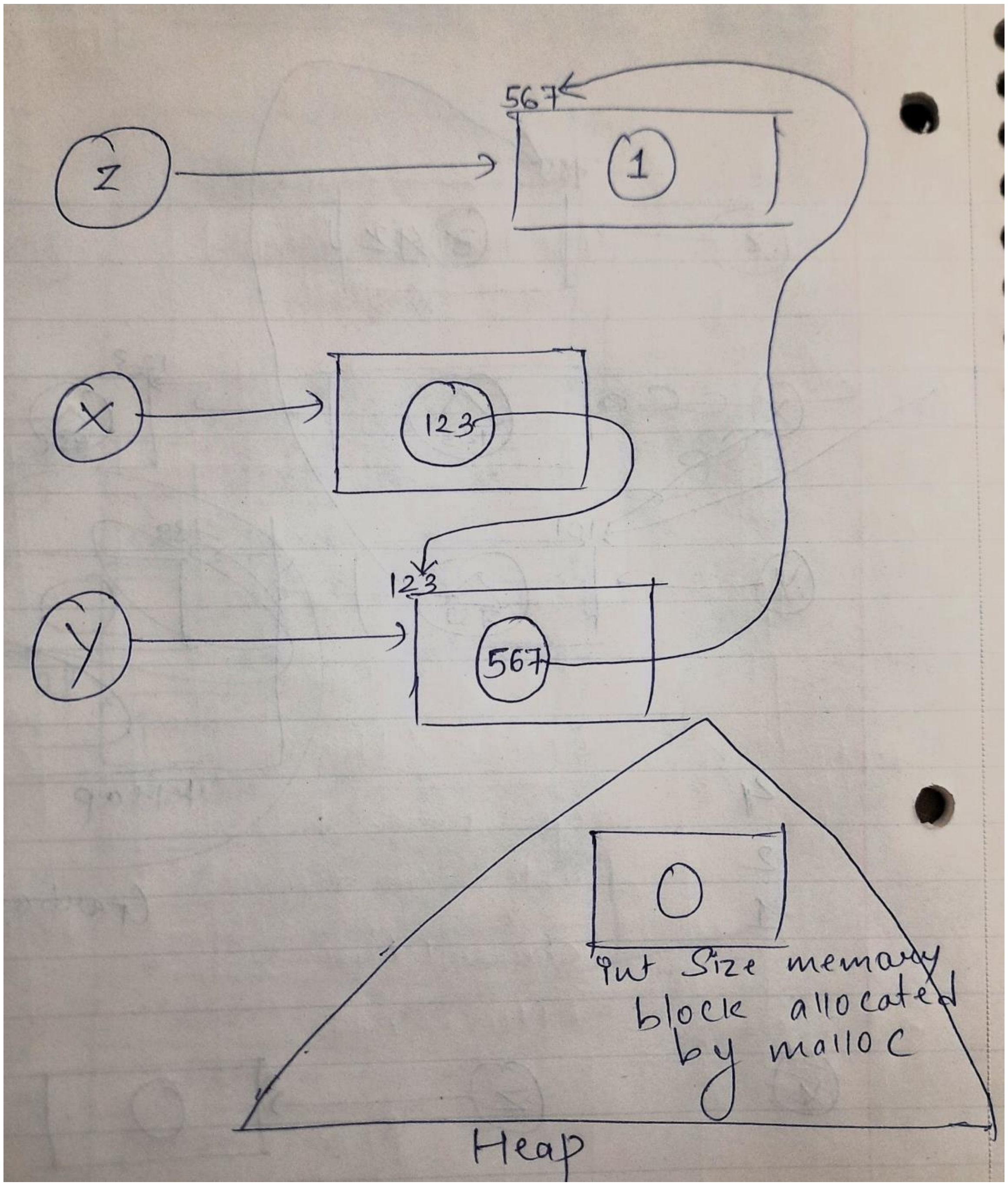
Answer: No, program does not produce dangling reference because none of memory locations pointed by the variables was made free while execution. Also, scope of any memory location allotted is not ending before the scope of variable.

(b) **[4 points]** Does the program produce garbage? Why?

Answer: Yes, program produces garbage because initially y stored address of a memory location allocated by malloc function and later address of z was assigned to y. After this assignment address of the memory location allocated by malloc is lost and it becomes a garbage block.

(c) **[4 points]** Draw the box and circle diagram (including all variables, together with the heap) when the program reaches the last line where "printf" for variable z is.

Answer: Note that address of y i.e 123 and address of z i.e 567 shown in below diagram are just for example and does not depict actual memory addresses.



3. [10 points] In the class, we mentioned that C++ supports call-by-reference, such as an example as follows:

```
int p(int& x) {
        x = x + 1;
        return x;
}
int z = 8;
int y = p(z);
```

One of the reasons why programmers do not find call-by-reference essential is that it can be simulated by call-by-value. In other words, it is possible to write a C++/C program that only uses call-by-value to “mimic” the behavior of the program above. Write a logically equivalent program for the code snippet above that allows variables y and z to be 9 and 9 (just as the call-by-reference program) at the end of the execution. (Hint: use pointers).

Answer:

```
#include <stdio.h>

int p(int* x) {
        *x = *x + 1;
        return (*x);
}
int main()
{
int z = 8;
int* q = &z;
int y = p(q);
//value of both y and z becomes 9
}
```

5. [30 points] Read paper **“The nesC Language: a Holistic Approach to Networked Embedded Systems”** and answer the following questions. The paper can be downloaded at: . https://sing.stanford.edu/site/publications/pldi03gay.pdf.

(a) **[5 points]** In our class, we mentioned that a static language design such as in Fortran 77 is ancient. nesC on the other hand is a relatively new language (younger than C, C# and Java for example), but it also calls for a static language design. To explain their motivations, the paper mentions that such a design makes “whole program analysis and optimization significantly simpler and more accurate”. Why are these goals so important for their application domain, sensor networks?

Answer: nesC calls for static language design as this makes “whole program analysis and optimization significantly simpler and more accurate”. These goals are so important for their application domain, sensor network because of below reason:

Whole-program analyses allows features like **data-race detection, which improves reliability** and **aggressive function inlining, which reduces resource consumption.** By performing whole-program optimizations and compile-time data race detection, nesC simplifies application development, reduces code size, and eliminates many sources of potential bugs. An important goal is to reduce run-time errors, since there is **no real recovery mechanism in the field except for automatic reboot**. These features are crucial to support lowpower “motes,” each of which execute concurrent, reactive programs that must operate with severe memory and power constraints. Also, nesC programs are subject to whole program analysis (for safety) and optimization (for performance).

(b) **[5 points]** nesC frowns upon the use of function pointers. Name one advantage and one disadvantage of this design decision.

Answer:
Advantage: Non-use of function pointer makes the whole program analysis easier for nesC compiler. In addition to static data race detection, the nesC compiler performs static component instantiation, whole-program inlining, and dead-code elimination. Use of function pointers will make it much more difficult to achieve these goals.

Disadvantage: We will loose all the flexibility of calling functions that function pointers can give us. It will restrict dynamic use functions in our program. Will not be able to decide which function to call during program execution.

(c) **[5 points]** Are memory leaks possible in nesC? If yes, write a short program that demonstrates the scenario. If not, explain why.

Answer:
Memory leaks possibilty in nesC is not seen throughout the paper because nesC programs are subject to whole program analysis (for safety) and optimization (for performance). nesC addresses safety through reduced expressive power. There is no dynamic memory allocation and the call-graph is fully known at compile-time.
The nesC compiler uses the application call-graph to eliminate unreachable code and module boundary crossings as well as to inline small functions.

(d) **[5 points]** In Figure 4, why doesn't `SurgeM` implement the `getData` function that is also declared in interface `ADC`?

Answer: The providers or an interface implement the commands, while the users implements the events. Since SurgeM is using interface ADC, it only implements the events and not the commands. Commands are implemented by the provider.

(e) **[5 points]** In Figure 4, the `StdControl.init` function calls `Timer.start` function. The `SurgeM` module however does not implement the `Timer.start` function. The interface `Timer` also does not contain the implementation of this function. Which function body will be called then?

Answer: TimerC, the TinyOS timer service, is implemented as a configuration, TimerC is built by wiring the two subcomponents given by the components declaration: and HWClock (access to the on-chip clock). It maps its StdControl and Timer interfaces to those of TimerM (StdControl = TimerM.StdControl, Timer = TimerM.Timer) and con nects the hardware clock interface used by TimerM to that provided by HWClock (TimerM.Clk -> HWClock.Clock).

(f) **[5 points]** The paper repeatedly mentions "inlining". In Figure 7, if we were to perform "inlining" to the function body of `sendData(),` what happens?

Answer: If we perform inlining to the function body of sendData(), all the calls to functions will be replaced by the actual code inside any function that is being called. And the call to commands will be resolved based on the parameters passed. Also it explicitly mentions uint16_t sensorReading due to which specific function is called.
This is shown below:

```
task void sendData() { // send sensorReading
adcPacket.data = sensorReading;
//call Send.send(&adcPacket, sizeof adcPacket.data);
adcPacket->amId = uint16_t; msg = data; ...
return SUCCESS;
}
```