CS-571 B HW01 13

Homework 1

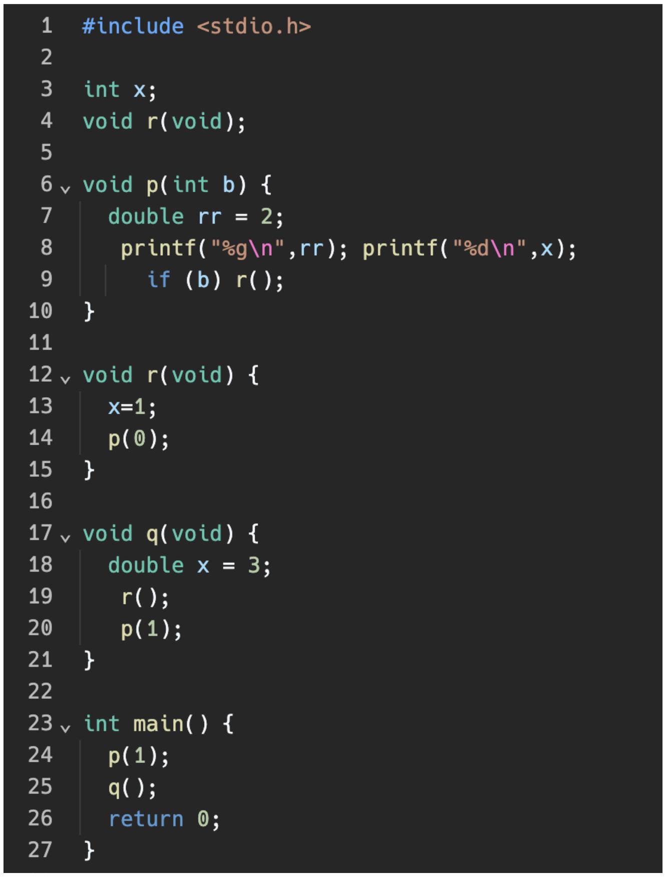
Shailee Manojkumar Patel

spatel31@binghamton.edu

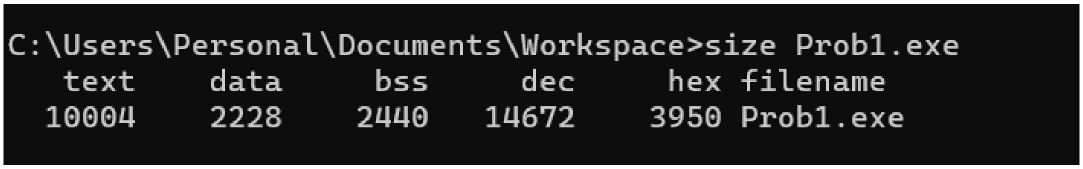
Harsimran Singh Avtar Singh Dhillon

hdhillon@binghamton.edu

1. **The following questions pertain to the attached prob1.c file.**

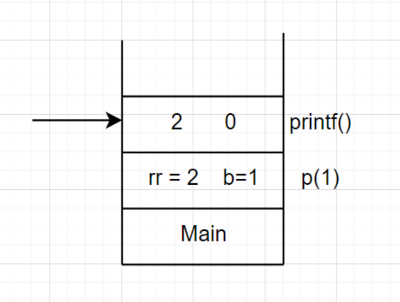


```
 1  #include <stdio.h>
 2
 3  int x;
 4  void r(void);
 5
 6  void p(int b) {
 7    double rr = 2;
 8     printf("%g\n",rr); printf("%d\n",x);
 9       if (b) r();
10  }
11
12  void r(void) {
13    x=1;
14    p(0);
15  }
16
17  void q(void) {
18    double x = 3;
19     r();
20     p(1);
21  }
22
23  int main() {
24    p(1);
25    q();
26    return 0;
27  }
```



```
C:\Users\Personal\Documents\Workspace>size Prob1.exe
   text    data     bss     dec     hex filename
  10004    2228    2440   14672    3950 Prob1.exe
```

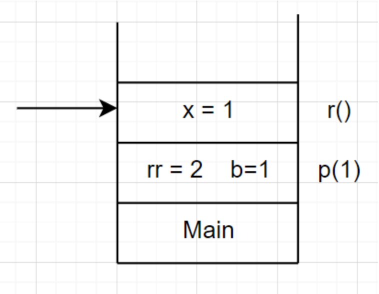
a. Describe the program stack (with activation records) when execution of the instructions in prob1.c reaches line 8 for the first time. When you describe the activation records, make sure you include control links, and all fields included in each activation record, including parameters and local variables.



Control Link: p(1)
Parameters: rr (Value = 2), x (Value = 0, by default initialization of any variable is 0)
Local Variables: None
Called From: p(1)
Return To: The next instruction after printf() in p(1)
Return value: None(Printf function does not return any value)
Pointer to caller's activation record: p(1)

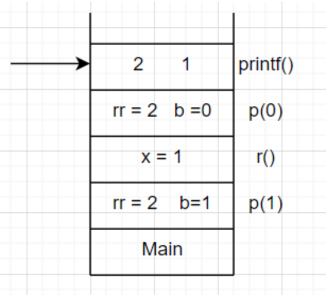
b. Describe the program stack (with activation records) when execution of prob1.c reaches line 13 for the first time.

Control Link: p(1)
Parameters: None
Local Variables: x = 1
Called From: p(1)
Return To: The next instruction after "x = 1" in r()
Return value: None
Pointer to caller's activation record: p(1)



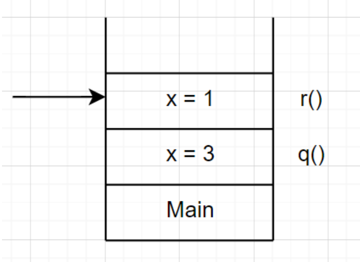
c. Describe the program stack (with activation records) when execution of prob1.c reaches line 8 for the second time.

Control Link: p(0)
Parameters: rr (Value = 2), x (Value = 1)
Local Variables: None
Called From: p(0)
Return To: The next instruction after printf() in p(0)
Return value: None(Printf function does not return any value)
Pointer to caller's activation record: p(0)



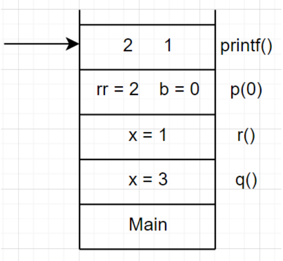
d. Describe the program stack (with activation records) when execution of prob1.c reaches line 13 for the second time.

Control Link: q()
Parameters: None
Local Variables: x = 1
Called From: q()
Return To: The next instruction after "x = 1" in r()
Return value: None
Pointer to caller's activation record: q()



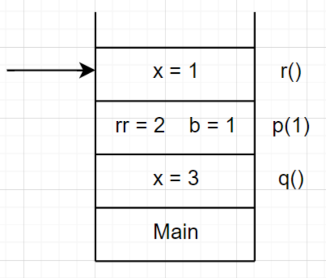
e. Describe the program stack (with activation records) when execution of prob1.c reaches line 8 for the third time.

Control Link: p(0)
Parameters: rr (Value = 2), x (Value = 1)
Local Variables: None
Called From: p(0)
Return To: The next instruction after printf() in p(0)
Return value: None(Printf function does not return any value)
Pointer to caller's activation record: p(0)



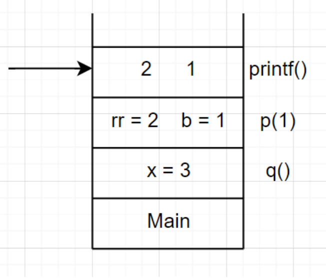
f. Describe the program stack (with activation records) when execution of prob1.c reaches line 13 for the third time.

Control Link: p(1)
Parameters: None
Local Variables: x = 1
Called From: p(1)
Return To: The next instruction after "x=1" in r()
Return value: None
Pointer to caller's activation record: p(1)



g. Does the execution of prob1.c reach line 8 for a fourth time? If so, describe the program stack. If not, explain why.

Control Link: p(1)
Parameters: rr (Value = 2), x (Value = 1)
Local Variables: None
Called From: p(1)
Return To: The next instruction after printf() in p(1)
Return value: None(Printf function does not return any value)
Pointer to caller's activation record: p(1)

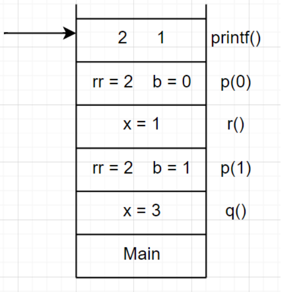


h. Does the execution of prob1.c reach line 13 for a fourth time? If so, describe the program stack.If not, explain why.

No, the program does not reach line 13 for a fourth time. Because after the execution of line 8 for the fourth time which is done by function p(0). Here the value of the 'b' variable will be 0 which is passed by parameter is 0. Since the value of the variable 'b' is 0, the if condition won't be satisfied and the r() function won't be called.

i. Does the execution of prob1.c reach line 8 for a fifth time? If so, describe the program stack. If not, explain why.

Control Link: p(0)
Parameters: rr (Value = 2), x (Value = 1)
Local Variables: None
Called From: p(0)
Return To: The next instruction after printf() in p(0)
Return value: None(Printf function does not return any value)
Pointer to caller's activation record: p(0)

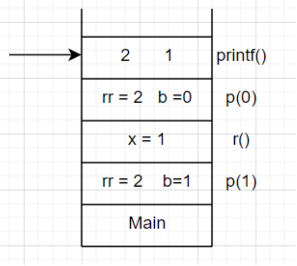


j. Does the execution of prob1.c reach line 13 for a fifth time? If so, describe the program stack. If not, explain why.

No, the program was not able to execute line 13 for the fourth and the fifth time, because the condition required for r() to be executed was unable to meet. The r() function was able to be executed only 3 times, where 2 times it was called by the p() function and 1 time by the q(). q() function called the r() 1 time. Whereas the execution of r() from p() was dependent on the value of the variable 'b'. The condition was satisfied only twice, making the total number of executions of the r() function to be 3 times i.e the Line 13 was executed only 3 times.

k. Describe where variables rr and x are located when the program execution reaches line 8 for the second time.

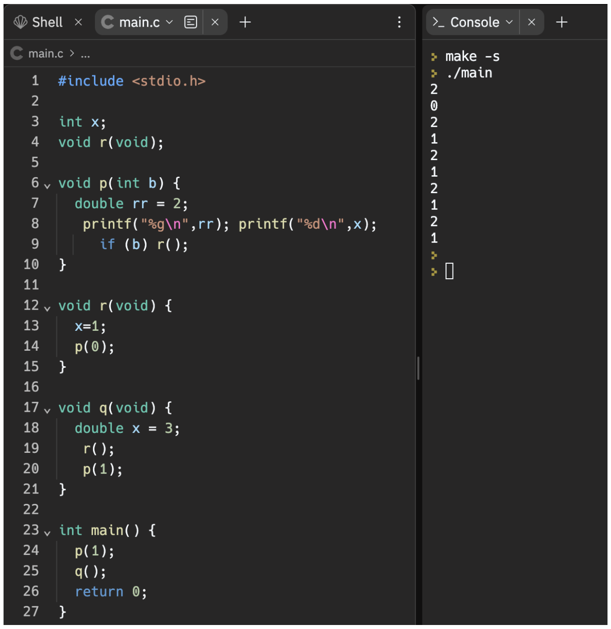
The variable rr is a local variable declared and initialized **(rr = 2)** by the function p(int b) and the variable x is a global variable. When the program execution reaches line 8 for the second time, the value of **rr corresponds to 2** and value of **x corresponds to 1** which is Initialized by the function r(). This printf() statement is called by the function p(0) which was called by the function r().



l. What do the instructions in prob1.c print out when executed?

Output when instructions in prob1.c is executed :

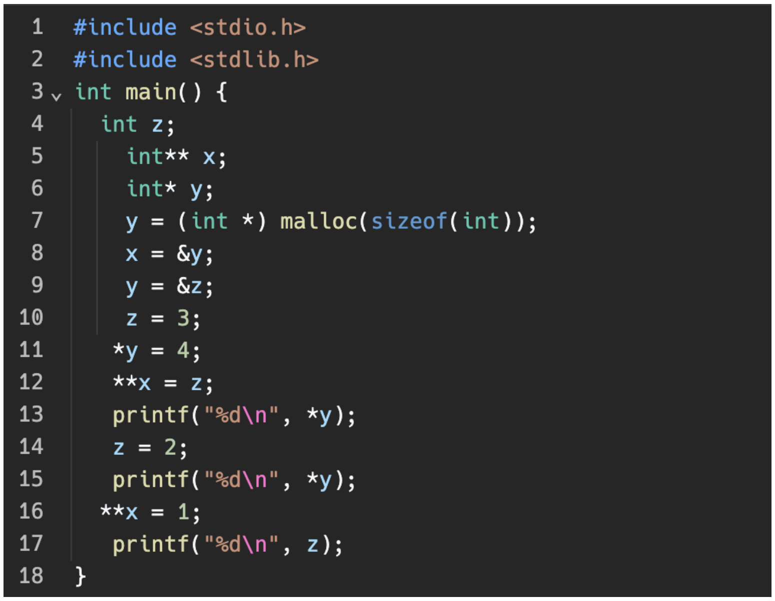
2 —> value of rr, when p(1) is called
0 —> value of uninitialised variable x is by default '0'
2 —> value of rr, when p(0) is called under function r()
1 —> value of x is set as 1 in function r()
2 —> value of rr, when r() is called under function q()
1 —> value of x, when r() is called under function q()
2 —> value of rr, when p(1) is called under function q()
1 —> value of x, when p(1) is called under function q()
2 —> value of rr, when p(0) is called under r() function in q()
1 —> value of x, when p(0) is called under r() function in q()



```
1  #include <stdio.h>
2
3  int x;
4  void r(void);
5
6  void p(int b) {
7    double rr = 2;
8     printf("%g\n",rr); printf("%d\n",x);
9       if (b) r();
10 }
11
12 void r(void) {
13   x=1;
14   p(0);
15 }
16
17 void q(void) {
18   double x = 3;
19    r();
20    p(1);
21 }
22
23 int main() {
24   p(1);
25   q();
26   return 0;
27 }
```

```
> make -s
> ./main
2
0
2
1
2
1
2
1
2
1
>
>
```

2. **The following questions pertain to the attached prob2.c file**



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4    int z;
5      int** x;
6      int* y;
7      y = (int *) malloc(sizeof(int));
8      x = &y;
9      y = &z;
10     z = 3;
11    *y = 4;
12    **x = z;
13    printf("%d\n", *y);
14    z = 2;
15    printf("%d\n", *y);
16   **x = 1;
17    printf("%d\n", z);
18 }
```

(Img 2.)

a. **Does the program produce dangling references? Why?**

No, the program doesn't produce dangling references. Initially, 'y' is allocated the memory at line 7 using 'malloc' and 'x' is pointed to the address of 'y' at line 8. Later, at line 9, 'y' is pointed to the address of 'z', which breaks the link between 'x' and allocated memory of 'y'. At line 11 and line 12, 'x' still points to the address of 'y' but 'y' points to the address of 'z'. As a result, memory is allocated using 'malloc' and it doesn't produce dangling references.

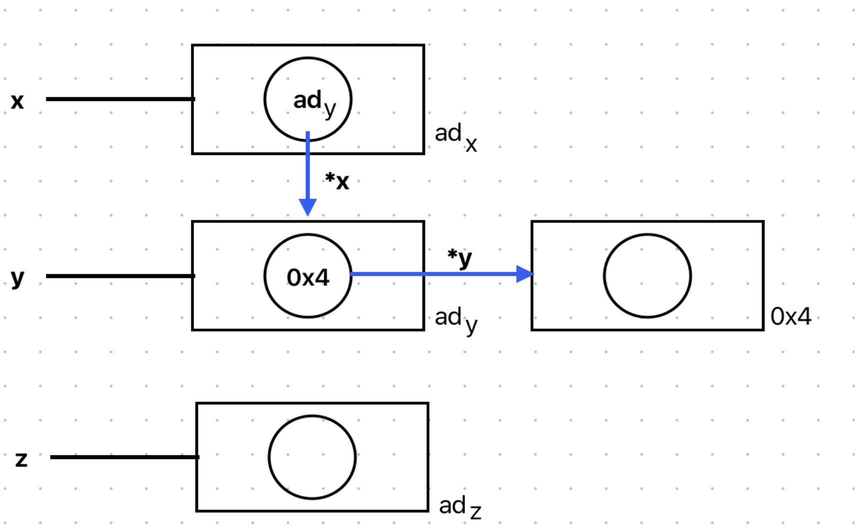
b. **Does the program produce garbage? Why?**

The program produces garbage as the memory is not deallocated but reference has been removed.

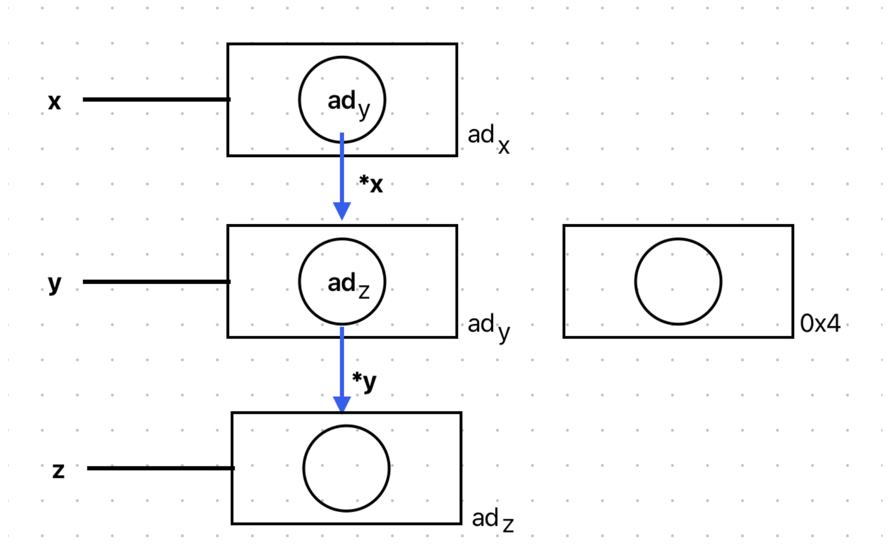
c. **Draw the box and circle diagram, including all variables together with the heap when the program reaches the last line that prints the value of variable "z".**

#(ad represents address of the variable)

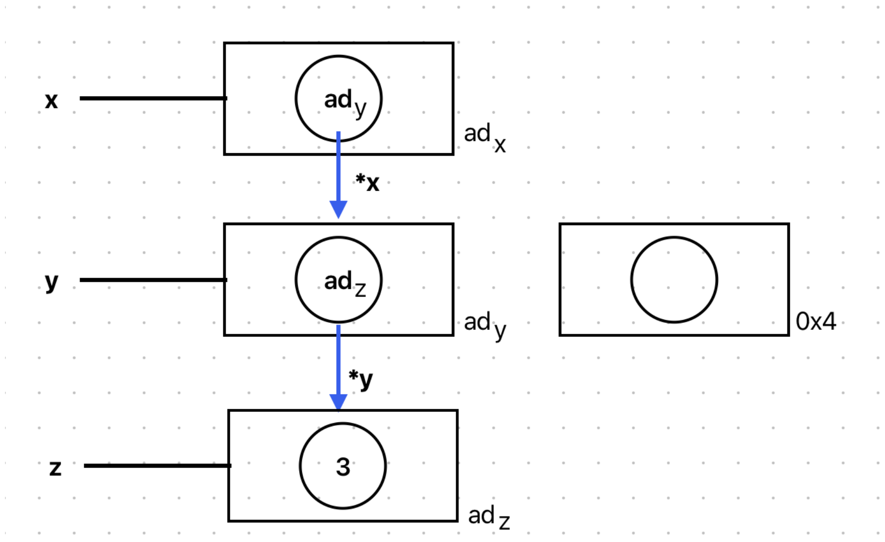
```
int z;
int** x;
int* y;
y = (int *) malloc(sizeof(int));
x = &y;
y = &z;
z = 3;
*y = 4;
**x = z;
printf("%d\n", *y);
z = 2;
printf("%d\n", *y);
**x = 1;
printf("%d\n", z);
```



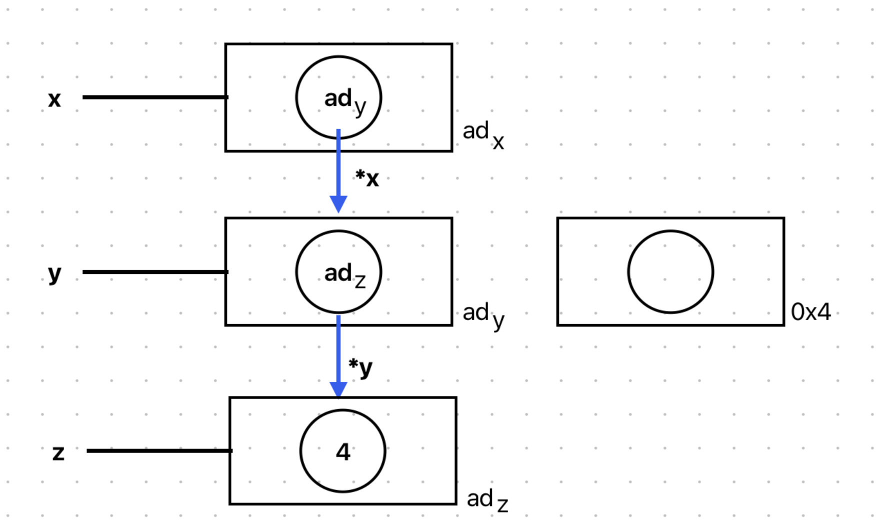
```
int z;
int** x;
int* y;
y = (int *) malloc(sizeof(int));
x = &y;
y = &z;
z = 3;
*y = 4;
**x = z;
printf("%d\n", *y);
z = 2;
printf("%d\n", *y);
**x = 1;
printf("%d\n", z);
```



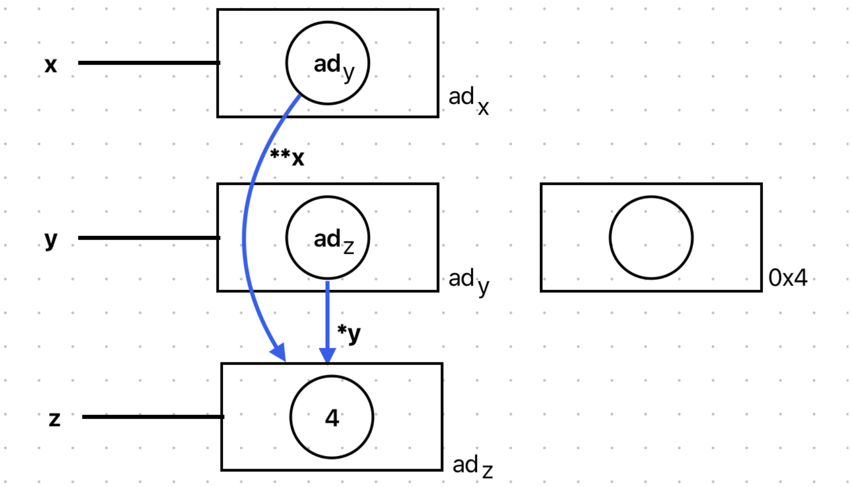
```
int z;
int** x;
int* y;
y = (int *) malloc(sizeof(int));
x = &y;
y = &z;
z = 3;
*y = 4;
**x = z;
printf("%d\n", *y);
z = 2;
printf("%d\n", *y);
**x = 1;
printf("%d\n", z);
```



```
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);
```

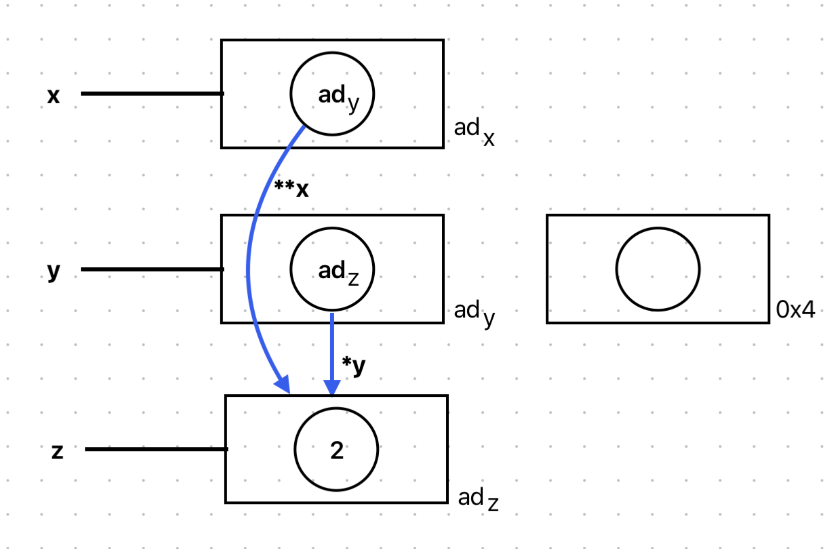


```
int z;
int** x;
int* y;
y = (int *) malloc(sizeof(int));
x = &y;
y = &z;
z = 3;
*y = 4;
**x = z;
printf("%d\n", *y);
z = 2;
printf("%d\n", *y);
**x = 1;
printf("%d\n", z);
```



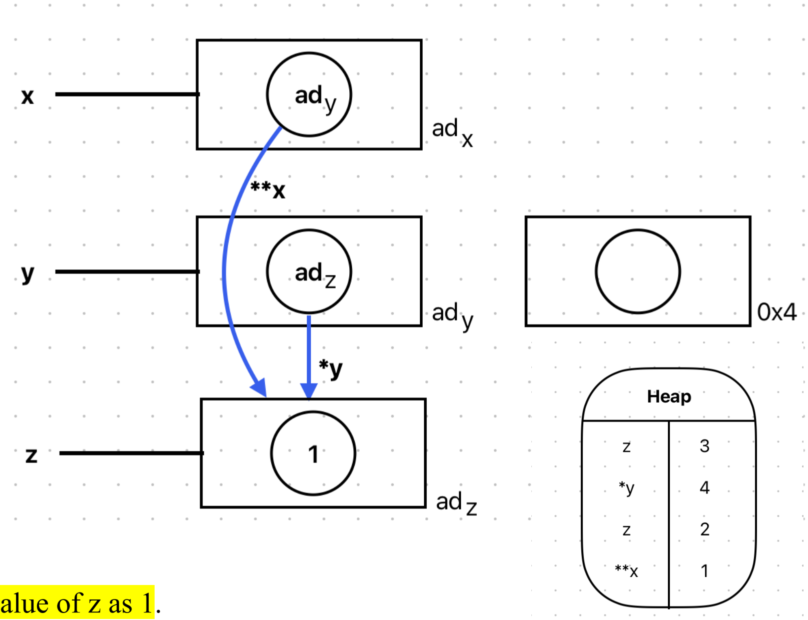
The print statement after ‘**x = z’, prints value of *y as 4.

```
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);
```



The print statement after ‘z=2’, prints value of *y as 2.

```
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);
```



z	3
*y	4
z	2
**x	1

The print statement after ‘**x = 1’, prints value of z as 1.

3. **Given the following program, in C-like pseudo-code:**

```
int i;
int a[3];
void swap(int x, int y) {
  x = x+y;
  y = x-y;
  x = x-y;
}
int main() {
  i=1;
 a[0]=2;
 a[1]=1;
 a[2]=0;
 swap(i,a[i]);
 printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
 swap(a[i],a[i]);
 printf("%d %d %d\n", a[0], a[1], a[2]);
 return 0;
}
```

a. What would this program print out, assuming parameters are passed using the call by value method?

1 2 1 0

2 1 0

In Pass by value we only pass the value to the x and y variables of the swap function. The arithmetic operations are performed on the x and y variables to swap the values which will be resulting in the changes of value of x and y only. There will be no change in the values of i and a[i] which were passed as an argument to the swap function. So while printing the values of i and the array a[], we will see no changes in the value and they will be printed as they were initialized. Even for the second swap function where we are passing a[i] and a[i], there will be no change in the values of a[i] and while printing we will get the same value as initialized.

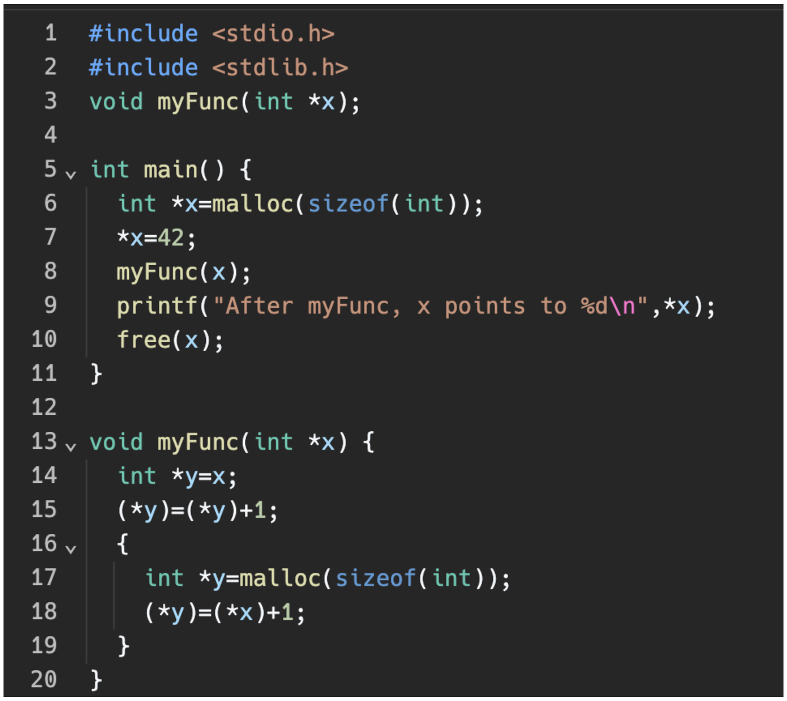
b. What would this program print out, assuming parameters are passed using the call by reference method?

1 2 1 0

2 0 0

In Pass by reference we pass the reference of the variable to the x and y variables of the swap function. So the arithmetic operations performed on x and y variables of the swap function will also change the values of i and a[i]. And the new values will be updated for these variables. As for the first swap function where we are passing the value of i as 1 and a[i] i.e a[1] as 1 we are getting the same values after swapping because the arithmetic operations are resulting in the same value. The second swap function is taking in the values of a[1] as 1 and the value returned after swapping for a[1] is 0. That's why we have got the value as 2 0 0 while printing the values of a[0], a[1] and a[2] respectively.

4. The following questions pertain to the attached prob4.c file



```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  void myFunc(int *x);
 4
 5  int main() {
 6    int *x=malloc(sizeof(int));
 7    *x=42;
 8    myFunc(x);
 9    printf("After myFunc, x points to %d\n",*x);
10    free(x);
11  }
12
13  void myFunc(int *x) {
14    int *y=x;
15    (*y)=(*y)+1;
16    {
17      int *y=malloc(sizeof(int));
18      (*y)=(*x)+1;
19    }
20  }
```

(Img 4.)

a. Does the program in prob4.c have any aliases? If so, list all the names or expressions that reference the same memory, and identify the scope of those aliases.

Yes, the program in prob4.c has aliases that reference the same memory. Following is the list of aliases:

a. In the 'main' function (line 6 in the Img 4)
int *x = malloc(sizeof(int));
'x' pointer is in the 'main' function and refers to a block of dynamically allocated memory.

b. In the 'myFunc' function (line 14 in the Img 4)
int *y = x;
The 'y' pointer is in the same scope of the 'myFunc' function and refers to the same memory as the 'x' pointer. It is an alias for 'x' within the function scope.
In the 'myFunc' function, there are 2 names that reference the same memory, 'x' and 'y'.

b. Does the program in prob4.c have any dangling references? If so, where was the memory allocated? (A line number in the prob4.c file is good enough for an answer.)

The program in prob4.c doesn't have any dangling references. The memory for 'x' was allocated on line 6 (Img 4) and the memory was correctly freed at line 10 (Img 4) before the program existed.

c. Does the program in prob4.c produce any garbage? If so, where was the memory allocated? (Again, a line number is good enough.)

Yes, the program in prob4.c produces garbage memory. The memory was allocated on line 17 for a new y. (refer Img 4)

d. What does prob4.c print out when it is executed?

<u>It prints:</u>

After myFunc, x points to 43