15-213: Introduction to Computer Systems
Written Assignment #5

This written homework covers Design, Debugging, Testing, Optimization and Linking.

Directions

Complete the question(s) on the following pages with single paragraph answers. These questions are not meant to be particularly long! Once you are done, submit this assignment on Canvas.

Below is an example question and answer.

Q: Please describe benefits of two's-complement signed integers versus other approaches.

A: Other representations of signed integers (ones-complement and sign-and-magnitude) have two representations of zero (+0 and −0), which makes testing for a zero result more difficult. Also, addition and subtraction of two's complement signed numbers are done exactly the same as addition and subtraction of unsigned numbers (with wraparound on overflow), which means a CPU can use the same hardware and machine instructions for both.

Grading

Assignments are graded via *peer review:*

1. Three other students will each provide short, constructive feedback on your assignment, and a score on a scale of 1 to 15. You will receive the maximum of the three scores.
1. You, in turn, will provide feedback and a score for three other students (not the same ones as in part 1). We will provide a *rubric*, a document describing what good answers to each question look like, to assist you. You receive five additional points for completing all of your peer reviews.

Due Date

This assignment is due on **Wednesday, July 13th by 11:59pm** Pittsburgh time (currently UTC−4). Remember to convert this time to the timezone you currently reside in.

Peer reviews will be assigned roughly 12 hours later, and are due a week after that.

Question #1

Explain why compiler optimization is necessary. What are the two types of compiler optimization and give an example of each.

Explains at least 1 reason why compiler optimization is necessary in terms of improvising the speed of code execution or reducing the size of code.

Mentions the two types of compiler optimization:

1. Local
    - Example associated with constant folding, strength reduction or dead code elimination
2. Global
    - Example associated with loop transformation or code motion

Question #2

What would be the expected output of the following code:

```c
#include <stdio.h>
#include <string.h>

void test () {
   int x = 100;
   size_t s = strlen("I am enjoying 1x-x13!");

   if (s) {
       printf("Condition 1\n");
   } else {
       printf("Condition 2\n");
   }

   x -= 100;

   if (x && s) {
       printf("Condition 3\n");
   } else if (x) {
       printf("Condition 4\n");
   } else {
       printf("Condition 5\n");
   }
}

void main () {
   test();
}
```

What are the different parts of this code that can be eliminated/substituted?

Output:
Condition 1
Condition 5

Optimizations that can be made:

1. No need for **int x = 100;** as it is being changed later on and isn't used anywhere before that. It can directly be initialized as **int x = 0**
2. Size of the string can directly be hardcoded.
3. Remove the unnecessary dead conditions as the code will never go there:
    a. Condition 2
    b. Condition 3
    c. Condition 4
4. The conditions can be fully removed and the final code can look something like:

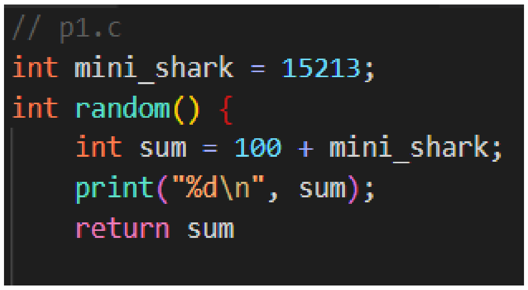
```
void test () {
   size_t s = 21;
   int x = 0;

   printf("Condition 1\n");
   printf("Condition 5\n");
}
```

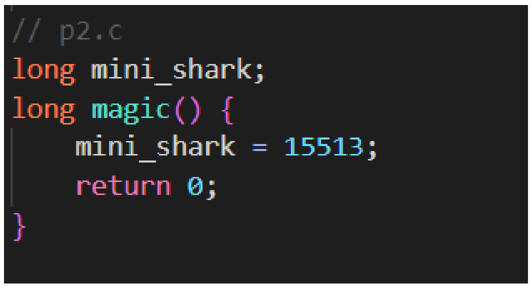
Question #3

Consider the following linker puzzles:

1. What might get printed in random() if we run the following two files? Briefly explain your answer.

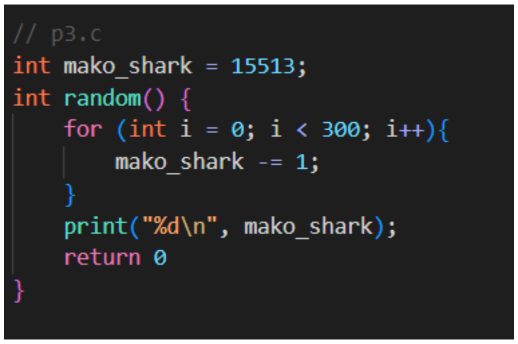


```c
// p1.c
int mini_shark = 15213;
int random() {
    int sum = 100 + mini_shark;
    print("%d\n", sum);
    return sum
```

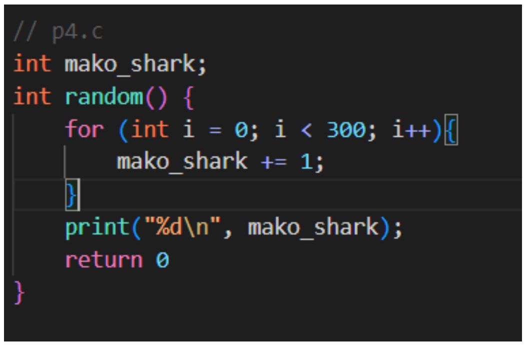


```c
// p2.c
long mini_shark;
long magic() {
    mini_shark = 15513;
    return 0;
}
```

2. What might happen if we run the following two files? We didn't specify the value of mako_shark before referencing it in p4.c, but the program won't raise a variable declaration error, why is this the case? Briefly explain your answer.

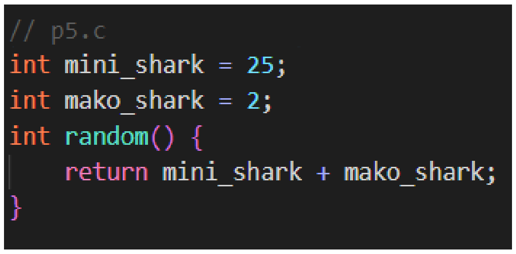


```c
// p3.c
int mako_shark = 15513;
int random() {
    for (int i = 0; i < 300; i++){
        mako_shark -= 1;
    }
    print("%d\n", mako_shark);
    return 0
}
```

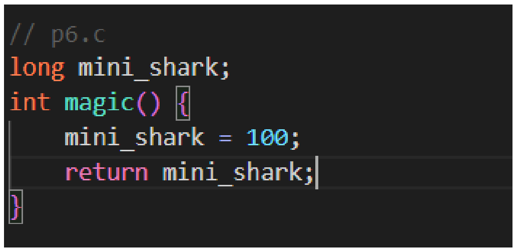


```c
// p4.c
int mako_shark;
int random() {
    for (int i = 0; i < 300; i++){
        mako_shark += 1;
    }
    print("%d\n", mako_shark);
    return 0
}
```

3. What might happen if we run the following two files? Briefly explain your answer. Note you don't have to state numerical values in this question, a general description of the program's behavior would suffice.



```c
// p5.c
int mini_shark = 25;
int mako_shark = 2;
int random() {
    return mini_shark + mako_shark;
}
```



```c
// p6.c
long mini_shark;
int magic() {
    mini_shark = 100;
    return mini_shark;
}
```

1. When there are weak and strong definitions, the linker chooses the strong definition which means p2.c's mini_shark is at the same memory location as p1.c's mini_shark so writing to mini_shark from p2.c will affect the value seen by p1.
   The number gets printed depends on which function gets called first. If random() gets called first, 15313 will be printed. If magic gets called first, the assignment to mini_shark will overwrite the original value of mini_shark and have the value 15513 (due to little endian format). Thus 15613 will get printed.

2. Explains that a link time error will be raised as we have two identical strong symbols (random).
   [If successfully compiled, references to mako_shark will refer to the same initialized variable in the .global section for both files.]

3. Explains that writing to mini_shark in p6.c might overwrite mako_shark in p5.c. The linker will assign the same place in the .global section for both files. Each file will treat that section of memory differently, either as an int or a long.