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.

Compiler optimization can improve the performance of a program by minimizing the number of instructions, avoiding waiting for memory, and avoiding branching.

For example,
1. Constant folding: `int num = 1 << 10;` is turned into `int num = 1024;`
2. Inlining:

```
bool larger(int a, int b) {
    return a > b;
}
void work(int* a, int* b) {
  if (larger(*a, *b)) {
      int tmp = *a;
      *a = *b;
      *b = tmp;
  }
}
```

is turned into

```
void work(int* a, int* b) {
  if (*a > *b) {
      int tmp = *a;
      *a = *b;
      *b = tmp;
  }
}
```

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?

The output is:

```
Condition 1
Condition 5
```

1. test() can be inlined
2. strlen() and x-=100 can be constant-folded
3. then the if branches can be eliminated
4. then x and s became unused values, so they can be eliminated as well

Then the entire program can be optimized into (ignoring the headers):

```
void main () {
   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.
mini_shark defined in p1.c is strong, while the one in p2.c is weak. So when two files are linked together, the definition in p1.c is chosen. Therefore, the mini_shark in both files refer to `long mini_shark = 15213`.
Then if magic() is not called before random(), then random() prints 15313.
Otherwise random() prints 15613.

2.
The linker will raise an error because there are two strong definition of random().

The mako_shark defined in p4.c is weak, the linker might make it refer to a strong definition in another source file.
A variable declaration error is not raised because C allows uninitialized global variables. Their symbols are stored in .bss section and will be stored in allocated memory when the program is loaded.

3.
The mini_shark in p5.c is a strong definition, while the one in p6.c is not.
Therefore, mini_shark symbol in p5.c refers to an integer with value 25.
The mini_shark symbol in p5.c refers to a long with value equal to 25 and 2 placed side by side in memory (0x200000019).

Of course, magic() prints 100 in any situation.
If magic() is called before, random() prints 100, since a long with value 100 in memory is an integer 100 and an integer 0 placed side by side.
Otherwise, random() prints 27.