Written Homework 1

Due: Monday 23rd January, 2017 by 9pm

Name: ________________________________

Andrew ID: ________________________________

Section: ________________________________

This written homework is the first of two homeworks that will introduce you to the way we reason about C0 code in 15-122. It also makes sure that you have a good understanding of key course policies.

Instructions

You can prepare your submission in one of two ways:

**Just edit (preferred)** Use any PDF editor (e.g., Preview on Mac, iAnnotate on mobile, Acrobat Pro on pretty much anything) to typeset your answers in the given spaces — you can even draw pictures. *That’s it.*

**Print and Scan** Alternatively, print this file, write your answers *neatly* by hand, and then scan it into a PDF file. *This is pretty labor-intensive.*

Once you have prepared your submission, submit it on Gradescope.

<table>
<thead>
<tr>
<th>Question</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points:</td>
<td>1</td>
<td>1.5</td>
<td>3</td>
<td>2.5</td>
<td>3.5</td>
<td>11.5</td>
</tr>
</tbody>
</table>

Score: ________________________________

---

1 Acrobat Pro is installed on all non-CS cluster machines.
1. **Running C0 programs**

The file `bar.c0` contains a function `bar` that takes an integer argument and returns an integer. Additionally, there is a `bar-test.c0` file that contains the following:

```c
int main() {
    return bar(230117);
}
```

(a) From the command line, show how to display the value returned by `bar(230117)` using the C0 compiler.

(b) From the command line, show how to display the value returned by `bar(230117)` using the C0 interpreter.
2. Policies

(a) **Collaboration Policy**

Read the collaboration policy on the course website. For each of the following situations, decide whether or not the students’ actions are permitted by the policy. Explain your answers.

1. Tom is having a hard time installing C0 on his dorm computer. The deadline for homework 1 is in less an hour. He Skypes his friend Hyrum who walks him through the installation process. With 15 minutes to go, they get cc0 working and Tom manages to turn in 3 out of 4 exercises.

<table>
<thead>
<tr>
<th>Tom's actions</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skyping with Hyrum to install C0</td>
<td>Permitted by policy</td>
</tr>
<tr>
<td>Turned in 3 out of 4 exercises</td>
<td>Permitted by policy</td>
</tr>
</tbody>
</table>

2. Iliano is having a really hard time with 122. So much so that he has hired an expert tutor, Rob, who goes over the lecture notes and the assignment writeups with him, nearly line by line. As a result, he manage to get 74% in the first homework.

<table>
<thead>
<tr>
<th>Iliano's actions</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hired expert tutor, Rob</td>
<td>Not permitted by policy</td>
</tr>
<tr>
<td>Got 74% in first homework</td>
<td>Permitted by policy</td>
</tr>
</tbody>
</table>

3. Frank and Penny write out a solution to Problem 2 on a whiteboard in GHC. Then they erase the whiteboard and go to class. In the evening, sitting at opposite sides of a computer cluster, each student types up the solution.

<table>
<thead>
<tr>
<th>Frank and Penny's actions</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write out solution on whiteboard</td>
<td>Permitted by policy</td>
</tr>
<tr>
<td>Erase whiteboard</td>
<td>Not permitted by policy</td>
</tr>
<tr>
<td>Type up solution</td>
<td>Permitted by policy</td>
</tr>
</tbody>
</table>

4. Andre is working on a problem alone on a whiteboard in the GHC commons. He accidentally forgets to erase his solution and goes to a cluster to write it up. Later, Tom walks by, reads Andre’s notes, and then writes the solution when he gets home a few hours later. Is Tom in violation of the policy? Is Andre?

<table>
<thead>
<tr>
<th>Tom and Andre's actions</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes solution in cluster</td>
<td>Not permitted by policy</td>
</tr>
<tr>
<td>Reads Andre's notes</td>
<td>Not permitted by policy</td>
</tr>
</tbody>
</table>
(b) **MOSS**

The course relies on a software service called MOSS to check for plagiarized code. What does MOSS do exactly? A quick online search will bring up examples of MOSS output, possibly for languages other than C/C++. If a student is pressed for time, what is likely to be least time consuming: writing his/her own code or “borrowing” code from a friend and modifying it in such a way that MOSS won’t flag them as similar? Explain.

---

(c) **Academic Integrity Contract**

Now that you had a chance to reflect on the collaboration policy of the course, we ask you to complete and sign the contract on the next page. By doing this, you declare that you understand the course policy on academic integrity and commit to abide by it. Like any contract, read it carefully. Please reach out to the course staff if you have any questions.

Although this task is worth 0 points, failure to complete and sign the contract will carry a penalty of **-500 points**, i.e., guaranteed failure in the course.
The value of your degree depends on the academic integrity of yourself and your peers in each of your classes. It is expected that, unless otherwise instructed, the work you submit as your own will be your own work and not someone else’s work or a collaboration between yourself and other(s).

Please read the University Policy on Academic Integrity carefully to understand the penalties associated with academic dishonesty at Carnegie Mellon. In this class, cheating/copying/plagiarism means copying all or part of a program or homework solution from another student or unauthorized source such as the Internet, having someone else do a homework or take an exam for you, knowingly giving such information to another student, or giving or receiving unauthorized information during an examination. In general, each solution you submit (quiz, written assignment, programming assignment, midterm or final exam) must be your own work. In the event that you use information written by another person in your solution, you must cite the source of this information (and receive prior permission if unsure whether this is permitted). It is considered cheating to compare complete or partial answers, discuss details of solutions, read other students’ code or show your code to other students, or sit near another person who is taking the same course and try to complete the assignment together.

It is a violation of this policy to hand in work for other students.

Your course instructor reserves the right to determine an appropriate penalty based on the violation of academic dishonesty that occurs. Penalties are severe: a typical violation of the university policy results in the student failing this course, but may go all the way to expulsion from Carnegie Mellon University. If you have any questions about this policy and any work you are doing in the course, please feel free to contact your instructor for help.

We will be using the Moss system to detect software plagiarism.

It is not considered cheating to clarify vague points in the assignments, lectures, lecture notes, or to give help or receive help in using the computer systems, compilers, debuggers, profilers, or other facilities, but you must refrain from looking at other students’ code while you are getting or receiving help for these tools. It is not cheating to review graded assignments or exams with students in the same class as you, but it is considered unauthorized assistance to share these materials between different iterations of the course. Do not post code from this course publicly (e.g., to Bitbucket or GitHub).

By signing below, I have read the statements above and have reviewed the course policy for cheating and plagiarism, and I will abide by these policies in this course.

Andrew ID ____________________________________________

Name (print) ________________________________ Section ____________

Signature ________________________________ Date ____________
3. Preconditions and postconditions

For the following functions, either check the box that says the postcondition always holds when the function is given inputs that satisfy its preconditions or give a concrete counterexample: specific values of the inputs such that the preconditions (if there is one) holds and the postcondition does not hold. You don’t have to write any proofs.

```c
int f1(int x, int y)
//@requires 0 <= x && x < y;
//@ensures \result >= 0;
{
    return x - y;
}
@ensures always true?  
x =  
y =  
```

```c
int f2(int x)
//@requires x % 2 == 0;
//@ensures x >= 0 || \result < x;
{
    return x / 2;
}
@ensures always true?  
x =  
```

```c
int f3(int x, int y)
//@requires y < 0;
//@ensures \result < -y ;
{
    return x % y;
}
@ensures always true?  
x =  
y =  
```

```c
int f4(int x, int y)
//@requires x + y == 42;
//@ensures \result - x == y;
{
    return 42;
}
@ensures always true?  
x =  
y =  
```

```c
int f5(int x, int y)
//@ensures \result > 0;
{
    if (x < 0) x = -x;
    if (y < 0) y = -y;
    if (y > x) {
        return y - x;
    } else {
        return x - y;
    }
}
@ensures always true?  
x =  
y =  
```

```c
int f6(int x, int y)
//@ensures \result <= 0;
{
    if (x <= 0) x = -x;
    if (y <= 0) y = -y;
    if (y >= x) {
        return y - x;
    } else {
        return x - y;
    }
}
@ensures always true?  
x =  
y =  
```
4. Thinking about loops

When we think about loops in 122, we will always concentrate on a single iteration of the loop. A loop will almost always modify something; the following loop modifies the local assignable `i`.

```java
while (i < n) {
    i = i + 4;
}
```

In order to reason about the loop, we have to think about the two different values stored in the local assignable `i`.

We use the variable `i` to talk about the value stored in the local `i` before the loop runs (before the loop guard is checked for the first time).

We use the “primed” variable `i'` to talk about the value stored in the local `i` after the loop runs exactly one time (before the loop guard is next checked).

(a) Consider the following loop:

```java
while (i < n) {
    k = j + k;
    j = j * 2 + i;
    i = i + 1;
}
```

- If `i = 7`, `j = 3`, and `k = 9`, then assuming `7 < n`,
  
  `i' = \_\_\_\_\_\_\_\_` , `j' = \_\_\_\_\_\_\_\_` , and `k' = \_\_\_\_\_\_\_\_`

- If `i = 2y`, `j = x - y`, and `k = y`, then assuming `2y < n`, in terms of `x` and `y`,
  
  `i' = \_\_\_\_\_\_\_\_` , `j' = \_\_\_\_\_\_\_\_` , and `k' = \_\_\_\_\_\_\_\_`

- If `j = k`, then assuming `i < n`, in terms of `i` and `k`,
  
  `i' = \_\_\_\_\_\_\_\_` , `j' = \_\_\_\_\_\_\_\_` , and `k' = \_\_\_\_\_\_\_\_`

- In general, assuming `i < n`, then in terms of `i`, `j`, and `k`,
  
  `i' = \_\_\_\_\_\_\_\_` , `j' = \_\_\_\_\_\_\_\_` , and `k' = \_\_\_\_\_\_\_\_`

Note that we always say “assuming (something) < n,” because if that were not the case the loop wouldn’t run, and it wouldn’t make any sense to be talking about the values of the primed variables.
(b) Consider this loop:

```java
while (...) {
    i = i + 3;
    j = j * 2 + i;
    k = k + i - j;
}
```

Be careful, it looks similar but is trickier! Give simplified answers.

- If \(i = 7\), \(j = 3\), and \(k = 9\), then assuming the loop guard evaluates to true,
  \[i' = \underline{ }, \quad j' = \underline{ }, \quad \text{and} \quad k' = \underline{ }\]

- In general, assuming the loop guard evaluates to true, then in terms of \(i\), \(j\), and \(k\),
  \[i' = \underline{ }, \quad j' = \underline{ }, \quad \text{and} \quad k' = \underline{ }\]

(c) Consider this loop:

```java
while (a > 0 && b > 0) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
```

- If \(a = 94\) and \(b = 12\), then
  \[a' = \underline{ } \quad \text{and} \quad b' = \underline{ }\]

- If \(a = x + y\) and \(b = x\), where \(x\) and \(y\) are both positive integers, then
  \[a' = \underline{ } \quad \text{and} \quad b' = \underline{ }\]

- If \(a = x\) and \(b = x + z\), where \(x\) is a positive integer and \(z\) is a non-negative integer, then
  \[a' = \underline{ } \quad \text{and} \quad b' = \underline{ }\]

- If \(a > 0\) and \(b > 0\), one of the two cases above will always be the case. Therefore, we can conclude which of the following about the values stored in \(a\) and \(b\) after an arbitrary iteration of the loop? (Check all that apply)
  - \(a' \geq 0 \text{ and } b' \geq 0\)
  - \(a' > 0 \text{ and } b' \geq 0\)
  - \(a' \geq 0 \text{ and } b' > 0\)
  - \(a' > 0 \text{ and } b' > 0\)
5. Proving a function correct

In this question, we’ll do part of the proof of correctness for a function `compute_square` relative to a specification function `SQUARE`. We won’t prove that the loop invariants are true initially, and we won’t prove that they’re preserved by an arbitrary iteration of the loop.

```c
int compute_square(int n) {
    int total = 0;
    while (n > 0) {
        total += 2*n - 1;
        n--;
    }
    return total;
}
```

(a) Complete the specification function below with the simple mathematical formula that gives the square of the numbers `n`.

```c
int SQUARE(int n)
//@requires 0 <= n && n < 10000;
{
    return ________________________________;
}
```

Give a postcondition for `compute_square` using this specification function.

```c
int compute_square(int num)
//@requires 0 <= num && num < 10000;
//@ensures ________________________________;
{
    int n = num;
    int total = 0;
    while (n > 0)
        //@loop_invariant 0 <= n;
        //@loop_invariant n <= 10000;
        // Additional loop invariant will go here
        {
            total += 2*n - 1;
            n--;
        }
    return total;
}
```

Note: in the real world we wouldn’t have an efficient closed-form solution used as a specification function for an inefficient loop-based solution. We usually use the slow, simple version as the specification function for the fast one!
(b) Why was it necessary to introduce the new local `num` in the second version of `compute_square` above?

Give a suitable extra invariant that would allow us to prove the function correct.

```c
//@ loop_invariant ________________________________ ;
```

Which line numbers would we point to to justify that \( n == 0 \) when the loop terminates?

Substitute in 0 for \( n \) in your loop invariant on line 11 and then simplify.

When you substitute \( \text{result} \) for \( \text{total} \) in the simplified version, you should have exactly the postcondition on line 4. This proves that the loop invariant and the negation of the loop guard imply the postcondition.

(c) Termination arguments for loops (in this class, at least) must have the following form:

During an arbitrary iteration of the loop, the quantity \( ____ \) gets strictly larger, but from the loop invariants, we know this quantity can’t ever get bigger than \( ____ \).

or

During an arbitrary iteration of the loop, the quantity \( ____ \) gets strictly smaller, but from the loop invariants, we know this quantity can’t ever get smaller than \( ____ \).

Assuming that your loop invariants are true initially preserved by every iteration of the loop (which we didn’t prove), why does the loop in `compute_square` terminate?

During an arbitrary iteration of the loop, the quantity \( _____ \) gets strictly \( _____ \) but from the loop invariants, we know that this quantity can’t ever get \( _____ \) than \( _____ \).