15-122: Principles of Imperative Computation, Spring 2020

Written Homework 1

Due on Gradescope: Monday 20th January, 2020 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.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***pdfescape*** or ***dochub***, two web-based PDF editors that work from anywhere.
- ***Preview***, the Mac's PDF viewer.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though*.

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	4	5	6	Total
Points:	1	1	1.5	2	2.5	4	12
Score:							

1pt 1. **Pretest**

Please complete the pretest at `http://cs.cmu.edu/~15122/misc/pretest.html` to the best of your abilities. Your score will be based uniquely on attempting the questions (you are not expected to know the answers to all questions).

2. **Running C0 Programs**

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

```
int main() {
    return num(20200120);
}
```

0.5pts **2.1** From the command line, show how to display the value returned by `num(20200120)` using the C0 compiler.

0.5pts **2.2** From the command line, show how to display the value returned by `num(20200120)` using the C0 interpreter.

3. **Policies**

1pt

3.1 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.

(a) 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.

(b) Stephanie and Anna are working on problem 5 on a whiteboard in the 9th floor of GHC. An hour later, they have come up with a complex solution. They go their separate ways but Anna takes a picture so that she can improve the solution. The next day, both type up their solution.

(c) Andre is working on a problem alone on a whiteboard in the GHC commons. He 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?

(d) 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.

(e) Jean realizes that problem 2 is very similar to when she took 15-122 last time, a semester ago. She retrieves her old solution, fixes the one issue where she lost points, and submits.

(f) Dilsun is really stuck on problem 3, and the deadline is looming. She texts Pat, who took 15-122 two semesters ago, for help. Pat texts her back highlights of his solution. She changes the wording and submits. Is Dilsun in violation of the policy? Is Pat?

0.5pts

3.2 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/C0. 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.

0pts

3.3 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 tasks 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.

15–122 — Principles of Imperative Computation, Spring 2020

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 carefully the academic integrity policy of this course and 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, giving such information to another student, having someone else do a homework or take an exam for you, reusing answers or solutions from previous editions of the course, 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, copy or adapt others' solutions, read other students' code or show your code to other students, or sit near another person who is taking the same course and complete an assignment together. Writing code for others to see (e.g., on a whiteboard) is never permitted. It is also considered cheating for repeating students to reuse their solutions from a previous semester, or any instructor-provided sample solution.

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

Your course instructors reserve 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 instructors for help.

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

By checking the second box below, you commit to performing a chicken dance in front of the TAs at office hours. Most people do not check this box.

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).**

☐ I have read the statements above and reviewed the course policy for cheating and plagiarism.

☐ I agree to the clause in paragraph 6.

By signing below, I commit to abiding by these policies in this course.

Andrew ID _____

Name (print) _____ Section _____

Signature _____ Date _____

2pts 4. **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 are any) holds and the postcondition does not hold. You don't have to write any proofs.

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

`@ensures` always true? ☐

x = ☐ y = ☐

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

`@ensures` always true? ☐

x = ☐

```
int f3(int x, int y)
//@requires y < -1;
//@ensures \result > y ;
{
  return x % y;
}
```

`@ensures` always true? ☐

x = ☐ y = ☐

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

`@ensures` always true? ☐

x = ☐ y = ☐

```
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 = ☐

```
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 = ☐

5. **Thinking about Loops**

When we think about loops in 15-122, 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`.

```
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` during an iteration.

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 more time (before the loop guard is next checked).

1pt **5.1** Consider the following loop:

```
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.

0.5pts **5.2** Consider this loop:

```
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,

  $i' =$ [ ], $j' =$ [ ], and $k' =$ [ ]

- In general, assuming the loop guard evaluates to true, then in terms of i, j, and k,

  $i' =$ [ ] $j' =$ [ ], and $k' =$ [ ],

1pt **5.3** Consider this loop:

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

- If $a = 94$ and $b = 12$, then

  $a' =$ [ ] and $b' =$ [ ]

- If $a = x + y$ and $b = x$, where x and y are both positive integers, then

  $a' =$ [ ] and $b' =$ [ ]

- If $a = x$ and $b = x + z$, where x is a positive integer and z is a non-negative integer, then

  $a' =$ [ ] and $b' =$ [ ]

- 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)

  ☐ $a' \geq 0$ and $b' \geq 0$

- ☐ $a' > 0$ and $b' \geq 0$
- ☐ $a' \geq 0$ and $b' > 0$
- ☐ $a' > 0$ and $b' > 0$

6. **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.

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

1pt **6.1** Complete the specification function below with the simple mathematical formula that gives the square of the numbers n.

```
1  int SQUARE(int n)
2  //@requires 0 <= n && n < 15122;
3  {
4    return ____________________________________________________;
5  }
```

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

```
7  int compute_square(int num)
8  //@requires 0 <= num && num < 15122;
9
10 //@ensures ____________________________________________________;
11 {
12   int n = num;
13   int res = 0;
14   while (n > 0)
15   //@loop_invariant 0 <= n;
16   //@loop_invariant n <= 15122;
17   // Additional loop invariant will go here
18   {
19     res += 2*n - 1;
20     n--;
21   }
22   return res;
23 }
```

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!

2pts **6.2** Why was it necessary to introduce the new local `n` in the second version of `compute_square` above?

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

```
17 //@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 17 and then simplify.

When you substitute `\result` for `res` in the simplified version, you should have exactly the postcondition on line 10. This proves that the loop invariant and the negation of the loop guard imply the postcondition.

1pt **6.3** 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 and are 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_____.