This written homework is the first of two homeworks that will introduce you to the way we reason about C0 code in 15-122.

Print out this PDF double-sided, staple pages in order, and write your answers *neatly* by hand in the spaces provided.

The assignment is due by 1:30pm on Monday 18th January, 2016.

You must hand in the assignment at the beginning of the lecture.

You must hand in your homework yourself;
do not give it to someone else to hand in.
<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
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:

```c0
int main() {
    return bar(15122);
}
```

(a) From the command line, show how to display the value returned by `bar(15122)` using the C0 compiler.

Solution:

(b) From the command line, show how to display the value returned by `bar(15122)` using the C0 interpreter.

Solution:
2. 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) {
    return y - x;
}
```

@ensures always true: □

```c
int f2(int x) {
    return x / 2;
}
```

@ensures always true: □

```c
int f3(int x, int y) {
    return x % y;
}
```

@ensures always true: □

```c
int f4(int x, int y) {
    return 5;
}
```

@ensures always true: □

```c
int f5(int x, int y) {
    if (y < x) {
        return y - x;
    } else {
        return x - y;
    }
}
```

@ensures always true: □

```c
int f6(int x, int y) {
    if (y <= x) {
        return y - x;
    } else {
        return x - y;
    }
}
```

@ensures always true: □
3. 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.

```cpp
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:

```cpp
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 then 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:

```c
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,
  \[i' = \underline{\text{}}\quad j' = \underline{\text{}}\quad \text{and} \quad k' = \underline{\text{}}\]

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

(c) Consider this loop:

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

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

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

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

- 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 \(b' > 0\)
4. Proving a function correct

In this question, we’ll do part of the proof of correctness for a `compute_sum` relative to a specification function `SUM`. 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_sum(int n) {
    int total = 0;
    while (n > 0) {
        total = total + n;
        n = n - 1;
    }
    return total;
}
```

(a) Complete the specification function below with the simple mathematical formula that gives the sum of numbers from 0 to n.

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

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

```c
int compute_sum(int num_ints)
//@requires 0 <= num_ints && num_ints <= 10000;
//@ensures ________________________________;
{
    int n = num_ints;
    int total = 0;
    while (n > 0)
        //@loop_invariant 0 <= n;
        //@loop_invariant n <= 10000;
        // Additional loop invariant will go here
    {
        total = total + n;
        n = n - 1;
    }
    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 add the new local `num_ints` in the second version of `compute_sum` above?

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

```cpp
//@
@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 15 and then simplify.

When you substitute `\texttt{result}` for `total` in the simplified version, you should have exactly the postcondition on line 8. 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_sum` terminate?

During an arbitrary iteration of the loop, the quantity

but from the loop invariants, we know that this quantity can’t ever get