Administrivia and general advice

Welcome to 15-122 recitation! Take a moment to fill in the particulars for this section, so that you know which section you’re in as well as your TAs name and email.

Section: __________________________________________________________
TA Names: ________________________________________________________
TA Emails: _________________________________________________________

Checkpoint 0

Identify and correct the syntax errors in the following code to make it valid C0:

```c0
#include conio

int fib(int i):
    if (((i == 0) || (i == 1)) == true) {
        return i;
    }
    return fib(i - 1) + fib(i - 2)

int main():
    print("This is the fibonacci sequence starting from: 0\n");
    for (int i=0, i < 10, i++) {
        printf("fib(%d) = %d\n", fib(i));
    }
```

**Printing:** The simplest way to print something in C0 is to use `printf`. For example,

```c0
printf("Today is %d %s %d!\n", day, month, year);
```

The first argument is the string that you wish to print, optionally containing *format specifiers* among `%d`, `%s` and `%c`. During printing these format specifiers will be replaced by the values of the subsequent arguments, in the order (here the variables `day`, `month` and `year`). You use `%d` to print an `int` value, `%s` for a `string`, and `%c` for a `char`. Printing is *buffered* in C0, which means that output will be displayed on the terminal only when a *new line character* (written `\n`) is encountered. See the documentation for the `<conio>` library for more on printing in C0.

Although `printf` is most commonly used, C0 provides other printing functions, `print` to print a string, `printint` to print an integer, etc. See the documentation for the `<conio>` library for a complete list.
Executing C0 Programs

There are two ways to execute a C0 program: run it through the interpreter or compile it. Using coin (the C0 interpreter) allows you to execute individual functions with your chosen inputs interactively. The cc0 compiler for C0 first translates your C0 code into an executable file (the default name is a.out), which you can then run to execute the main function (and any function main calls). There must be exactly one main function when using cc0, but no main is needed for coin.

Consider the following example where our program has been split into three files. What it does isn’t important: what matters is that dotproduct.c0 calls a function in multiply.c0, which itself calls a function in add.c0.

<table>
<thead>
<tr>
<th>File add.c0</th>
<th>File multiply.c0</th>
<th>File dotproduct.c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>int add(int x, int y) { return x + y; }</td>
<td>int multiply(int x, int y) { //@requires y &gt;= 0;</td>
<td>int dot_product(int x, int y, int a, int b) {</td>
</tr>
<tr>
<td></td>
<td>{ int result = 0; while (y &gt; 0) { result = add(result, x); y--;</td>
<td>int first = multiply(x, a); int second = multiply(y, b);</td>
</tr>
<tr>
<td></td>
<td>} return result; }</td>
<td>return add(first, second); }</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>int main() {</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>return dot_product(150, 61, 100, 2); }</td>
</tr>
</tbody>
</table>

How would we compile this program with contract checking enabled?

```
% 
```

How would you run the resulting executable?

```
% 
```

How would you compile this program without contract checking enabled so that the executable is called dptest? How would you then run it?

```
% 
```

How would you use coin to check that multiply(3,2) evaluates to 6? What about using coin to find out what multiply(-3,-2) produces? You may want to enable contracts.

```
% 
```

C0 interpreter (coin) 1.0.0 'Quarter' ...

```
and coin support many more flags. A particularly useful one in coin is -l which loads a system library. For example to load <conio> so that we can use printf, we would invoke it as

\%
coin -l conio
C0 interpreter (coin) 1.0.0 ‘Quarter’ ...
--> printf("hi!\n");
hi

Contracts

There are 4 types of contract annotations in C0 (for convenience, we’re using exp here to mean any Boolean expression):

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>//@requires exp;</td>
<td>before function execution</td>
</tr>
<tr>
<td>//@ensures exp;</td>
<td>before function returns</td>
</tr>
<tr>
<td>//@assert exp;</td>
<td>wherever you put it in the code</td>
</tr>
<tr>
<td>//@loop_invariant exp;</td>
<td>before the loop condition is checked</td>
</tr>
</tbody>
</table>

There are certain special variables and functions you have access to only in annotations. One of these is $result$. It can be used only in @ensures statements and it will give you the return value of the function. (There are other such variables/functions that we’ll get to later in the semester.)

To help you develop an intuition about contracts, here are some explanations of the different kinds of annotations:

- **@requires**: For checking

- **@ensures**: For checking

  Allow use of the special expression

- **@loop.invariant**: We can only write these immediately after the beginning of a while loop or for loop.

  When are these checked?

- **@assert**: Assertion statements don’t play the special role in reasoning that @requires, @ensures, and @loop.invariant statements do. They can be very helpful for debugging code and summarizing what you know, especially after a loop.

Proving the correctness of the mystery function

We use contracts to both test our code and to reason about code. With contracts, careful point-to reasoning and good testing both help us to be confident that our code is correct.

The code for the mystery function from lecture is reproduced on the left below. To its right is an annotated control flow diagram that highlights the different parts in in what sequence they are executed.
Proving that this function is correct is divided into four steps. For each of them, give a brief intuitive explanation of what it achieves (“what”), determine the proof technique you are expected to use to prove it, write a precise statement of what you need to prove (“to show”), and list the lines of code you are allowed to use to prove this statement (“allowed lines”).

INITialization:

What: show that the loop invariants hold before the first iteration of the loop
Proof technique: point-to reasoning
To show: 
Allowed lines: 

PREServation:

What: show that each loop invariant is preserved by an arbitrary iteration of the loop
Proof technique: point-to reasoning
To show: 
Allowed lines: 

EXIT:

What: show that the postconditions follow from the loop invariants and the negation of the loop guard
Proof technique: point-to reasoning
To show: 
Allowed lines: 

TERMination:

What: show that the loop terminates
Proof technique: operational reasoning (although point-to reasoning is also possible)
To show: 
Allowed lines: 

```c
int f(int x, int y)
//@requires y >= 0;
//@ensures POW(x, y) == \result;
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 0)
        //@loop_invariant e >= 0;
        //@loop_invariant POW(b, e) * r == POW(x, y);
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    return r;
}
```
Having done this, express the allowed justification lines in terms of the parts of the control flow diagram — all proofs you will do in this class will follow this more general template.

We often think of INIT and PRES as part of a single larger step that shows that the loop invariants are valid.

**Preservation of loop invariants**

Showing that a loop invariant is preserved can be a bit confusing: we need to assume that the loop invariant, like \( e \geq 0 \) or \( r \cdot \text{POW}(b, e) = \text{POW}(x, y) \), is true just before we start executing the body of the loop (it is checked just before the loop guard), and use this information (together with the knowledge that the loop guard evaluated to \text{true}) to show that it is also true at the end of the loop (just before the loop guard is evaluated again). We do this relative to an arbitrary iteration of the loop. Here's a different loop body from what we saw in class.

```c
while (e > 0)
    //@ loop_invariant e >= 0;
    //@ loop_invariant r * \text{POW}(b, e) = \text{POW}(x, y);
    {
        r = r * b;
        e = e - 1;
    }
```

Let's show that both loop invariants are preserved by an arbitrary iteration of this loop. We will express the various steps of these proofs using mathematical symbols (e.g., \( e \geq 0 \)) but you are welcome to use code (e.g., \( e \geq 0 \)).

Recall that we use primed variables (e.g., \( e' \)) to denote that value of this variable after the current iteration has been completed, while we omit the prime (e.g., \( e \)) for its value at the start of the current iteration.

**Loop invariant on line 2 \((e \geq 0)\):**

**Assume:**

**To show:**

a. \( e' = \) by 
b. 

c. 

d. 

**Loop invariant on line 3 \((rb^e = x^y)\):**

**Assume:**

**To show:**

a. \( rb^e = x^y \) by assumption (or by line 3)

b. \( r' = \) by 

c. \( e' = \) by 

d. \( b' = \) by 

e. 

f. 

g. 

REFERENCE: Basic syntax for C0 programs

Semicolons: Statements are terminated by semicolons. At the end of most lines, you’ll need a semicolon. (Exceptions are if statements, function definitions, #use statements, and loops.)

Types: Some of the types in C0 are:

- **int**: Integers \( x \), where \(-2^{31} \leq x < 2^{31}\).
- **bool**: Either true or false. Useful for conditionals, loops, and more.
  - \( a \ || \ b \) is true if either \( a \) or \( b \) are true
  - \( a \ &\& \ b \) is true if both \( a \) and \( b \) are true
  - \( !a \) is true if \( a \) is false
  - \( x < y, x <= y, x > y, x >= y \) all compare two integers \( x \) and \( y \) and return a bool
  - \( x == y \) and \( x != y \) compare most C0 types and return a bool. You can’t compare strings this way, though, and it’s usually bad style if you’re writing code like \( e == true \).

- **string**: An ordered sequence of characters enclosed in double quotes like "Hello!

- **char**: A single character enclosed in single quotes, like 'c', 'z', 'F', or '?'

**Boolean operators**: Both &\& and || are short-circuiting infix operators.

They are *infix* (like other operators +, -, %, etc.) because they take two arguments and the operator is placed between the two arguments. The compiler mentions the word "infix operator" if you make a mistake with them, so it’s good to be aware of this name for them.

They are *short-circuiting* because, if the expression to the left of &\& evaluates to false or if the expression to the left of || evaluates to true, then the expression to the right will never get executed. This means that, even though evaluating the expression \( y/x == 0 \) will cause an error if \( x \) is zero, evaluating the expression \( x == 0 \ || \ y/x == 0 \) can never cause an error.

**Locals**: Locals (also called “local variables,” “assignable variables,” “assignables,” or “variables”) are explicitly declared along with their type. Locals can never change type after they are declared.

```c
int x = 5; // x is initialized to 5
int y; // We can’t use y until we assign to it!
string str = "hello";
y = x + 4; // y is now equal to 9
x = x + 1; // x is now equal to 6
x = "world"; // ERROR! string and int are different types!
```

**Conditionals**: These are one way we use bool values. Here’s an example of if statements in C0:

```c
def (condition) {
    //do something if condition == true
}
def else if (condition2) {
    //do something if condition2 == true and condition == false
}
def else {
    //do something if condition == false and condition2 == false
}
```
Loops: There are two kinds of loops in C0 — **while** loops and **for** loops.

- **while** loop: It takes a condition (something that evaluates to a Boolean). The loop executes until the condition is false.

- **for** loop: It takes three statements separated by semicolons. Execute the first statement once at the beginning of the loop, loop until the second statement (a condition) is false, and execute the third statement at the end of each iteration.

```plaintext
<table>
<thead>
<tr>
<th>while loop</th>
<th>for loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 0;</td>
<td>for (int x = 0; x &lt; 5; x++) {</td>
</tr>
<tr>
<td>while (x &lt; 5) {</td>
<td>printf(&quot;%d\n&quot;, x);</td>
</tr>
<tr>
<td>printf(&quot;%d\n&quot;, x);</td>
<td>}</td>
</tr>
<tr>
<td>x++;</td>
<td>}</td>
</tr>
</tbody>
</table>
```

These two examples do the same thing. Here, the **for** loop is preferred but there are cases (like binary search in an array, which we’ll discuss later this semester) where **while** loops are cleaner.

**Function definition:** This example defines a function called **add** that takes two **int**s as arguments and returns an **int**.

```plaintext
int add (int x, int y) {          
    return x + y;                 
}
```

**Comments:** Use `//` to start a single line comment and `/* ... */` for multi-line comments.

**Indentation and braces:** Your code will still work if it’s not indented well, but it’s really bad style to indent poorly. Python’s indentation rules are good and you should generally follow them in C0 too. C0 uses curly braces (i.e., `{ and `} to denote the starts and ends of blocks, as seen above. For single-line blocks it’s possible to omit the curly braces, but that can make debugging very difficult if you later add in another line to the block of code. For that reason, you may want to use braces, even for single-line statements.

<table>
<thead>
<tr>
<th>Very Bad</th>
<th>Okay</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (x == 4)</td>
<td>if (x == 4)</td>
<td>if (x == 4) {</td>
</tr>
<tr>
<td>printf(&quot;x is 4&quot;);</td>
<td>printf(&quot;x is 4&quot;);</td>
<td>printf(&quot;x is 4&quot;);</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

You can find more information about the C0 syntax on the C0 wiki.