15-122: Principles of Imperative Computation

Recitation 1 Josh Zimmerman, Nivedita Chopra, Manu Garg

Administrivia and general advice

Welcome to 15-122 recitation! I'm Manu Garg and you can email me at manug@andrew.cmu.edu. Before we get started, I want to give you some information that'll be useful to you, as well as some advice.

I have office hours, which I encourage you to go to if you're having any trouble with the material. The sooner you ask questions, the sooner I can help you. (Everyone else on course staff has office hours too. You can find those, and their contact information, on the course website.)

If at any point you don't understand something I say, *ask questions!* If I explained something in a way that made you confused, you're almost certainly not the only one, and I want to clarify what I said.

There's a lot of useful reference material about C0 at http://c0.typesafety.net/. If you have any questions about anything related to C0 , it's an excellent first resource.

- Office hours: Tues, Thurs 4.30 - 6.30 GHC 4215
- ASK ALL THE QUESTIONS! 
- http://c0.typesafety.net/ is useful!

Basic syntax

Semicolons: statements are terminated by semicolons. What this means is that at the end of most lines, you'll need a semicolon. (exceptions are if statements, function definitions, `use` statements, and loops.)

Variables: variables must be explicitly declared and all variables have a type. Variables can never change type after they are declared. Some of the types in C0 are:

- `int`: whole numbers x where $-2^{31} \leq x < 2^{31}$
- `bool`: Either `true` or `false`. Useful for conditionals, loops, and more.
- `string`: An ordered sequence of characters like "Hello!"
- `char`: A single character, like 'c'
- `t[]`: An array with elements of type `t`. Arrays are declared with `alloc_array`: `alloc_array(int, 10)` will make an array that can hold 10 `int`s. This is a big distinction from Python and other languages: arrays have fixed size, so you need to know how long your array will be at the time you declare it.

Conditionals: It's an error to put something that isn't a `bool` in a conditional. Note that `a || b` is true if either `a` or `b` are true (and false otherwise), and `a && b` is true if both `a` and `b` are true (and false otherwise). These are called *infix* operators. The compiler mentions them if you make a mistake with them, so it's good to be aware of this name for them.

Here's an example of `if` statements in C0 :

```
if (condition) {
   //do something if condition == true
}
else if (condition2) {
   //do something if condition2 == true (and condition == false)
}
else {
   //do something if condition == false and condition2 == false
}
```

Loops: There are two kinds of loops in C0 : while loops and for loops. While loops execute the loop until the condition they're given is false. For loops execute the first statement they're given once, loop until the second statement is false, and execute the last statement at the end of each iteration. Both these loops are entry controlled as opposed to do while loops (c0 doesn't support do while loops) which are exit controlled. These two examples given below do the same thing. Here, the for loop is preferable but there are cases (like binary search in an array, which we'll discuss later this semester) where while loops are cleaner.

While loop

```
int x = 0;
while (x < 5) {
   printint(x);
   print("\n");
   x++;
}
```

For loop

```
for (int x = 0; x < 5; x++) {
   printint(x);
   print("\n");
}
```

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

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

Comments: use // to start a single line comment and /* ...*/ for multi-line comments. It's good style to have a * at the beginning of each line in a multi-line comment.

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 ({ 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, I *highly* encourage you to always use braces, even for single-line statements.

Bad

```
if (x == 4)
   println("x is 4");
```

Good

```
if(x == 4) {
   println("x is 4");
}
```

Another important note about indentation is that you should choose either tabs or spaces and stay consistent, since mixing styles makes your code unreadable if someone views your code with a different number of spaces per tab.

Fix syntax my!

Now, ssh in to unix.andrew.cmu.edu and run these commands (don't forget the "." at the end of the cp):

```
$ cd private
$ mkdir -p 122
$ cd 122
$ cp /afs/andrew.cmu.edu/usr5/jzimmerm/public/badSyntax.c0 .
```

If you're using the csh shell (which is the default, you should run the following command, which allows you easy access to tools you'll need to use for the course. (If you're using bash, you should first run the command csh, run exit after you're done, and then log out and log back in from the andrew machine.)

```
$ source /afs/andrew.cmu.edu/course/15/122/bin/setup-c0.csh
```

Then, use either emacs or vim (I personally prefer vim) and the C0 compiler (cc0) to correct the syntax errors in that file.

Compile using the following command.

```
$ cc0 badSyntax.c0 -o badSyntax
```

In this command, cc0 refers to the C0 compiler. Next comes the name of the file in which we've written our program: badSyntax.c0. This file is called the *source code*. Then we can specify the file that we want to store the compiler's output in using the -o option. (This file is called an *executable*). Here this file is badSyntax. If the -o option is absent, the compiler's output is stored in a.out.

Then, when cc0 no longer gives errors, run with the following command. (in general, to execute a file, you need to either put it in a "special" location like /bin or to specify a path to that file. In this case, the file is in the current working directory so we prepend ./.

```
$ ./badSyntax
```

When you're done, your compiled version should output:

```
0
1
1
2
3
5
8
13
21
```

```
34
0
```

Checkpoint
If my first command was `cc0 badSyntax.c0` what command would I use to run the executable?

Contracts

There are 4 types of annotations in C0 (note: for convenience, I use `exp` to mean any boolean expression):

Annotation	Checked...
`//@requires exp;`	before function execution
`//@ensures exp;`	before function returns
`//@loop_invariant exp;`	before the loop condition is checked
`//@assert exp;`	wherever you put it in the code

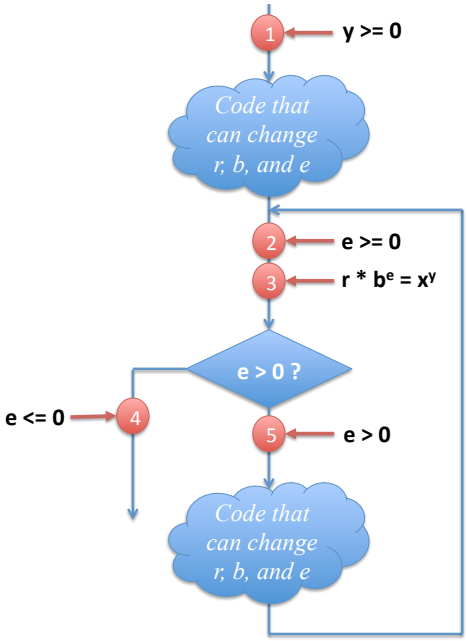
There are certain special variables and functions you have access to only in annotations. One of these is `\result` . In `//@ensures` statements, it will give you the return value of the function. (There are others that we'll get to later in the semester.)

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

- `//@requires` : Something that the *caller* needs to make sure is true before calling the function. `//@requires` statements are used to make sure that users of the function use it in ways that make sense. For instance, if you were writing a factorial function it wouldn't make sense to ask for the factorial of a negative number, so you might say `//@requires n >= 0;` as a precondition of your function. Using a `//@requires` statement allows you to clearly express how a function you write is used. If someone calls your function and violates a `//@requires` statement, anything can happen and it's their fault, not yours. (You warned them!)
- `//@ensures` : If the caller satisfies all requires statements, the function *must* make all `//@ensures` statements true. `//@ensures` statements are useful because they allow users of functions you write to make assumptions about your function's behavior.
- `//@loop_invariant` : Loop invariants are very useful when trying to verify that a function is correct. A loop invariant should directly imply the postcondition in most cases (the exception being when your function does something after the end of the loop). If your loop invariant doesn't directly imply the postcondition, you should strengthen it until it does or figure out why you can't strengthen it enough and fix any bug in your function that is stopping you from strengthening it.
- `//@assert` : Assert statements are useful if at some point in your function you want to be sure that a certain condition holds. This can be useful to help you debug part of a loop (for example, if the loop invariant doesn't work, assert statements might help you find out why) and also in cases where you do work after the end of your loop (to help you prove the postcondition).

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

Here's a different way of looking at our mystery function. Once we have loop invariants for the mystery function set, we can view the whole thing as a control flow diagram:



The circle labeled **1** is a _____ of the function, and the circles labeled **2** and **3** are _____. The circles labeled **4** and **5** just capture information we get from the result of the loop guard (or loop condition), but we might write **4** as an `//@assert` statement.

To prove this function correct, we need to reason about the two pieces of code (pieces that this diagram hides in the two cloud-bubbles) to ensure that our contracts never fail:

- When we reason about the upper code bubble, we assume that _____ is true before the code runs and show that _____ are true afterwards.
- When we reason about the lower code bubble, we assume _____ are true before the code runs and show that _____ are true afterwards.
- To reason that the returned value r is equal to x^y, we combine the information from circles _____ to conclude that $e = 0$. Together with the information in circle _____, this implies that $r = x^y$.

In addition, we have to reason about termination: every time the lower code bubble runs, the value e gets strictly smaller.

Greatest common divisor

Let's take a different look at contracts, proofs, and tests. Imagine we're given a function that we're told gives us the greatest common divisor of two numbers.

```
1 int gcd(int x, int y)
2 //@requires x > 0 && y > 0;
3 //@ensures \result > 0 && x % \result == 0 && y % \result == 0;
```

This isn't a great contract – it doesn't require the result to be the *greatest* common divisor of two numbers, just that it be some divisor. If we don't have access to the implementation of this function, the best we can do is *test* it. We're looking for two kinds of errors:

- Cases where the contracts don't hold: given positive integers, the function returns a quantity that isn't a positive divisor. Call these *contract failures*.
- Cases where the answer was wrong even though the contract was right. Call these *contract exploits*.

To test for contract failures, we just have to run the gcd function on some good test cases. To test for contract exploits, we can use the `assert(exp)` statement to enforce that we're actually calculating greatest common divisors. `assert(exp)` is like the contract `//@assert exp`, but it is checked whether or not `-d` is on. As a result, we use `assert(exp)` mostly for writing tests.

```
#use <util>
#use <conio>

int main() {
   assert(gcd(42, 4) == 2); //random
   printint(gcd(13,5)); print("\n");
   assert(gcd(13, 5) == 1); //primes
   assert(gcd(1, 7) == 1); //input 1
   assert(gcd(12, 48) == 12); //multiple inputs
   assert(gcd(12, 12) == 12); //same number
   assert(gcd(int_max(), 2) == 1); //int_max
   assert(gcd(int_max(), int_max()) == int_max()); //both int_max

   println("All tests passed!");
   return 0;
}
```

Now we've that we've tested this implementation a bit, maybe we're a little bit more confident that it's correct. But maybe it's too slow, or maybe we're just nervous that we can't see and reason about the correctness of this code. We can instead use this secret implementation of the greatest common divisor as a specification and write our own implementation:

```
1  int fast_gcd(int x, int y)
2  //@requires x > 0 && y > 0;
3  //@ensures \result == gcd(x, y);
4  {
5     int a = x;
6     int b = y;
7     while (a != b)
8     //@loop_invariant a > 0 && b > 0;
9     //@loop_invariant gcd(a, b) == gcd(x, y);
10    {
11       if (a > b) {
12          a = a - b;
13       }
14       else {
15          b = b - a;
16       }
17    }
18    return a;
19 }
```

But does it actually work? Using the fact that $\gcd(a, b) = \gcd(a - b, b)$ if $a > b > 0$ (and that $\gcd(a, b) = \gcd(b, a)$), let's try to prove that this function is correct.