15-122: Principles of Imperative Computation

Recitation 2

Josh Zimmerman

Lecture recap

This lecture was mainly about contracts and ensuring correctness of code.

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

Annotation	Checked...
`//@requires e;`	before function execution
`//@ensures e;`	before function returns
`//@loop_invariant e;`	before the loop condition is checked
`//@assert e;`	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, it may behave in undefined ways.
- `//@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).

GCD

Let's get some practice with contracts and proofs. Here's a reference implementation of a function that calculates the greatest common divisor of two numbers.

```
1 int ref_GCD(int x, int y)
2 //@requires x > 0 && y > 0;
3 {
4    int cd = 1;    /* initialize common divisor */
5    int i = 1;
6    while (i <= x && i <= y) {
7       if (x % i == 0 && y % i == 0) {
8          /* i is a common divisor of x and y
9           * and is larger than any common divisor we already saw
10          */
11         cd = i;
12      }
13      i = i+1;
14   }
15   return cd;
16 }
```

Unfortunately, this implementation is kind of slow. We'd like to do better. So, using the fact that $\gcd(a, b) = \gcd(a - b, b)$ if $a > b > 0$ (and that $\gcd(a, b) = \gcd(b, a)$), we can write a faster version of this function:

```
1 int fast_gcd(int x, int y)
2 //@requires x > 0 && y > 0;
3 //@ensures \result == ref_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 ref_GCD(a, b) == ref_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 }
```

This is certainly faster, but does it actually work? Try to prove that `fast_gcd` satisfies its postcondition this on your own or with someone else now for a few minutes. Then, we'll work through the solution on the board.

Another buggy program

I added another function here to give you more practice with invariants and proving correctness of functions.

Here's a function that's supposed to add its arguments. (Admittedly, this particular case is a bit unrealistic, since you wouldn't ever want to implement a slower version of something you already have a fast version of.) The function doesn't work.

To figure out why, add annotations (I've left spaces everywhere you might want to do that), try to prove that it's correct, and see where that proof fails.

```
1 int add (int x, int y)
2
3
4 {
5    // These two variables will let you keep track of old values of x and y
6    // in case you need them in any loop invariants or assert statements.
7    int a = x;
8    int b = y;
9
10   while (b >= 0)
11
12   {
13
14      a++;
15      b--;
16
17   }
18
19   return a;
20 }
```