# 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 `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.)

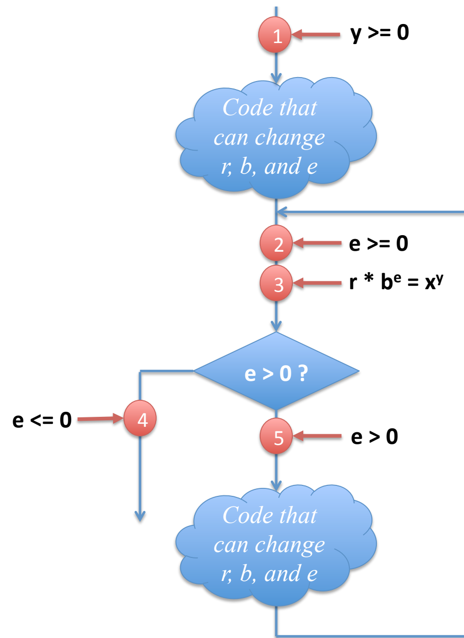
To help you develop an intuition about contracts, here are some explanations of the different kinds of annotations:

- `//@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() {
   // Check for contract errors only










   // Check for contract errors & contract exploits















   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 $\text{gcd}(a, b) = \text{gcd}(a - b, b)$ if $a > b > 0$ (and that $\text{gcd}(a, b) = \text{gcd}(b, a)$), let's try to prove that this function is correct.