15-122: Principles of Imperative Computation

Recitation 2 Solutions

Josh Zimmerman, Nivedita Chopra

Lecture recap

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

There are 4 types of annotations in C0 (for convenience, we're using `exp` here 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` . 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** : 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!). `//@requires` statements are called *preconditions* because they are checked before the function is executed.

- `//`**@ensures** : If the caller satisfies all preconditions, 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. `//@ensures` statements are called *postconditions* because they are checked after the function has been executed.

- `//`**@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).

Checkpoint 0

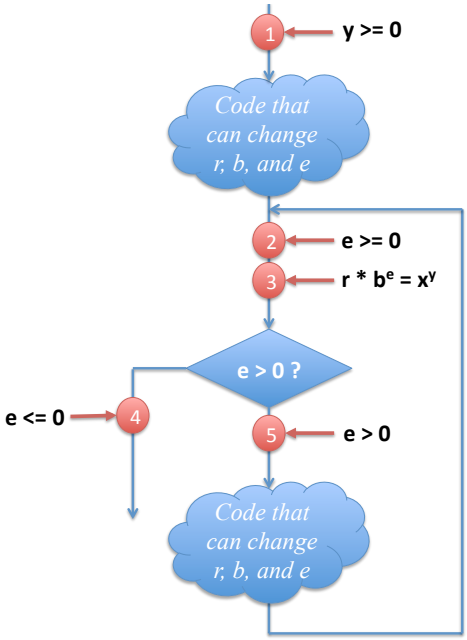
What command would you compile with to enable contract checking in a file named `fastpow.c0`?

Solution: `cc0 -d fastpow.c0`

Proving correctness of the mystery function

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 the mystery function from lecture yesterday. 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 `precondition` of the function, and the circles labeled **2** and **3** are `loop invariants`. 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 __1__ is true before the code runs and show that __2__ and __3__ are true afterwards.
- When we reason about the lower code bubble, we assume __2__, __3__, and __5__ are true before the code runs and show that __2__ and __3__ are true afterwards.
- To reason that the returned value r is equal to x^y, we combine the information from circles __2__ and __4__ to conclude that $e = 0$. Together with the information in circle __3__, 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. We'll call these *contract failures*.
- Cases where the answer was wrong even though the contract was right. We'll 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)` performs the same function as the contract `//@assert exp`. However, since it's a function and not an annotation, it's always checked, whether or not we compile with the `-d` flag. 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.

Solution:

Part 1: Precondition implies loop invariant

First loop invariant: `a > 0 && b > 0`

By precondition, `x > 0 && y > 0`. By line 5, `a == x` and by line 6, `b == y`. Thus, `a > 0 && b > 0`.

Second loop invariant: `gcd(a, b) == gcd(x, y)`

We know that `a == x` and `b == y`, so thus `gcd(a, b) == gcd(x, y)`

Part 2: Preservation of loop invariants

First, assume that both invariants hold at the start of some iteration.

We have three cases: either `a > b`, `b > a`, or `a == b`.

If `a == b`, then we don't enter the loop and so we don't need to consider this case, since there are no more checks of the loop invariants after the last one that succeeded by assumption.

If `a > b`, we enter into the case on lines 11-13.

For the first loop invariant, we know that `b' = b` and that `a' = a - b`. Since `b > 0`, `b' > 0` and since `a > b`, we know that `a' > 0`.

For the second loop invariant, we can apply the mathematical observation made above: If $a > b$, $\gcd(a, b) = \gcd(a - b, b)$.

Since `a > b`, we know that `gcd(a, b) = gcd(a - b, b)` and since `a' == a - b` (and `b' == b`), we know that `gcd(a, b) = gcd(a', b')`. By our assumption, `gcd(a, b) == gcd(x, y)`, so `gcd(a', b') == gcd(x, y)` by the transitivity of equality.

The case when `b > a` is essentially the same, but switching the letters we use.

Thus, the loop invariant holds at the end of the iteration.

Part 3: Loop invariant and negation of the loop exit condition imply postcondition

By the loop invariant, when we exit the loop `gcd(a, b) == gcd(x, y)`. Further, by the negation of the loop exit condition, `a == b`. Since $\gcd(a, a) = a$ for all positive a, we know that `gcd(x, y) == gcd(a, b) == a`. We return `a`, which is `gcd(x, y)` so therefore our postcondition is satisfied if the loop terminates.

Part 4: Termination of the loop

We're almost done now—we just need to argue that the loop does, in fact, exit.

The argument for this is somewhat subtle. The trick is that we look at the quantity $\max(a, b)$. By the first loop invariant, we know that $\max(a, b) > 0$. So, if we can show that after every iteration of the loop, $\max(a', b') < \max(a, b)$, then we know that eventually the loop must exit since $\max(a, b)$ can never go below 1. Now, let's look at that proof.

We again have three cases: either `a > b`, `b > a`, or `a == b`.

If $a > b$, then $\max(a, b) = a$, and $\max(a - b, b) < a$. (Since $b < a$ and $a - b < a$.) Thus, $\max(a', b') < \max(a, b)$ in this case.

If $a == b$, then we exit the loop and terminate.

If $b > a$, then we have an essentially identical case to when $a > b$ and can argue that $\max(a, b)$ decreases simply by changing a few letters.

Therefore, `\result == gcd(x, y)`.

Another buggy program

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 (there are 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 }
```

Solution: Here's the annotated version of the source:

```
1 int add (int x, int y)
2 //@ensures \result == x + y;
3 {
4    int a = x;
5    int b = y;
6    while (b >= 0)
7    //@loop_invariant a + b == x + y;
8    {
9       a++;
10      b--;
11   }
12   // We need b == 0 here because we can't get a strong enough statement from
13   // just the loop invariant. If b < 0 , the loop invariant would be
14   // true, and the result might not be correct from just the loop invariant.
15   //@assert b == 0;
16   return a;
17 }
```

Note that the `//@assert` fails when we call `add(1, 1)`. Adding some print statements to investigate this leads us to the discovery that `b == -1` when we check the `//@assert` . Now, if we look at the loop condition again, we can see that it is incorrect. It should say `while(b > 0)` — otherwise, we're adding one too many times. So, we fix that, and we see that `add(1, 1)` works with no annotation failures and gives 2, as expected.

We're not done, though—we should still try to prove that the function always works, because it's possible that there's another case that doesn't work.

We don't have any `//@requires` statements yet, so let's just try to prove that `add` works for all integers `x` and `y`.

Before the first iteration of the loop, `a == x` and `b == y`, so `a + b == x + y`.

Now, we want to prove that if `a + b == x + y`, then `a' + b' == x' + y'` (we write the values of `a` and `b` after the next iteration as `a'` and `b'`).

After the loop, `a' == a + 1` and `b' == b - 1`. (Note: these calculations may overflow, but that's OK since the overflow would happen if we did `x + y` as well.) This means that `a' + b' == a + 1 + b - 1`, which is equal to `a + b`. By our assumption, that's equal to `x + y`.

Therefore, our loop invariant holds. Now, we need to prove that the `//@assert` holds. We know by the loop exit condition that `!(b > 0)`, so therefore `b <= 0`. However, there's nothing stopping `b` from being negative at this point, which is problematic. We need to require `y >= 0` to ensure a non-negative result at the end of the function. It's also helpful to add a loop invariant that `b` will always be non-negative. At this point, after the corrections we've made, here's our add function:

```
1 int add (int x, int y)
2 //@requires y >= 0;
3 //@ensures \result == x + y;
4 {
5    int a = x;
6    int b = y;
7    while (b > 0)
```

```
 8    //@loop_invariant a + b == x + y;
 9    //@loop_invariant b >= 0;
10    {
11       a++;
12       b--;
13    }
14    // We need b == 0 here because we can't get a strong enough statement from
15    // just the loop invariant. If b < 0 , the loop invariant would be
16    // true, and the result might not be provably correct based on
17    // just the loop invariant.
18    //@assert b == 0;
19    return a;
20 }
```

We must now prove that the second loop invariant holds. It's true initially by the precondition and the fact that `b == y` before we enter the loop. Now, assume that it's true before some iteration of the loop. We also know that, since we're executing another iteration of the loop, `b > 0` by the loop guard. Further, `b' = b - 1`. Since `b > 0`, `b' >= 0` and the loop invariant holds.

Now, let's return to proving that the `//@assert` holds. By the loop invariant, `b >= 0`. Since `!(b > 0)`, or in other words `b <= 0` as well, we know that `b == 0`.

Now, we want to show that that fact combined with the loop invariant implies that the function is correct.

We know that `a + b == x + y` and that `b == 0`. Thus, `a == x + y`.

Finally, we should prove that the function always terminates. `b >= 0` when we start looping, and we decrease `b` each iteration of the loop. Thus, `b` will eventually be 0 and we'll exit the loop.

Checkpoint 1

Write some tests for this add function in the same format as we tested the gcd function earlier. Testing is an invaluable skill, as you will soon see on the programming assignments.

Solution:

```
1 #use <util>
2 #use <conio>
3
4 int main() {
5    assert(add(3, 5), 8); //random
6    assert(add(0, 12) == 12); //first input 0
7    assert(add(12, 0) == 12); //second input 0
8    assert(add(0, 0) == 0); //both inputs 0
9    assert(add(-1, 5) == 4); //first input negative
10   assert(add(2, -1) == 1); //second input negative
11   assert(add(-3, -3) == 6); //both inputs negative
12   /* You can also add in tests with int_max() that will cause overflow
13    * We'll discuss overflow in more depth next recitation
14    * so we're not including those tests here
15    */
16   println("All tests passed!");
17   return 0;
18 }
```