# 15-122: Principles of Imperative Computation

## Recitation 0

Josh Zimmerman, Nivedita Chopra

## 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 TA's name and email.

**Section :**

**TA's Name :**

**TA's Email :**

Also: remember that there's a quiz today! You should complete it online by 10pm. Answers will be available by tomorrow afternoon.

## Basic syntax for C0 programs

**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`: integers $x$, 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 some other languages: arrays have fixed size, so you need to know how long your array will be at the time you declare it. And you need to respect the array size whenever you use 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). `&&` and `||` (as well as other operators like +, -, etc.) are called *infix* operators, 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. Here's an example of `if` statements in C0:

```
1 if (condition) {
2   //do something if condition == true
3 }
4 else if (condition2) {
5   //do something if condition2 == true and condition == false
6 }
7 else {
8   //do something if condition == false and condition2 == false
9 }
```

**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.

`while` loop

```
1 int x = 0;
2 while (x < 5) {
3    printint(x);
4    print("\n");
5    x++;
6 }
```

`for` loop

```
1 for (int x = 0; x < 5; x++) {
2    printint(x);
3    print("\n");
4 }
```

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`.

```
1 int add (int x, int y) {
2    return x + y;
3 }
```

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

Very Bad

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

Okay, but Risky

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

Good

```
1 if(x == 4) {
2    println("x is 4");
3 }
```

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.

## Checkpoint 0

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

```
1 #use <conio>
2
3 def fib(i):
4    if(i == 0 or i == 1){
5      return i;
6    }
7 return fib(i − 1) + fibi(i − 2)
8
9 int main():
10  for int i=0; i < 10; i++
11  printint(fib(i))
12  print(\n)
13  return 0
```

## Contracts

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** : 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.

## Checkpoint 1

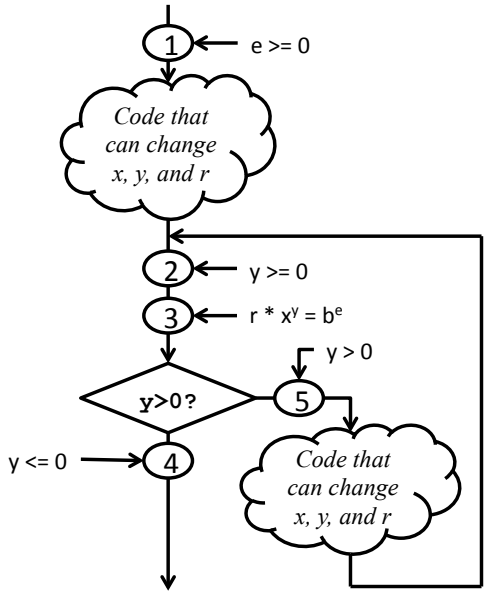
What command would you compile with to enable contract checking in a file named `fastpow.c0`?

---

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

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

To summarize, in general there are four steps for proving the correctness of a function with one loop using loop invariants:

- _____
- _____
- _____
- _____

## Preservation of loop invariants

Preserving loop invariants can be a bit confusing, because we have to assume that the loop invariant, like `y >= 0` or `r * POW(x,y) == POW(b,e)` is true before the loop invariant is checked by assuming that the loop invariant was true the last time the loop invariant was checked (we also assume that the loop guard subsequently evaluated to `true`).

```
1   while (y > 0)
2   //@loop_invariant y >= 0;
3   //@loop_invariant r * POW(x,y) == POW(b, e);
4   {
5      r = r * x;
6      y = y - 1;
7      x = x;
8   }
```

When an arbitrary loop begins, we know _____ and

_____.

After an arbitrary iteration of the loop, we use primed values to represent the new values in terms of the old ones:

$x' =$ _____

$y' =$ _____

$r' =$ _____

We need to show that _____

This is true because _____

This terminates because _____

_____

Because we haven't changed any loop invariants, the rest of the correctness proof for exponentiation is the same as it was in class. By keeping the loop invariant the same, we still have a proven-correct function, even though we tore out loop body and replaced it with a different (and less efficient) one!

## Greatest common divisor

Consider this specification function:

```
1 int gcd(int x, int y)
2 //@requires x > 0 && y > 0;
```

Assuming that this specification function correctly captures the notion of *greatest common divisor*, then work through the process of showing that `fast_gcd` is correct:

```
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 }
```