First Steps

William E. Skeith III

The most important thing I want you to take away from this course is how to carefully formulate **procedures to solve problems**. As you may have gathered already, in order to explain your procedure to a computer, there can be a lot of hassle involved. The main issue perhaps, is that you must specify what seems like *every* tiny detail of your idea, as the computer has no inherent facility to understand you or guess what you mean. But there are some positive flip-sides to this:

1. You can generally count on the computer doing *exactly* what you say.
2. Your programs may look complicated, but most of the *building blocks* you will use to make them are extremely simple and intuitive.

With a bit of practice, the process of translating your ideas into the computer's language will become almost second nature. But first you have to have ideas! Let's focus on developing some procedural problem solving intuition for now. We'll integrate some program code later.

SECTION 1

A Simple Example

Perhaps think of programming as a kind of puzzle or game. I'm going to tell you the rules and the pieces, and then it will be up to you and your wits to find a way to win. **Forget about computers for now.** If you have it handy, get **two** post-it notes (maybe in two different colors) and a pencil with an eraser:

The problem is as follows: your friend has a bunch of very large numbers (each one is around 20 digits!). She will read the numbers aloud to you, and she challenges you to figure out which number is the largest. Naturally your solution should satisfy the following rules:

1. You have to write each number down in order to compare it to anything (each one is 20 digits long, so you can't keep even a single one in your head).
2. Each post-it note can store **one** number only.[1] Keep in mind you can overwrite old numbers since you have an eraser.

---

[1] The ability to write a few more numbers won't really help much anyway.

Now find a friend that's full of random numbers (or just use your imagination) and try to solve an instance of the puzzle. How can you do it using only two post-it notes, pencil and eraser? Do yourself a favor, and actually try this out. Right now! Don't just skip ahead.[2]

Okay, ready for the ʙɪɢ reveal? Let's say we have a blue note and a green one. One way to do it (the solution is far from unique) would be to always keep the *largest number you've heard so far* on the green note, and use the blue one to write down the next number you hear. Then all you have to do is compare what's on the two notes, and overwrite the green one with what's on the blue one whenever the blue is larger. When your friend tells you she's out of numbers, just read what's on the green post-it, and you know it will be the largest of them all. Easy, right? Intuitive, yes? I hope to show in what follows that much of computer programming is just solving games and puzzles like this. And while there are some brutally difficult ones[3], the most basic elements of a programmatic solution are essentially those described above. It might sound crazy, but it's true – at a fundamental level, *every* computer program reduces to numbers and post-it notes.[4]

Here's a quick summary of our process:

1. Get number from friend, write it on the blue note.
2. Compare number on blue note with what's on the green note. If it is bigger, copy it to the green note.
3. Repeat from step 1 until there are no more numbers.
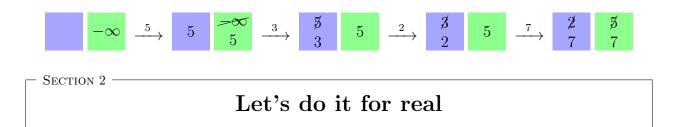4. Green note will contain the largest number from the list.

See anything wrong with the summary? Well, if you think about it carefully, step 2 doesn't make sense the first time around, as the green note will be blank, right? It is exactly this necessity for hyper-detailed thinking that trips up many beginners. Any human would quickly improvise her way through that little gap in your instructions, but the computer won't. So what can we do instead? A few things come to mind: either (a) write the first number you hear on both notes, or (b) begin with the green note containing the symbol $-\infty$ so that step 2 will always copy the first time around.

Here's an illustration of how the notes' contents would change if the sequence of numbers was $5, 3, 2, 7$. The green note will always store the largest number we've seen so far, the blue note is used to store the input, and the next number read is shown over each arrow.

---

[2]Keep in mind that to simulate the game properly, once you hear / see a number, *you can't look back at it ever again* unless you wrote it down on one of your two post-its, so don't just start with a list of numbers written down.

[3]As well as **provably** unsolvable ones!

[4]A more formal expression of this idea is the *Church-Turing thesis*, which states that any reasonably computable function is in fact computable by a simple device called a Turing machine, which bears some resemblance to our post-it note analogy, and indeed subsumes our familiar computing devices (laptop, phone, etc). However, as we will see, it will be convenient at times to build layers of abstractions and concepts upon this foundation so we don't always have to think in such primitive terms. It's almost like we're doing... math! (PS, we are.)

Let's do it for real

Now that we have a clear picture of **the procedure** for solving this problem, let's explore the few details needed to teach that procedure to your computer. As you may have guessed, your computer doesn't actually work with post-it notes. But from your point of view, it may as well – we will work almost entirely *in abstractions.* The specific details of the physical realization of those abstractions will seldom come into play for us in this course, so you can think about your solution in any form that happens to be convenient for you (the post-it notes were just one concrete option that I thought would be helpful, as it lines up reasonably well with the hardware[5]).

To begin with, we need our post-it notes. The programming term for these is **variables**. Here's how you make one in C++:

```
int x; /* <-- you can store an integer in here! */
```

For emphasis, here it is alongside the picture that you should imagine:

```
int x;
```


The obvious questions are how to read and write to x, but first a quick note about **datatypes**. The `int` part in the above is the *datatype* of the variable x. In terms of our silly post-it note analogy, C++ has different shapes and sizes of post-it notes to accommodate different types of information. But this should make good sense: a long string of text—for example the contents of this lecture—probably would require a larger post-it note than one for any integer you're likely to come across. By the way, the semicolon at the end is what delimits one *statement* from another. It's like punctuation – you put a period at the end of each sentence. And now let's see how to write something to x:

```
x = 145; /* store the integer 145 in variable x */
```

This is called an *assignment statment.* Again, a picture for emphasis:

```
x = 145;
```


---

[5]I would like to stress that some kind of mental pictures or abstractions should almost certainly be the **primary** terms in which you think about solving these problems. If your first step is trying to guess some program code, you're (probably) doing it wrong.

A familiar subway stop, but the number 145 doesn't have much to do with our program of computing the largest value. What we really wanted was $-\infty$. We can't quite store that number in our computer, but there is something similar we can use:

```
x = INT_MIN; /* put −∞ in variable x. */
```

Here, INT_MIN is a special constant that works like $-\infty$, in the sense that no int in C++ can be smaller.[6] Note that the direction of information is from right to left – you are writing $-\infty$ to the variable x. Pretty simple so far, right? Now let's see another way to change the contents of a variable. In particular, we need to "listen" for a number from our friend. For this, we appeal to a library function for input and output.

```
int n; /* variable for input */
cin >> n; /* read an integer from 'standard input' into n */
```

In this case, the little >> tells you the direction that information is going: left to right (into n). This is unfortunately backwards from the way assignment works (recall that in x = INT_MIN above, information went from right to left, changing x). The variable cin is defined in the library iostream, but let's not get distracted with that yet – accepting it as a bit of magic for now won't hinder your understanding much. Two things remain: we need to know how to compare things and **conditionally** take an action dependent on the outcome, and we need to know how to do stuff over and over again. Let's do the conditional first. It looks like this:

```
if (n > x) {
    x = n;
}
```

Using what you've just learned, you can almost read this aloud: "if n is greater than x, then overwrite x with n." There are some subtleties about the types of things you can use as conditions, but again, let's try not to get distracted just yet. Alright – we're almost there! All that's left is to do the above steps over and over until there are no more numbers. Here's a first approximation, in which we just do this forever:

```
while (true) {
    cin >> n;
    if (n > x) {
        x = n;
    }
}
```

The while statement accepts a condition, just like the if statement, but instead of doing what's inside those braces 0 or 1 times, it will do it over and over until the condition is no longer true. Since we simply set the condition as the value true, it will go on forever!

---

[6]It is a finite value though, likely equal to $-2^{31} = -2147483648$ on most machines.

Not quite what we wanted. At some point our list of numbers will run out, and we need to know when that happens so we can stop and print our results. There are explicit ways to do this (the function `cin.good()`), but I will offer a different option: It turns out that almost **every expression in C++ has a value**.[7] This includes `cin >> n`. Fortuitously, `cin >> n`, when treated as a true/false value, takes a value of `true` precisely when a number was successfully read. Hence we can do the following, which will naturally stop when there are no more numbers to read:[8]

```
while (cin >> n) {
    if (n > x) {
        x = n;
    }
}
```

Not so bad, right? Here's the entire thing, including the necessary wrapper stuff that was explained in class:

```
#include <iostream> /* needed for cin and cout */
using std::cin;
using std::cout;
#include <limits.h> /* needed for INT_MIN */

int main()
{
    int x;              /* In our analogy, this is the green note. */
    x = INT_MIN;        /* put −∞ in variable x. */
    int n;              /* variable for input (the blue note). */

    while (cin >> n) { /* write number you heard on the blue note */
        if (n > x) {   /* if blue note's contents are larger... */
            x = n;     /* copy from blue to the green. */
        }
    }

    cout << "Largest number was " << x << "\n";
    /* (The answer is on the green note.) */

    return 0;
}
```

And here is how you would test this program in a unix-like environment, using GCC:

---

[7]There are some exceptions that arise from the type `void` of which there are no values.

[8]Note the **side-effects** of our loop condition: every time we check the condition `cin >> n`, we are also trying to change the value of `n`. Two for the price of one :D

```
# assuming our C++ file is named 'largest.cpp', compile like this:
$ g++ largest.cpp   # now you have an executable program named 'a.out'
# Suppose we have some numbers in a file called 'numbers'
$ cat numbers   # print contents of 'numbers'
3 2 7 8 5
# Then we can run our program with all those numbers as input via:
$ ./a.out < numbers
Largest number was 8
# It worked.  Yay.
```

SECTION 3

Summing up

The above example, as simple as it was, illustrates many of the key ideas, and I hope has shown you the typical process you'll go through when writing programs. Here are the important features and steps to keep in mind:

1. When imagining the solution, you actually need not think about any computery details. The abstractions suffice, and in fact should be the **primary** way you think.

2. Once the process is clear in your mind (via some abstractions like the post-it note analogy), the translation to a working program can be done **piece by piece**, in a fairly straightforward way. Each abstraction has a counterpart in the programming language, and with practice, the translation process will become smoother and smoother. Eventually it will seem almost automatic.

SECTION 4

Practical stuff + exercises

A few more notes on running and testing your program:

- As shown above, remember that you can compile this program via `g++ largest.cpp` (provided that's the name of the file).
- You can enter numbers manually (instead of getting them from a file). Just run `./a.out`, type some numbers, hit `Enter`, and then hit `Ctrl-d` to indicate the end of your input (this will cause the next `cin >> n` to evaluate to false and end your loop).
- You can make a file of numbers without even opening an editor by running, for example, `echo 3 2 7 8 5 > numbers`.

A few potentially useful tips:

1. You should **compile often**. Even if you **know** you don't have a working solution yet, get your work in progress to the point where it compiles *ASAP*, and then compile again and again after each small interval of work. This way, you know exactly where any compile errors must be lurking – it's in one of the 5 lines you added since the last compile. (Once you have some experience, maybe you'll feel comfortable waiting a 100 lines instead of just a handful, but to be honest, I seldom wait that long and I've been doing this a while.)
2. If (or shall I say *when*) your program doesn't do what you want, master the process of the *manual trace*. Basically this amounts to pretending you are the computer (perhaps using post-it notes!) and following your own instructions **very carefully**. As mentioned, the post-it notes are a fine analogy for what is actually going on, so it ought to work on paper if and only if it works on the computer. If not, use `cout` to print out intermediate states of variables and see where your misunderstanding lies. Also check out that `gdb` tutorial at some point.

And lastly, some exercises:

1. If your solution to the problem of finding the largest number was different than the one given above, try to write a program that corresponds to your answer.
2. Figure out a way to swap the contents of two variables, using the post-it note analogy. Then write and test a bit of code for it.
3. Write a program that computes the *smallest* value seen on standard input. Compile and test it.
4. Write a program that computes the *second smallest* value, rather than the absolute smallest value. For example, if the input were 3,2,5,2,4, the output should be 3 (even though there were multiple copies of the value 2). *Hint: three integer variables will suffice.*
5. Write a program that computes the sum of all the numbers read from standard input.