# 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 (or two small pieces of paper) and a pencil with an eraser. The problem you are to solve is to listen to a sequence of numbers, and at the end, report the *largest* one. I know it sounds trivial, but there are a few rules that make it a little less so:

1. It won't be realistic to remember the numbers you hear, as each one could be as many as **20 digits long**. So you'll have to write things down.[1]
2. Each post-it note can store **one** number only.[2] Keep in mind you can overwrite old numbers since you have an eraser.

[1] If you can remember long sequences of 20 digit numbers, then maybe you *are* a computer and you don't need this class to begin with, haha.

[2] Maybe you could fit two with small writing, but these are the rules. And as you'll probably see, 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? Here's a hint, just in case: on one of the post-its, always copy the number you just heard.[3] Since space is so limited, try to think about what might be the most important thing you can write on the other paper... Do yourself a favor, and actually try this out. Right now! Don't just skip ahead. **Note:** 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.

Okay, ready for the big reveal? One way to do it (the solution is far from unique) would be to just write the *largest number you've heard so far* on the other paper. Then all you have to do is compare what's on the two pieces, and overwrite the second one with what's on the first whenever the first is larger. When your friend tells you she's out of numbers, just read what's on the second 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[4], 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.[5]

Here's a quick summary of our process:

1. Get number from friend, write it on the first note.
2. Compare number on first note with what's on the second note. If it is bigger, copy it to the second note.
3. Repeat from step 1 until there are no more numbers.
4. Second 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 second 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 pieces, or (b) begin with the second note containing the symbol $-\infty$ so that step 2 will always copy the first time around.

[3] Remember, they are long, and your brain might not be able to remember even a single one without writing it down!

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

[5] 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 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.)

Section 2

# 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[6]).

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! */
```

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 */
```

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.[7] 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.

[6] 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.

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

```
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, which is into `n`, and is (unfortunately, in my estimation) backwards from the way assignment works (the `x = INT_MIN` above). 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! Not quite what we wanted. Instead, we'll make use of the following trick: it turns out that most **every expression in C++ has a value**. This includes `cin >> n` for example. And fortuitously, `cin >> n` takes a value (when interpreted as a true/false 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;
    }
}
```

[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`

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; /* <-- you can store an integer in here! */
    x = INT_MIN; /* put -∞ in variable x. */
    int n; /* variable for input */

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

    cout << "Largest number was " << x << "\n";

    return 0;
}
```

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, to the point of becoming second nature.

Section 4

# Practical stuff + exercises

Remember, you can compile this program via `g++ min.cpp` (provided that's the name of the file). Once you've done so, you'll be left with an executable named `a.out`. You can test it by running it (`./a.out`) and entering numbers manually, hitting `Ctrl-d` to indicate the end of your input (this will cause the next `cin >> n` to evaluate to false and end your loop). Or if you have a file of numbers called `numbers`, you could run `./a.out < numbers` and get the same effect with less hassle. A few potentially useful tips:

1. You should **compile often**. Even if you **know** you don't have a working solution yet, try to get what you have to the point where compiles right away, 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. Write a similar program that computes the smallest value seen on standard input.
2. Write a program that computes the *second smallest* value, rather than the absolute smallest value. *Hint: three integer variables will suffice.*
3. Write a program that computes the sum of all the numbers read from standard input.