

COMP 161 - Lecture Notes - 07 - Procedures for Effect: I/O and State

February 10, 2015

In these notes we look at effects. We'll discuss how they might arise in through program design and consider recipes for designing and developing effect-based procedures.

Interactive Testing

So far, you've only interacted with your functions by writing and running tests. While having frameworks like gTest are great. It's also really nice to interact with the code yourself. If we put this in the context of testing, it'd be nice to be able to choose an input on the fly, feed that to the function, and see the result. We want to interact with the code in a dynamic fashion. This can be an important part of debugging and problem solving, so we should learn how to do it. It also turns out that the programming tools we need to do interactive testing are the same tools we need to write interactive programs. So, while we're just going to think about the problem of testing functions. You should be imagining how these new tools lead to the kind of interactive functionality we know of love from computers.

The Function

Let's look at problem 1 from section 4 of the HTDP1e online problem set¹. I've renamed the function to conform with our C++ standards rather than BSL standards.

¹ <http://htdp.org/2003-09-26/Problems/4.html>

Develop the function *isWithin*, which consumes three numbers representing the x and y coordinates of a point and the radius r of a circle centered around the origin. It returns true if the point is within or on the circle. It returns false otherwise. The distance of the point to the origin is the square root of $x^2 + y^2$.

Here's the end product of our development process²

² Recreate this process for practice

```
// In the header
```

```
namespace TwoD{
```

```
/**
```

```
 * Determine if the point (x,y) is on or within a
```

```

    * circle with center (0,0) and radius r
    * @param x the x coordinate of the point
    * @param y the y coordinate of the point
    * @param r the radius of the circle
    * @return true if (x,y) is in or within the circle
    * @pre radius r is positive or 0
    */
    bool isWithin(double x, double y, double r);
}

```

```
// In the tests
```

```

TEST(isWithin,all){
    using namespace TwoD;

    // Positive Tests
    EXPECT_TRUE(isWithin(0,0,0));
    EXPECT_TRUE(isWithin(0,0,0.5));
    EXPECT_TRUE(isWithin(0,0,2.5));
    EXPECT_TRUE(isWithin(1.0,0.5,2.5));
    EXPECT_TRUE(isWithin(2.5,0.0,2.5));
    EXPECT_TRUE(isWithin(0.0,2.5,2.5));

    // Negative Tests
    EXPECT_FALSE(isWithin(1.0,0,0));
    EXPECT_FALSE(isWithin(0,1.0,0.5));
    EXPECT_FALSE(isWithin(2.5,2.5,2.5));
    EXPECT_FALSE(isWithin(0,0.501,0.5));
    EXPECT_FALSE(isWithin(2.5,0.1,2.5));
    EXPECT_FALSE(isWithin(0.0,3.0,2.5));
}

```

```
// In the Library Implementation
```

```

#include <cmath>

namespace TwoD{

    bool isWithin(double x, double y, double r){
        return r >= sqrt(x*x + y*y);
    }
}

```

```
}
```

With this kind of function we can expect to run into some double arithmetic rounding problems. In particular, when we're checking for the point one the edge of the circle, the equality case, things could easily get off and produce erroneous results. So, it might be nice to interactively test this function to see when that kind of problem crops up. It might also be nice to try some new values without the need to write new tests.

Goal: Two Programs for Interactive Testing

Let's identify two programs that would allow us to test *isWithin* in an interactive fashion:

1. an primitive REPL³ program that prompts you for input values and prints the results until you close it. The looping part is really optional. We could just do one test, then relaunch the program for more tests as needed.
2. a CLI program that lets the user pass the *isWithin* inputs as CLI arguments. This is like the previous program without a loop and without any prompts.

³ Read Evaluate Print Loop

We'll first focus on program number one, the REPL program. It's a bit easier to get off the ground and exposes to wide and wonderful world of effect-based, imperative, procedural programming. We'll also checkout program number two, but will likely hand-wave over a few details that we'll explore in depth later on down the line.

Problem: Effects

Before we dive into the guts of these programs, let's make it clear why we need effect-based code. Clearly we need I/O⁴ effects as we must READ data from the CLI input, aka `STDIN` and feed it as inputs to our function. We then want to WRITE the results of our computation to the CLI output, `STDOUT`. Racket did this kind of thing implicitly in the interactions window. We have no such system now and need to explicitly manage it ourselves.

⁴ Input/Output

What's less clear is why we need STATE MUTATION. The short answer is because you cannot read from `stdin` directly into a function. You must first read into a variable⁵, then pass the value of that variable to the function. This means declaring and initializing a variable, then using MUTATION to change the value to what is read from `stdin`. There are good reasons for this requirement. Sometimes the read fails. If that happens when you're attempting to call the function,

⁵ i.e. memory

then your program will probably crash. It's also just a bad idea to assume that what's coming in from *stdin*, or input devices in general, is well-formed, valid data. You might be expecting a number, but the user could, for whatever reason, give you string data. This can be an honest mistake or malicious intent. In web-based systems, you want to check all user input from a form before acting on it in order to avoid SQL injection⁶.

⁶ http://en.wikipedia.org/wiki/SQL_injection

When dealing with CLI arguments as we'll do with program two, we don't have to worry about reading from *stdin*. The operating system takes the CLI command, parses it, and sends it to our program as C strings. This is great if we needed strings, but we need numbers. The solution is actually to use a C++ library that allows us to treat a string as a input device. We can then read from the string(s) in the exact way we read from the CLI. The end result is the same, we just have to work through an intermediate step.

Program 1: Interactive REPL

Let's dive right into the first program. We'll first look at the whole main procedure, then we'll break down the logical steps. I've included the *#include* statements to highlight the standard C++ I/O library, *iostream*⁷.

⁷ <http://www.cplusplus.com/reference/iostream/>

```
#include <iostream> //std::cout std::cin std::boolalpha
#include "ln7.h"    // TwoD::isWithin

int main(int argc, char* argv[]){
    using namespace TwoD;
    using namespace std;

    // Declare Variables to hold user input
    double x{0.0};
    double y{0.0};
    double r{0.0};

    // explicit infinite loop
    // Use Ctrl-C to kill the program at the CLI
    // This is a terrible idea more often than not, but
    // works for this application.
    while(true){

        // Prompt and get x,y
        cout << "Enter point coordinates: ";
```

```

cin >> x >> y;
// Prompt and get r
cout << "Enter circle radius: ";
cin >> r;

// Compute and report results
cout << "isWithin( " << x << " , " << y << " , " << r << " ) -> " <<
    boolalpha << isWithin(x,y,r) << '\n';

} //end while

return 0;
}

```

The first two lines of the procedure are the familiar *using namespace* statements. The first namespace is for *isWithin*. The later, *std*, is the *standard* namespace that contains all the elements of the *iostream* library.

Variable Declaration and Initialization

The first step of the procedure is to declare and initialize variables for our user input.

```

// Declare Variables to hold user input
double x{0.0};
double y{0.0};
double r{0.0};

```

Here we see three variable of type double, *x*, *y*, and *r*, each assigned an initial value of 0.0 using brace-initialization. This form of initialization is new to the c++11 standard⁸. Prior to the c++11 standard you'd see alternative initialization statements such as:

```

double y; //pure variable declaration. type and name
y = 0.0; // then immediately assign "initial" value

double x = 0.0; //declare and initialize with assignment

double r(0.0); //constructor function style with parenthesis

```

The first option is really a C style. It's only an initialization in that nothing happens between declaration and the assignment. I don't prefer this because you're tempted to not assign an initial value and you should always assign an initial value. The second line of this pair bears some comment. This is a STATE MUTATION or *variable assignment* statement. It is not a mathematical equation. It is a statement

⁸ this means we need to add the -std=c++11 compiler option to the Makefile rule for building objects

that reads: “Assign the value 0.0 to the variable with name *y*” or “Assign the value 0.0 to the memory location labeled with the name *y*”. We’ll talk in more detail about assignment shortly.

The second option for initialization is nice because it’s one cohesive statement, but because it uses the assignment operator `=` it can sometimes confuse people about the usage of variable type annotation. They see the *double* in this statement and want to then continue to use *double* wherever they are referring to, and specifically assigning values to, *x*. I prefer my initialization statements to look completely different from my general purpose assignment statements.

The final option is great from the perspective that it looks different than assignment and therefore reminds us that initialization and general purpose assignment are different things. The only reason we’ll use braces instead of parenthesis is that the parenthesis become more meaningful as we explore more involved kinds of data. Curly brace initialization was introduced to provide a consistent means of initializing data for standard C++ types.

We’ll always use the new brace initialization style as it really sets apart initialization and is fairly uniform across the types of data we’ll use. You should recognize other initialization styles because you may need to read or write them someday, but *always initialize and always use brace initialization in this class*⁹.

⁹ second reminder: this requires the compiler option `-std=c++11`

Infinite Loop

Next you’re confronted with an infinite loop, specifically an infinite *while* loop.

```
while(true){
    // Loop body
}
```

A *while* loop will repeat everything in it’s curly brace block¹⁰ for as long as the boolean expression in the parenthesis evaluates to *true*¹¹. In this case, we’ve explicitly put *true* in parenthesis so this loop will repeat forever. We’ll explore loops quite a bit soon, but it’s really hard to separate loops constructs like this one from effects. A great many loops drive variable mutation over time or I/O like this one. In short, a loop is rarely, if ever, a functional, black-box process; they almost always have some kind of effect on the overall system with respect to your program.

There aren’t a lot of good reasons to write infinite loops. This one makes sense from the perspective that we’re writing a program only meant to be run by us and with a very special purpose. It’s a quick and dirty solution to the problem of creating a REPL and

¹⁰ called the loop body

¹¹ If you can’t wait to learn more:
<http://www.cplusplus.com/doc/tutorial/control/>

we're not imagining this program will have much of a life span or user base. As soon as your program is going to be around awhile or used by people not you, you should find a better way than an infinite loop¹²Be warned, I'll rarely expect an explicitly infinite loop as a solution to a problem on your homework.

¹² Like an explicit quit key

Prompt and Input Sequences

The body of our infinite loop contains our first I/O statements. What we see is a pattern of prompting the user for input then actually getting their input. Let's look at the first example.

```
cout << "Enter point coordinates: ";
cin >> x >> y;
```

We want to think of these statements in terms of the binary STREAMING OUTPUT OPERATOR `<<`¹³ and the binary STREAMING INPUT OPERATOR `>>`¹⁴. Let's start with the first statement– the output statement. To the left of the operator is the output stream *cout*¹⁵. It is our direct connection to the CLI standard output. The effect of this statement is to print the string literal "Enter point coordinates: ".

¹³ "put to" operator

¹⁴ "get from"

¹⁵ or `std::cout` because it's in the *std* namespace

The input stream *cin* connects us to *stdin*. It reads values from the CLI and, in this case, writes them to variables. Data from *cin* is, by default, separated by white space. In this case, we'll take the first TOKEN and read it like a double, because *x* is a double, and save it to *x*. The next token is also read as a double and then saved to *y*. You can separate the two tokens with spaces, tabs, or even newlines as those all qualify as whitespace characters. It's important to note that the program will read *any* data as if it were a double. If you type the string *hello*, the program will attempt to interpret as a double literal and interesting and unexpected things will happen.

Our procedure continues with one more prompt and input sequence:

```
cout << "Enter circle radius: ";
cin >> r;
```

Let's just restate the effect of these statements in plain English:

Put the string literal "Enter circle radius: " to the standard output via the stream *cout* get a value from the standard input stream *cin* and write it as a double to the variable *r*.

It's important to get a feel reading these statements as applications of streaming I/O operators. In doing so you'll be less attached to specific streams and more able to adapt this logic to alternate I/O streams.

There is a great deal of effect-based programming going on under the hood of the streaming I/O objects *cin* and *cout*. We'll expose some details as needed. For now, it's enough to know that they buffer¹⁶ data from their receptive devices and that the I/O operators mutate the contents of those buffers when you use them. They are your first example of STATEFUL OBJECTS. They are not black-boxes. They have hidden internal state that combines with input to determine their output and behavior.

¹⁶ temporarily store

More Involved Output

The last statement in our loop is a fairly involved output statement.

```
cout << "isWithin( " << x << " , " << y << " , " << r << " ) -> " <<
    boolalpha << isWithin(x,y,r) << '\n';
```

First note that this is a long statement and I split it across two lines. The compiler typically treats all white space the same and generally ignores it. This means we're free to split a line as long as we don't break up something important.¹⁷ To understand why we have a lot of << operators in this statement remember that the operator is binary only with one operand being the stream and the other a single datum. So, in order to output *n* things, we'll need *n* operators.

¹⁷ Don't split the string literal or a <<

Before going further, let's see an example of the kind of output this produces:

```
isWithin( 3.4 , 5.6 , 10 ) -> true
```

All of the spacing in that output *is not the result of breaking things up with <<*. Put another way, *additional << operators do not introduce any kind of white space to the output*. If you go back and look at all the string values, you'll notice carefully placed spaces. This is how the actual output comes out nice and readable. Without those spaces it would all be smashed together.

We could also get the exact same output with two statements instead of just the one.

```
cout << "isWithin( " << x << " , " << y << " , " << r << " ) -> ";
cout << boolalpha << isWithin(x,y,r) << '\n';
```

This alternative sequence of output statements tells us that *new output statements do not produce new lines in the output*.

Students first starting out with C++ I/O assume too much of their operators. So, commit this thought to memory now:

If you want spaces and newlines in your output, then you must explicitly put it there.

There is one final thing of note in this statement. One of the tokens is not actual output, but a token used to modify how data in the stream following it is displayed. Any boolean values after the *boolalpha*¹⁸ token will be printed as *true* or *false* rather than 1 or 0. We call these tokens I/O MANIPULATORS. They are used to set the state of the I/O stream object and modify the way in which data is displayed. They allow us to format how data is displayed.

¹⁸ std::boolalpha

Before moving on, let's restate this doozy of an output statement in plain English:

The the stream *cout* put the following data, in this order: the string literal "isWithin(" the value stored in the variable *x*, the string " , ", the value stored in *y*, the string " , ", the value stored in *r*, and the string ") -> ". Next, put the *boolalpha* manipulator to the *cout* stream in order to format boolean output to alphabetic. Finally, put the result of *isWithin(x,y,r)* to *cout* and the newline character.

Recap

This relatively straight forward example illustrates several key issues surround basic C++ stream I/O and variables. Be sure you understand what's happening with this concrete example. In the following sections we'll explore the big picture and establish some general guidelines for variable and I/O effects.

State Variables

We previously encountered variables when dealing with the linux CLI and noted that three logical operations tend to govern our interaction with variables.

- **INITIALIZATION**
Assigning a starting, initial value to the variable.
- **MUTATION**
Assigning a new value to a previously initialized variable
- **ACCESS**
Viewing or getting the current value in the variable.

Let's look at basic statements for each of these operations.

Declaration and Initialization

In C++, variables are typed. This means they can only hold values of that type. When we first add, or *declare*, a variable to our program we must state its type and name and initialize its value. Variable declarations simply provide the type and the name with no initial value— *you*

should never just declare a variable. We'll be using the C++11 brace initialization syntax and so our variable declarations and initializations have the following template:

```
TYPE NAME{INITIAL-VALUE};
```

You've seen a few examples already, but here's a few more:

```
char  achar{'a'};
bool  isOK{false};
int   size{15};
double amplitude{0.707};
```

Accessing Variables

When we talk about accessing a variable we mean retrieving the value currently stored in that variable. To do this we simply refer to the variable by name. You saw this in the output statements from our testing program and when we called the function. Here's a few more examples of accessing variables. I've reused the variables from above and written examples of accessing them in the context of gTest unit tests so that you get a sense for the substitution of variable name for value.

```
EXPECT_EQ('a', achar);
EXPECT_EQ('A', toupper(achar));
```

```
EXPECT_FALSE(isOK);
EXPECT_TRUE(!isOK);
EXPECT_TRUE(isOK || !isOK);
```

```
EXPECT_EQ(15, size);
EXPECT_EQ(-3, size-18);
EXPECT_EQ(33, 2*size+3);
```

```
EXPECT_DOUBLE_EQ(0.707, amplitude);
EXPECT_DOUBLE_EQ(1.414, 2*amplitude);
```

Variable Mutation

To mutate a variable means to assign it a new value. This is carried out via the assignment operator `=`. This operator is binary. Its left-hand operand must be something with an associated location¹⁹, i.e. the name of a variable. Its right-hand operand must be an expression, i.e. something with an associated value²⁰. Let's look at a few examples and then break them down.

¹⁹ L-VALUE

²⁰ R-VALUE

```

size = 5;
achar = toupper('R');
size = size + 5;
achar = toupper(achar);

```

The first example is straight forward: “Assign the value 5 to the variable *size*”. The second example assigns the return result of *toupper*(‘R’) to the variable *achar*. The next example looks nuts if you confuse assignment = with how we read = in mathematics. As a C++ assignment statement it reads: “Assign the value of *size*+5 back to *size*”. If the current value in *size* is 10, then it changes to 15 as a result of this statement. If we read this statement like mathematics then it says: “*size* is the same as itself plus 5”. That’s nuts. No number is equal to five plus itself. The final assignment example is like the second in that we’re first computing the *toupper* of the current value in *achar*, then assigning that back to *achar*.

Assigning a variable an updated value based on its current value is so common that several shortcut operators exist for just this purpose.

Operator	purpose
<code>+=</code>	$a += b$ is the same as $a = a + b$
<code>*=</code>	like <code>+=</code> but with <code>*</code>
<code>-=</code>	like <code>+=</code> but with <code>-</code>
<code>/=</code>	like <code>+=</code> but with <code>/</code>

Streaming I/O

The C++ streaming I/O system provides a fairly uniform way of approaching I/O tasks. Streams provide buffered I/O access to a device and the operators `<<` and `>>` are used to put and get, respectively, data from those streams. Our testing REPL exposed us to the standard output stream *cout* and the standard input stream *cin*. The *iostream* library also provides the output stream *cerr*, which is the standard error output. The *fstream* library allows us to create I/O streams for files. The *stringstream* library allows us to treat string objects like streaming I/O devices and in doing so lets us leverage the streaming I/O capabilities for interpreting characters as typed data, and vice versa, when processing string data. Our first use for this will be working with arguments passed to our programs from the command line.

The following examples will all use *cout* and *cin* as our representative stream objects. Later we’ll see that we can replace *cout* with another output stream object and *cin* with an input stream object and get the same general results, just to different I/O devices.

Streaming Output

As we've seen already, streaming output is done via the "put to" operator `<<`. Our example program really highlights all the key issues for working with streaming output, so it bears close examination.

The main things to remember are that:

- the only white space that gets written is the white space you explicitly add to the stream
- you chain together multiple uses of `<<` with a single stream as long as its one datum per operator

To illustrate these points I give you one example.

```
cout << 7 << 8.36 << " isn't ";
cout << 7 << " " << 8.36 << "\n" << "hi\n";
```

The result of these two statements is the output:

```
78.36 isn't 7 8.36
hi
```

The cursor ends immediately below the h in hi.

Streaming Input

Once again, our example problem pretty much covers all the bases. Go back and read it carefully. Some key points to remember are:

- You cannot do streaming input without variables because you must have a place to store the value that `>>` gets from the stream
- Data from the stream is broken up by whitespace. This includes the enter key, so hitting enter doesn't terminate input, reading tokens does.
- The way in which data is read from the stream is determined by the type of the variable in which the data is being stored and not by how it appears in the stream.
- You can chain input together to read multiple values with a single statement.

Program 2: CLI Input

The CLI is built on top of strings. When we want to take input from the CLI for our C++ programs, then we have to take it as strings. So, to get our second program working, we need to start looking

at strings. We'll start with a quick and dirty overview of just some essentials for managing this program and get into more details in the next set of lecture notes.

Without further ado, here's a version of this program that lets you pass in your test values from the CLI. The tradeoff here is that there is no loop. We have to re-run this for each new set of data²¹.

²¹ or write a looping Bash program to run this maybe

```
#include <iostream> // std::cout, std::cerr
#include <sstream> //std::istringstream
#include "ln7.h"

int main(int argc, char* argv[]){
    using namespace TwoD;
    using namespace std;

    double x{0.0};
    double y{0.0};
    double r{0.0};

    // Quick error check for enough arguments
    if( argc != 4 ){
        cerr << "Not enough arguments. Usage: " << argv[0] << " x y r\n";
        return 1;
    }

    // Let's assume we typed numbers as arguments
    istringstream xstream{argv[1]};
    xstream >> x;

    istringstream ystream{argv[2]};
    ystream >> y;

    istringstream rstream{argv[3]};
    rstream >> r;

    cout << "isWithin( " << x << " , " << y << " , " << r << " ) -> " <<
        std::boolalpha << isWithin(x,y,r) << '\n';

    return 0;
}
```

First off, take note that we're using two libraries: *iostream* and *sstream*. The first is familiar. It provides basic I/O capabilities and ac-

cess to `std::cout` and `std::cin`. The second library is for string streams²². It provides us with the `stringstream`²³ type that lets you treat a string as if it were an a streaming input device.

²² <http://www.cplusplus.com/reference/sstream/>

²³ input string stream

At a very high-level this program proceeds as follows:

Declare variables to store the CLI input. Check that a sufficient number of CLI arguments were given and if not, end the program and signal and error. Create input string streams from each of the CLI input strings and read the data from those streams to the variables. Output the results.

CLI Inputs and main's arguments

When we run our programs from the CLI, the operating system will take the command run, package each piece as a string, count the number of pieces, and hand all of this off to the program as arguments to main.

Let's look at main's signature:

```
int main(int argc, char* argv[])
```

The first parameter, the integer *argc*, is the number of arguments or the argument count. This includes the name of the command as well as all the other arguments. For example, if our executable is named *runIsWithin* and we run the following CLI command,

```
./runIsWithin 2.3 4.5. 10.6
```

then the value of *argc* is 4. The arguments themselves are stored in the parameter *argv*.

The parameter *argv* is an *array of C-Strings*. We know this because the type *char** is, informally, the C-String type and when you put a set of square brackets after a variable or parameter name you're indicating that its an array. Arrays are important structures and we'll look at them more in the future. For now it's enough to know that they are collections of homogeneous data. In this case, the array *argv* is a collect of C-strings. If we had the parameter *int vals[]* or *double vals[]*, then we'd be working with a collection of integers or doubles, respectively. Arrays are an INDEXED collection such that each datum is accessible via its index number. The first element in the collection is always 0²⁴ and the index values increase sequentially from there.

²⁴ say it again, "the first element in an array is always zero"

Returning to our example from above, we demonstrate how to access the array *argv* and how CLi arguments are stored through some gTest unit tests²⁵.

²⁵ EXPECT_STREQ is used to test two C-Strings for equality

```
EXPECT_STREQ("./runIsWithin" , argv[0]);
EXPECT_STREQ("2.3" , argv[1]);
EXPECT_STREQ("4.5" , argv[2]);
EXPECT_STREQ("10.6" , argv[3]);
```

Restating the first test in plain English, we'd say, "Expect the value of *argv* at 0 to be the same as the string `./runIsWithin`".

It's a recipe for disaster to attempt to access an array at a location where no data exists. We call this an INDEX OUT OF BOUNDS error and it's a classic bug in software. Thankfully, the value of *argc* always tells us exactly which index values are OK and which are not— it tells us the size of *argv*²⁶. For our program we need exactly 4 arguments, the executable name and the three numbers. It's easy enough to check that we have exactly 4 and we should do so.

²⁶ arrays of size s have index values from $\{0, \dots, (s - 1)\}$

```
// Quick error check for enough arguments
if( argc != 4 ){
    cerr << "Not enough arguments. Usage: " << argv[0] << " x y r\n";
    return 1;
}
```

Here we check the value of *argc* and if it's not 4, we output a message to *cerr*²⁷. Notice we include the executable name in the message and attempt to remind the user, i.e. ourselves, how to use the program properly. Finally, we end *main* by returning 1. Here we assume that a non-zero return value signals an error condition of some kind.

²⁷ where errors go

Streaming Strings

The central obstacle to our CLI-based program is that we're given C-string data and need doubles. The C++ streaming I/O system has already solved the problem of reading a series of characters and determining what numerical value they represent, so let's just re-use that code by way of using string streams. Specifically, we'll turn our C-Strings into input string streams, which we can then use along with the streaming input operator. Here's the first example:

```
istringstream xstream{argv[1]};
xstream >> x;
```

Our first statement declares the input string stream²⁸ *xstream* and initializes it with the first numerical argument. The second statement then gets the first value from that stream and reads it, as a double, to the variable *x*. We repeat this two more times, once for each remaining argument.

²⁸ istringstream

```
istringstream ystream{argv[2]};
ystream >> y;
```

```
istringstream rstream{argv[3]};
rstream >> r;
```

Assuming the strings can be read as double literals, then that's it. We've now read our CLI arguments in as doubles. The final statement in our program is the same as version 1, we just report the results with an output statement.

This Doesn't Scale Well

You've seen the basics of streaming I/O and started to get some sense of how to work with variables through basic statements in C++. The problem with only knowing statements is that it doesn't scale. Complex effects typically need to be abstracted away from main, or whatever procedure initiated them, and turned into a procedure in their own right. Doing this simplifies program logic by breaking it down to task-specific procedures and provides us with an opportunity to test our effect-based code.

As it stands, the only way we can tell if our testing programs are producing the desired effect is to run them. This is mainly due to the fact that the effect-based statements are in main and not abstracted away into a helper procedure. The other problem is that the effects are generally taking place on the standard input and output devices and there is no way to tell the computer to inspect those devices.

We can easily solve the first problem by laying down some ground rules for designing and writing procedures whose purpose is to carry out a desired effect. We can work around the later problem by designing our procedures in such a way that they leverage the Object-Oriented design of the C++ streaming I/O system. We'll be able to write general purpose I/O procedures and test them relative to string streams. We can then use them on cin, cout, and even file streams without changing the code. Before we can do any of this, we really need to take a moment to study the world of strings in C++.