COMP 161 - Lecture Notes - 09 - Procedures for Effect

February 17, 2015

In these notes we re-evaluate our procedure design and development process to account for procedures whose primary purpose is variable mutation or I/O effects.

Designing for Effect

We can build entire systems of purely functional principles, but we would not to do so using C++. Program design with C++ lends itself better to a systems approach in which we interact with different components of a larger computing system. Pure functions like we've been writing are still highly useful in this context. We can imagine a pure function as something that only involves the CPU and is independent of other parts of the system. While this independence is great, it 's necessary to have some procedures interact with systems like memory or I/O.

We'll consider the design of three types of effect-based procedures:

- Variable Mutation
- Output to a Stream
- Input from a Stream

To look at mutation procedures we'll consider a classic example: swapping the values of two variables. We'll then return to our program from lecture notes 7 and revise and update it using I/O procedures

Pass-by-Value and Pass-by-Reference

So far we've seen variable mutation happen one of two ways: assignment operators and mutator methods. For example,

```
double a{0.0};
string d{"This is a string"};

a += 5; // aka a = a + 5
EXPECT_DOUBLE_EQ(5.0,a);

d.push_back('!');
EXPECT_EQ(string("This is a string!",d);
```

The logic of both of these could be expressed as procedures. Let's just pretend we have the procedure *addN* which adds a value to a variable and *push_back* which adds a character to the end of a string. If these procedures existed, then we might write the above examples as follows:

```
double a{0.0};
string d{"This is a string"};

addN(a,5.0);
EXPECT_DOUBLE_EQ(5.0,a);

push_back(d, '!');
EXPECT_EQ(string("This is a string!",d);
```

The idea is to pass the variable(s) to be mutated along with any other needed values to the procedure. The procedure then modifies the variable(s), leaving them changed from that point forward in the execution of our program.

There's something subtle going on here. For this to work we need to make certain that *the mutator function has access to the variable itself*. Normally, when we use a variable name in an expression, like a procedure call, then the system interprets that as a request for the value inside of the variable. This isn't what we want. If it were, then really our call to *addN(a,5.0)* would be equivalent to *addN(0.0,5.0)*. What this all tells us is that we must somehow signal to the compiler that when a variable is passed to a mutator, it should treat that variable differently than it does the rest of the time. Namely, the system should treat the variable in the context of the mutator as an expression of a location and not an expression of a value.

We've actually seen this before. Consider the expression $a = a + 5$. To the right of the assignment operator a is an expression of value, the current contents of the variable a. To the left of the assignment operator a is an expression of location, the place to store the result of $a+5$. This duality of meaning is so fundamental to many programming languages that the two values have formal names. The L-VALUE is the location value of a variable and the *r-value* is the content value of a variable.[1].

[1] Remember l for left side of assignment, r for right side

Now consider the following function declaration:

```
int foo(int x);
```

What we're really declaring here is that the parameter x is an integer value. We call this PASS-BY-VALUE because *foo* accepts any integer value as an argument. This can be a literal *or* the r-value of a variable. If, instead, we wish to pass *foo* an integer *l-value*, the location where

an integer is stored[2], then we need to use PASS-BY-REFERECE. One small change in syntax creates this major change in semantics[3].

[2] i.e. an integer-typed variable

[3] syntax = how to write it. semantics = what it means

```
int foo(int& x);
```

The type *int&* is the integer reference type. Appending the character & to any type signals to the compiler that you're concerned with l-values of that type, references to locations where that type can be found.

By declaring a procedure parameter as a reference type, you've made a pretty drastic change to how arguments to that parameter are treated. This is best illustrated through a concrete example.

Swap

A procedure to swap the values stored in two variables is extremely useful to have around. So much so that it's already defined in a C++ library[4], but we'll reinvent the wheel a bit here in order to explore mutation procedures and pass-by-reference vs pass-by-value semantics.

[4] `http://www.cplusplus.com/reference/algorithm/swap/`

The procedure *swap* is clearly a mutator; it changes the value stored in not one, but two variables. In order to mutate a variable within a procedure we must declare that variable parameter as a reference type. So, if we're going to swap the values of two double variables, then we need swap to work on two parameters of type *double&*. What about the return type? Its not at all clear that *swap* should return anything, so what we'd like to do is declare that swap return nothing. This is done with the type *void*[5].

[5] *void* isn't so much a type as a signal to the compiler that no value, of any type, is returned

```
namespace pbr{

  /**
    * Swap the value of two double variables
    * @param x reference to first double object
    * @param y reference to second double object
    * @return none
    * @pre none
    * @post the values in x and y have been swapped
    */
  void swap(double& x, double& y);

} //end namespace pbr
```

To further illustrate why pass-by-value won't work here, let's declare a second version.

```
namespace pbv{

  /**
    * Swap the value of two double variables. This
    * version does not work due to pass-by-value semantics
    * @param x first double
    * @param y second double
    * @return none
    * @pre none
    * @post the values in x and y have been swapped (not really)
    */
  void swap(double x, double y);

} //end namespace pbv
```

We expect the first version to actually change the values of variables it is given as arguments. We expect the later to do nothing. Let's express this as some tests.

```
  TEST(swap,all){

    double a{3.141};
    double b{2.718};

    EXPECT_DOUBLE_EQ(3.141,a);
    EXPECT_DOUBLE_EQ(2.718,b);

    pbv::swap(a,b); //has no effect

    EXPECT_DOUBLE_EQ(3.141,a);
    EXPECT_DOUBLE_EQ(2.718,b);

    pbr::swap(a,b); //mutates a & b

    EXPECT_DOUBLE_EQ(2.718,a);
    EXPECT_DOUBLE_EQ(3.141,b);

  }
```

It is important to note that mutation tests must work by doing a before and after check of the value. When the before value is clearly an initial value, we can usually leave off the before test. Either way, the key observation is that the purpose of a mutator is to change the value in a variable, so our tests must check for just that. We don't test against a return value because there is no return value and if there were it would be secondary to the purpose of the procedure: *to*

change the value of variables.

It's important to point out that the names of the variables we're swapping[6] do not match those of the parameters to *swap*[7]. There's no reason they should. Names, or IDENTIFIERS, are, in this case, abstractions over memory addresses. Rather than refer to variables by address, we refer to them by name. In doing so, we're free to give a single address multiple names. When you pass a variable by reference you are creating an ALIAS of the variable being passed. For the remainder of the procedure execution, the reference parameter name is an alias to a variable back in the calling procedure. In our tests, the *x* and *y* in *pbr::swap* are alias to *a* and *b* in the test space. So what happens when you pass by value? You make *copies not alias*. The *x* and *y* in *pbv::swap* are copies of *a* and *b* in the test space. This observation is at the heart of how pass by value and pass by reference actually work. As we deal with more complicated objects, the copy effect of pass-by-value can have drastic effects. Imagine that you're passing the complete works of Shakespeare as a value. Do you really want to copy all that text? When this becomes an issue, we'll address it. For now, you need to get your head around the copy vs alias nature of pass by value and pass by reference.

[6] *a* and *b*
[7] *x* and *y*

Before we look at the actually logic of swapping, let's talk about stubbing out this *void* return type function. Recall that stubs are meant to complete a definition that satisfies the signature. Up until now that's meant returning a value of the same type as the return type. Now, there's no requirement to return a value. So, we use return simply to terminate the procedure and go back to the program at the point where the procedure was called.

```
void swap(double &x, double &y){
return;
}
```

You can, in fact, omit the return and stub *void* return type functions like this:

```
void swap(double &x, double &y){


}
```

I'm not a fan of this style. The return is implicit and making it explicit costs you nothing but a few seconds of typing.

Now, about the actual at of swapping. At first glance you might decide to reassign the two values:

```
void swap(double &x, double &y){
x = y;
y = x;
```

```
return;
}
```

This is wrong. To see why, we should trace through the statement sequence.

Let's assume that x and y are alias for a and b as declared in our test[8]. The first assignment reassigns to x the value stored in y. At this point you should see the problem. Both x and y contain the same value and we've lost the original contents of x all together. To solve this, we need some temporary storage. First save the value of x. Then write the value of y to x. Finally, write the saved value of x to y. Here's how we say that in C++.

[8] x == 3.141 and y == 2.718

```
\
void swap(double &x, double &y){
double temp{x}; // save x in temp
x = y; // overwrite x with y
y = temp; // write old x to y
return;
}
```

We can do this same sequence of assignments in the pass by value version.

```
void swap(double x, double y){
double temp{x}; // save x in temp
x = y; // overwrite x with y
y = temp; // write old x to y
return;
}
```

We need to be sure we understand why this doesn't work. It's clear that *a* swap occurs between x and y. If you're not convinced, throw in some output statements to see what's happening.

```
void swap(double x, double y){
cout << "x=" << x << " y=" << y << '\n';

double temp{x}; // save x in temp
x = y; // overwrite x with y
y = temp; // write old x to y

cout << "x=" << x << " y=" << y << '\n';

return;
}
```

What you'll see is the values swap. Back in the test, or wherever you're calling *pbv::swap* from, no swap happens. We know why– the procedure variables[9] *x* and *y* are copies of their arguments, variables *a* and *b* from our gTests, not alias. They are two totally independent sets variables and the changes made to one set are not changes to the other. More formally, the swap parameters for *pbv::swap* do not share l-values with the arguments like they do in *pbr::swap*.

[9] yes. parameters are variables

Mutator and Pass by Reference Recap

We started with mutators because they introduce pass by value vs pass by reference semantics without any other baggage. It turns out that for I/O procedures to work they way we'd like them to, we must use pass by reference. So, let's review the key details before moving on:

- Reference parameter types are declared by appending an & after the usual type name used for value parameter types.
- Value parameters can take as their arguments any expression that can be interpreted as the appropriate type. This includes literal values, arithmetic expressions, and variables.
- Reference parameters can take as their arguments only variables of the appropriate type. They require something with an l-value.
- When objects are passed as arguments to a pass by value procedure parameter, then the procedure's parameters are copies of those objects.
- When objects are passed as arguments to a pass by reference procedure parameter, then the procedure's parameters are alias of those objects.

I/O Procedures

Our inspiration for I/O procedures is the C++ string procedure *getline*. It is an input effect procedure that takes two arguments, a stream and a string. The input is done relative to the stream and the result is stored in the string. We can tease out a couple of general principles from this: take the stream as a parameter and input procedures are also mutators. The later observation clearly means we'll be using pass-by-reference on input procedures. What isn't obvious is that streams must be passed by reference.

Stream objects are non-trivial. We know they maintain a buffer for the data being input or output as well as other state to control how the data in that buffer should be displayed. In this regard they

are COMPOUND DATA[10] and the copy cost incurred by pass by value might be costly. This alone might give us reason to use pass-by-reference. Creating an alias is cheap compared to copying multiple values. If this were our only concern, however, we'd use a technique called PASS BY CONST REFERENCE where we declare the stream a constant to prevent inadvertent copying. We'll see this pop us soon enough as it's a common occurrence when writing functional procedures for object types with potentially costly copy costs.

We're not just worried about the cost of copying stream objects though. Most of the operations we're going to use on our streams have the potential to modify the stream and we're likely to need those changes to propagate forward in the execution our program. Imagine using multiple input procedures to pull multiple inputs from a stream. We'd like each procedure to modify that stream by taking out the tokens relevant to its purpose. This is, in fact, exactly how the I/O operators behave. Each invocation of $<<$ or $>>$ is clearly modifying the same stream as they're used in succession.

[10] Like Racket's struct types

A Little Polymorphism

Throughout our time with C++, we'll be interested in three kinds of I/O streams:

- Standard streams like cout, cin, and cerr
- File streams for reading from and writing to files
- String streams for I/O-like management of strings and unit testing of I/O procedures.

At first glance we have a bit of a problem as each of these streams are defined as different classes. The standard streams are objects from the *ostream* and *istream* classes[11]. File streams are of objects from the *ofstream* and *ifstream* classes[12]. String streams are objects from the class *ostringstream* and *istringstream*[13]. So far we've had to be very specific about types. If we wrote a procedure for doing output to a ostringstream object, then it wouldn't work for *cout* because the types are different.

[11] both defined in the *iostream* library

[12] from the fstream library

[13] both in *sstream*

The C++ streaming I/O libraries leverage a key feature of Object-Oriented Programming[14], POLYMORPHISM. Polymorphism, in this context, refers to a variables ability to take on objects of different types so long as those types are clearly defined as proper subtypes, or extensions of the variables type. In this case the file and string stream libraries are defined as extensions of the stream libraries in such a way that the compiler recognizes them as subtypes of streams. More specifically, we can use ostringstream and ofstream anywhere

[14] OOP

we'd use an ostream and istringstream and ifstream where we'd use istream. These subtypes do everything that the supertype does and then some. The result is that if we write an output procedure such that it works for cout, an ostream, then we can pass it an ostringstream or an ostream instead and the compiler will allow it. If we write procedures instead for something like an ostringstream, then we cannot pass it the other output stream types because the compiler cannot guarantee that the procedure won't do something that is specific to the output string stream subtype.

To see this at work let's revisit our testing program from lecture notes 7. We'll set two goals for our new version:

1. Set it up so that pass no CLI arguments defaults to the REPL version, pass 3 runs a single test like our CLI command version, and passing any other number of arguments results in an error.

2. Use basic I/O procedures to reuse code between the two versions.

A side-effect of writing I/O procedures[15] is that we can unit test them using polymorphism. Procedures we intend to use on stream objects like *cout* and *cin* can instead be tested with string stream objects.

[15] and polymorphism

Input Procedures

Our *isWithin* tester needs to read in a 2D point and a circle radius. The REPL version does so from *cin*, an *istream*. The CLI-command version does so from *istringstream*s initialized with the contents of *main*'s argv array. Our new version will utilize two input procedures, *getPoint* to read the point coordinates and *getRadius* for the circle radius. When we're executing the REPL version, we'll read from *cin*. When we're executing the CLI-command version we'll have to setup one or more istringstreams with our data.

In both cases we can make use of the same pair of procedures.

```
/**
 * Retrieve (x,y) coordinates from an input stream
 * @param in input stream where user input is found
 * @param x variable that stores x coordinate
 * @param y variable that stores y coordinate
 * @return none
 * @pre in will produce two double tokens
 * @post the value of x and y have been changed to data from
 *  the stream and the stream has had two doubles read from it
 */
void getPoint(std::istream& in, double& x, double& y);
```

```
/**
 * Retrieve circle radius r from an input stream
 * @param in input stream where user input is found
 * @param r variable that stores circuit radius
 * @return none
 * @pre in will produce one double token
 * @post the value of r has been changed to data from stream in and in
 *   has had one double read from it.
 */
void getRadius(std::istream& in, double& r);
```

Let's start with the signatures. Both procedures take *istream* objects reference parameters– that is the rule for passing streams to procedures. The remaining parameters are all by reference because they're the variables into which the input is stored. Remember, we already observed that input procedures, and statements, are by their very nature variable mutations.

In terms of documentation, the most important thing to notice is probably the pre and post conditions. Preconditions we're familiar with. Here we're assuming that the streams contain enough data tokens to fill our variables. Postconditions are the things that have changed about the system after the procedure has been executed. When we're talking about procedures called for effect, then *postcondtions are where we document the expected effect.* It should be clear from this statement that you always, *always*, write postconditions for I/O procedures. For these procedures we need to clearly document the fact that we've modified variables and removed data from the stream.

Writing stub definitions for these procedures is the same as *swap*– when the return type is void, you simply *return* with no return value. Let's look at some tests for these procedures and see polymorphism at work.

```
TEST(getPoint,all){
  using namespace std;

  istringstream in{"2.3 4.5"};
  double a{0},b{0};

  getPoint(in,a,b);

  EXPECT_DOUBLE_EQ(2.3,a);
  EXPECT_DOUBLE_EQ(4.5,b);
  EXPECT_TRUE(in.eof());
}
```

First, let's point out the polymorphism. The procedure *getPoint* was written to take an *istream* object as its first argument. In these tests we passed it an *istringstream* object, namely *in*. The class *istringstream* is defined as a subtype, or extension, of the *istream* class, in a way the compiler recognizes and so this does, in fact, pass the type checker. Next, take careful note of the before-after nature of this test. We omit the before tests because we're clearly dealing with fresh initialized variables. If the before values of our objects weren't immediately clear, we would do some EXPECT tests to verify the values prior to mutation. The procedure itself is run as a single statement. It's not a part of a test statement. Finally, we check for the three expected changes: the two variables have new values and the stream is empty. The method *eof* returns true if the stream's contents have all been read. It's short for "end of file", even though the stream may not be reading from a file.

The implementation of these procedures is really straight forward. All we're really doing is a wrapping a little abstraction around the statements we used in our previous program.

```
void getPoint(std::istream& in, double& x, double& y){
  in >> x >> y;
  return;
}

void getRadius(std::istream& in, double& r){
  in >> r;
  return;
}
```

In practice, we're might to want to do some error checking and validation relative to the input values and the stream so that the post-conditions are strictly enforced.

Output Procedures

Writing output procedures generally follows suit with input procedures with adjustments for writing data rather than reading it. In both cases the stream is passed by reference. For output, we typically pass some variables whose values need to be written to the stream. Those variables should not be changed as a result of writing output[16] and are passed by value, not reference.

For our new program we want two output procedures: *reportResults* and *CLIError*. The former is used to write the results of *isWithin* in a clearly readable manner. The later is used to report usage errors to the user when they pass in the wrong number of CLI arguments.

[16] If you need data to look a certain way, format the output. Don't change the data.

```
/**
   * Print results of function isWithin to an output stream
   * @param out the stream where results are printed
   * @param x the x coordinate
   * @param y the y coordinate
   * @param r the circle radius
   * @return none
   * @pre r >= 0.
   * @post Results of isWithin(x,y,r) are written to the stream out
   */
  void reportResults(std::ostream& out, double x, double y, double r);

  /**
   * Print error message for bad CLI call to output stream
   * @param out output stream for error message
   * @param num_args number of arguments
   * @param cmd_name name of command executable
   * @return none
   * @pre none
   * @post Error message written to the output stream
   */
  void CLIError(std::ostream& err, int num_args, std::string cmd_name);
```

Once again, a stub definition just requires a return without a value. This is the general pattern for *void* return types. Let's skip writing it here and go right to tests.

```
TEST(reportresults,all){
    using namespace std;

    ostringstream out{""};
    string expected{"isWithin( 2.3 , 4.5 , 10 ) -> true\n"};

    reportResults(out,2.3,4.5,10);

    EXPECT_EQ(expected,out.str());

}

TEST(clierr,all){
    using namespace std;

    ostringstream out{""};
    string expected{"Given 1 arguments but expected 3.\n"};
    expected.append("Usage: test! x y r\n");
```

```
CLIError(out,1,string("test!"));

    EXPECT_EQ(expected,out.str());
}
```

Once again, polymorphism lets us use *ostringstream* to test these functions that we eventually intend to use with streams like *cout* and *cerr*. The only effect we're looking at is the addition of data to the stream. We can tests this by comparing the string, as returned by the *str* method, to a string we hard coded. The *CLIError* test shows you one strategy for managing large strings. The expected result string, *expected*, is built up in pieces. Part of the string is set as the initial value, then the *append* method is used to modify the string by adding the remaining characters to the end of the initial string.

The actual definitions for these procedures are, once again, basic abstractions over what we had in our programs originally.

```
void reportResults(std::ostream& out, double x, double y, double r){
    out << "isWithin( " << x << " , " << y << " , " << r << " ) -> ";
    out << std::boolalpha << TwoD::isWithin(x,y,r) << "\n";
    return;
}

void CLIError(std::ostream& err, int num_args, std::string cmd_name){
    err << "Given " << num_args << " arguments but expected 3.\n";
    err << "Usage: " << cmd_name << " x y r\n";
    return;
}
```

A new isWithin tester

Now that we have a library of I/O functions to use in developing our *UI* for *isWithin*, we can return to our updated program.

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

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

  if( argc == 1 ) // no arguments, drop to REPL
    while(true){
      std::cout << "Enter x & y coordinates :";
      getPoint(std::cin,x,y);
      std::cout << "Enter circle radius: ";
      getRadius(std::cin,r);
```

```
      reportResults(std::cout,x,y,r);
    }

  else if( argc == 4 ){  // 3 arguments for isWithin

    istringstream clistrm{string(argv[1]) + "  "+
    string(argv[2]) + " " +
    string(argv[3])};


    getPoint(clistrm,x,y);
    getRadius(clistrm,r);
    reportResults(cout,x,y,r);
  }

  else{ //argc is not 4 or 1
    CLIError(cerr,argc-1,string(argv[0]));
    return 1;
  }

  return 0;

}
```

We used a basic conditional to branch on *argc*. When only the program name is used at the CLI, we drop into our REPL program. When three additional arguments are passed, we assume their doubles for *isWithin* and carry-out the CLI version. Otherwise we got a weird number of arguments and need to print an error message. Notice that we're leveraging polymorphism in order to reuse our I/O functions here as well. The input procedures are called both with *cin* and an *istringstream*. It's also worth noticing that we opted to create one long istringstream with all three input values for our CLI version. The fact that our input procedures modify the stream let's us THREAD the object through multiple procedures, modifying it as we go. The last thing we should notice is the variable declaration. When you're declaring multiple variables of the same type, you can do so in a single line separating each initialization with a comma like this

```
TYPE NAME{VAL},NAME{VAL},... ;
```

Procedure Chaining and Returning References

If you look at the signatures for the string function *getline*, you'll notice that they return a reference to an istream. This is a common

practice for I/O procedures and in several other scenarios we'll see this semester. The intent is usually to allow *chaining* of procedure calls[17]. A perfect example of this is the streaming I/O operators.

[17] or operators or method calls

Both << and >> return references to streams and by doing so they allow us to do things like this:

```
cout << 1 <<  3 << 'a' << b;
```

Let's parenthesize this statement to highlight order of execution:

```
((((cout << 1) <<  3) << 'a') << b);
```

First we put 1 to *cout*. The return result is *cout with 1 added to the buffer*. Let's just call that *cout'*. Now we effectively have *cout'* << 3 to do next. This process continues until the end. The key observation is that the modified stream returned by one operator becomes the stream input to the next operator. This is operator *chaining*.

To see how we can leverage this with procedures, let's go back and redefine our input procedures to return references and then chain them together in our main program. First we need to revisit the basic declaration and documentation.

```
/**
 * Retrieve (x,y) coordinates from an input stream
 * @param in input stream where user input is found
 * @param x varibale that stores x coordinate
 * @param y variable that stores y coordinate
 * @return reference to the input stream
 * @pre in will produce two double tokens
 * @post the value of x and y have been changed to data from
 *  the stream
*/
std::istream& getPoint(std::istream& in, double& x, double& y);

/**
 * Retrieve circle radius r from an input stream
 * @param in input stream where user input is found
 * @param r variable that stores circuit radius
 * @return reference to the input stream
 * @pre in will produce one double token
 * @post the value of r has been changed to data from stream in
 */
std::istream& getRadius(std::istream& in, double& r);
```

The difference is fairly subtle. We've added a return type, a reference to an istream, and we've documented that return type. Nothing else about the procedure changes.

Now let's stub these procedures. Remember we *must* return a stream reference. The smartest thing to do, the thing we'll ultimately do anyway if we want chaining to occur, is to return the stream argument.

```
std::istream& getPoint(std::istream& in, double& x, double& y){
   return in;
}

std::istream& getRadius(std::istream& in, double& r){
return in;
}
```

Testing is a little bit more complicated now. Before writing the tests lets lay out our expectations in plain English. We have three. The first two are hold overs from before, the last is to account for the return value.

- The variable arguments must be mutated relative to the contents of the stream
- The stream has had data removed from it.
- The return value is literally the same stream as the input stream.

The new expectation is a different beast than we've seen before. In the past our concern was with equality of value, specifically *r-value*. Now, to enable chaining we need to be certain that the *l-value*, the address of the input object and the returned object are the same. Now let's say all of this in C++.

```
TEST(getpoint,v2){
    using namespace std;

    istringstream in{"2.3 4.5"};
    double a{0},b{0};

    EXPECT_EQ(&in,&getPoint(in,a,b));

    EXPECT_DOUBLE_EQ(2.3,a);
    EXPECT_DOUBLE_EQ(4.5,b);
    EXPECT_TRUE(in.eof());
}

TEST(getradius,v2){
```

```
    using namespace std;

    istringstream in{"10"};
    double a{0};

    EXPECT_EQ(&in,&getRadius(in,a));

    EXPECT_DOUBLE_EQ(10.0,a);
    EXPECT_TRUE(in.eof());
}
```

Most of the tests are hold-overs from before that check our first two expectations. The new element is checking the return value. We can manage this in the same way we did for our functional procedures by comparing the return value of the procedure to an expected value. In both cases we want to be certain that *in* and the return value are the same. I've demonstrated two ways of approaching this test.

The test for *getPoint* utilizes the "address of" operator & to get the actual memory address of the two objects. So *EXPECT_EQ(&in,&getPoint(in,a,b))* literally compares the address of the two objects and passes if and only if they're the same. You can use the operator & on any object or variable to effectively get the *l-value* for the object. The effect of this operator also gives you some insight as to the source of the pass-by-reference syntax.

Finally, we implement.

```
std::istream& getPoint(std::istream& in, double& x, double& y){
    in >> x >> y;
    return in;
}

std::istream& getRadius(std::istream& in, double& r){
  in >> r;
  return in;
}
```

The only real difference here is the return statement. Just like when passing objects as reference arguments, it's not your job to do anything special to the return variable. By declaring the return type a reference and returning an object, the system will manage the r-value/l-value distinction.

So, what did all of this buy us? Not much really. For our problem we got the option to chain the input procedures together. It's not clear that this is an improvement really. However, this technique is critical to some code we'll be writing soon, so this is a good opportunity to being our exposure to the idea of returning references and

chaining procedures.

Here we modify our REPL prompt slightly to leverage chaining in both the REPL and the CLI versions.

```
using namespace std;

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

  if( argc == 1 ) // no arguments, drop to REPL
    while(true){
      cout << "Enter point x & y coordinates and a circle radius r: \n";
      UI2::getRadius(UI2::getPoint(cin,x,y),r);
      UI::reportResults(std::cout,x,y,r);
    }
  else if( argc == 4 ){     istringstream clistrm{string(argv[1]) + "  "+
        string(argv[2]) + " " +
        string(argv[3])};

    UI2::getRadius(UI2::getPoint(clistrm,x,y),r);
    UI::reportResults(cout,x,y,r);
  }
  else{ //argc is not 4 or 1
    UI::CLIError(cerr,argc-1,string(argv[0]));
    return 1;
  }

  return 0;

}
```

Again, there is really no significant gain from our change to chainable procedures for the problem we're working on here. On the other hand, composition of functions is clearly a boon to our ability to program. Composing systems is a corner stone of computing. What we've exposed here is composition while passing state, namely the stream, through the thread of execution. Chaining procedures like we did here not only allows us to compose two lines into one, but it allows us to propagate effects across the composition. Using this technique we can next effects in the same fashion we nested functions.