# COMP 161 - Lecture Notes - 14 - Revisiting I/O Procedures

*Spring 2014*

In these notes we explore more involved I/O procedures while also looking at generalized I/O procedures.

## Get and Validate Procedures

So far we've just looked a very simple input methods that just get input. We'd then wrap that in validation code in the *main* procedure. A really good idea is to move the validation code out of main and over to the input method. This allows us to test our validation code using gTest. It also makes the validation code usable in other programs. Let's look at an example.

```
/**
 * get5to15int gets an integer from [5,15] from
 * the standard input.
 * @param uInRef is a reference to the int variable for user
 *    input
 * @return none
 * @pre none
 * @post the value in the variable referenced by uInRef is now
 *  a value in [5,15]
 */
void get5to15int(int &uInRef);
```

So far the only thing that's changed is the stronger post condition. Time to write tests. Let's get creative with these tests and borrow some ideas from how we tested randomized procedures. Rather than write tests for specific inputs. Let's write a test that runs this procedure 5 times[1]. We, the user, than then use those 5 executions to test some good and bad situations. We'll still document some possible values, but the tests will not be dependent on those values.

[1] we can use some other $n > 1$ if we want

```
TEST(get5to15int,all){
  int uin(0);

  for( int count(0); count < 5; count++){
    uin = 0;
    EXPECT_EQ(0,uin);
    // try bad inputs like: 2,3,-15, 25 ,100
    // and good inputs like: 5,6,7,10,15
    get5to15int(uin);
    EXPECT_TRUE(5 <= uin && uin <= 15);
  }
}
```

These tests are not dependent on specific values entered, but if our procedure properly validates user input, then this should only pass after we've entered exactly 5 valid inputs. We're free to test bad inputs all we want as our procedure shouldn't terminate and return until we've entered something valid. There's still the chance of some infinite loops, but such is life in programming.

Now we implement. Let's use the more involved validation loop from previous notes and check not just for bad integers but bad input in general.

```
void get5to15int(int &uInRef){
  using namespace std;

  do{
    // prompt
    cout<<" Enter an integer from [5,15]: ";
    cin >> uInRef;

    //error check and validate
    if( cin.fail() ){
      //input went poorly.
      // error prompt
      cout<<"Bad input. I got "<< uInt << " try again.\n";
      // set to invalid value to force loop.
      uInRef = 0;
       // clear error flags
       cin.clear();
       // ignore everything else left in the stream
       cin.ignore(numeric_limits<streamsize>::max());
     }
     else if( uInRef < 5 || uInRef > 15 ){
       // int, but out of range
      cout<<"Bad input. I got "<< uInt << " try again.\n";
     }

  }while( uInRef < 5 || uInRef > 15 );
}
```

This validation loop has some new C++, but the overall logic is the same as the loops we saw before. First, we cleared and flushed the stream in the conditional block corresponding to the failed input.

Second, we used *numeric_limits<streamsize>::max()* to get the system's value for the largest possible stream size. This requires that we include the *limits* library[2].

[2] #inlclude<limits>

This design has lots of opportunities for helper procedures. Things like predicates for checking the validity of the value make a lot of sense. Also, some output procedures for the prompts and error messages would hide those statements as well. However, the boolean expressions and output prompts in this particular problem are very simple. So, maybe the helpers would be overkill in this case. It's hard to know when to write a helper and when to use in-place statements. When in doubt, err on the side of helper procedures. They break up the problem in to smaller problems, are testable, and reusable. These are all good things in programming.

## *Generalized I/O procedures*

What if we wanted to output the same thing to a file and to the standard output[3]? We could write two procedures, one for *cout* and another for the file. However, we can also take advantage of some Object-Oriented Programming features and write one procedure to do both. Let's see how.

[3] think program logs

Say we want to print an integer score to a file as well as to the standard output. Something like *You got 50 points* or *You got 5 points*. There are two variables involved here, the first is obviously the score. The second is the output stream. At first it seems like we have a problem. The file stream is of type *ofstream* and *cout* is of type *ostream*[4]. It turns out that all *ofstream* objects can also be viewed as *ostream* objects by the compiler. That is, the compiler recognizes a HIERARCHY OF TYPES built into the streaming I/O classes. In this hierarchy the *ofstream* class is a SUB-CLASS of the *ostream* class. Equivalently, we can say that *ostream* is a SUPER-CLASS of *ofstream*.

[4] `http://www.cplusplus.com/reference/iostream/cout/`

The ability to create these type hierarchies is called called TYPE INHERITANCE. It is a major feature of object-oriented languages like C++. We're starting to see its benefits with this problem. We get to write a procedure that takes an *ostream* object as an input and can then pass pass it *cout*, an ostream, or an instance of an *ofstream*.

Objects in the *ofstream* class don't just inherit the *ostream* class type. They also inherit the core *ostream* class functionality. If you look at the *ofstream* documentation[5] you'll notice some methods listed as *Public member function inherited from ....* These methods are safe to use in our new output procedure as they come from the *super-class*. The methods under the *Public member functions*, no "inherited from", listing are specific to *ofstream* objects. This means they will not work if called on *cout* which is an *ostream* but not an *ofstream*. Because this

[5] `http://www.cplusplus.com/reference/fstream/ofstream/`

listing includes or open/close methods, we'll need to be certain our files streams are open before we pass it to the output procedure; This means some new preconditions.

We're almost ready to document and declare this procedure, but we've never passed streams to procedures before. When passing streams we *must* pass our objects by reference. This makes sense. We are, after all, planning to create an effect relative to the stream. Let's declare our score writing procedure.

```
\**
 * outputScore will print the score to a stream
 * @param score is the score as an int
 * @param outStreamRef is a reference to the output stream
 * @return none
 * @pre score is a valid score and outStreamRef refers to an
 *    ostream already opened on a device or file
 * @post the score message has been written to the stream
 */
void outputScore(int score, std::ostream &outStreamRef);
```

And now some tests. Let's do one case for *cout* and one case for a file.

```
TEST(outputScore,toStdOut){
  using namespace std;

  // You got 50 points
  outputScore(50, cout);

  // You got 5 points
  outputScore(5,cout);

  // You got 10 points;
  outputScore(10,cout);
}

TEST(outputScore,toFiles){
  using namespace std;

  ofstream testLog("outputScore-tests.txt");

  // You got 50 points
  outputScore(50, testLog);

  // You got 5 points
  outputScore(5, testLog);

  // You got 10 points;
  outputScore(10, testLog);

  testLog.close();

}
```

In truth, if the *cout* tests pass, then the only reason the file tests will fail is because the file fails to open. A more interesting possibility is that if it works for the file, it should definitely work for *cout*. We could just test by writing to the file and then inspecting the file at the command line. If this works, then *cout* will work just fine as well. Not using cout in our tests keeps the test output from clogging up the standard output, which is already getting pretty full with gTest output. If we wanted to take this a step further, then we might consider reading in the contents of *outputScore-tests.txt* and testing them against our expectations. This might be overkill, but let's see what it might look like. For this problem we'll need to use getline[6] because our output is a string with spaces in it. Using *cin* would on get us the string up to the first space.

[6] http://www.cplusplus.com/reference/string/string/getline/

```
TEST(outputScore,toFiles){
  using namespace std;

  ofstream testLog("outputScore-tests.txt");

  // You got 50 points
  outputScore(50, testLog);

  // You got 5 points
  outputScore(5, testLog);

  // You got 10 points;
  outputScore(10, testLog);

  testLog.close();

  // now read in the data and test
  ifstream postTestLog("outputScore-tests.txt");
  string result("");

  getline(postTestLog,result);
  EXPECT_EQ("You got 50 points\n",result);
  getline(postTestLog,result);
  EXPECT_EQ("You got 5 points\n",result);

  getline(postTestLog,result);
  EXPECT_EQ("You got 10 points\n",result);
}
```

Again, this might be overkill; your eyes do a pretty outstanding job of verifying expected output. The advantage of this approach is that its *completely automated*. If your tests pass, then we know our output procedure worked and we don't need to look at anything other than the green text of gTest output. So, while this might take longer to code, verifying that the tests passed will typically go much faster. This could also let us save human intervention for *usability testing*[7] rather than correctness testing.

[7] http://en.wikipedia.org/wiki/Usability

OK. We declared, documented, and wrote tests. Let's implement.

```
void outputScore(int score, ostream &outStreamRef){

  outStreamRef << "You got " << score << " points\n";
  return;

}
```

Other than passing a stream and using a stream other than *cout*, nothing new is happening in this procedure.

We could now apply this same general technique to input procedures. Rather than use the *ostream* type, we'd need use the *istream* type which covers *cin* as well as *ifstreams*. We'd still have to manage opening and closing file streams outside of the procedure, but we could at least create a generalized input procedure. We could apply the same automated testing technique to generalized input procedures as well. First create a file by hand, then read from that file rather than *cin*. Think of it as pre-typing your intended input.