COMP220 - Project 1

Fall 2015

Abstract

For your first project you'll be working on an extension to exercise 15 from chapter 6 of the text. This problem demonstrates the *finite state machine* abstraction, which is a fundamental principle in computing and algorithm design. Your finished project is a library for working with mathematical number types and a simple program that tests and demonstrates the usage of that library.

1 Project and Document Overview

You'll be developing a library called *Numbers* for dealing with different mathematical number types. This library can and should be spread across several files. Each class should get its own header,implementation, and tests. Any procedures not associated with a specific class should also get grouped together in another "sub-library" with its own header, implementation, and test trio of files. Section 2 describes the different parts of the library.

In addition to testing your library with gtest unit tests, you will write a basic program that uses your library. This program lets you test the overall usability of your library and test how different units work in tandem. Designing libraries is an exercise in tool building and some applications let you gauge the effectiveness of your tool in the wild. This program is described in section 3.

This project encompasses two lab periods. In the first lab you're required to begin the process of converting the description of the project given in this document to a high-level, plain language list of program requirements. This vital part of the process lets you establish the discrete elements of the overall program. From there you can begin the process of translating that to and class method declarations, documentation, tests, and implementation. This lab assignment is discussed in more detail in section 4.

The final section of this document breaks down the grading rubric and clearly states the due dates.

2 The *Numbers* Library

The core task for this project is the development of a library for working with mathematical number types. The goal of this library is to enable a programer to work in terms of the mathematical number types integer, rational, real, and complex. For our purposes, an integer is equivalent to the C++ *int* type and a real is equivalent to the C++ *double* type. We'll not discuss these types here, so if you have questions about integers and reals, you need to seek out the instructor and ask them. Much the required functionality for these types is already built into C++. Rationals are discussed and implemented in Chapter 6, section 3 of the text. Complex numbers are likely new to you, so we'll discuss them briefly later in this document.

All the definitions and declarations for this library should be placed within a namespace called *Numbers*. The library includes several thing:

- A *Rational* class for rational numbers.
  Much of this is complete in the text already. You'll mostly need to modify and add to the existing code.

- A *Complex* class for complex numbers.
  You'll develop this class from scratch.

- A *NumberScanner* class for reading string representations of numbers.
  This class is essentially a more constrained version of the *TokenScanner* class from text. It is designed to scan number tokens rather than more arbitrary tokens.

A programmer using this library should be able to:

- Convert between these types as appropriate through the combination of overloaded constructors or by using built-in C++ static casting.
- Carry out addition, subtraction, multiplication, and division for two values of the same type using overloaded operators.
- Get the absolute value, aka Magnitude, of any number type through a combination of the *cmath* library or class methods called *abs*.
- Compare two values of any two types for numerical equality or lack thereof using overloaded instances of the operators $==$ and $! =$.
- Output all types to a stream using the $<<$ operator.
- Input all types from a stream using the $>>$ operator.
- Produce a string-based representation of the number.
- Read a string representation of the number as the appropriate number type using the NumberScanner class
- Read a stream or string of separated values (i.e. comma or whitespace separated) to a vector of the most appropriate type using NumberScanner.

Any library defined classes should also provide the expected set of constructors, selectors, mutators, and other necessary interface methods.

2.1 Integers and Reals

You do not need to re-implement any integer or real functionality that's already built-in to the C++ *int* and *double* types. For the most part, you'll just have to integrate ints and doubles with the new types you're defining, rationals and complex numbers.

2.2 Rationals

You should be able to use most of the Rational class found in the text as is. We do, however, want to get rid of the use of the *friend* modifier used to implement the overloaded operators. Beside this change, you may, or may not, need to extend the existing implementation with additional methods.

2.3 Complex Numbers

An imaginary number is any real multiple of $i = \sqrt{-1}$. The imaginary unit i has the important property that $i^2 = -1$ which helps smooth out some rough algebraic edges. We typically write imaginary numbers:

```
i
3i
-2.5i
0.25i
```

The i is effectively a named constant like π or e. The imaginary number i derives its name from the fact that you cannot actually take the square root of negative number so must simply imagine that $\sqrt{-1}$ is an actual numerical value. In doing so we can start to deal with square roots of negative because $\sqrt{-n} = \sqrt{-1 * n} =$

$\sqrt{n} * \sqrt{-1} = \sqrt{n}i$. It's probably not at all clear to you why we'd play this kind of of "what if" game with numbers, but doing so leads to some very, very important ideas in mathematics.

A complex number, the number type you need to deal with directly, is the sum of a real number a and an imaginary number bi written as $a + bi$. For example,

```
0.5 + 3i
-0.25 + 0.5i
3 + 0.5i
3 - 0.5i
0 + i
1 + 0i
```

For the imaginary number $z = a + bi$, we refer to a as the real part of z and b as the imaginary part or z. From this perspective we can look at complex numbers as pairs of real numbers. Two complex number are equivalent if and only if their real and imaginary parts are equivalent.

The basic arithmetic operations are all defined in terms of standard real arithmetic involving a and b. They are well documented on wikipedia: `https://en.wikipedia.org/wiki/Complex_number#Elementary_operations`. The definitions are provided below without explanation.

$$\begin{aligned}(a+bi)+(c+di) &= (a+c)+(b+d)i \\ (a+bi)-(c+di) &= (a-c)+(b-d)i \\ (a+bi)*(c+di) &= (ac-bd)+(bc+ad)i \\ (a+bi)/(c+di) &= \left(\frac{ac+bd}{c^2+d^2}\right)+\left(\frac{bc-ad}{c^2+d^2}\right)i\end{aligned} \tag{1}$$

Notice that when the right hand operand of division is 0 or $0 + 0i$, then you encounter a division by zero problem just like with other number types.

Another important operation for complex numbers is that of complex conjugation. This is typically written as $\overline{z}$ and sometimes as $z^\dagger$ for complex number $z = (a + bi)$.

$$\overline{(}z) = \overline{(a+bi)} = a - bi \tag{2}$$

Finally, with an appeal to geometry, we can define the absolute value, or magnitude, of a complex number. For $z = (a + bi)$,

$$|z| = \sqrt{z\overline{z}} = \sqrt{a^2+b^2} \tag{3}$$

The above discussion of complex numbers tells us that in addition to a suite of constructors, a Complex number class should minimally provide the following methods to enable basic mathematical work with complex values:

- Selectors and Mutators *Re* and *Im* for the real and imaginary parts.
- An *isZero* predicate that determines if the
- Predicates *isReal*, *isImag*, *isComplex*, and *isZero* that determine if the number is real (zero imaginary part), imaginary (zero real part), complex (non-zero real and imaginary part), or zero.
- A method *conj* returning the complex conjugate of the complex number
- An *abs* method which computes the absolute value of the complex number
- Operators for addition, subtraction, multiplication, and division of complex numbers
- Equality and non-equality operators

2.4 Equivalencies and Conversion

Mathematics tells us that the sets of integer, rational, real, and complex numbers in that order are subsets of one another. In mathematical notation we'd write this as:

$$\text{Integers} \subset \text{Rationals} \subset \text{Reals} \subset \text{Complex} \tag{4}$$

What this means is that all integers are also rationals, all rationals are also Reals, and all Reals are also Complex. For example, 25 can be written as the rational $\frac{25}{1}$, which can be written as the real 25.0, which can be written as the complex $25 + 0i$. While we can always convert from a subset (type on the left of $\subset$) into a value from one of its supersets (type on the right of $\subset$), the reverse is not always true. Conversions from superset to subset suffer the same loss of precision problems as converting an integer to a double. For example, while we can convert $\frac{5}{1}$ into the integer 5, we cannot convert $\frac{17}{3}$ into any integer value without losing a rational quantity, namely $\frac{2}{3}$. It's worth noting that there are 6 possible left to right conversions for these four types. Some you'll need to write and some that already exist.

The compiler will implicitly convert ints to doubles whenever you use both types in an arithmetic expression, i.e. $2.0 + 5$ actually computes $2.0 + 5.0$. We can also explicitly convert values using *static_cast* like this:

```
EXPECT_DOUBLE_EQ(5.0,static_cast<int>(5));
EXPECT_DOUBLE_EQ(5.0,static_cast<int>(2));
EXPECT_DOUBLE_EQ(2.5,static_cast<int>(5)/static_cast<int>(2));
```

To provide conversions to and from rational and complex numbers you should provide constructors and methods including, but not necessarily limited to, those implied by the following:

```
EXPECT_EQ(Numbers::Rational(5,1),Numbers::Rational(5));
EXPECT_DOUBLE_EQ(Numbers::Rational(5,2).toReal(),2.5);
```

2.5 Writing Numbers as Strings

Exercise 15 in chapter 6 of the text describes how to capture the rules for writing a Real valued number into a finite state machine which in turn provides the logic basis for traversing the characters of a real value written as a string. We'll go over this important idea in class and discuss how to work a state-machine into a traversal pattern and thereby into a procedure.

You'll need to establish a state-machine for character by character traversals of integers, rationals, and complex numbers. The plain English descriptions of the rules for writing these strings are given below. We'll assume for this project that no well-written number string uses spaces.

- *Integers*: Integers optionally begin with a $-$ sign if they're negative. They then continue with a series of one or more digit characters.
- *Rationals*: Rational numbers optionally begin with a $-$ sign if the number is negative. They then continue with one or more digits, then a / then finish with one or more digits.
- *Complex Numbers:* Complex numbers begin with a real number, then a $+$ or a $-$, then a positive real number, and finally the i. Notice that the $+$ or $-$ in the middle determines the sign of the imaginary component. We do not expect the imaginary coefficient to be explicitly negated so strings like $3.2 - -4.0i$ are not allow and should instead be written as $3.2 + 4.0i$.

2.6 *NumberScanner*

A NumberScanner is a very constrained version of the TokenScanner discussed in chapter 6 of the text. Its purpose is to enable the tokenization and thereby traversal of a string or stream (of strings) in which textual representations of numbers are separated by a specific character. For example, a common format for storing data in files is *comma separated values* (aka csv) in which each value (number in our case) is separated by a comma. These are all csv number sequences:

```
1,2,3,4
1,2.0,3/1,4.0+0.0i
2.3-1.0i,1/2,-5/7,4
```

If one were trying to read a series of comma separated integers out as reals, or doubles, and store them in a Vector of doubles, then they the might expect to use the NumberScanner like this:

```
NumberScanner some_ints("1,2,3,4,5,6");
// set seperator character
some_ints.setSeperator(',');
// set output type
some_ints.setOutputType(Numbers::NumType::REAL);

Vector<double> nums;
while( some_ints.hasNext() ){
  nums.add( some_ints.next() );
}
```

This exposes several key requirements of the Scanner:

- The ability to determine if there's another token with something like *hasNext*
- The ability to get the next token with something like *next*.
- The ability to specify the separator character.
- The ability to set the expected output type.

This type of process is not the only one a user might want NumberScanner to support though. As our example csv sequences illustrated, streams and strings might contain a mixture of the four number types, and scanning such a mixture might need to produce the most appropriate type for each value. This type of flexibility implies several other methods for working with the type of the next token:

- A method that returns the type of the next token and/or a predicate method that can be used to determine if the next token's most appropriate type is a specific type. Something like *some_ints.nextType()* versus *some_ints.nextIsInt()* or *some_ints.nextIs(Numbers::NumType::INTEGER)*.
- A method that returns the next token as a specified type rather than the current Scanner default. For example, *some_ints.nextInt()* would attempt to return the next token as an int even if the user previously set the output type to Real.
- The ability to set the expected output type to "most appropriate" which could mean doing something like *some_ints.setOutputType(Numbers::NumType::MIXED*.

Methods such as those listed above would enable traversals like these:

```
NumberScanner nums("1 2.3 -5/2 4-5i");
nums.setSeperator(' ');
nums.setOutputType(Numbers::NumType::Mixed);
while( nums.hasNext() ){
  ... nums.next()...
}
// or programmer checked types/typed-selection
while( nums.hasNext() ){
if( nums.nextIsInt() ){
   ... nums.nextInt()...
}
else if( nums.nextIsRational() ){
   ... nums.nextRational()...
```

```
}
else if( nums.nextIsReal() ){
   ... nums.nextReal()...
}
else{ // if( nums.nextComplex() {
       ... nums.nextComplex() ...
}
}
```

Finally, an enum class for the types (int, real, rational, complex, mixed) seems useful as well.

2.6.1 On Errors

How should *NumberScanner* handle this sequence?

```
this,is,a,sequence
```

It shouldn't. We could pass the buck and say, "if you don't give me proper numbers bad things will happen." A more robust solution is to determine if and when a number is bad and generate a run time error.

We'll discuss the basics of *throw*ing errors and testing for errors in class. The short version is we can use statements like these to generate an error:

```
throw std::runtime_error("error message here");
throw std::logic_error("error message here");
```

The the runtime_error and logic_error classes are documented on cplusplus.com.

Google's testing framework has test statements to check for an error and verify the type if needed.

```
EXPECT_THROW( throw std::runtime_error("Whoops!") , std::runtime_error);
EXPECT_ANY_THROW(throw std::logic_error("Whoops!"));
```

These test types are documented in the gTest advanced guide: `https://code.google.com/p/googletest/wiki/AdvancedGuide#Exception_Assertions`

Any procedure or method that might throw and error should document that error with an @throws annotation.

3 Be Your Own Client

Library design is about predicting and understanding the needs of the users, or clients, of the library. Sometimes you are the client and sometimes not. For this project, we're imaging that you're not the client. We have no real-world application in mind. This library is not something you cooked up to solve some other problem. Your goal is simply to enable a more robust set of number types. Even still, we should come up with some toy problems in which we will be the client of our library. This lets us test the actual usability of our design and acts as a way to test the big picture. Building a library is building a tool and tools should be as easy to use as possible.

For this project, you should write a CLI program that takes as a command-line argument a file name. That file contains a comma-separated series of numbers of mixed type. Your program should first read the file contents into a vector of the least superset that can contain all the numbers. For example, if the file contains a mixture of integers and rationals, then the vector should be all rationals because rationals can express all the numbers in the file. Put another way, if the file contains at least one rational, then our vector contains rationals, if there's at least one real, then it contains reals, if there's at least one complex number, then it contains complex numbers. Once you've read the file into a vector, you want to determine the max absolute difference between any pair of numbers in the vector. That is, for vector v and for all indices i,j $< v.size()$ with $i \neq j$, we want:

$$\max |v[i] - v[j]| \tag{5}$$

This toy program is just complex enough that you're going to want a least a few helper procedures for this toy program. This also means it's just right for getting a realistic feel for how usable the library is and how it will perform for the client.

4 First Lab Assignment: Find the Trees

Due Monday 11/2 at the start of Class

We spend a lot of time just working procedure design and implementation. These little one-off procedures have their own challenges but the lack the problems that come with larger-scale programs. To practice managing the complexities of scale, we turn to projects like this one.

Like any realistic program, there is *a lot to do* . It's easily overwhelming if you look at the sum of the parts, the forest. We practice on individual procedures for a reason. Step one of managing the complexity of a program is to identify the parts, the trees within the forest. You do not have to know how all of these discrete parts fit together yet, you just have to know they exist and that they're required for the final product. We'll call these things the requirements.

For this lab you need to begin laying down a list of requirements in mostly plain English. Again, don't worry too much about how things fit together or even relationships between the requirements, that comes next. You just need to list down each little thing that needs to happen. Don't get hung up on how these things happen. Just identify tasks. The plan is to turn these tasks into methods and procedures.

It's clear from the description that we need three classes: Rational, Complex, and NumberScanner. Start by listing all the things you need to be able to do with each of these classes. Anything that doesn't seem to fall within one of these classes can go in a fourth list for "other". Finally, write a list for the requirements for the main program. To be clear, this is a list of five lists: Rational, Complex, NumberScanner, Other, and Main.

Both in lab and on Monday in class you'll be put in groups to compare and contrast your requirements lists. There is no one perfect list. We all see problems a bit differently. You're conferring with group mates in the hope that having many eyes on the problem will see all the angles that need seeing. *Hand in a hard copy of your requirements list at the start of class on Monday.*

4.1 Next Steps and Getting Done

Once you have your base requirements list, you should convert every item on the list to a procedure or class method, document and declare those methods in source files, and write tests for each of them. By the time you finish this you should start to see how things will come together. If not, don't worry. There's still time for that.

Start implementing all of the things you just set down in the source documents. As you do this you might decide that a helper procedure is needed, go ahead and (1) add it to your requirements list then (2) document, declare, and write tests for that helper. You are effectively iterating on the design as laid out by the requirements list. In the case of the Rational class, you need to verify that the implementation provided by the text meets your requirements and modify it where it does not.

I recommend you get working on the main program early in the process. It gives you a sense of the big picture. If you get it working for a diverse set of number sets, then you'll likely have a good chunk of all the library done and working. Just remember, that this problem doesn't cover all the use cases of the library and the goal is a robust library, not the main. Library design is tough because they themselves do nothing, they enable others to do something. This is why writing toy programs for your library is important. It keeps you focused on the goal, to enable programers to write new and better code.

5 Logistics

The following are notable dates for this project:

Date	Activity
10/29	Project 1 Lab
11/2	Project 1 Lab **due** at start of class
11/5	In-Lab Work Time
11/9	Project **due**. Submit all source code via handin as assignment *proj1*

Your grades are based on the following criteria:

Points	Category
15	Neatness and Good Style
25	Library Design Completeness
25	Library Implementation Correctness
15	Main Program Design Completeness
10	Main Program Implementation Correctness

Completeness means that all the necessary code is accounted for by way declared and documented classes and procedures with complete test coverage. It has nothing to do with implementation, with tests passing. It's all about *what* and not all all about *how.* At this point in your studies there is a very high bar for completeness. It's worth 40 of the 100 points of this assignment. If you're clearly missing vital functionality or you're packing too much functionality into a small set of procedures rather than using helpers, then you can expect to receive no better than a C or C+.

Correctness means that your code compiles and that a set of tests with complete coverage would all pass. The main procedure should do what it's supposed to do as well. Correctness is purely a measure of completed implementation. If you submit code that doesn't compile, you can expect to receive no better than a D+. This is an exception to the above rubric.

To receive a B- or better your program design should be complete in the sense described above. This means that all the requirements are accounted for by declared and documented procedures and methods with complete tests. At that point your grade is a function of how much of that design is implemented and the quality of that implementation. The key observation here is that you're expected to identify everything that needs doing in a problem and be able to express those things as procedural elements of a program. Once you know what to do, then and only then should you set about determining how to do those things.