PIC 10B Section 1 - Homework # 5 (due Sunday, May 5, by 5 pm)

You should upload each .cpp and .h file separately and submit them to CCLE before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h and .cpp.
Be sure you upload files with the precise name requested of you and that it matches the names you used in your editor environment otherwise there may be linker and other errors when your homeworks are compiled on a different machine. Also be sure your code compiles and runs on Visual Studio 2017.

At the end of this work you should submit a header file **Complex.h**, with the class interface, function declarations, and constexpr function definitions, and **Complex.cpp** that includes all the other implementations.

Do not submit a main routine!

COMPLEX NUMBERS

Before you panic (if you are math-phobic), this is not really about the math. All the math information you need to solve the homework is given in this document. Just read it carefully. We're effectively going to work with special numbers with special rules (that are given to you) for how +, -, *, and / work.

Background: real numbers are the numbers everyone is familiar with in daily life: -4.3, 19, $\sqrt{27}$, π, etc.

The **complex numbers** are pairs of real numbers (a, b) often written in the form

$$a + bi$$

where i is called the "imaginary" unit. If z is a comlex number and $z = a + bi$, we call a the "real" part of z and b the "imaginary" part of z.

There are some special rules for their arithmetic. You don't need to know why, all you need to know is what these rules dictate. Suppose that a, b, c, and d are real numbers. Then:

- $(a+bi)+(c+di)=(a+c)+(b+d)i$ so, for example $(4+3i)+(1-6i)=(4+1)+(3-6)i=5-3i$.
- $(a+bi)-(c+di)=(a-c)+(b-d)i$ so, for example, $i-(2-3i)=(0+1i)-(2-3i)=(0-2)+(1-(-3))i=-2+4i$.
- $(a+bi)(c+di)=(ac-bd)+(bc+ad)i$ so, for example, $(1+i)(2+i)=1+3i$.
- $(a+bi)/(c+di)=\frac{ac+bd}{c^2+d^2}+\frac{bc-ad}{c^2+d^2}i$ so, for example, $\frac{1+i}{3+4i}=\frac{7}{25}-\frac{1}{25}i$.

Given a complex number $z=a+bi$, we refer to the process of switching the sign of the imaginary part as *complex conjugation* and write it as $\bar{z}=a-bi$.

In this homework, you will need to write a **Complex** class to represent complex numbers and overload many operators for the **Complex** class to behave just like "normal" arithmetic. You should also make heavy use of **constexpr**. Everything that can be **constexpr** should be.

Firstly, your **Complex** class should have

- two floating point (use **double**!!!) private member variables to store the real and imaginary parts;
- *a single constexpr constructor* that, through the use of default arguments, behaves as a default constructor that sets both the real and imaginary parts to 0; when given a single **double** as a parameter, the real part is set to this value and the imaginary part is set to 0; and when given two **double** inputs, it sets the real and imaginary parts to these values in respective order; and
- a constructor that accepts a string input, *containing no spaces, for simplicity*, that can parse a string containing addition/subtraction of complex numbers such as "i", "2+3.2-4i", "1+2+3-4i-5i-6i", etc., and turn it into a complex number. Given an invalid input like the empty string or one that cannot be parsed, it should throw a **std::logic_error** with message "bad input string: [THE STRING]".

Then, you should overload

- binary **operator+=** to add two complex numbers, changing the left-hand side;
- binary **operator+** to add two complex numbers, returning the sum;
- unary **operator+** to return a copy of the complex number;
- **operator++**, both prefix (unary) and postfix (binary), such that the real part (and only the real part) is incremented by 1;
- binary **operator-=** to subtract two complex numbers, changing the left-hand side;
- binary **operator-** to subtract two complex numbers, returning the difference;
- unary **operator-** to return a complex number with negated real and imaginary parts;
- **operator- -**, both prefix (unary) and postfix (binary), such that the real part (and only the real part) is decremented by 1;
- binary **operator*=** to multiply two complex numbers, changing the left-hand side;
- binary **operator*** to multiply two complex numbers, returning the product;
- binary **operator/=** to divide two complex numbers, changing the left-hand side;
- binary **operator/** to divide two complex numbers, returning the quotient;
- unary **operator˜** to return the conjugate of the complex number;
- a **call operator taking no arguments** setting the complex number to 0;
- a **subscript operator overloaded on const** taking either "real" or "imag" as arguments and returning the real or imaginary parts of the number respectively (possibly as references), and should the subscript be invalid an **std::out_of_range** exception should be thrown, without being caught, with an error description "invalid index: [WHATEVER THE BAD INDEX WAS]";
- **operator<<** to print/display the complex number with an output stream such that if the imaginary part is positive it should be displayed as "a+bi"; if the imaginary part is negative, it should be displayed in the format "a-bi"; and if the imaginary part is 0 then it should be displayed as "a" and with "1i" and "-1i" displayed as "i" and "-i";
- **operator>>** to read in two double values, separated by white space, respectively setting the real and imaginary parts of the complex number on the right-hand side of an input stream;

- all of the comparison operators **operator**<, **operator==**, **operator**>, **operator**<=, **operator**>=, **operator!=** such that $a + bi$ and $c + di$ are compared lexicographically, i.e., by first comparing the real parts numerically and then the imaginary parts numerically if necessary ($3 + 4i < 3.1 + 4i$, $1 - i < 1 - 0.5i$, $4 + i = 4 + i$, etc.)*;
- a conversion operator from **Complex** to **std::string** (if the real part is 7 and the imaginary part is -8.36 then the output should be "7-8.36i", etc.); and
- a user-defined literal expression so that a code expression such as **3.3_i** will be converted into the complex number **0+3.3i**.

* mathematically, there is no "ordering" of the complex numbers in that no mathematical definition of < can satisfy the axioms of an ordered field. That doesn't mean we can't order the numbers in a way that is intuitive for a user of the class as we do here.

To reiterate on the formatting: a number with 0 real and 0 imaginary part should display as "0"; a number with a real part of 5 and an imaginary part of -3 should display as "5-3i"; a number with a real part of -1 and an imaginary part of +4.4 should display as "-1+4.4i"; a number with real part 9.73 and imaginary part of -1 should display as "9.73-i"; etc. There should not be unnecessary 0's floating around after a decimal (-1+4.40000 is not acceptable, for example).

Further requirements:

- You may **only** use the following headers:
  - iostream
  - string
  - sstream
  - stdexcept

  In particular, may may not use the <**complex**> header!
- You must adhere to the conventions in class with regards to which operators are member/nonmember functions.

Given the main routine below, the output is provided.

```
#include "Complex.h"
#include<vector>
#include<algorithm>
#include<iostream>

int main(){
  // some warmups
  Complex u{ 2,3 }, v{ 1,1 };
  std::cout << "simple stuff: " << u + v << '\n' << u - v <<
    '\n' << u*v << '\n' << u / v << '\n';

  // constexptr stuff
  constexpr Complex z0 = 1.77_i;
  constexpr Complex z0_conj = ~z0;
  constexpr Complex z0_copy = +z0;
  constexpr Complex z0_negated = -z0;

  // display this constexpr stuff
  std::cout << "z0, z0 conj, z0 copy, z0 neg: " << z0 <<
     " " << z0_conj << " " << z0_copy << " " << z0_negated << '\n';

  // Create Complex numbers through constructors
  constexpr Complex z1, z2{ 3, 4 };
  constexpr Complex z3 = (Complex{1} += (4 - 3._i));
  Complex z4 = z3;
  z4 *= 2; // multiply by 2 (which will be converted to Complex(2,0)

  // Create with user-defined literal
  Complex z5 = 3.14 + 14.3_i;

  std::cout << "Numbers z1, z2, z3, z4, z5: " << z1 <<
    " " << z2 << " " << z3 << " " << z4 << " " << z5 << '\n';

  // turn z5 to its conjugate
  z5 = ~z5;
  std::cout << "z5 after conjugation: " << z5 << '\n';

  std::cout << "z5*z3: " << z5*z3 << '\n';

  // Do some arithmetic to them and check values
  z4 += z3; z4 /= z5;
  Complex z6 = z1 + z2 + z3 - z4;
  Complex z7 = z6;
  z7();
```

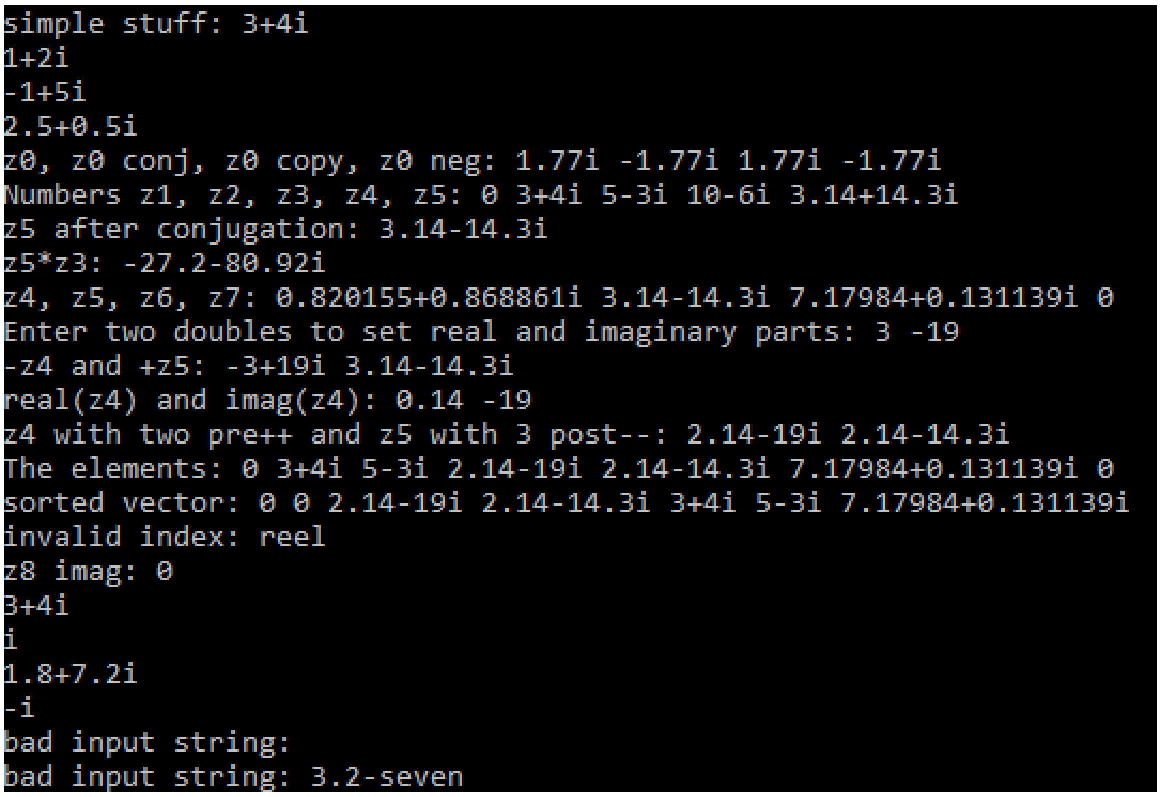
```
std::cout << "z4, z5, z6, z7: " << z4 << " " << z5 <<
  " " << z6 << " " << z7 << '\n';

// And read in with std::cin
std::cout << "Enter two doubles to set real and imaginary parts: ";
std::cin >> z4;
std::cout << "-z4 and +z5: " << -z4 << " " << +z5 << '\n';

// Access real and imaginary parts
z4["real"] = 0.14;
std::cout << "real(z4) and imag(z4): " <<
  z4["real"] << " " << z4["imag"] << '\n';

// increment and decrement operators
++++z4, z5------;
std::cout << "z4 with two pre++ and z5 with 3 post--: "
  << z4 << " " << z5 << '\n';

// store items in a vector, sort them
std::vector<Complex> vec{ z1, z2, z3, z4, z5, z6, z7 }; // vector of complex

// vector of strings because of conversion operator
std::vector<std::string> vec2{ z1, z2, z3, z4, z5, z6, z7 };

std::cout << "The elements: ";
for (const std::string& complexString : vec2) { // print each as a string
  std::cout << complexString << " ";
}

std::sort(std::begin(vec), std::end(vec)); // sort the Complex numbers
std::cout << "\nsorted vector: ";

for (const Complex& number : vec) { // print the sorted list
  std::cout << number << " ";
}
std::cout << '\n';

try{ // try accessing invalid index
  z2["reel"];
}
// upon failure, print the error and continue to run the program
catch (const std::out_of_range& ORR){
  std::cerr << ORR.what() << '\n';
  }

  // access real part of a constant number
  const Complex z8 = z7;
  std::cout << "z8 imag: " << z8["imag"] << ‘\n’;

  // testing the tostrings...

  std::cout << Complex{ "3+4i" } << ’\n’ << Complex{ "i" } << ’\n’ <<
    Complex{ "i+2i-3.2+4.2i+6-1" } << ’\n’ << Complex{ "-i" } << ’\n’;

  std::string empty;
  std::string also_bad("3.2-seven");

  try { // outer try
    try { // inner try, makes a Complex from empty string
      Complex will_fail(empty);
      std::cout << will_fail << ’\n’;
    }
    catch (const std::logic_error& L) { // manage that exception
      std::cout << L.what() << ’\n’;
      Complex not_gonna_work = also_bad; // but give another bad string
      std::cout << not_gonna_work << ’\n’;
    }
  }
  catch (const std::logic_error& L) { // catch any throws here
    std::cerr << L.what() << ’\n’;
  }

  std::cin.get();
  std::cin.get();

  return 0;
}
```

Some initial guidance to get you on your way...

1. Most of this is following exactly what was done in lecture: look at the slides and the examples. Besides different rules for addition, multiplication, etc., you are effectively redoing the **Time** class from lecture.

2. Your literal operator should take a **long double** as an input.

3. To do the **std::string** conversion, you should use a **string stream** and good old **operator**<<. But to suitably overload **operator**<< and adhere to the formatting



```
simple stuff: 3+4i
1+2i
-1+5i
2.5+0.5i
z0, z0 conj, z0 copy, z0 neg: 1.77i -1.77i 1.77i -1.77i
Numbers z1, z2, z3, z4, z5: 0 3+4i 5-3i 10-6i 3.14+14.3i
z5 after conjugation: 3.14-14.3i
z5*z3: -27.2-80.92i
z4, z5, z6, z7: 0.820155+0.868861i 3.14-14.3i 7.17984+0.131139i 0
Enter two doubles to set real and imaginary parts: 3 -19
-z4 and +z5: -3+19i 3.14-14.3i
real(z4) and imag(z4): 0.14 -19
z4 with two pre++ and z5 with 3 post--: 2.14-19i 2.14-14.3i
The elements: 0 3+4i 5-3i 2.14-19i 2.14-14.3i 7.17984+0.131139i 0
sorted vector: 0 0 2.14-19i 2.14-14.3i 3+4i 5-3i 7.17984+0.131139i
invalid index: reel
z8 imag: 0
3+4i
i
1.8+7.2i
-i
bad input string:
bad input string: 3.2-seven
```

requirements, you may need to condition what you display based on the sign of the imaginary part.

4. Save the constructor taking in a string for the end. You will likely need string streams for it!

Hint: Just like file streams, string streams also have an **eof** member function that behaves analogously.