Kiwis can be complex. Kiwis have a queen.
But mostly they're just fuzzy. Fuzzy, sweet and green.

kiwis can be complex

Complex Numbers

Solving this quest will earn you 1.0 real tickets to board the Argand plane.

It's a plane that can take you on flights of fancy into the deepest reaches of your imagination.

To ace this quest, you simply need to know how to throw and handle exceptions. You also get to override operators for this quest, but since you've already done that in previous quests, I'm not considering that as something new.

Overview

Obviously, this is not a primer on Complex Numbers. Refer to other sources for that. This quest will require you to have a basic understanding of complex numbers (as the sum of real and imaginary numbers). Here are some selected complex number features (among others) that you get to implement in this quest:

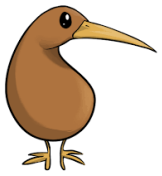
- The Complex Object: You encode it as a pair of double precision floating point numbers which represent the real and imaginary components of the complex number.
- The plus operator: The complex sum is simply another complex number in which the real component is the sum of the real components of all the operands and the imaginary component is the sum of the imaginary components of all the operands.
- The minus operator: The complex difference between two complex numbers is simply another complex number in which the real component is the difference between the real components of the corresponding operands and the imaginary component is the difference between the imaginary components of the corresponding operands.
- The times operator: The complex product of two complex numbers is another complex number. I'll give you more detail on how to calculate it in its miniquest description..
- A reciprocal method: The reciprocal of a complex number. I'll give you more detail about this also in its miniquest description.

Most of the implementation of this Complex class is routine, just like all the other classes you've implemented already. What's different with this one is that some of its methods may encounter exceptional situations (such as attempts to divide by 0, as when calculating the reciprocal). In situations like this you will make and throw an exception.

Note: In this spec, I often refer to a complex number `z = x + yi` using the tuple `(x,y)`.

This can be your chance to experiment with and explore the world of exceptions. Find out how they can make your programming life easier.

Starter code



I asked the Questmaster for some starter code.

But he said "*No way! Go away. This quest is way too easy.*"

So I said "*Ok. Another day. But don't mind me being nosy.*"

Then I started poking around in his things and managed to get a quick (but) fuzzy photo of the class def from his screen. See if it helps.

```
19 class Complex {
20 private:
21     double _real, _imag;
22
23 public:
24     static constexpr double FLOOR = 1e-10; // Threshold
25
26     Complex (double re = 0.0, double im = 0.0) : _real(re), _imag(im) {};
27     double get_real() const { return _real; }
28     double get_imag() const { return _imag; }
29     void set_real(double re) { _real = re; }
30     void set_imag(double im) { _imag = im; }
31
32     string to_string() const;
33     double norm_squared() const { return _real*_real + _imag*_imag; }
34     double norm() const { return sqrt(norm_squared()); }
35
36     Complex reciprocal() const;
37     Complex& operator= (const Complex & rhs);
38
39     class Div_By_Zero_Exception {
40     public:
41         string to_string() { return "Divide by zero exception"; }
42         string what() { return to_string(); } // more conventional
43     };
44
45     // operators (only the most common ones)
46     Complex operator+(const Complex& that) const;
47     Complex operator-(const Complex& that) const;
48     Complex operator*(const Complex& that) const;
49     Complex operator/(const Complex& that) const;
50
51     bool operator==(const Complex& that) const;
52     bool operator!=(const Complex& that) const { return !(*this == that); }
53     bool operator<(const Complex& that) const;
54
55     friend ostream& operator<<(ostream& os, const Complex& x);
56
57     friend class Tests; // Don't remove this line
```

Your first miniquest - Constructors make things

Implement the default and non-default constructor for the Complex class:

```
Complex(double re, double im);
```

The constructor should take default values for both the real and imaginary values. The constructor with no parameters should give you `0` (which is `0 + 0i`). The constructor with a single real valued parameter `x`, should give you `x` (which is `x + 0i`). And the constructor with two parameters `x` and `y` should give you the complex number `x + yi`.

If your constructor is correctly implemented, then you get a freebie implicit constructor. You can say something like:

```
Complex c = 3.14;
```

Although the RHS is a single real number, the compiler will auto-invoke your constructor implicitly as if you had written:

```
Complex c = Complex(3.14, 0);
```

Your second miniquest - Equal means equal

Implement the equality comparison operators:

```
bool operator==(const Complex& that);

bool operator!=(const Complex& that);
```

Ideally, one of these is implemented in terms of the other. But here's an opportunity to reward me with some discussion about whether it is better to do it that way rather than implement each comparator independently.

Your third miniquest - Assign foretells incoming goodies

Yeah! Do you even have to implement this puppy?

Or is this, like, a TOTALLY FREEBIE reward!

I can't believe it, peeps.

Your fourth miniquest - What's the norm around here?

The norm of a Complex number `c = a+bi` is defined as follows:

$$|c| = \sqrt{a^2 + b^2}$$

Implement:

```
double norm() const;
```

which will return the norm of the Complex, `c`, on which it is invoked.

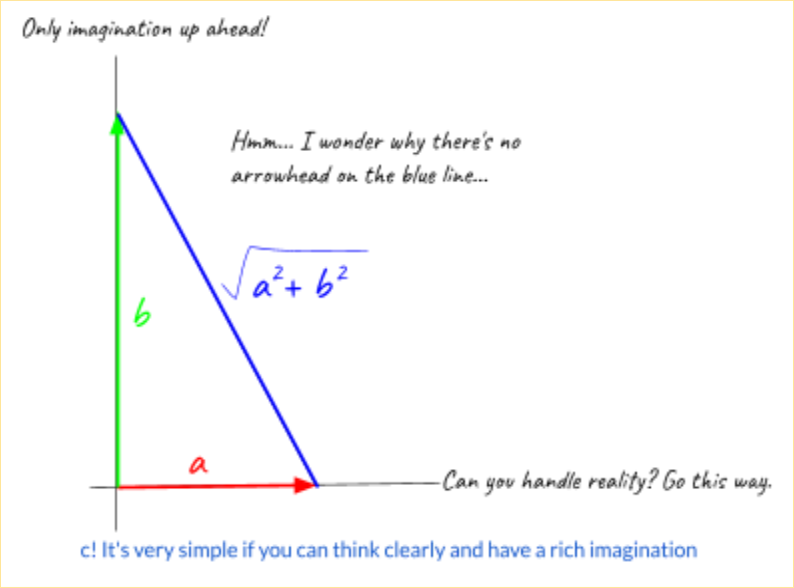


Figure 1 (The norm as the hypotenuse on the Argand plane)

Your fifth miniquest - Compare and contrast

Implement the quantitative comparison operators:

```
bool operator<(const Complex& that) const;
```

Someone said "Hey! I thought you said operators? But I only see one operator up there."

That would be right. You can implement all of them, but you only need to implement the other comparison operators to make your interface sweeter to REAL users of your Complex class. As for me, I'll use your < operator to manufacture the other ones (How?)

The only downside I can see is that if you got your < wrong, then all your comparison operations will be wrong. Still... I think it's less work to get it right.

We'll define less than to refer to the norms of the complex numbers internally. That is, $c_1 < c_2$ if $norm(c_1) < norm(c_2)$.

Your sixth miniquest - Plus... don't be nonplussed

Implement

```
Complex operator+(const Complex& that);
```

which returns a new complex number that is the sum of its two operands (`*this` + `that`)

Your seventh miniquest - Merry Minus

Yeah, I know it sounds like a quest in itself. But it's really quite simple. Implement:

```
Complex operator-(const Complex& that);
```

which returns the difference between its operands (`*this - that`)

Your eighth miniquest - Times and Times again

The product of two complex numbers `(a, b)` and `(c,d)` is defined as `(ac-bd, ad+bc)`.

Is it commutative? Why? (or why not?)

Regardless, implement

```
Complex operator*(const Complex& that);
```

which returns the product (assuming `*this = a + bi` and `that = c + di`)

Your ninth miniquest - A reciprocal arrangement

Implement a reciprocal arrangement that works well enough to handle all non-empty quantities.

```
Complex reciprocal() const;
```

Assuming that a complex number, `z`, is not zero, we want `1/z`, that is, a number z^{-1} such that $z*z^{-1} = 1$.

if `z = a + bi`, then we want:

$$1/z = 1/(a + bi)$$

$$= \frac{a-bi}{(a+bi)(a-bi)}$$

$$= \frac{a-bi}{a^2+b^2}$$

In your code, you can leverage the fact that the denominator is simply the square of the norm. The reciprocal is the complex number $(a/(a^2+b^2), -b/(a^2+b^2))$

Your tenth miniquest - The Daily Divide

If you nailed the basic reciprocal miniquest, the daily divide is practically a freebie.

Implement the division operator well enough to work on everyday denominators.[1]

```
Complex operator/(const Complex& that);
```

For a non-zero complex z_2 (which is all this miniquest needs to handle):

$z_1/z_2 = z_1 *$ (reciprocal of z_2)

Your eleventh miniquest - Impossible to repay!

Now we're talking.

[1] I mean, you don't get to divide by zero everyday, do you?

Extend your reciprocal function to handle the case when it's invoked on the zero complex number. Really, I should be saying *a zero* complex number because we defined our zero to be a range of numbers between two limits.

When asked to reciprocate with something for nothing, it's always better not to throw a fit. But when asked to find the reciprocal of zero, it's always better to throw an exception (since there is no single correct answer).[2]

The exception object you will create and throw must be defined as an inner public class within the `Complex` class (as `Complex::Div_By_Zero_Exception`)

This public inner class, `Div_By_Zero_Exception`, must provide two public methods:

```
string Div_By_Zero_Exception::what();
string Div_By_Zero_Exception::to_string();
```

Either method (one may simply relay the other if you want), if invoked on this exception object, should return the exact string: "`Divide by zero exception`"

Usually, the person who asked you to do something is the one who has immediate responsibility for your actions. And this includes exceptions. So the caller of your method is responsible for catching and handling any exceptions your method may throw. If they don't catch it, the exception will pass over to their caller, and so on.

The idea here is that anytime you come to a situation you can't handle, you make an informative exception and throw it. Someone had better catch that exception. If nobody catches it, it will roll all the way down to the run-time system which will KILL all programs that throw things at it.

In computer programs, exceptions often allow us to easily propagate all exceptional situations to an appropriate common layer of processing so we can factor our error handling in the same way as we factor the rest of the code.

Here is how I would create and throw a brand new `Div_By_Zero_Exception` object, maybe in my reciprocal code:

```
...
 if (my_denominator <= Complex::FLOOR) // watch for round-off
     throw Div_By_Zero_Exception();
...
```

[2] I think Ken Thompson (Unix) said this. But you may want to look up the source: *Don't test for errors you don't know how to handle.* (Throw an exception and let your caller handle it if they know how).

Define a suitably small number in your complex class and use it as a proxy for zero. When you're working with doubles and floats (which are more similar to density functions than discrete distributions - Discuss), it helps to define ranges a value should fall in rather than be exactly equal to. The range I picked for my Complex 0 is -10^{-10} to $+10^{-10}$

So I defined my `Complex::FLOOR` to be `1e-10`. If any denominator is smaller than this `FLOOR`, I consider that an attempt to divide by zero.

Here is how I might catch your exception object:

```
try {
    ...
    // I try to do things with complex numbers here
    ...
} catch (Complex::Div_By_Zero_Exception e) {
     cerr << e.what() <<endl;
     exit(-1);
 }
```

If anything about this miniquest is unclear, all you have to do is ask (in our subreddit channel)

Your twelfth miniquest - The Great Divide

Just like before, extend the division operator's domain to handle division by 0. But rather than make and throw an exception yourself, you can blissfully ignore the possibility and let your reciprocal calculation handle the exception throwing.

Oh wait! Isn't that already done? Umm.. ok. Do you got to do anything at all for this miniquest?

Bummer... Looks like I didn't really think this one through. Still a reward is a reward. You earned it if you got this far.

Your thirteenth miniquest - To string

Implement

```
string Complex::to_string() const;
```

When invoked on a complex number a + bi, return the string `(a,b)`. There is no space after the comma. To format the numbers, I use:

```
sprintf(buf, "(%.11g,%.11g)", _real, _imag);
return string(buf);
```

You may already know how to format numbers for stream output using c++ precision specifiers. This is your chance to learn about an ancient and powerful formatting technique that is making

a comeback. Look up "printf formatting" online to see what the above specification means. Share your finding summaries on our subreddit.

Your fourteenth miniquest - … and beyond

Make a friend. Implement:

```
friend ostream& operator<<(ostream &os, const Complex& x);
```

Something tells me that this reward is also as good as a freebie if you've aced the previous miniquest.

Do you think that's way too many freebies for a quest?

How long is a well written solution?

Well-written is subjective, of course.

But I think you can code up this entire quest in under about 150 lines of clean self-documenting code (about 25-50 chars per line on average), including additional comments where necessary.

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret code for this quest in the box.
3. Drag and drop your `source` files (`Complex.*`) into the button and press it.
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions.

Happy hacking,

&