Testing a Complex ADT

This is a **warmup** exercise. It is **not compulsory**, and may be completed **individually or with your lab partner**.

Heads up! Make sure you've done *The Complex ADT* before attempting this exercise.

So far, we've seen how to implement an ADT, and how an ADT enforces its abstraction. Now, let's talk about how we can use it, and while we're at it, write some tests for our brand-new ADT.

Testing Philosophy

There are four types of tests:

1. **Bad tests**: "I've written this program… now, let's write some tests… my program passed, woohoo!"

   There's a conflict of interest here. On one hand, you want your program to be correct. On the other hand, you're confident your program is right when it is wrong. If you miss some logical error when writing your code, you'll likely also blindly not test it.

2. **Black box tests**: "I've written this program, now let's get someone else to test it for me!"

   The tester does not care how your program works, just that its inputs and outputs are correct. A tester could treat your program as a magical black box, where information goes in, and information comes out.

   The tester also does not have as much of an interest in finding your program correct as you would.

3. **White box tests**: "I've written this program, can you look through it, and check it's right?"

   The tester, with knowledge of your code, trawls through it looking for errors. They can use black-box testing to see if there is an error there.

4. **Unit tests**: "As well as testing my whole program, I'll test each of the small parts of it."

   An error in one of our small units might not easily be found trying inputs to our program.

   It is much faster and easier to check each of the small units in isolation; we'll still test the program as an integrated whole, of course.

Whenever we find an error in a unit of our program, you should

- create a test to catch that bug out,
- run our tests, and check the unit under test fails,
- fix the error,
- rerun our tests, and check the unit now passes!

We only add tests to our test file, we never take them out. After working on a program for a while, you should have a big suite of comprehensive tests!

And, whenever we make a change, we rerun the tests! This way, you know if you've broken something or not, and you can (hopefully easily) identify the breaking change you've made.

Consuming an ADT

To use our `Complex` ADT, we do need to `#include` our header file to expose our interface –

```
#include "Complex.h"
```

— and now we can start to call functions.

```
    Complex w = newComplex (0, 0);
```

Already, there's plenty of things to talk about here. First, we declare a new `Complex` variable `w`, whose value is $0 + 0i$. The value is established by calling the constructor, `newComplex`. Now, in the system, we have this magical Complex value that's moving around, and we can't see inside it.

No, really. Try and stab into it.

```
useComplex.c:9:3: error: incomplete definition of type 'struct _complex'
        w->re = 0;
        ~^
Complex.h:7:16: note: forward declaration of 'struct _complex'
typedef struct _complex *Complex;
               ^
1 error generated.
```

The type quite literally doesn't exist. We don't know, in our program, what's in this structure, or how it's laid out. That's good!

You might be tempted to think, "oh, I'll just copy the `struct` definition in, then." Don't do it! That defeats the purpose of the abstraction; we've deliberately hidden that from the user.

By convention, ADT interface functions tend to take the ADT as the first parameter. So, to correct our above mistake, we might do:

```
    double re = complexRe (w);
    printf ("The real part is %lf.\n", re);
```

And that will compile and work.

The final piece is *compilation*: to compile an ADT and a user, you need to tweak your compilation step slightly, so the ADT implementation is compiled in. Otherwise, you'll get lots of angry linker errors about undefined symbols.

```
$ dcc -o useComplex useComplex.c Complex.c
```

Testing an ADT

With all that in mind, let's talk about how we can test an ADT. From the outside, we can easily write black-box tests: black-box tests only require a well-defined interface, and it just so happens we have one of those.

Our ADT's interface does fairly sensible things, so you could write a test to check that the real portion you put in is the same as the real portion that comes out.

You can verify that with our good friend `assert()`. `assert()` allows us to state an *invariant* in our program: something that is always true, and if it is not, our program ceases to exist. Some programmers use `assert` as a primitive error handling mechanism; in this course, that's strictly forbidden.

```
    double expect = 0;
    Complex w = newComplex (expect, expect);
    assert (w != NULL);
    assert (complexRe (w) == expect);
    assert (complexIm (w) == expect);
```

That's pretty much it! That's how you'd write black-box tests for an ADT.

One caveat to be aware of: you cannot test destructors. Once you call a destructor, you've almost definitely called `free()`, and you've promised the memory allocator that you'll never use that memory again.

The Exercise

Go forth, and write tests for the functions in the interface. Think about the edge cases, and the combinations of values that might cause your tests to fail.

There aren't any Autotests yet... stay tuned.

Once you've written these tests, you might like to try *A More Complex ADT*, and run your tests against it.

To run Styl-o-matic:

```
$ 1511 stylomatic testComplex.c
Looks good!
```

You'll get advice if you need to make changes to your code.

Submit your work with the *give* command, like so:

```
$ give cs1511 wk08_testComplexADT
```

Or, if you are working from home, upload the relevant file(s) to the `wk08_testComplexADT` activity on Give Online.