A Complex ADT

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

This week, we're taking some of the new data types that we've seen over the last few weeks, and asking, "can we make that an ADT?"

First, though, let's talk about ADTs: **Abstract Data Types**.

Concrete vs Abstract

Over the last few weeks, we've seen a few variations on a structure for holding complex numbers. For example, in the code you're provided for assignment 1, you get the following structure in `mandelbrot.h`:

```
typedef struct _complex {
    double re;
    double im;
} complex;
```

We say that a type is *concrete* if a "user" of that type has knowledge of how it works, and conversely, a type is *abstract* if a user has no knowledge of how it works. By this definition, `complex` is a *concrete* type: you may know what a `complex` value's real and imaginary parts are directly.

Using concrete types is dangerous: once a user of a type knows about the insides of that data type, and *especially* if they are given the power to directly use it or change it, you cannot change the "insides" of the type without breaking current software. If, say, we wanted to change our `complex` structure to instead store the modulus and argument (or magnitude and phase, if you prefer), we would have to change absolutely everywhere that used our type.

Abstraction is our friend here: if we only expose the real and imaginary part through a pair of functions (and the modulus and argument likewise) it doesn't really matter how the data is stored – how the **implementation** works – so long as the **interface** works.

So far, we've used .h files to achieve this, where the functions are declared but not defined.

More powerfully, we can make types abstract: in C, we can refer to a structure without knowing its contents if that structure sits behind a pointer. Other languages have similar features, but fundamentally, they boil down to this exact idea.

Compare the above concrete type with this abstract type:

```
typedef struct _complex *Complex;
```

That's much better! You can expose this to a user, and all they know is that there's a reference to a structure, but not necessarily what's in that structure.

So, what's in that structure? We don't know, and it doesn't really matter, because all we need is for the interface functions to work.

So, let's make an ADT.

An ADT's Interface

Download `Complex.h`, or copy it into your current directory on a CSE system by running

```
$ cp /web/cs1511/17s2/week08/files/Complex.h .
```

> **Don't change *Complex.h*!** If you do, the charge on the protons that make up your teeth will be suppressed.

In `Complex.h`, there are a few new bits of syntax that we'll be seeing much more often now we're working with all the moving parts of ADTs. First up, *header guards*, which allow you to include a file multiple times without it conflicting with itself.

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

// ... code here ...

#endif // !defined(_COMPLEX_H_)
```

Next, the `typedef`, which creates our data type. C lets us `typedef` a pointer type before we know what the type is, because pointers are all the same size; this trick won't work in any other place.

```
typedef struct _complex *Complex;
```

And finally, some function prototypes. It's a good habit to document the functions prototyped in a header file. We've listed the prototypes below, along with brief documentation of what they do as provided in the header file, along with some notes on how you should implement them.

- `Complex newComplex (double re, double im);`
  Create a new `Complex` number from two `double` s.

  We call this a *constructor*, and its role is to allocate sufficient resources to hold the value(s) required.

  For this particular constructor, you'll need to use `calloc` to make an allocation of enough bytes to store a `struct _complex` . You should verify that your allocation succeeded, and exit sensibly if not.

- `void destroyComplex (Complex c);`
  Release all resources associated with a `Complex` number.

  You made an allocation, and, in accordance with Newton's third law of memory management, for every allocation, there should be an equal and opposite `free` . When working with ADTs, we would call this function a *destructor*, and its sole purpose is to call `free` , to release the memory allocated for this structure.

- `double complexRe (Complex c);`
  Retrieve the real part of the `Complex` number.

  This function really needs to do two things: first, check the parameter `c` is a valid pointer, and then return the real part by stabbing `c` . In the new parlance of ADTs, this is a *getter* function.

- `double complexIm (Complex c);`
  Retrieve the imaginary part of the `Complex` number.

  Again, this function does two things. This function is also a *getter*.

- `Complex complexAdd (Complex w, Complex z);`
  Add two `Complex` numbers together.

  This function does four things: it verifies both `w` and `z` are valid pointers, it adds the real parts and imaginary parts, it creates a **new** Complex, whose real and imaginary parts are the values we just calculated, and it returns that complex. This function shouldn't change `w` or `z` , but should return a strictly new Complex number, which has to be destroyed separately.

- `Complex complexMultiply (Complex w, Complex z);`
  Multiply two `Complex` numbers together.

This function is much the same as `complexAdd`, but it returns the multiple of the two Complex values. Again, this creates a **new** Complex, which has to be destroyed separately.

- `double complexMod (Complex w);`
  Take the magnitude (or modulus, or absolute) of a `Complex` number.

  This function does three things: it verifies that `w` is a valid pointer, takes the square root of the sum of the squares of the real and imaginary parts of `w`, and returns that value.

  You should use the `sqrt` function from `math.h`, which we've included for you. If you're not using *dcc* to compile your code, you'll need to also specify the compiler flag `-lm`.

- `double complexArg (Complex w);`
  Take the argument (or angle) of a `Complex` number.

  This function is very similar to `complexMod`, but you should take the argument. The argument of a complex number z is $\tan^{-1}(\mathbf{Im}(z)/\mathbf{Re}(z))$.

  You should use the `atan2` function from `math.h`; it calculates the arc-tangent of y/x, and its prototype is `double atan2 (double y, double x)`,

Implementing an ADT

Download `Complex.c`, or copy it into your current directory on a CSE system by running

```
$ cp /web/cs1511/17s2/week08/files/Complex.c .
```

You should implement these functions, based on the notes above. If you want, you could try doing so without the stub code provided; otherwise, we've provided enough code to make `Complex.c` compile.

As an ADT implementation, `Complex.c` **should not** contain a `main` function. How do you run it, or test for correctness, then? You should write an ADT *user*, and the Testing a Complex ADT activity shows how to do that. (We also have an Autotest set up, designed in exactly the same way.)

To run some simple automated tests:

```
$ 1511 autotest complexADT
```

To run Styl-o-matic:

```
$ 1511 stylomatic Complex.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_complexADT
```

Or, if you are working from home, upload the relevant file(s) to the `wk08_complexADT` activity on Give Online.