Enel 452 – Fall 2023 – Assign 1

A Complex Number Calculator
Handed Out: [2023-09-05 Tue]
Due: [2023-09-15 Fri 23:55]

Introduction

You will write a C or C++ program[1] that implements a command-line complex number calculator. In a future assignment we will ask you to run this code on the embedded nucleo board, so you should keep portability in your mind as you design your solution.

Your calculator will present the user with a prompt, at which they can type a number of commands. The interface is a REPL, or Read-Evaluate-Print Loop. The four major commands are `A`, `S`, `M`, and `D`, corresponding to Add, Subtract, Multiply, and Divide. Any of these commands must be followed by two pairs of floating-point numbers, each pair corresponding to a complex number in *rectangular* form. All commands must be case-insensitive. In addition the character `Q` or `q` should quit the program.

To *add* the complex numbers (45.67, -170) and (9.2, 15) with your calculator, you might type the following at the prompt

```
Enter exp: A 45.67 -17e1 9.2 15
54.87 - j 155
Enter exp:
```

Things to note:

1. I chose the prompt: '`Enter exp:  `', but you can choose whatever you like
2. you must press the *enter* key after each command
3. an empty line, or one with only whitespace, is ignored
4. commands and arguments may have an arbitrary amount of leading and trailing whitespace
5. the program input supports standard scientific `e` notation, numbers with a decimal point, and integers, but all numbers are converted to `double` before processing

[1] in either case your program must *compile under at least gcc/g++-9 or clang 10*

6. it's case insensitive: you can use `a` instead of `A`

7. The calculator prints out the answer in the form

   ```
   real + j imag
   ```

   This format must be *precisely* followed or your program will fail all the provided acceptance tests.

Writing a filter

An important constraint in this assignment is that your program must act as a *filter*, that is, it must read from the *standard input* and write to the *standard output* and *standard error*. I have provided a sample filter program (`filter.c`) with the handout which explains this a bit. Basically any program executing on a UNIX-derived OS or on Windows (also UNIX-derived) will own three open file handles:

file 0: known as `stdin` in C and `cin` in C++. Normally associated with the *keyboard*.

file 1: `stdout` in C and `cout` in C++. Normally associated with the *screen*.

file 2: `stderr` and `cerr`, respectively. Normally associated with the *screen*.

Every process is free to simply read from and write to those 'files' without dealing with any file-system open/close/configure commands. Normally `stdout` is used for the regular console output, and `stderr` is used to inform the user of some error.

We will use `stdout` and `stderr` in a slightly off-beat way: all the important *program* outputs will go to `stdout` and all "lesser" output (prompts, informative messages, system help, *some* error messages) will go to `stderr`. By doing this, we gain great power because our program can now seamlessly interact with the console, with disk files, with devices, or with any data source and sink.

Below I've shown a complete console session on my computer, but I've drawn all user input in red, all arithmetic results in blue, and all `stderr` output in gray. The first line, where I invoke the program, is irrelevant to the discussion.

```
$ ./assign1
Complex calculator

 Type a letter to specify the arithmetic operator (A, S, M, D)
 followed by two complex numbers expressed as pairs of doubles.
 Type Q to quit.

Enter exp: a 1 2 3 4
4 + j 6
Enter exp: s 1 2 3 4
-2 - j 2
Enter exp: m 1 2 3 4
-5 + j 10
Enter exp: d 1 2 3 4
0.44 + j 0.08
Enter exp: q
```

As you will see, it is vitally important that your calculator write only *calculated* outputs and *error codes* to `stdout`. *Every* other output: prompts, help messages, etc., must be written to `stderr`.

Given this, the same session could be run in "batch" mode, with all the commands read from a disk file, named, say, `mycommands.txt`. Here is this file's content:

```
a 1 2 3 4
s 1 2 3 4
m 1 2 3 4
d 1 2 3 4
q
```

To run `assign1` with the input taken from `mycommands.txt` we must override the default `stdin` association (keyboard) and force input to come from the file. This is called *redirecting* the input. The `<` can be thought of as an "arrow" telling us the input is coming from the file instead of the keyboard.

```
$ ./assign < mycommands.txt
```

Before the program executes, the OS *redirects* file handle 0, so that it streams data from the file, rather than the keyboard. We could explicitly show that it's file 0 with a "0" prefix, however, in practice you rarely see this because it defaults to file 0.

```
$ ./assign 0< mycommands.txt
```

And if we want to shunt aside all the `stderr` stuff, saving it in `junk.txt`, and write the `stdout` stuff to a file named `output.txt` here is how:

```
$ ./assign < mycommands.txt 1> output.txt 2> junk.txt
```

Here I've prefixed a "1" to clearly show `output.txt` is receiving data from `stdout`, but that's redundant because it defaults to `stdout`. If I run this commandline, nothing appears on the screen and nothing is read from the keyboard. After running the command, here is the content of `junk.txt`:

```
Complex calculator

 Type a letter to specify the arithmetic operator (A, S, M, D)
 followed by two complex numbers expressed as pairs of doubles.
 Type Q to quit.

Enter exp: Enter exp: Enter exp: Enter exp: Enter exp:
```

And here is the content of `output.txt`:

```
4 + j 6
-2 - j 2
-5 + j 10
0.44 + j 0.08
```

The modifiers `<`, `1>`, and `2>` provide the ability, *at the command line*, to redirect console I/O to and from *disk files*. However this does not adequately convey the *real* earth-shaking innovation provided by UNIX and its descendants: the treatment of *any data source or sink* as a "file". This allows, for example, a program's I/O to be replaced at invocation time with *live processes*, through a mechanism called *pipes and filters*.

I hope this is clear that now we've achieved two things:

1. An entire series of calculations can be automated, even though the program was written to be strictly interactive

2. the irrelevant output can be separated and discarded, and the important output can be extracted and fed into another program for processing. Below I'll describe a test program I've provided which can run in parallel with your program, and determine its correctness.

Error handling

There are certain inputs that will result in an error. Your program *is required* to produce a specific error code for each condition. In this instance, the error codes are to be sent to `stdout` rather than `stderr`. This is because the test program introduced below currently only uses `stdout` and ignores `stderr`.

Error 0: Command executed correctly, no error. Just print the result as usual, to `stdout`.

```
a 1 2 3 4
4 + j 6
```

Error 1: Illegal command (anything besides upper/lowercase `A`, `S`, `M`, `D`, or `Q`). Suppress any arithmetic output, and emit the indicated string to `stdout`:

```
Y 1 2 3 4
error code: 1: illegal command
```

Error 2: Missing arguments. For example, supplying fewer than 4 floats for an arithmetic operation. Suppress any arithmetic output, and emit the string to `stdout`:

```
a 1 2 3
error code: 2: missing arguments
```

Error 3: Extra arguments. For example, supplying more than 4 floats for an arithmetic operation, or supplying any arguments to a quit command. Suppress any arithmetic output, and emit the string to `stdout`:

```
a 1 2 3 4 blabla
error code: 3: extra arguments
```

Error 4: Divide by zero. In a divide operation, a zero denominator was encountered. Suppress any arithmetic output, and emit the string to `stdout`:

```
d 1 2 0 0
error code: 4: divide by zero
```

If you discover any errors beyond 1-4, feel free to add error strings for them, and document what you did in your `readme` file.

Other general requirements and hints

1. You must define your own `Complex` data type. You're likely aware that the standard C and C++ libraries both provide implementations of complex numbers and arithmetic. *Don't use these!* Use four separate subroutines, which perform the 4 complex arithmetic functions. For example a prototype of the add routine might be:

   ```
   void add(Complex z1, Complex z2, Complex* result);
   ```

2. Use your compiler to help you catch bugs: by enabling the following warnings. I will use these when I compile your code, and if there are any warnings you'll lose most of the as1 marks.

   ```
   -Wextra -Wall -Wfloat-equal -Wconversion -Wparentheses
   -pedantic -Wunused-parameter -Wunused-variable -Wreturn-type
   -Wunused-function -Wredundant-decls -Wreturn-type -Wunused-value
   -Wswitch-default -Wuninitialized -Winit-self -Werror
   ```

   - I suggest only turning on `-Werror` when you've successfully eliminated all warnings. This flag turns all warnings into errors, and kills the build, so it might be more of an irritant than an assistant in the early stages.

3. If possible, write the program so it can be compiled under C++, *even if your program is pure C*. The C++ compiler does more stringent checking.

4. Include a `readme.txt` file that briefly discusses your program, its use, its limitations, how it handles erroneous input, any extra features you added, any required features you didn't implement, *etc.* Also provide information on how your program is to be built. For example if you use make, include the `Makefile`.

Snoopy issues

First of all: keep in mind the eventual goal is to be able to compile the exact same source for snoopy *and* for the nucleo board.

So try to anticipate this. Compile your source in Keil, flash the ELF file to the nucleo board, see if it runs. Probably the biggest hurdle you'll encounter is with I/O, and the early labs will give you assistance in solving it.

You can use either C or C++ for this assignment (and all assignments and labs), but you might find I/O functions from standard C slightly easier to get running on the nucleo board because they tend to rely less on the heap.

If you use the standard C I/O library, I *strongly* recommend using the `fgets()` function to read the input as a string in one fell swoop, `sscanf()` to parse it, and `fprintf()` to print the output. These functions are also available in C++.

People have floundered when they use something like `fscanf` because it has no respect for line boundaries, and the program gets totally confused about the input. For the same reason, you must not use `cin`. Other similar "C++ native" I/O functions exist in the standard C++ library. See cplusplus.com for a good reference on the the C/C++ I/O library.

You *don't have to do all your work on snoopy.* You can develop on any standard platform: Windows, Mac, Linux, BSD. You'll need to install the GCC toolchain.

> *I will test your code on snoopy, so make sure you at least try it there!.* If your code passes `runtests.py` then you automatically get full marks for the functionality tested.

Runtest.py

In addition to GCC, you will also need to install Python3. I have provided a Python3 test script (`runtests.py`). This script interacts with your calculator *as a filter*, and it expects you to follow the I/O rules above: read exclusively from `stdin`, and rigorously enforce correct writes to `stdout` and `stderr` as described.

Using runtest.py

If your current directory contains your executable (`as1` or `as1.exe`) and also contains `runtests.py` you would run the tests at the command prompt like this:

```
$ ./runtests.py ./as1
```

Under Windows you might need to use backslashes (\) and the executable would be `as1.exe`. `Runtest.py` must be marked as executable, that is, the `execute` bit on the script must be set. NOTE WELL: you must provide the relative (or absolute) path to both the `runtest.py` and `as1`, regardless of where they reside.

`Runtest.py` will work in any development environment: Linux, Mac, Windows, but *not on the embedded target* because the embedded target has *no operating system.*

To submit, push your code to your `git` repository.

Note: Late assignments will be rejected and given a grade of zero.