**Department of Electrical and Computer Engineering**

Homework Assignment No. 06:

**Real Programs**

submitted to:

Professor Joseph Picone

ECE 3822: Software Tools for Engineers

Temple University

College of Engineering

1947 North 12th Street

Philadelphia, Pennsylvania 19122

October 5, 2015

prepared by:

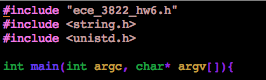
Robert Irwin  
Email: robert.irwin@temple.edu

# Problem

In this assignment we are reading and displaying the contents of binary files in C. We will write a program that accepts command line arguments, and inputs piped from standard-in. These inputs will be read and displayed as 4 byte floating-point numbers, or short integers depending on the inputs to the command. Our read function, as well as constants used in the program, will be imported from a user-defined header file.

# Approach

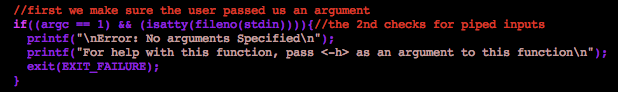
To begin solving the problem, we had to make sure we could read command line arguments. This is done in the definition of main. The syntax is shown below in Figure 1.



*Figure 1: Syntax for Accepting Command Line Inputs*

Argc holds the number of command line arguments, and argv is a pointer to the command line arguments. Argv[0] always points to the function itself, so the smallest value for argc is 1. Notice the header file include by the quotation marks. This is the custom header, and it is in the same directory as our function. The contents of this header will be shown later.

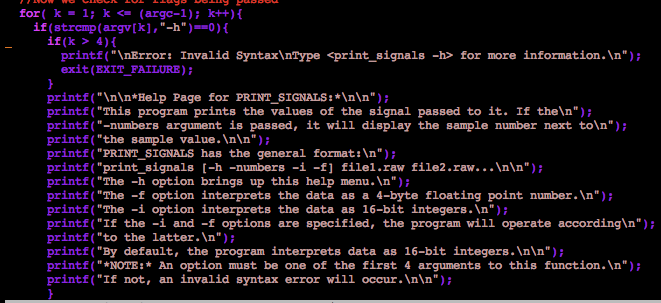
The next step was making sure our program actually received inputs. The process for this is shown below in Figure 2.



*Figure 2: Checking for Inputs*

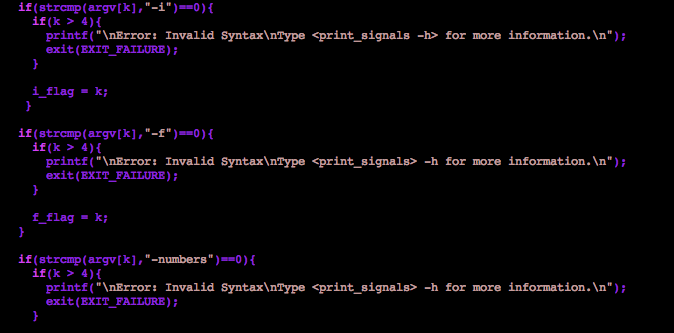
Notice that we check for argc == 1. As explained previously, argc is 1 if we received no command line inputs. Because we also accept inputs piped from standard-in we also check for the status of isatty(fileno(stdin)). Fileno returns the file descriptor of its input, which for stdin is 0. This 0 is passed to isatty, which determines if a file descriptor is a terminal object or not. If standard-in is not piped, this returns 1.

It is also important to note that we mention a help menu. The contents of the help menu are shown now.



*Figure 3: Help Menu*

In this piece of code, we loop through all command line arguments. Each time, we check to see if the argument given is the “-h” option. If it is, and it is not the 5th argument or later, we print the help menu shown above. A similar method is checking for the –i, -f, and –numbers flags. This is shown below.



*Figure 4: Checking for Other Flags*

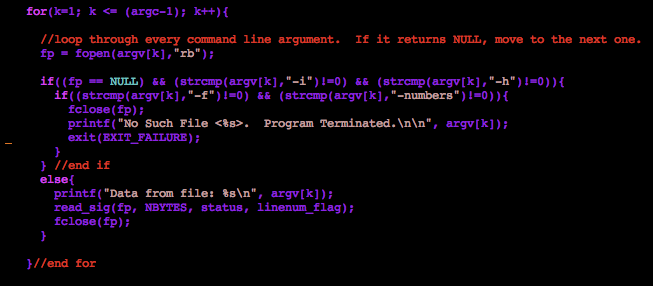
It is important to note that both the –i and –f options can be specified. If this happens, we decided to accept whichever argument was passed last. We know which one was passed last by the k value associated with the flag. Notice we set i\_flag and f\_flag equal to the k-value used as the index of argv. Then we check which is greater, and set “status” accordingly. This is shown below in Figure 5.



*Figure 5: Setting Status*

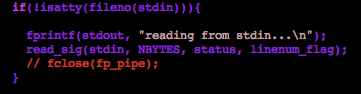
It is important to note that we initialize the status flag to be 0, which corresponds to short integer mode.

Now that we have taken care of all the flags, it is time to start reading the files. The code that does this is shown below in Figure 6.



*Figure 6: Processing Command Line Inputs*

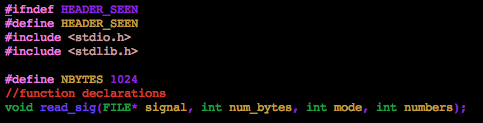
Again, we loop through every command line input, try to open it, and check the file pointer for equality with NULL. If it is not NULL, we know we have opened a file, and we pass it to our function read\_sig(). The contents of read\_sig(), which is contained in the header file, ece\_3822\_hw6.h, will be shown after we review the code for processing the piped inputs.



*Figure 7: Processing Piped Inputs*

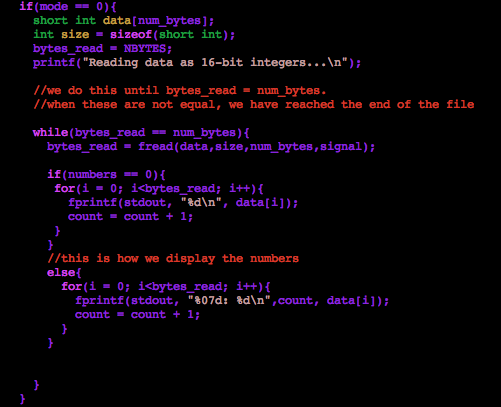
Notice that we use the same argument as was used in Figure 2, with an inverter thrown in. This time, if standard in is not a terminal argument, we execute the code within the if statement. Also notice that we use stdin just as we use fp. This is because stdin is a file pointer to the standard input.

Now we show the contents of the header file.



*Figure 8: Definitions in Header File*

The first two lines of this header check to see if the header file was already called. Sometimes, the compiler will throw errors if it tries to compile the same header file more than once. Then we bring in the standard C headers, which is why we didn’t have to include them in our main, and we define NBYTES, which we also saw used in main without a declaration. Note the inputs to the function.



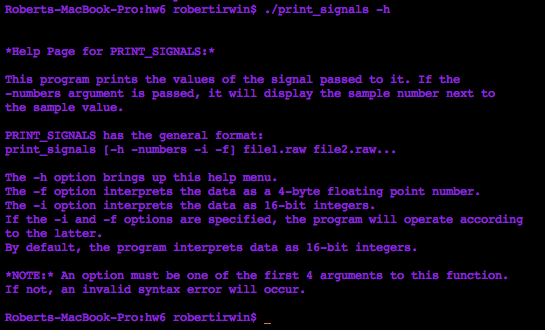
*Figure 9: Function read\_signal()*

The first thing we do in this function is determine if we are in –i or –f mode. We define the buffer accordingly, and use fread in a while loop to read 1024 bytes at a time. In between each fread, we print the contents of the buffer using a for-loop. When we print the contents of the buffer, we check the numbers flag. If it is one, we format and print a counter in front of each value.

The code for the float option is identical, with all “short int”s replaced by “float”s.

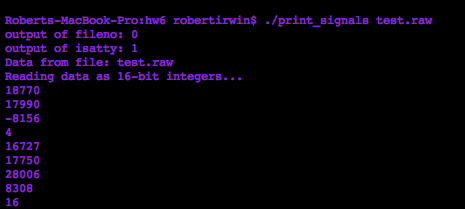
# Results

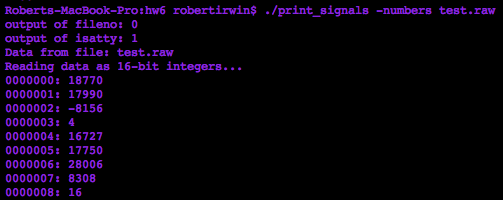
First we show the help file.



*Figure 10: Help Menu*

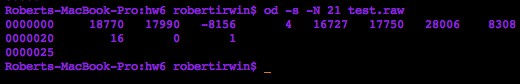
Now we look at the output from the command line arguments for short integers, with and without the indices.





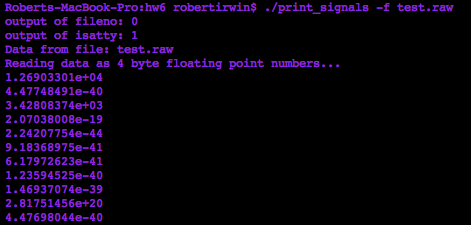
*Figure 11: Short Int Output*

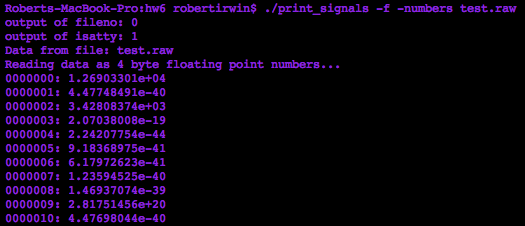
We now compare this to the first few data points of od.



*Figure 12: Output from Od for Verification*

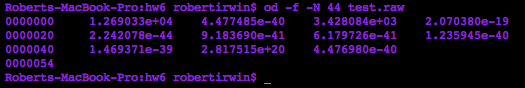
We see that the outputs match. We will now do the same thing for the floating-point numbers.



**

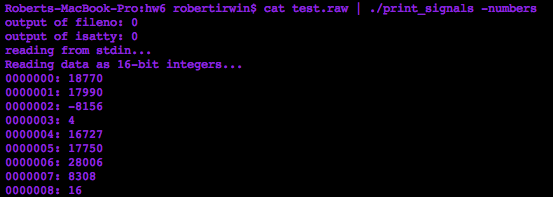
*Figure 13: Output for Floating Point Numbers*

Now we compare against the output of od.



*Figure 14: Output of Od for Verification*

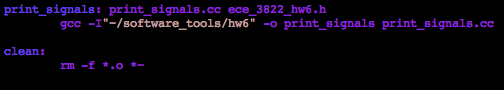
Now we will look at the same results when we pipe the arguments to our functions.



*Figure 15: Output from Piped Input*

We see that our function accepts piped inputs from standard in.

In order to make compiling our program a little easier, we create a makefile. The makefile is shown below. It is easier because our program not only depends on the main program, but also the contents of the header file!



*Figure 16: Makefile*

The –I option on gcc allows us to specify a path for the compiler to search. If we wanted to use our header in a program we were writing in a different directory, it would still find the header using this option. It is also important to note that our executable, print\_signals, is defined to be dependent on both print\_signals.cc and ece\_3822\_hw6.h. If we told it that print\_signals only depended on print\_signals.cc, we could add functions to the header that our main program would not have access to unless we recompiled the header separately. This makefile takes care of this detail for us.

# Analysis

In this assignment, we saw how to write a C program that accepts command line arguments and arguments from the standard input. We saw how to make our functions more dynamic by allowing the user to define how the program should be run. It was also interesting to see how the use of a header file can decrease the complexity of the main program.

The most important lesson to take from this assignment is how C reads data from files. We can see from this assignment, that the values you obtain from a file are dependent on how they read, and how they are displayed.