AERE 361

Computational Techniques for Aerospace Design

Programming Mission Manual

Fall 2021

Prof. Matthew E. Nelson

Contents

1 | Mission 6 - Functions and Files

Reference — **C Programming Language, 2nd Edition***:*
Chapter 7 - Input and Output

1.1 Mission Briefing

This mission will introduce students to the use of functions and files in C. We will cover some of the basics which includes how to declare a function, how to call a function and passing data in and out of functions. The student is expected to at least understand the basic concepts of these from their previous courses, so we will focus on how these are used in C.

We will also work with some files including reading and writing a file. Students will get experience with working text based files and how to store data into the file. Students will also gain some experience with reading from a text based file.

1.1.1 What you will need

For this homework, you will need access to a computer with Linux or a virtual machine version of Linux (WSL or Docker). That image will need to have GCC for this mission. If you are using the Engineering Linux Servers, these have the required Linux and GCC environments. If you are using Visual Studio Code with our Docker image, this will also have the necessary files.

1.1.2 Background

Functions are a fundamental part of the C programming language. They allow us to easily re-use blocks of code and they are flexible by allowing data and values to be passed in and the result to be passed out of the function. Functions can help us to streamline our code and make writing code more efficient.

Files also play a critical role when we need to store data for long term use or retrieve that data for analysis. We may also want to store the data to be analyzed in other program. For these reasons, we will want to write and read from files stored by the operating system.

1.1.3 Objectives

- Opening and closing files; managing input, output, and error streams
- Functions
- Passing data into and out of functions
- Learning more on Loops and Statements

Deliverable	Points
Exercise 1	5
Exercise 2	10
Exercise 3	10
Exercise 4	15
Exercise 5	15
Report	50
Total Score	**105**

Table 1.1: Rubric for Mission 6

1.1.4 Methodology

As we stated in Mission 2, you have two choices when it comes to doing your assignments. I often refer to this as your development environment and may also use the term workflow. The development environment is the system and software that you are using to compile and test your code. The workflow, generally, refers to the processes you take, write, test, and upload your code. Yes, some of that overlaps, but you can think of your workflow as the steps needed and the development environment as the system you are using.

As stated in Mission 2, you can either use the Engineering Linux machines or the Visual Studio Code setup we have embedded into the GitHub Classroom assignment. Either will work; it boils down to preference, but in both cases, we are using Linux and the GCC compiler to create our programs for this mission.

Students could use other sources such as the HPC or the Engineering Linux servers. Those machines have GCC installed and can be used. Other IDE's could be used, which include the online Repl.IT that I used last semester or even programs like Eclipse. However, a word of caution. While it is true you can use these; you are on your own. The support that myself, my TA, and graders can give you will be limited. You have been warned.

A word about compiling with LaTeX. You can edit your LaTeXdocument in Visual Studio Code. There is a way to compile your report in Visual Studio Code if you have TexLive installed on your machine. There is another approach where you can use a TexLive Docker image as well. You can also use Overleaf, but as I have cautioned before, there are a few things to be aware of in using Overleaf. What do I suggest? Edit your document in a text editor and then push those changes to GitHub. We have a GitHub action that will compile your document. Make sure you do a "git pull" once you know that the action is done. You can check on the status of any action by going to your repo in GitHub and then clicking on "Actions". If the "latex_build" action has a red check-mark, it means it failed to compile. You can click on it for more information. If it has a green check-mark, it compiled, and you can see the PDF in the repo branch under "reports".

> *To Do:*
> In your mission report, under the methodology section, explain what you used to work on your programming. Don't forget the objectives, which you can copy and past from this write-up. That will conclude the Mission Briefing section of the report.

1.2 Grading

There are 5 exercises in this mission. Please see the rubric in Table 1.1. This point breakdown is also in the README.md file and you should check things off in there just like you did in earlier missions. An AutoGrader will be used to grade that you completed the exercises provided. The remaining 50 points will be from your mission report. A copy of the rubric can also be found on Canvas.

1.2.1 Report

In the report folder you will find a starter "main.tex" file. Like the last lab, this template is very barren as you are expected to fill it out. At this point, you are now expected to read the lab writeup and when asked, fill out the appropriate section.

> *Be Careful:*
> Make sure your report compiles. It needs to compile using our LaTeX builder script in GitHub. We will always grade what the GitHub action generates. This happens during a push and can be triggered manually.

> *Remember:*
> Every time you do a push, our GitHub action compiles your report. It will also push a copy to your repo. For that reason, once you do a push, do a pull about 5 minutes later. This will help to avoid an error message saying you behind what is on GitHub and doing a git pull periodically is good practice when working in a team.

1.3 Con-Ops

1.3.1 Getting Started

To start with, go to the link found in Canvas for this lab. Click on it, and then clone the repo. You should have an icon for Visual Studio Code in your Readme.MD file. You can click on that to clone this to VS Code and start working. This is the same process you did for Mission 5 if you decieded to use VS Code.

> *Be Careful:*
> Don't forget that your first step is to create a new branch called "devel-netid". You will do all of your programming work in this branch.

If you decide to not use VS Code, then you can use the normal *git* commands to pull the repo, create a branch, and do all of your commits. Any computer running GCC and has the Git CLI tools installed should work. But, as we stated, you are own your own with this.

1.3.2 Functions

So far, you have used the "main" function as well as the "printf" and "scanf" functions. Consider your program as a system, the functions are all modules to build your system. To keep code organized and useful, a function should do exactly one job and stay relatively small (under 100 lines). There are many more rules of style that will be introduced as we dive deeper into C. The general form of a function definition in the C programming language is as follows:

```
return_type function_name( parameter_list ) {
   body
}
```

Where the parts are defined as:

- **Return Type**. A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword `void`.
- **Function Name**. This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters**. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. These are the function arguments. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body**. The function body contains a collection of statements that define what the function does.

1.3.3 File Management

Redirection

One way to read input into a program or to output from a program is to use standard input and standard output, respectively. That means, to read in data, we use scanf() (or a few other functions) and to write out data, we use printf(). When we need to take input from a file (instead of having the user type data at the keyboard) we can use input redirection:

```
$ a.out < inputfile
```

Here "a.out" is the executable file compiled from the C code. The same applies to the following. This allows us to use the same scanf() calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., inputfile above) instead of the keyboard. Similarly, there is output redirection:

```
$ a.out > outputfile
```

The above command allows us to use printf() as before, but causes the output of the program to go to a file (e.g., outputfile above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time...

```
$ a.out < inputfile > outputfile
```

C File I/O

While redirection is very useful, it is really part of the operating system (not C). In fact, C has a general mechanism for reading and writing files, which is more flexible than redirection alone.

Include file. There are types and functions in the library **stdio.h** that are used for file I/O. Make sure you always include that header when you use files.

File Type. For files you want to read or write, you need a file pointer, e.g. FILE *fp; What is this type "FILE *"? This is a special kind of pointer, and we will talk about pointers in the next mission. For now, just think of it as some abstract data structure, whose details are hidden from you. In other words, the only way you can use a FILE * is via the functions that C gives you.

> Note: In reality, FILE is a structure that holds information about the file. We must use a FILE * because certain functions will need to change that information, i.e., we need to pass the information around.

Functions. Reading from or writing to a file in C requires 3 basic steps:

- Open the file.
- Do all the reading or writing.
- Close the file.

The following describe the functions needed to accomplish each step.

Mode	Description
r	Opens an existing text file for reading purposes.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first resets the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Table 1.2: File open modes in C.

Open a file

You can use the fopen() function to create a new file or to open an existing file. This call will initialize an object of the type FILE, which contains all the information necessary to control the stream. The prototype of this function call is as follows:

```
FILE *fopen( const char * filename, const char * mode );
```

Here, "filename" is a string literal, which you will use to name your file, and access "mode" can have one of the values shown in Table 1.2.

If you are going to handle binary files, then you will use following access modes instead of the above mentioned ones

```
"rb", "wb", "ab", "rb+/r+b", "wb+/w+b", "ab+/a+b"
```

Read from or write to a file

fscanf/fprintf. Once a file has been successfully opened, you can read from it using fscanf() or write to it using fprintf(). These functions work just like scanf() and printf(), except they require an extra first parameter, a FILE * for the file to be read/written. An example to use fscanf and fprintf is shown in Listing 1.1.

Listing 1.1: fscsanf example

```
1  /* fscanf example */
2  #include <stdio.h>
3
4  int main (void) {
5    char str [80];
6    float f;
7    FILE * pFile;
8
9    pFile = fopen ("myfile.txt","w+");
10   fprintf (pFile, "%f %s", 3.1416, "PI");
11   rewind (pFile);  /*set the FILE pointer to the beginning*/
12   fscanf (pFile, "%f", &f);
13   fscanf (pFile, "%s", str);
14   fclose (pFile);
15   printf ("I have read: %f and %s \n",f,str);
16   return 0;
17 } /*end main*/
```

This sample code creates a file called myfile.txt and writes a float number and a string to it. Then, the stream is rewinded and both values are read with fscanf. It finally produces an output similar to:

```
I have read: 3.141600 and PI
```

fgets/fputs. You can also use fgets/fputs to read and write a buffer (string) to a file. An example is shown in Listing 1.2.

Listing 1.2: fgets and fputs example

```c
/* using fgets() and fputs() */
#include <stdio.h>
#define MAXLINE 20
int main(void)
{
    char line[MAXLINE];
    while (fgets(line, MAXLINE, stdin) != NULL &&
              line[0] != '\n')

       fputs(line, stdout);

    return 0;
}
```

In the above code, "stdin" are "stdout" are two special FILE pointers corresponding to the standard input and output. We will introduce them more in the following section. Compile and run this program. Try to give the program two inputs: "hello world" and "hello world hello world hello world hello world". Can you find anything strange?

> For the second input, its length is larger than 20, which exceeds the maximal length of the buffer "line" in the program. In your testing, does this program work fine? Why?

Function	Description
fscanf	Read (raw) data from a file.
fprintf	Write (raw) data to a file.
fgets	Read a buffer (string) from a file.
fputs	Write a buffer (string) to a file.
fread	Read a data structure from a file.
fwrite	Write a data structure to a file.

Table 1.3: File read/write APIs in C (most requently used).

fread/fwrite. The third option to read and write a file is to use fread/fwrite. These two APIs are more often used to read/write binary files. Examples for these two APIs are omitted. Table 1.3 shows a summary of the introduced FILE read/write APIs and their differences.

Close a file

When you're done with a file, it must be closed using the function fclose(). Closing a file is very important, especially with output files. The reason is that output is often buffered. This means that when you tell C to write something out, e.g.,

```
fprintf(ofp, "Whatever!\n");
```

it doesn't necessarily get written to the disk right away, but may end up in a buffer in memory. This output buffer would hold the text temporarily:

When the buffer is full, the file is closed, or is explicitly asked (with a "flush" command), the data is finally written to the disk. So, if you forget to close an output file then whatever is still in the buffer may not be written out, causing data loss!

Special file pointers

There are 3 special FILE * pointers that are always defined for a program. They are "stdin" (standard input), "stdout" (standard output) and "stderr" (standard error).

Standard input is where things come from when you use scanf(). In other words,

```
scanf("%d", &val);
```

is equivalent to the following fscanf():

```
fscanf(stdin, "%d", &val);
```

Standard Output Similarly, standard output is exactly where things go when you use printf(). In other words,

```
printf("Value = %d\n", val):
```

is equivalent to the following fprintf():

```
fprintf(stdout, "Value = %d\n", val);
```

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless redirection is used.

Standard error is where you should display error messages. For example,

```
fprintf(stderr, "Can't open input file in.list!\n");
```

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

```
$ a.out > outfile
```

only redirects stuff going to standard output to the file "outfile". Anything written to standard error goes to the screen.

1.3.4 Writing a file from scratch

In future missions we may not provide you with a starter file, so you will need to create the file from scratch. Future missions may or may not have any starter code depending on the learning objectives for that mission. However, this section will walk through what you should or need to include. It also does not hurt to look back at previous labs for examples as well.

1.3.5 Taking owner ship of your C program

All programs need to have a header that has the following:

```
AerE 361
Mission #
Exercise #
YOUR NAME HERE
```

Replace the "#" with the number of the Mission and exercise and of course replace the "YOUR NAME HERE" with your name. You might think it is silly that I need to state that and yet, you have no idea how many students leave that information there.

Remember
Don't forget that the information above must be in a comment. You can use `//` for a single line or the `/* */` to create a block. Look at the other exercises, you should be able to mimic those.

Setting up your C program

Next recall that you will need to add your includes. These are important as it will bring in the standard C libraries that we will use. You can look at the code below on how you can add these includes. You will need both the `stdio.h` and the `string.h` headers to be included. Listing 1.3 shows an example of this.

Listing 1.3: Example of including libraries in C

```
#include<stdio.h>
#include <stdlib.h>
```

And finally, you will need to include your main function. As we have done in previous labs, we will be passing in arguments from the autograder, so you will need to do the same. Look at the code in Listing 1.4 for how this can be done. You can also look at previous code. Also, don't forget that at the end of your code, you need to close the main function with a `}`. Without that, your code will not compile.

Listing 1.4: Example of a main function.

```

/*begin main function*/
```

Now that we have our main function, we need to have an if statement that handles between asking the user for the information or if we have data passed in (like from the autograder) then we need to use that data. You have seen this before in previous labs. Now you will need to write this yourself. A big hint on what this looks like for Exercise 2 is shown in Listing 1.5.

Listing 1.5: Example of checking for argument or user input.

```
// Do not change anything below.
// -----------------------------------------------------------------
        lines = atoi(argv[2]);
        if ( argc > 1){
                strcpy(filename,argv[1]);
        }
        else{
        printf("Please let me know the filename: ");
        fgets(filename,100,stdin);
        length = strlen(filename);
```

1.4 Exercises

Hint:
Don't forget that you can test your work. A `make test` will test all programs. Look at the Makefile we provided, you should be able to see how you can call other tests as well. In most cases, it is straightforward such as `make test_number_gen`. If your program passes there, it should pass on GitHub. Use them! Also, the Makefile will compile your programs as well, again, look in the Makefile

1.4.1 Exercise 1

Exercise 1. (5 points) - Use the provided template file called `number_gen.c` and create a program that creates a file based on the name provided by the user. If the file already exists, exit with an error message

and a non-zero exit code. If the file does not exist, create the file and populate it with numbers from 1 through 100 with each number on a separate line.

For Exercise 1, your file should contain numbers that look like the following:

```
1
2
3
4
5
.
.
.
98
99
100
```

Execute your program and tell it to save the numbers in a file called `number_list.txt`. Once you are done, do not delete this file, you will use this file in the next labs. For report, include both the code that you wrote and a copy of the output generated from your program. Put the output of your program and a copy of your code in the **Results** section. In the **Analysis** section, explain how you generated the numbers used in your program.

1.4.2 Exercise 2

Exercise 2. (10 points) - Create a program that will now ask the user for a file name to read in and also how many lines starting at the top they would like to have viewed. This will be almost identical to the `head` program that you can use in Linux.

When running this program, use the `numbers_list.txt` that you generated in Exercise 1. If you tell it to use that file and to show you three lines, then you should see something like what is below. If you accidentally deleted your file, just re-run your `number_gen` program to recreate it.

```
1
2
3
```

Execute your program and copy the output generated. For the report, include both the code that you wrote and a copy of the output generated from your program. Put the output of your program and a copy of your code in the **Results** section. In the **Analysis** section, explain how you were able to look at only the first few lines from a text file starting at the top. This should include things like any loops or statements you used and why.

1.4.3 Exercise 3

Exercise 3. (10 points) - Create a program that will now ask the user for a file name to read in and also how many lines starting at the *bottom* they would like to have viewed. This will be almost identical to the `tail` program that you can use in Linux. Your C program must be a in a file called `tail.c`

Refer back to Exercise 2 and make sure that you again, add the correct main function and add your includes. The standard libraries needed will be the same as in Exercise 2. Don't forget to add the header with your name on it and of course, don't forget to close your main function with a `}`. When executed, your program should produce something like what is below, using the number file we generated and telling it we wish to see three lines.

```
98
99
100
```

Execute your program and copy the output generated. For the report, include both the code that you wrote and a copy of the output generated from your program. Put the output of your program and a copy of your code in the **Results** section. In the **Analysis** section, explain how you were able to look at only the first few lines from a text file starting at the bottom. This should include things like any loops or statements you used and why.

1.4.4 Exercise 4

Exercise 4. (15 points) - Create a program that will now ask the user to enter a value. This value can be a float (double is fine). This value must be a positive value so your program must check for that. Then ask the user for an allowable error. Your program must check that the user entered a value and the value should be between 0 and 1. Finally your program will calculate the Maclaurin Series and make sure the series falls within the error specified.

Your C file for this must be named `mac_exp.c` in order for it to work with our scripts. Again, you need to have code that checks if arguments from the main function and store the value and the error into the same variables that you use in your program. You can use `atof` to convert from the character from the argument to a double.

Listing 1.6: The print statement you should use in your program

```
1 ||          printf("After %d terms in the series, exp(%lf) is approx. %lf with an error of %lf \
  ||              n", itr, x, ans, error_temp);
```

For this program you must use the print statement as in Listing 1.6. You will also use the built in math function to calculate the power of the values in the series. In order to do this, two things need to be done. First, you need to include `math.h` which can be done as shown in Listing 1.7.

Listing 1.7: Include statements you should have in your program

```
1 ||#include <stdio.h>
2 ||#include <stdlib.h>
3 ||#include <math.h>
4 ||#include <string.h>
```

When compiling your program, you now need to add a new flag to GCC. Before this, you were simply using `-o` to indicate the output file. Now we will add `-lm` which tells the compiler to link to the math library for C. To issue this at the command line, it would look like this:

```
gcc mac_exp.c -o exp.out -lm
```

You can also see that we have this in the Makefile we provided you. So you can use the Makefile to compile this as well. Remember, you can use this to both compile and test your code.

You will write a function called `factorial` that will take in an integer and return a double. Recall that a factorial is defined in Equation 1.1. You can use a for loop for this.

$$n! = n * (n-1) * (n-2) * (n-3) \cdots 3 * 2 * 1 \tag{1.1}$$

Now, we will use this function in a Maclaurin Series. A Maclaurin Series is a special case of a Taylor Expansion function defined in Equation 1.2. This is a power series that allows us to approximate the function $f(x)$ for input values close to zero.

$$\sum_{n=0}^{\infty} f^{(0)}(0)\frac{x^n}{n!} = f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + \cdots \tag{1.2}$$

This may seem fairly complicated, but in our case we can simplify this. We can use a loop to loop through this until we get to a low enough error for our approximation. We can boil this down to Equation 1.3.

$$y = y + \frac{x^{itr}}{!itr} \tag{1.3}$$

Where itr is the iteration we are in, in our loop, and is an integer value. The value x is the number we obtain from the user and y is the result calculated with an initial value of 0. Within the Math Library, we have a function called `pow`. This function takes in two values, the base value and the power value. For example, you can use it like in Listing 1.8.

Listing 1.8: Example of implementing the Maclaurin Series

```
1 ||                ans = ans + (pow(x, itr) / factorial(itr));
```

To calculate the error, we simple take the absolute value between the current value calculated in the series and the previous value calculated. With the C Math library, this can be done by calling the `fabs` function. If the error is greater than our user inputted error, then we continue in the loop. If not, then we will continue in the series.

To finish this exercise, put the output of your program and a copy of your code in the **Results** section. In the **Analysis** section, explain how you wrote the functions. This should include what kind of loops you used or any other snippets of code.

1.4.5 Exercise 5

Exercise 5. (15 points) - Create a program that is capable of doing a brute force adder, a Gauss adder, a Factorial, and a Series Value. This program must have a user interface where the user can select what they wish to do and enter in the appropriate values. The program must have an exit option to quit the program. Like all all programs, it must also be able to support having values passed in so the autograder can test your code.

Now we will create a fairly fully operational program. Save this in a file called `361_calc.c`. Recall from Mission 5 that you created a program that implemented a brute force adder and Gauss adder. We will now expand on this with a few changes.

- Move the adder code into functions that can be easily called.
- Add the ability to compute a factorial
- Add the ability to compute a Series Value

You have done a factorial in Exercise 4. Since this is already in a function, you should be able to copy this into this program. For the Series Value, we will use a much simpler version. This is shown in Equation 1.4.

$$\sum_{k=0}^{\infty} \frac{a}{!k} \tag{1.4}$$

Here, a is a float value from the user and k is the number of items we are computing this series. This can be done simply in a for loop. Another hint is that you can use `+=` to add and store the result into a variable.

As before, make sure you have code setup to handle whether there is a user input or arguments being passed in. For this exercise, I want you to focus more on the functions and converting a mathematical expression

into C code. So I will provide you the code that will accommodate handling this information. You should however student and understand how this code works. You can see this code in Listing 1.9.

Listing 1.9: Code for checking for arguments and and user input

```
do{
        if (argc == 1){
                print_menu();
                scanf(" %d", &reply);

                if (argc==1 && reply !=0 && reply <= 3){
                        printf("Enter a number: ");
                        scanf(" %lu", &x);

                }
                if (argc==1 && reply == 4){
                        printf("Enter a number: ");
                        scanf(" %lf", &y);
                        printf("Enter a value for x: ");
                        scanf(" %lu", &x);

                }
        }
```

Finally, you should create a function that generates your menu system. Since no information needs to be passed in or out of this function, this should be of type `void`. Your menu should look like what is in Listing 1.10.

Listing 1.10: Example of the menu system

```
        printf("Welcome to the 361 Math program.\n");
        printf("==================================\n\n");
        printf("Please select from one of the following options.\n\n");
        printf("1....Brute Force Adder\n");
        printf("2....Gauss Adder\n");
        printf("3....Compute Factorial\n");
        printf("4....Compute Series Value\n");
        printf("0....Exit Program\n\n");
        printf("Enter option: ");
```

Don't forget your header information and for includes, you should once again include `stdio.h`, `stdlib.h`, `string.h`, `stdbool.h`, and `math.h`. Don't forget to commit your changes. And that is it, you are done with this exercise.

To finish this exercise, put the output of your program and a copy of your code in the **Results** section. In the **Analysis** section, explain how you wrote the functions. This should include what kind of loops you used or any other snippets of code.

1.5 Report

Your report should have your code outputs from the programs you ran above. It should also include copies of the code you wrote. Finally, make sure you explained how you solved the problems for each Exercise in the **Analysis** section. Remember that your report must be in a file called `main.tex` in a directory called `report` and must build without errors.

1.6 Wrapping Up

Congratulations! You have finished Mission 6. When you are finished with your lab, check that you have everything done below, and enjoy the rest of your day. Don't forget that you can always check your work by

running `make test` and make sure everything passes in GitHub. If everything passes, you should be good to go.

☐ **Read the mission write-up**

☐ **Clone this repo into your work environment, create your devel-netid branch**

☐ **Complete all 5 exercises**

☐ **Push your changes to the devel-netid branch**

☐ **Complete your mission report**

☐ **Open a new Pull Request (and leave it open for the grader to find)**