ECE 264: Advanced C Programming

Programming Assignment 4 - Files, Pointers, and Structures, Due 7/2/19

This assignment continues from PA03. In PA03, the function to be integrated is not part of the arguments to integrate. This assignment makes the function part of the argument. As a result, the integration function is more flexible because everything needed is passed as arguments.

1 Learning Goals

You will learn how to:

- Create a type for functions that have the same types of arguments and return value
- Make a structure with an attribute as a function pointer
- Write a function that takes a pointer to this structure (modified from PA03)
- Write Makefile to test
- Write a main function that test integration function
- Write a function that integrates multiple functions
- Read data from a file and write output to another file
- Detect errors and return proper code (true, false, EXIT_SUCCESS, or EXIT_FAILURE)

Please be aware that this you need to write a lot more for this assignment. Please start early. You are encouraged to study PA01 - PA03 thoroughly before starting this assignment. This assignment is designed with the assumption that you are familiar with the details in PA01 - PA03.

2 Background

2.1 Function Pointers

Every function in a C program refers to a specific address. You have seen that in PA02. It is possible to create pointers to store functions' starting addresses, and then call functions by using those addresses, instead of the function name directly.

In C, a function pointer is declared by this syntax:

typedef type (* typename) (arguments)

For example:

typedef int (* funcptr)(int, int);

creates a type for pointers to functions that take two integers as input arguments and return one integer.

Here is an example of creating and using function pointers:

```
//create a function pointer type called "myfuncptr"
//myfuncptr functions take one int as an argument and return an int
typedef int (* myfuncptr)(int);

myfuncptr f = foo; //f now refers to the function foo
```

```
int x = f(3); //equivalent to: int x = foo(3)
```

In this assignment, you need to create a type called funcptr (you must use this name) for the functions you want to integrate. A function of type funcptr should take as argument one double, and return a double.

With this type, it is possible to create an array of five functions called func1, func2, ..., func5.

The program can go through these five functions in the same way as going through normal array elements.

2.2 Function Pointer as Attribute/Field

An important concept in developing high-quality software is making a function's behavior controlled by the argument(s) and nothing else. In PA03, the function to be integrated was not in the argument – we assumed it was a function called func. This can easily create confusion. Similarly, global variables and static variables can create confusion because they can make functions behave in ways that are not completely controlled by the input arguments. Thus, you should avoid using global and static variables.

After creating the type for function pointers, it is possible to make a function pointer an attribute of a structure. Now, everything needed to run the integration function is passed into the integration function as the argument.

2.3 Reading and Writing Files

This assignment asks you to read and write data from files. You can see examples of doing this in earlier assignments (PA01 and PA03)

The key to working with files is to understand the basic file manipulation API (an "Application Program Interface" — the functions that provide certain behavior):

- FILE * : All files are manipulated through pointers to FILE structures. We will call the pointer to a FILE structure a file "handle". FILE * inf; declares a file handle called inf.
- FILE * fopen(const char * path, const char * mode): Open a file whose file name is stored in the character array path. The character array mode indicates how you want to access the file:
  - "r": open the file for reading, starting at the beginning of the file.
  - "r+": open the file for reading and writing, starting at the beginning of the file
  - "w": open the file for writing; this deletes any content already in the file.
  - "w+": open the file for reading and writing; this deletes any content already in the file.
  - "a": open the file for appending, starting at the end of the current contents of the file.
  - "a+": open the file for reading and appending: reads start at the beginning of the file, writes append to the end of the file.

  inf = fopen("test1", "r") will open the file named "test" for reading, and return the necessary file info to the FILE pointer inf. If inf is NULL after calling fopen, then the operation failed. You can find out more by typing man fopen on the ecegrid machines.
- int fscanf(FILE * file, const char * format, ...): This works just like scanf, but reads from the file pointer file. fscanf(inf, "%lf
  n", &d) reads a double (%lf) from the input file and stores the result in d. You can find out more by typing man fscanf on the ecegrid machines.

- int fprintf(FILE * file, const char * format, ...): This works just like printf but writes to the file pointer file. Note that for fprintf to work, the file you pass should have been opened in a mode that allows writing. You can find out more by typing man fprintf on the ecegrid machines.
- int fclose(FILE * file): Close the file. You should always close files when you are done using them.

3 What do you need to do?

Your job is to write an integration function using the same integration method as PA03 (you can even re-use your code from there). Unlike the integration function in the previous assignment, however, your integration method will need to accept a new kind of structure, one that contains a function pointer pointing to the function you want to integrate. This will require both defining the function pointer type you need, as well as a new structure that contains all the necessary information to perform an integration.

3.1 Files You Need to Modify

Unlike in previous assignments, this assignment asks you to modify many files. The exact details of what each file should contain are provided in the initial files for the assignment (pay careful attention to the comments in each file!), but briefly, here are the files you need to modify, and what you need to do:

- pa04.h: You will define a function pointer type and a new structure that contains all the components necessary to perform an integration, including a pointer to the function to be integrated, as well as a field to store the result of the integration. Please follow the naming convention mentioned in pa04.h strictly. While calculating partial credit, if your fucntion names and variable names do not match with the instructor's key, your program will fail to compile and you will be given zero credit.
- answer04.c: This file contains your integration functions: integrate, which takes the structure you defined in pa04.h and uses that information to perform a numerical integration, and runIntegrate, which takes the name of an input file and an output file as arguments and:
  1. Reads the upper limit, lower limit, and number of intervals from the input file.
  2. Uses integrate to numerically integrate five functions (provided for you, called func1, func2, func3, func4, and func5).
  3. Writes the results of each integration out to the output file
- pa04.c: This file contains your main function, which gets the names of the input and output files from the command line (argv[1] and argv[2]) and calls runIntegrate.
- Makefile: You will need to write your own Makefile to build and test your program (the template we provide only defines one "recipe": clean). See below.

3.2 Improve Makefile

By now, you have seen at least three Makefile in PA01, PA02, and PA03. Please read them carefully and understand what they do.

You should have discovered that every Makefile so far has this pattern

```
file.o: file.c
    $(GCC) $(CFLAGS) -c file.c
```

and file is replaced by specific file names.

If five object files are needed, this pattern has written five times. This is tedious and error prone (do you remember DRY vs. WET code)?

Fortunately, there is a simple way to write this as a rule.

```
c.o:
    \$(GCC) \$(CFLAGS) -c \$*.c
```

This means whenever a .o file is needed, Makefile will look for the corresponding .c file and use gcc (with the flags) to compile it. This can dramatically reduce the effort writing Makefile

Makefile can do much more than creating executable files. You can run test cases using Makefile. You have seen this in PA01 and PA03.

In PA01, testsome and testall show how to run multiple test cases. In PA03, pa03-test1 and pa03-test2 also show how to run multiple test cases.

Your Makefile for PA04 needs to run the five test cases stored in testdir, using a target called testall. **Remember to have your testall recipe compile your files if needed!** Look at previous Makefiles we have given you to see how to add this dependency. **The name of the executable should be 'pa04'**. This is essential as we will use your executable file to test some other testcases as well. Also, when you run the five test cases using 'testall' target, the output files generated for each input test file should be named output_<test file name> e.g. for file test1, ouptput file name should be output_test1.

Don't forget to define the flags RUN_INTEGRATE and TEST_INTEGRATE to make sure that the code you write in answer04.c gets compiled!

3.3 main function

This is the first assignment that requires a complete main function.

3.4 Partial Credits

It is very important that you follow the instructions precisely (including functions' names and variables' names). These names allow the grading programs to replace some of incorrect parts by correct parts (written by the teaching staff) and give you some points.