Appendix C

File Handling

C.1 Overview

Up to this point in the course you have used functions from the standard C library to display information on the screen and receive information from the user. This laboratory will introduce you to a set of very similar functions which allow you to both write and read information to/from files stored on the computer or a network drive.

In the final part of the lab you will use this to store information from the program you have been working as part of you graphics or music option.

C.2 Getting Access to Files

Since very nearly the beginning of this course you have been using the `printf` function for output to the screen and the `scanf` function for input from the keyboard. This lab will use very similar functions, `fprintf` and `fscanf`, to output and input to/from a file.

Before you can either read or write to/from a file you have to request access to the file. When you do this, the file you have requested is usually *locked* meaning that only your program can access the file at that time. This is called *opening* the file. When you are finished with the file you must *close* it again.

When you open a file you are given a pointer which acts a shorthand for the file you are talking about. This pointer is called a *file handle* or a *file pointer*. You get a different file handle for each file that you open. Once you have closed the file, the file handle is no longer valid. A file handle variable is declared like this:

```
FILE* file_handle;
```

To open a file you should use the `fopen` function. This function requires two arguments, both of which are strings: the name of the file you want to open and the *access mode* you wish to have. The access mode is to do with what kinds of operations you wish to carry out on the file. Specifying `"w"` indicates that you wish to write to the file; `"r"` specifies you wish to read and `"w+"` specifies that you want to be able to both read and write. For example, the following line of code:

```
file_handle = fopen("number.txt", "w");
```

opens the file `numbers.txt` for writing. If the `fopen` function could not open your file, for whatever reason, the file handle (in this case the `file_handle` variable) is set to `NULL`.

When you are finished with the file, you must close it again using the `fclose` function. This function takes only one argument: the file handle of the open file you wish to close:

```
fclose(file_handle);
```

Function Reference: `fopen` — Opens a File

```
FILE* fopen(char* filename, char* access_mode);
```

e.g.

```
file_handle = fopen("data.txt", "w");
secret_file = fopen("important.dat", "r");
temp_file = fopen("temporary.tmp", "rw");
```

The `fopen` function attempts to open the file specified by the `filename` argument with the access mode specified by the `access_mode` argument. Both of these arguments should be strings. Valid access modes are:

- `"r"` open file for reading, the file must already exist;
- `"w"` open file for writing, if the file already exists it will be overwritten;
- `"a"` open file for writing, if the file already exists it will be not overwritten, writing takes place at the end of the file (this is called *append* mode);
- `"r+"` open file for both reading and writing, the file must already exist;
- `"w+"` open file for both reading and writing, if the file already exists it will be overwritten;
- `"a+"` open file for both reading and writing, if the file already exists it will be not overwritten, reading and writing take place at the end of the file.

If the file cannot be opened the `fopen` function returns `NULL`.

The `fopen` function is defined in `stdio.h`.

Function Reference: `fclose` — Closes an Open File

```
int fclose(FILE* file_handle);
```

The `fclose` function closes an open file identified by the argument `file_handle`. If there is a problem during the close operation the `fclose` function returns the predefined constant `EOF`. If there were no problems the function returns zero.

The `fclose` function is defined in `stdio.h`.

C.3 Simple File Read and Write Operations

Change the target to "appendixb" and open "appendixb.c" in the IDE. The source code for the "appendixb.c" file is shown below:

```
/*
 * A Program to Demonstrate File Handling
 * Lab appendixb
 */

#include <stdio.h>

int main(void)
{
    FILE* file_handle;
    int number;

    /* Ask the user for a number */
    printf("Please enter an integer number: ");
    scanf("%d", &number);

    /* Get access to the file by opening it */
    file_handle = fopen("number.txt", "w");

    /* Write a single number to the file */
    fprintf(file_handle, "%d", number);

    /* We're done with the file, close it */
    fclose(file_handle);

    return 0;
}
```

This simple program first obtains an integer number from the user. It then opens the file `number.txt` for writing. The next line of code uses the `fprintf` function to write to the file. The `fprintf` function works exactly like the `printf` function, or the `sprintf` function. The difference between the `fprintf` function and the `printf` function is that the `fprintf` function takes an extra first argument: the handle of the file to write to. For example, the code:

```
fprintf(file_handle, "Hello");
```

Outputs the text "Hello" to the file identified by the file handle `file_handle`. The line in the "appendixc" program:

```
fprintf(file_handle, "%d", number);
```

Outputs the integer number held in the `number` variable to the file as text.

Function Reference: `fprintf` — Outputting to a file using `printf`-like placeholders

```
int fprintf(FILE* file_handle, char* format_string, ...);
```

e.g.

```
fprintf(file_handle, "%d", int_variable);
fprintf(result_file, "The answer is %d", integer_answer);
num_chars = fprintf(temp_file, "%lf", double_variable);
```

The `fprintf` function works like the `printf` function except that, instead of outputting its results onto the screen, it places them into the file specified by the `file_handle` argument. It is called in exactly the same way as `printf` but with an extra first argument (`file_handle`).

`fprintf` takes at least two arguments but, like `printf`, it can take more, depending on how many variables you want to include.

- `file_handle` is a handle to a file, previously opened with `fopen`, which will be used as the destination for all `fprintf` write operations;
- `format_string` is a string containing placeholders, just like with `printf`;
- after the first two arguments (where the `...` is) you should include variables whose values will be substituted for the placeholders in the `format_string`.

The placeholders you will use most often are:

- `%d` for integer (`int`) variables;
- `%lf` `double` variables;
- `%c` for single characters (`char`);
- `%s` for strings (`char[]`).

These are the same as for `printf`.

The return value of `fprintf` is an integer, which specifies how many characters were written into the file (not including the null termination). This number will be negative if an error occurs.

The `fprintf` function is defined in `stdio.h`

Exercise C.1: Investigating File Writing

Try building and executing the "appendixb" program. When it has terminated. Go to the "File manager" and double click on your "clab" folder. Find the executable correspondng to appendixb (in the "Debug" folder). You should see a file called `number` or `number.txt`. If you double click on this file it will open. You should see that the number you entered into the "appendixc" program is stored in this file as text. Note, if you ran the program from within the Code::Blocks IDe it actually creates the file "number.txt" in the folder where the "clab" project folder sits (i.e. the "clab" folder.

Just as writing to a file can be done with `fprintf`, reading from a file can be done with `fscanf`. The `fscanf` function works just like the `scanf` function except that it takes an extra first argument: the handle of the file that it should read from. For example, the following line of code reads an integer from

a file:

```
fscanf(file_handle, "%d", &int_variable);
```

Function Reference: `fscanf` — Performs a `scanf`-like read from a file

```
int fscanf(FILE* file_handle, char * format_string, ...);
```

e.g.

```
fscanf(file_handle, "%d", &an_int_variable);
fscanf(text_file, "%c", &a_char_variable);
number_of_matches = fscanf(result_file, "%lf", &a_double_variable);
```

The `fscanf` function obtains textual input from the file identified by the `file_handle` argument. This handle must identify a file previously opened by the `fopen` fucntion for reading. The input is converted according to the `format_string` argument. The `format_string` argument specifies placeholders in an identical manner to the `scanf` function.

`fscanf` takes at least two arguments but, like `scanf`, it can take more, depending on how many variables you want to include.

- `file_handle` is a handle to a file, previously opened with `fopen`, which will be used as the source for all `fscanf` read operations;
- `format_string` is a string containing the place holders that `fscanf` will try to match input to;

`fscanf` uses the same place holder system as `scanf`. As a reminder, the most useful place holders are:

- `%d` for integer (`int`) variables;
- `%c` for single characters (`char`);
- `%lf` real valued (`double`) variables.

The return value of `fscanf` is the number of place holders that were successfully matched or the predefined constant `EOF` (End Of File) if there was a problem.

The `fscanf` function is defined in `stdio.h`.

Exercise C.2: Reading from a File

Alter the "appendixb" program so that it reads a number back from the `number.txt` file using `fscanf` and displays it to the user. Remember to change the access mode in the `fopen` function call.

C.4 Reading and Writing Many Lines to/from a File

The `fprintf` and `fscanf` operations are very powerful, they are not just limited to writing a single number into a file and reading it back again. There is no reason why you cannot place a `fprintf`

function call like:

```
fprintf(file_handle, "%d\n", numbers[index]);
```

into a loop to write many consecutive number to a file from an array. Notice the use of the new-line escape sequence (`\n`) to make sure that each number is on a new line in the file.

The next exercise will use the `rand_number` which is defined in the graphics library file `graphics_lib.h` or the equivalent function `random_number` which was discussed in the "music option" in lab 4 and is defined in the music library `music_lib.h` [1]. The function generates an integer random number within the bounds that are specified by the two arguments to the function. For example, the function call:

```
number = rand_number(1, 10);
```

Will assign a random number to the `number` variable. The random number will be anywhere between (and including) 1 and 10.

Function Reference: `rand_number` — Generates a random number

```
int rand_number(int lower_range, int upper_range);
```

The `rand_number` function returns a random number which could be anywhere in the range `lower_range` to `upper_range`, including the values `lower_range` and `upper_range`. For example:

```
x = rand_number(60, 71);
```

will assign a value to `x` that could be 60, 71 or any integer value in-between.

`rand_number` is defined in `graphics_lib.h`.

Exercise C.3: Writing Multiple Numbers to a File

Alter the "appendixb" program so that it fills an array of 40 elements with random numbers and writes all of the numbers to a single file. Each number should be on a new line in the file. Open the file you have written into to check the write operation worked correctly.

Adjust the program to write 500 random numbers between 1 and 100 to a file called `numbers.txt`.

Exercise C.4: Reading Multiple Numbers from a File

The "appendixb-a" program is a nearly-empty template for this exercise, change the target to "appendix-a" and open the source file in the IDE. Write a program that opens the file that was to written by the program in Exercise C.3. It should read 20 numbers from this file into an array. At the end of the program it should display the numbers.

[1] the reason why each library has an implementation of the same function with a different name is so that if both libraries were included in a program, no conflict would occur.

Exercise C.5: Making the Read Operation More Flexible

The "appendixb-a" program you have just written depends on there being 20 numbers in the file it is reading from. If there are more than that, the numbers do not get read. If there are fewer, there will be a problem. (In this case the `fscanf` function will return the constant `EOF`, which stands for End Of File).

Alter the "appendixb-a" program so that it will read any amount of numbers in from a file. The numbers should be read into an array, the memory for which is dynamically allocated using `malloc`. To do this the simplest thing to do is read the file twice [a]:

- once to determine how many numbers there are in the file so that you can allocate the array;
- a second time to read the numbers into the array.

[a] Using a standard C function called `realloc` which allows a redefinition of the memory required for an array makes it possible to acheive this with a single read of the file

C.5 File Operations with Structures

The examples we have looked at so far have only written or read a single number with each `fprintf` and `fscanf` function call. There is no reason why this has to be the case. A `fprintf` function call could insert multiple numbers into a file, like this:

```
fprintf(file_handle, "%d, %d\n", first_number, second_number);
```

This would write two numbers on a single line in the file, separated by a comma and a space. To read these numbers back in from the file we would use the following `fscanf` function call:

```
fscanf(file_handle, "%d, %d\n", &first_number, &second_number);
```

There is also no reason that we must only work with numbers. We may also work with strings.

In labs 6.1 you worked with a structure which stored information about a student. You created a type definition for this structure called `student_type`. To write information from this type of structure to disk you would use a piece of code like the following:

```
fprintf(file_handle, "%s, ", student.family_name);
fprintf(file_handle, "%s, ", student.given_name);
fprintf(file_handle, "%d, ", student.year_of_birth);
fprintf(file_handle, "%d\n", student.course_code);
```

Notice that each of the pieces of information about the student will all appear on the same line in the file, each separated by a space and a comma. A similar system could be used for reading information in.

Exercise C.6: Reading and Writing Structure Information to/from a File

Alter the "appendixb" program so that it prompts the user for details about a student and stores them in a structure. Write a separate function which will write the details of that student to a file. The student details should all appear on the same line in the file, separated by commas.

The "appendixb-b" program is another near-empty template. Basing your "appendixb-b" program on "appendixb-a", write a program which uses a separate function to read the details of a single student in from the file that was written to by the "appendixb" program. The program should display the details it reads in.

You might like to copy some of your code from the program you worked on in lab 6 to make this task easier.

Exercise C.7: Adding Save and Load Functionality to a Program

If you have been following the graphics option, change the target to "graphics3"; if you have been following the music option, change the target to "music3" and open the corresponding source code in the editor window. Each of these programs use an array of structures to store either a drawing (in the case of the graphics option) or a musical sequence (in the case of the music option). Based on the exercises you have done in this lab, add functionality to your program to allow the user to save the contents of the array to a file, or to load all the information from a file into the array. This will allow a user to save and then reload their drawing or sequence.

C.6 Summary

Now that you have finished this laboratory you should understand how to carry out basic file operations. You should know that:

- to begin operating with a file, you must *open* it with the `fopen` function;
- when you have finished using the file you should *close* it with the `fclose` function;
- when operating on the file you use an identifier that you obtained from the `fopen` function called a *file handle*;
- you can write to the file using the `fprintf` function, which is very similar to `printf`;
- you can read from the file using the `fscanf` function, which is very similar to `scanf`.

You should have used these concepts to add load and save functionality to the program that you have been working on as part of your option. This gives you the ability to save drawings as a file (in the case of the graphics option) or musical sequences as a file (in the case of the music option).