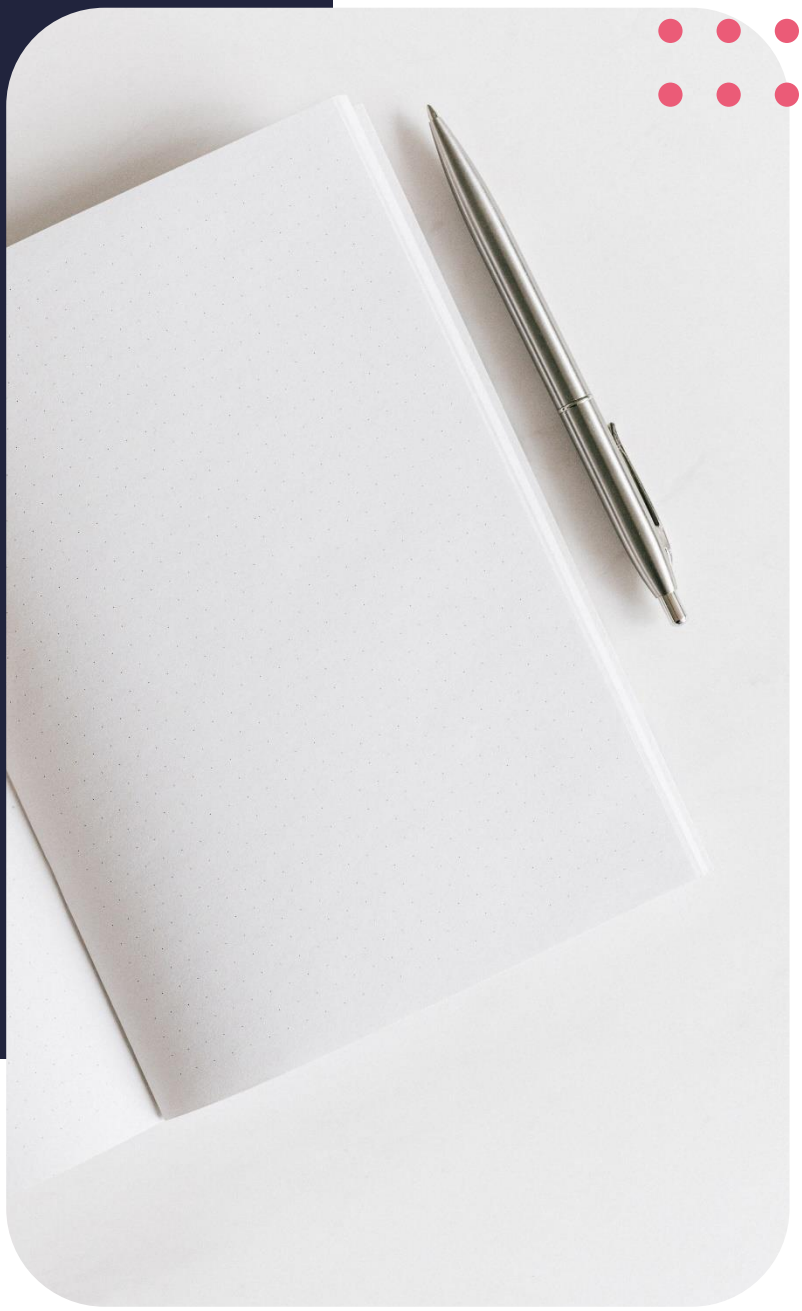
Diploma in Computer Science

File Management

Contents



Lesson objectives

By the end of this lesson, you should be able to:

- Understand the file system
- Explore how programs access files
- Distinguish between file types
- Use C code to access and manipulate filesAdjust the width and length of this pink bubble accordingly

Introduction

Welcome to our 4th lesson. Today we take a look at one of the most common things on the computer, the one thing that is used for literally everything from execution to storage to classification and all that: the file. In this lesson, we will be looking at how C handles files, and allows you to read file contents and write to files. We will also look at the different types of files and how the computer distinguishes between them. And we'll talk about command line arguments and see how they help us to pass arguments at runtime. Without wasting anymore time, let's get into it!

Why files?

A file is a very common object in computing. It is rare to talk about anything at all in terms of computing and not mention a file. Files are the basis of most computer systems out there. In fact, it's hard to think of any operating system at all without thinking of files. This is how we store information, including the programs that are run by that computer. You recall from module 1 that an operating system is just a collection of files that contain the programs that are run by the computer, configuration files, alongside your documents and pictures of your cats. It's like that everywhere. Everything is contained in some form of file. Let's look closely at how this is done.

Storage of data

We have mentioned previously that everything on a computer is contained on a file. A file is a computer's basic means of storing data. You will recall from module 1 that data in a computer is represented by a series of ones and zeros, known as binary digits.

Looking at binary digits, they can go on and on and look so meaningless to the untrained eye, yet the computer knows what each sequence means.

Each of these sequences needs to be placed in a container, so that the computer knows where to start reading and when to stop.

It also needs to tell one sequence apart from the other. This is where files come in.

What is a file?

A file is basically a container of binary digits that is formatted in a special way so that the computer knows that it is a single entity.

All the files that are contained in the computer are binary sequences. Each sequence is created in a unique way so that it serves a unique purpose. It is the job of the operating system to manage these files.

Files are stored in the computer's storage, either on the internal drive or on removable drives, such as flash drives.

Files are accessed using what is known as a file path. This file path is a string of characters that locates the files in a structure called a file directory.

All the files on the hard drive and any other storage for that matter are stored in a file directory. Think of those telephone directories that were used to use in the old days.

History of files

Just out of interest, let's take a brief look at the history of files.

The word 'file' first appeared in the context of computing around the 1940s. It was used extensively in reference to punched cards. If you remember from module 1, postcards were used as a method of saving programs for a computer to retrieve at runtime.

It would have been fun if we were still doing that to this day. Instead of sitting at a computer and typing code, we would have been sitting at some weird machine that punches holes into cards so that the computer can read those as instructions!

During this era, the term files referred more to the hardware in the computer than arbitrary data contents in storage.

When computers evolved and started using storage on discs and other similar mediums, the term 'file' also evolved to refer to arbitrary binary contents of these discs.

How do files fit into operating systems?

Operating systems are a point of interest when referring to computer files.

The operating system itself is simply just a collection of files that do various things.

These include executables such as your drivers and applications and the kernel, among others.

There are also non-executable files such as system configuration files, your budget spreadsheet, pictures of your cats and the party from last summer.

All these files are arranged in what is known as a file system, which controls how all the data is stored on and retrieved from the storage medium. W

Without the file system, it would be very confusing for the computer to know what's what. The file system itself contains guidelines for the structure and logic of the files that will be stored in that medium, along with any permissions that are enforceable on the files.

At some point you may have encountered files that cannot be opened for one reason or the other. This is typically because of restrictions applied on the file that require you to have certain permissions in order to access the file. The exact reason why we need to do this is a topic for another day. In fact, we can even have an entire course on that!

Directories for storage of configurations

File systems typically have directories that allow the computer to trace the root of the drive right up to the point where it gets to the file.

Directory site typically structured in the form of a tree. The drive that contains the file is at the very top of the hierarchy.

Each folder falls under this hierarchy and is also subdivided into folders and files. Each file is located on the drive using a path.

The file path is a traversal off the directory tree which leads to the exact node in which the file is located. We're going to be using this extensively in our code very soon.

Key elements of files

In short, files provide us with four key elements that makes software as useful as we know it.

Reusability: any data that is generated as the software runs can be saved and used at a later stage.

Large storage capacity: on a typical computer, RAM is smaller than storage devices. Writing to files allows you to write much larger amounts of data to storage without straining the computer.

Time-saving: if your program requires a lot of input from the user, you can get it from files instead of letting the user enter everything. This is how preconfigured software packages are typically delivered.

Portability: you can easily transfer the contents of a file from one computer system to another without having to worry about data loss.

Data portability

If you look at all the programs that we have written so far, we could do quite a lot of interesting things with functions variables and constants, but there is one glaring problem. The data that we work on disappears like mist as soon as we close the program. This limits the usefulness of the program because we cannot share the data with other users, and it's only limited to the PC on which the information is processed. In the real world, computer users deal with large volumes of data, and it's hardly practical for you to be entering so much data every time you want to use a program. Not only does it waste time by reducing the efficiency of the computer through the introduction of the human element, but it also introduces errors, which are what we humans do best. Try typing a sequence of 10,000 numbers that are not arranged in any particular order then repeating the sequence five times, and you will see that errors already start creeping in when you are at the second round.

Using files in C allows us to make our data portable.

C allows us to…

To overcome this, C allows us to read information from files and also write information to files.

This means that you can have your configuration settings stored to a file, which is then retrieved by the program as soon as it starts running.

It also means that the data that you work with in a program can be retrieved from a file, processed, and stored to the same file. Pretty nifty!

Configuration file example: display driver

If you look at nearly all the software that you have on your PC, and even on your phone, you will see that besides the executable file, they are a number of other files that are bundled with the executable. Quite often, you can open some of these files using a regular text editor and see what they contain. Typically, programs have their configuration settings stored as text files. An extremely common example of this is your display driver. If you dig up a bit in your system files, you will see a folder that contains your display driver. Information such as the resolution of your monitor, number of colour bits, refresh rate and many other properties of your display hardware are contained in a text file.

Such a file is what is known as a configuration file. Now imagine if programs didn't have the capability to access files. Each time you switched on your computer; you would have to punch in all those values so that the computer can recognise your display! That would really be silly and take up so much time that computers would be so hard to use!

File system example: Windows

Each type of operating system has its own file system. This means that whenever you are writing a program that accesses files from a storage device, you would have to follow the strict guidelines that are peculiar to that specific operating system. An example of this is the windows file system. Each storage device is represented by a letter, known as a drive letter, followed by a colon and a backslash. If you want to access the system drive on a computer that has default configurations, then you would likely be looking in The C drive which will look like this C:\. Any subsequent directories will be named after the backslash, and each subsequent backslash means going one directory lower. This string of characters ends with the

filename, a dot, and the file extension. You remember file extensions from header files that we have looked at previously, which carry the extension dot h. You also remember that each time you save a source file in your IDE, you save it with the extension dot c. Whenever programs on your computer request access to this file, such as when your IDE requests to open the file when you want to edit your code, it follows the path from the root of the drive, going through each sub directory up until it gets to the file.

File system example: Linux

On other systems such as Linux it is kind of a different story. Instead of drive letters and backslashes, Linux drives typically don't have a drive letter at all! Naturally, the next question would be how are the files accessed? How does the computer tell storage mediums apart?

In Linux, each storage device is listed under /dev, which you may have guessed, is short for device. The Linux filesystem unifies all physical hard drives and partitions into a single directory structure. It all starts at the top–the root (/) directory. All other directories and their subdirectories are located under the single Linux root directory. This means that there is only one single directory tree in which to search for files and programs. Notice that the slashes have changed direction this time. As you may have guessed already, the Linux directory structure users forward slashes instead of the backslashes used in windows. To access any folder in the Linux directory, you start with a forward slash followed by the name of the folder. This starts all the way at the top with the root directory as we have just seen and ends with the filename followed by a dot and the file extension.

We will not get into much detail beyond this as that is a whole topic for an entire course! We have just enough information to do what we need to do within C.

Types of files

Whenever you save a file in whatever software package you will be using, the file is in encoded in a specific format that is usually reflected by the file extension.

Files supported in C

We're going to be looking at two types of files that are supported in C, namely binary files, and text files.

Let's look at each of them in detail and see what they're all about

Text files

Text files are the normal files that can be opened by pretty much any system that has a text editor. Before you even install anything on Windows, fresh out of the box, you can already open text files using notepad. The same applies to Linux, which comes with various text editors such as vim. Text files typically have the file extension dot txt.

Opening text files

When you open these files, you will see that the contents are in plain text, just the way you typed. If you remember from module 1, various encoding standards are used to encode these files into a machine-readable state. Most text files use the ANSI and Unicode encoding. You may have seen this when you try to save a source file in your IDE, where it asks you which encoding format you want to use. Text files have the advantage of being very easy to open, edit and delete and have widespread compatibility. It is pretty much impossible to have a text file that opens on one system and refuses to open on another. Like mention before text files rely on standardised encoding which is present on pretty much every system under the sun. Text files require minimum effort to maintain because of their simplicity, but this comes at a price.

The problem with text files…

Text files offer the least amount of security. Think of it this way: if you have a board of jigsaw pieces and you paint directions to where you dug up a hole and stashed your gold bars, anyone who comes along can immediately read the directions in know exactly what it is all about. If you paint the board then proceed to scramble the pieces, only you know how to put them back together the way they were and so not everyone who comes along will be able to see what the jigsaw board is all about. This is exactly the same thing that happens with text files. Since nearly every system under the sun can open text files, there is very little in the way of security that you can get from them. Also, text files don't really offer any kind of compression, and

so require more space to store.

Binary files

Another type of file that is supported in C is a binary file. The only difference between these and text files is that binary files cannot be read by text editors. Binary files are very similar in structure to arrays of structures, except that this type of structure is further away from memory and is tucked away in disks. Because of this, you can create very large collections of data within them literally limited by the space available on your storage disk! Another remarkable observation that you will make on binary files is that you can jump instantly to any structure within the file which provides random access, just like you can do with arrays in memory. This also means that you can change the contents of the structure anywhere in the file at anytime.

Opening binary files

Binary files usually have faster read and write times then text files because binary images are stored directly in memory during execution. This is in contrast to a text file in which everything has to be converted back and forth between text and binary in this consumes a bit of time on the part of the CPU. C supports the file-of-structures concept very cleanly. Once you open the file you can read a structure, write a structure, or seek to any structure in the file. C supports the file-of-structures concept very cleanly. Once you open the file you can read a structure, write a structure, or seek to any structure in the file.

From this point on, the program uses pointers. The specific type of pointer that is used here is a file pointer. Then the file is opened, the pointer points to record 0. This is the first record in the file. Any read operation reads the currently pointed-to structure and moves the pointer down one structure. Any write operation writes to the currently pointed-to structure and moves the pointer down one structure. In this whole operation, seek moves the pointer to the required record. It is worth noting that C regards everything in the disk file as blocks of bytes read from disk into memory or read from memory onto disk. C uses a file pointer, but it can point to any byte location in the file. This means that you have to keep track of things.

What can you do with files in C?

There are five operations that you can perform on files in C. These are:

Create a file

Open a file

Read a file

Write to a file and

Closing a file.

We will look at each of these closely in a bit.

Header files and functions in C

There are quite a number of functions that are used for file input and output. These functions are contained in the standard library header file as you might expect. As mentioned earlier, C treats files as data structures and that means you need to declare a pointer of type file. This will be used for communication between your program and the file.

The basic syntax for this pointer will look like this:

```
FILE *fptr.
```

File operations in C

C provides various built-in functions associated with file handling. They are:

Creating a new file: fopen ()(pronounced f open, same for subsequent words)

Opening an existing file in your system: fopen ()

Closing a file: fclose ()

Reading characters from a line: getc()

Writing characters in a file: putc()

Reading a set of data from a file: fscanf()

Writing a set of data in a file: fprintf()

Reading an integral value from a file: getw()

Writing an integral value in a file: putw()

Setting a desired position in the file: fseek()

Getting the current position in the file: ftell()

Setting the position at the beginning point: rewind()

A closer look at C's built-in functions…

To create a new file the fopen function is used. The basic syntax for the fopen function looks like this:

```
File = fopen("file_name", "mode")
```

On the parameter's filename is pretty obvious, is it refers to the name of the file that you want to work with. The more interesting one is 'mode'. It refers to the manner in which the file will be opened.

This table summarises what each mode does.

R	Opens a text file in reading mode
W	Opens or create a text file in writing mode
A	Opens a text file in append mode
R+	Opens a text file in both reading and writing mode
W+	Opens a text file in both reading and writing mode
A+	Opens a text file in both reading and writing mode
Rb	Opens a binary file in reading mode
Wb	Opens or create a binary file in writing mode
Ab	Opens a binary file in append mode
Rb+	Opens a binary file in both reading and writing mode
Wb+	Opens a binary file in both reading and writing mode
Ab+	Opens a binary file in both reading and writing mode

Each of these modes is important as it affects the behaviour of the file as we are working on it. You may be wondering how a file is created yet it is not specified in that table. It is actually specified ! It is just not as obvious. Like I mentioned before the F open function is used for creating files.

A closer look at C's built-in functions

If the function does not find the specified file in the specified location, it will create a new file. If the file is found, then the program or use the specified mode to open the file. Let's look at this code snippet:

```
#include <stdio.h>
Int main(){
  FILE * file;
  If (file = fopen("hello.txt", "r")){
    Printf("File opened in read mode");
  }
  Else
  Printf("The file wasn't found, and we cannot create a new file using r mode");
  Fclose(file);
  Return 0;
}
```

This code snippet will open the file if it is found in the specified directory. Fopen cannot create a new file in read mode, so it will print the error message that we specified. We will learn more about errors in our next lesson. In C, when you write to a file, newline characters '\n' must be explicitly added.

The stdio library contains all the functions you need to write to a file:

Fputc(char, file_pointer): It writes a character to the file pointed to by file_pointer.

Fputs(str, file_pointer): It writes a string to the file pointed to by file_pointer.

Fprintf(file_pointer, str, variable_lists): It prints a string to the file pointed to by file_pointer. The string can optionally include format specifiers and a list of variables variable_lists.

This code snippet shows an example of how you can write to a file. #include <stdio.h>

```
Int main() {
    Int i;
    FILE * fptr;
    Char fn[80];
    Char str[] = "Hello world\n";
    Fptr = fopen("our_first_file.txt", "w"); // "w" defines "writing mode"
    For (i = 0; str[i] != '\n'; i++) {
      /* write to file using fputc() function */
      Fputc(str[i], fptr);
    }
    Fclose(fptr);
    Return 0;
  }
```

Our string is written character by character using the for loop, putting each character in our file until the "\n" character is encountered, then the file is closed using the fclose function.

We can also use the fputs function so we can get rid of the for loop.

Let's take a look at this code snippet:

```
#include <stdio.h>
Int main() {
    FILE * fp;
    Fp = fopen("testfile.txt", "w+");
    Fputs("This is an easier way to save data ,", fp);
    Fputs("We just got rid of the for loop!\n", fp);
    Fputs("way easier!\n", fp);
    Fclose(fp);
    Return (0);

  }
```

We have created and opened a file called fputs_test.txt in a write mode.

Then we did a write operation using fputs() function by writing three different strings

Then the file is closed using the fclose function.

A closer look at C's built-in functions

Our next stop is reading data from a file. When you have written data to a file as in the previous example, you need a way of retrieving it from the file at a later stage.

There are three different functions dedicated to reading data from a file:

Fgetc(file_pointer): It returns the next character from the file pointed to by the file pointer. When the end of the file has been

reached, the EOF (pronounced end of file) signal is sent back.

Fgets(buffer, n, file_pointer): It reads n-1 characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.

Fscanf(file_pointer, conversion_specifiers, variable_adresses): It is used to parse and analyse data. It reads characters from the file and assigns the input to a list of variable pointers variable_adresses using conversion specifiers. Keep in mind that as with scanf, fscanf stops reading a string when space or newline is encountered.

Another function that is used for input is fgets(). This function is used to read a string from a given file and copies the string to a memory location which is referenced by an array. Syntax: fgets(sptr,max,fptr); Where sptr is a string pointer, which points to an array. Max is the length of the array. Fptr is a file pointer, which points to a given file.

This function reads n-1 characters and places them into array which is pointed by sptr. It reads characters until either a newline or an end of the file or size of the array is encountered. It appends a null character (‘\0’) at the end of the string. It returns a null pointer if either an end of file or an error encountered.

A closer look at C’s built-in functions

A file must be closed when we have finished with it. This is done to save memory and to make sure that no data is accidentally written to the file. Some systems limit the number of files that can be open at once, so this also helps free up slots for other files. A file is closed using the fclose function, and its basic syntax looks like this:

```
Fclose(file_pointer)
```

Command line arguments

Our last stop for today is command line arguments. Some C programs can behave in many different ways, based on the user’s request. This gives the program the provision to add parameters or arguments inside the main function to reduce the length of the code. These arguments are called command line arguments.

Command line arguments are arguments that are specified after the name of the program in the system’s command line, and these argument values are passed on to your program during program execution.

Components of a command line argument

There are two components of Command Line Argument in C:

Argc: It refers to “argument count”. It is the first parameter that we use to store the number of command line arguments. It is important to note that the value of argc should be greater than or equal to 0.

Agrv: It refers to “argument vector”. It is basically an array of character pointers which we use to list all the command line arguments.

In order to implement command line arguments, generally, two parameters are passed into the main function: these are the number of command line arguments and the list of command line arguments.

The basic syntax for command line arguments looks like this:

```
Int main( int argc, char *argv[] )
{
// BODY OF THE MAIN FUNCTION
}
Or it can also be written as

Int main( int argc, char **argv[] )
{
```

```
// BODY OF THE MAIN FUNCTION
}
```

In C, when you run a program via the Linux command line, you can look at these command line values and alter the behaviour of your program as well.

The first step to tell a C program to "get" the command line values is to change the signature of the main function, like this:

```
Int
Main( int number_of_args, char* list_of_args[] )
{
  ...
}

Int main( int argc, char** argv )
{
  ...
}
```

When the program is invoked with ‘values’ after the name of the program, they are stored in the list_of_args. We normally use a loop to ‘search ‘through this list of strings to find (and set) the state of the program.

Processing Command Line Args

Usually, you will create a function in C which reads through all the command line args and returns the state.

For simple cases, where the command line args are well defined, this can be done in the first few lines of code of the main function.

And on that note, we have come to the end of our 4th lesson. In this lesson, we learned about managing files in our programs, and why this is important. We have also learned about nifty way of controlling our programs after we have compiled them through the use of command line parameters. Our journey continues - we still have quite a lot more to cover before we wrap up our course. That's all for today - see you next time.

References

Dataflair. (2019). Command Line Arguments in C - Don’t be Confused, be Practical! - dataflair. [online] Available at: https://data-flair.training/blogs/command-line-arguments-in-c/

Opensource.com. (2016). An introduction to Linux filesystems. [online] Available at: https://opensource.com/life/16/10/introduction-linux-filesystems

PREP INSTA. (2021). Files Directories in Operating System | prepinsta. [online] Available at: https://prepinsta.com/operating-systems/directory-structure/#tree-***

Programiz.com. (2021). C Files I/O: Opening, Reading, Writing and Closing a file. [online] Available at: https://www.programiz.com/c-programming/c-file-input-output

Tutorialtous.com. (2018). C File I/O,getw(),putw() etc. [online] Available at: http://tutorialtous.com/c/iofiles.php

Utah.edu. (2021). The C Programming Language - Command Line Arguments. [online] Available at: https://www.cs.utah.edu/~germain/PPS/Topics/C_Language/command_line_args.html