Problem Solving with C

Compiled by

M S Anand (anandms@pes.edu)

Text Book(s):

1. “How To Solve It By Computer”, R G Dromey, Pearson, 2011.
2. “The C Programming Language”, Brian Kernighan, Dennis Ritchie, 2nd Edition, Prentice Hall PTR, 1988.

Reference Book(s):

1. “Expert C Programming; Deep C secrets”, Peter van der Linden
2. “ The C puzzle Book”, Alan R Feuer

What is a file?

A **computer file** is a computer resource for recording data discretely in a computer storage device.

If your computer could only ever process data stored within the main memory of the machine, the scope and variety of applications you could deal with would be severely limited. Virtually all serious business applications require more data than would fit into main memory and depend on the ability to process data that's persistent and stored on an external device such as a disk drive. Now, we will explore how we can process data stored in files.

C provides a range of functions in the header file stdio.h for writing to and reading from external devices. The external device you would use for storing and retrieving data is typically a disk drive, but not exclusively. Because, consistent with the philosophy of C, the library facilities you'll use for working with files are device independent, they apply to virtually any external storage device.

The Concept of a File

With all the examples up to now, any data that the user enters are lost once the program ends. If the user wants to run the program with the same data, he or she must enter it again each time. There are a lot of occasions when this is not only inconvenient, but it also makes the programming task impossible.

If you want to maintain a directory of names, addresses, and telephone numbers, for instance, a program in which you have to enter all the names, addresses, and telephone numbers each time you run it is worse than useless! The answer is to store data on permanent storage that continues to be maintained after your computer is switched off. This storage is called a *file, and a file is usually stored on a disk.*

A file is essentially a serial sequence of bytes stored on a medium.

Positions in a File

A file has a beginning and an end, and it has a *current position, typically defined as so many bytes from the beginning*. The current position is where any file action (a read from the file or a write to the file) will take place. You can move the current position to any point in the file. A new current position can be specified as an offset from the beginning of the file or, in some circumstances, as a positive or negative offset from the previous current position. You can also move the position to the end of the file in some situations.

File Streams

The C library provides functions for reading and writing to or from data streams. A stream is an abstract representation of any external source or destination for data, so the keyboard, the command line on your display, and files on a disk are all examples of things you can work with as streams. You use the same input/output functions for reading and writing any external device that is mapped to a stream.

There are two ways of writing data to a stream that represents a file. First, you can write a file as a *text file, in* which case data is written as a sequence of characters organized as lines, where each line is terminated by a newline character. Obviously, binary data such as values of type int or type double have to be converted to characters to allow them to be written to a text file.

Second, you can write a file as a *binary file. Data written to a binary file are written as a series* of bytes exactly as they appear in memory, so a value of type double, for example, would be written as the 8 bytes that appear in memory.

Of course, you can write any data you like to a file, but once a file has been written, it just consists of a series of bytes. Regardless of whether you write a file as a binary file or as a text file, it ultimately ends up as just a series of bytes. This means that the program must know what sort of data the file represents to read it correctly. What a series of bytes represents is dependent upon how you interpret it. A sequence of 12 bytes in a binary file could be 12 characters, 12 8-bit signed integers, 12 8-bit unsigned integers, 6 16-bit signed integers, a 32-bit integer followed by an 8-byte floating-point value, and so on. All of these will be more or less valid interpretations of the data, so it's important that a program that is reading a file has the correct assumptions about what was written.

Accessing Files

The files that are resident on your disk drive each have a name, and the rules for naming files are determined by your operating system. It would not be particularly convenient if a program that processes a file would only work with a specific file with a particular name. If it did, you would need to produce a different program for each file you might want to process. For this reason your program references a file through a *file pointer or more accurately a stream pointer. You associate a stream pointer with a* particular file programmatically when the program is run. A program can associate a given stream pointer with different files on different occasions, so the same program can work with a different file each time it executes. A file pointer points to a struct of type FILE that represents a stream.

The FILE structure to which a file pointer points contains information about the file. This will be such things as whether you want to read or write or update the file, the address of the buffer in memory to be used for data, and a pointer to the current position in the file for the next operation. You don’t need to worry about the contents of this structure in practice. It’s all taken care of by the input/output functions. However, if you really want to know all the gory details of the FILE structure, you will find them in the code for the stdio.h library header file.

Opening a File

You associate a specific external file name with an internal file pointer variable through a process referred to as *opening a file. One way to open a file is by calling the fopen() function that returns the file pointer for a specific* external file. The fopen() function is defined in stdio.h, and it has this prototype:

FILE *fopen(const char *name, const char *mode);

The first argument to the function is a pointer to a string that is the name of the external file you want to process. You can specify the name explicitly as an argument, or you can use an array or a variable of type pointer to char that contains the address of the character string that defines the file name. You would typically obtain the file name through some external means, such as from the command line when the program is started, or you could arrange to read it in from the keyboard. Of course, you can also define a file name as a constant at the beginning of a program when the program always works with the same file.

The second argument to the fopen() function is a character string that represents the file mode. The file mode specifies what you want to do with the file.

Mode	Description
“w”	Open a text file for *write operations. If the file exists, its current contents are discarded.*
“a”	Open a text file for *append operations. All writes are to the end of the file.*
“r”	Open a text file for *read operations.*

Note

Notice that a file mode specification is a character string between double quotes, not a single character between single quotes.

These three modes only apply to text files, which are files that are written as characters.

Assuming the call to fopen() is successful, the function returns a pointer of type FILE* that you can use to reference the file in further input/output operations using other functions in the library. If the file cannot be opened for some reason, fopen() returns NULL.

The pointer returned by fopen() is referred to as either a *file pointer or a stream pointer.*

A call to fopen() does two things:
it creates a file pointer—an address—that identifies the specific file on a disk from the name argument you supply, and it determines what you can do with that file.

When you want to have several files open at once, they must each have their own file pointer variable, and you open each of them with a separate call to fopen().

How many files can you keep open at a given time?
There's a limit to the number of files you can have open at one time, which will be determined by the value of the symbol FOPEN_MAX that's defined in stdio.h. The C standard requires that the value of FOPEN_MAX be at least eight, including stdin, stdout, and stderr, so as a minimum you will be able to be working with up to five files simultaneously but typically it's many more, often 256, for example.

Closing a file

When you've finished with a file, you need to tell the operating system that this is the case and free up the file so it can be used by others. This is referred to as *closing a file. You do this by calling the fclose() function that accepts a* file pointer as an argument and returns a value of type int, which will be EOF if an error occurs and 0 otherwise.

The typical usage of the fclose() function is as follows:

```
fclose(pfile); // Close the file associated with pfile
pfile = NULL;
```

The result of calling fclose() is that the connection between the pointer, pfile, and the physical file is broken, so pfile can no longer be used to access the file it represented.

If the file was being written, the current contents of the output buffer are written to the file to ensure that data are not lost. It's good practice to always set the file pointer to NULL when you have closed a file.

Note

EOF is a special character called the *end-of-file character. In fact, the symbol EOF is defined in stdio.h as a* negative integer that is usually equivalent to the value –1. However, it isn't necessarily always this value, so you should use EOF rather than an explicit value. EOF indicates that no more data are available from a stream.

It's good programming practice to close a file as soon as you've finished with it. This protects against output data loss, which could occur if an error in another part of your program caused the execution to be stopped in an abnormal fashion. This could result in the contents of the output buffer being lost, as the file wouldn't be closed properly. You must also close a file before attempting to rename it or remove it.

Note

Another reason for closing files as soon as you've finished with them is that the operating system will usually limit the number of files you may have open at one time. Closing files as soon as you've finished with them minimizes the chances of you falling foul of the operating system in this respect.

Calling the fflush() function will force any unwritten data left in an output buffer to be written to a file. With your file pointer pfile, you could force any data left in the output buffer to be written to the file by using this statement:

fflush(pfile);

The fflush() function returns a value of type int, which is normally 0 but will be EOF if an error occurs.

Deleting a File

The remove() function that's declared in stdio.h deletes a file

remove("myfile.txt");

This will delete the file that has the name myfile.txt from the current directory. Note that the file cannot be open when you try to delete it. If the file is open, the effect of calling remove() is implementation defined. You always need to double check any operations on files, but you need to take particular care with operations that delete files. You could wreck your system if you don't.

Writing to a Text File

The simplest write operation is provided by the function fputc(), which writes a single character to a text file. It has the following prototype:

int fputc (int ch, FILE *pfile);

The function writes the character specified by the first argument to the file identified by the second argument, which is a file pointer. If the write is successful, it returns the character that was written, otherwise it returns EOF.

In practice, characters aren't usually written to a physical file one by one. This would be extremely inefficient. **Hidden from your program and managed by the output routine, output characters are written to a buffer until a reasonable number have been accumulated; they are then all written to the file in one go**.

Reading from a Text File

The fgetc() takes a file pointer as its only argument and returns the character read as type int. The typical use of fgetc() is illustrated by the following statement:

int mchar = fgetc(pfile); // **Reads a character into mchar**

The mchar is type int because EOF will be returned if the end of the file has been reached. EOF is a negative integer that cannot be returned or stored as type char when char is an unsigned type.

Behind the scenes, the actual mechanism for reading a file is the inverse of writing to a file. **A whole block of characters is read into a buffer in one go. The characters are then handed over to your program one at a time as you request them, until the buffer is empty, whereupon another block is read. This makes the process very fast, because most fgetc() operations won't involve reading the file but simply moving a character from the buffer in main memory to the place where you want to store it**.

A few sample programs

Program to copy one file to another – sample1

Program to find the size of a given file – sample2

Program to mimic the “more” command in Linux – sample3

Selection sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.
In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Program to perform selection sort – sample4

Example

arr[] = 64 25 12 22 11
// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64
// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 **12** 25 22 64
// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 **22** 25 64
// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 **25** 64

A sample implementation is here.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) -> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

The corresponding C program is here.

Recursive selection sort for singly linked list | Swapping node links

Given a singly linked list containing **n** nodes. The problem is to sort the list using recursive selection sort technique. The approach should be such that it involves swapping node links instead of swapping nodes data.

Searching algorithms

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found. It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

The program is here.

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

Program for Iterative Binary search is here.

Program for Iterative recursive search is here.

Reading and Writing Strings to a Text File

You can use the fgets() function to read from any stream. It has the following prototype:

```
char *fgets(char * str, int nchars, FILE * pfile);
```

The function reads a string into the memory area pointed to by str, from the file specified by pfile. Characters are read until either an '\n' is read or nchars-1 characters have been read from the stream, whichever occurs first.

If a newline character is read, it's retained in the string. A '\0' character will be appended to the end of the string in any event. If there is no error, fgets() returns the pointer, str; otherwise, NULL is returned. Reading EOF causes NULL to be returned.

For writing a string to a stream, you have the complementary fputs() function that has the prototype:

int fputs(const char *str, FILE *pfile);

The first argument is a pointer to the character string that's to be written to the file, and the second argument is the file pointer. The operation of the function is slightly odd in that it continues to write characters from a **string until it reaches a '\0' character, which it doesn't write to the file**. This can complicate reading back variable-length strings from a file that have been written by fputs().

It works this way because it's a character-write operation, not a binarywrite operation, so it's expecting to write a line of text that has a newline character at the end. A newline character isn't required by the operation of the function, but it's very helpful when you want to read the file back using fgets().

The fputs() function returns EOF if an error occurs and a positive integer under normal circumstances

Writing formatted data to a file

int fprintf(FILE *_stream_, const char *_format_, ...);

Return value

Upon successful return, this function return the number of characters printed (excluding the null byte used to end output to strings).

Reading formatted data from a file

int fscanf(FILE *_stream_, const char *_format_, ...);

Return Value

Returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure. The value **EOF** is returned if the end of input is reached before either the first successful conversion or a matching failure occurs. **EOF** is also returned if a read error occurs.

Managing file position indicators

int fseek(FILE *_stream_, long _offset_, int _whence_);

The **fseek**() function sets the file position indicator for the stream pointed to by *stream*. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.
On success, the return value is 0, otherwise, -1.

long ftell(FILE *_stream_);
The **ftell**() function obtains the current value of the file position indicator for the stream pointed to by *stream*.
Returns offset on success, -1 on failure.

Other file related functions

void rewind(FILE *_stream_);

The **rewind**() function sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to: (void) fseek(stream, 0L, SEEK_SET) except that the error indicator for the stream is also cleared

size_t fread(void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);
size_t fwrite(const void *_ptr_, size_t _size_, size_t _nmemb_, FILE *_stream_);

The function **fread**() reads *nmemb* items of data, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite**() writes *nmemb* items of data, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

Extract tokens from strings

#include <string.h>

char *strtok(char **str*, const char **delim*);

The **strtok**() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok**(), the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

Each call to **strtok**() returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, **strtok**() returns NULL.

A sequence of calls to **strtok**() that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. The first call to **strtok**() sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in *str*. If such a byte is found, it is taken as the start of the next token. If no such byte is found, then there are no more tokens, and **strtok**() returns NULL. (A string that is empty or that contains only delimiters will thus cause **strtok**() to return NULL on the first call.)

The end of each token is found by scanning forward until either the next delimiter byte is found or until the terminating null byte ('\0') is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and **strtok**() saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, **strtok**() returns a pointer to the start of the found token.

A program to tokenize the password file is here.

Unions

User defined data type just like a structure.

```
union sample
{
    char name [4];
    long  length;
    short s1;
};

union sample u1, u2;
```

Differences between structures and unions

A simple program is here.

A program to print the individual bytes of a long variable is here.

You can have a pointer to a union.

You can access members of the union by pointer->member

Bit fields

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

A sample program without using bit fields for date is here.

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

Program using bit fields is here.

Some interesting facts about bit fields

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the program.

2) We cannot have pointers to bit field members as they may not start at a byte boundary. Sample program.

3) Array of bit fields is not allowed. For example, the below program fails in compilation. The program.

Bit fields in unions

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.

A sample program without using bit fields for date is here.

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

Program using bit fields is here.