

File Handling

Objectives:

So far we have entered information into our programs via the computer's keyboard. This is somewhat laborious if we have a lot of data to process. The solution is to combine all the input data into a file and let our C program read the information when it is required.

Having read this section you should be able to:

- 1.open a file for reading or writing
- 2.read/write the contents of a file
- 3.close the file

The Stream File:

Although C does not have any built-in method of performing file I/O, the C standard library contains a very rich set of I/O functions providing an efficient, powerful and flexible approach. We will cover the ANSI file system but it must be mentioned that a second file system based upon the original UNIX system is also used but not covered on this course.

A very important concept in C is the stream. In C, the stream is a common, logical interface to the various devices that comprise the computer. In its most common form, a stream is a logical interface to a file. As C defines the term "file", it can refer to a disk file, the screen, the keyboard, a port, a file on tape, and so on. Although files differ in form and capabilities, all streams are the same. The stream provides a consistent interface and to the programmer one hardware device will look much like another.

A stream is linked to a file using an open operation. A stream is disassociated from a file using a close operation. The current location, also referred to as the current position, is the location in a file where the next file access will occur. There are two types of streams: text (used with ASCII characters some character translation takes place, may not be one-to-one correspondence between stream and what's in the file) and binary (used with any type of data, no character translation, one-to-one between stream and file).

To open a file and associate it with a stream, use `fopen()`. Its prototype is shown here:

```
FILE *fopen(char *fname,char *mode);
```

The `fopen()` function, like all the file-system functions, uses the header `stdio.h`. The name of the file to open is pointed to by `fname` (must be a valid name). The string pointed at for `mode` determines how the file may be accessed as shown:

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Open a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append a binary file for read/write

If the open operation is successful, `fopen()` returns a valid file pointer. The type `FILE` is defined in `stdio.h`. It is a structure that holds various kinds of information about the file, such as `size_t`. The file pointer will be used with all other functions that operate on the file and it must never be altered or the object it points to. If `fopen()` fails it returns a `NULL` pointer so this must always be checked for when opening a file. For example:

```
FILE *fp;
```

```
if ((fp = fopen("myfile", "r")) == NULL){
    printf("Error opening file\n");
    exit(1);
}
```

To close a file, use `fclose()`, whose prototype is

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

The `fclose()` function closes the file associated with `fp`, which must be a valid file pointer previously obtained using `fopen()`, and disassociates the stream from the file. The `fclose()` function returns 0 if successful and EOF (end of file) if an error occurs.

Once a file has been opened, depending upon its mode, you may read and/or write bytes to or from it using these two functions.

```
int fgetc(FILE *fp);
int fputc(int ch, FILE *fp);
```

The `fgetc()` function reads the next byte from the file and returns it as an integer and if error occurs returns EOF. The `fgetc()` function also returns EOF when the end of file is reached. Your routine can assign `fgetc()`'s return value to a char you don't have to assign it to an integer.

The `fputc()` function writes the bytes contained in `ch` to the file associated with `fp` as an unsigned char. Although `ch` is defined as an int, you may call it using simply a char. The `fputc()` function returns the character written if successful or EOF if an error occurs.

Text File Functions:

When working with text files, C provides four functions which make file operations easier. The first two are called `fputs()` and `fgets()`, which write or read a string from a file, respectively. Their prototypes are:

```
int fputs(char *str, FILE *fp);
char *fgets(char *str, int num, FILE *fp);
```

The `fputs()` function writes the string pointed to by `str` to the file associated with `fp`. It returns EOF if an error occurs and a non-negative value if successful. The null that terminates `str` is not written and it does not automatically append a carriage return/linefeed sequence.

The `fgetc()` function reads characters from the file associated with `fp` into a string pointed to by `str` until `num-1` characters have been read, a newline character is encountered, or the end of the file is reached. The string is null-terminated and the newline character is retained. The function returns `str` if successful and a null pointer if an error occurs.

The other two file handling functions to be covered are `fprintf()` and `fscanf()`. These functions operate exactly like `printf()` and `scanf()` except that they work with files. Their prototypes are:

```
int fprintf(FILE *fp, char *control-string, ...);
int fscanf(FILE *fp, char *control-string ...);
```

Instead of directing their I/O operations to the console, these functions operate on the file specified by `fp`. Otherwise their operations are the same as their console-based relatives. The advantages to `fprintf()` and `fscanf()` is that they make it very easy to write a wide variety of data to a file using a text format.

Binary File Functions:

The C file system includes two important functions: `fread()` and `fwrite()`. These functions can read and write any type of data, using any kind of representation. Their prototypes are:

```
size_t fread(void *buffer, size_t size, size_t num, FILE *fp);
size_t fwrite(void *buffer, size_t size, size_t num, FILE *fp);
```

The `fread()` function reads from the file associated with `fp`, `num` number of objects, each object size bytes long, into `buffer` pointed to by `buffer`. It returns the number of objects actually read. If this value is 0, no objects have been read, and either end of file has been encountered or an error has occurred. You can use `feof()` or `ferror()` to find out which. Their prototypes are:

```
int feof(FILE *fp);
int ferror(FILE *fp);
```

The feof() function returns non-0 if the file associated with fp has reached the end of file, otherwise it returns 0. This function works for both binary files and text files. The ferror() function returns non-0 if the file associated with fp has experienced an error, otherwise it returns 0.

The fwrite() function is the opposite of fread(). It writes to file associated with fp, num number of objects, each object size bytes long, from the buffer pointed to by buffer. It returns the number of objects written. This value will be less than num only if an output error as occurred.

The void pointer is a pointer that can point to any type of data without the use of a TYPE cast (known as a generic pointer). The type size_t is a variable that is able to hold a value equal to the size of the largest object supported by the compiler. As a simple example, this program write an integer value to a file called MYFILE using its internal, binary representation.

```
#include <stdio.h> /* header file */
#include <stdlib.h>
void main(void)
{

    FILE *fp; /* file pointer */
    int i;

    /* open file for output */
    if ((fp = fopen("myfile", "w"))==NULL){
        printf("Cannot open file \n");
        exit(1);
    }
    i=100;

    if (fwrite(&i, 2, 1, fp) !=1){
        printf("Write error occurred");
        exit(1);
    }
    fclose(fp);

    /* open file for input */
    if ((fp =fopen("myfile", "r"))==NULL){
        printf("Read error occurred");
        exit(1);
    }
    printf("i is %d",i);
    fclose(fp);
}
```

File System Functions:

You can erase a file using remove(). Its prototype is

```
int remove(char *file-name);
```

You can position a file's current location to the start of the file using rewind(). Its prototype is

```
void rewind(FILE *fp);
```

Hopefully I have given you enough information to at least get you started with files. Its really rather easy once you get started.

Command Line Parameters:

Many programs allow command-line arguments to be specified when they are run. A command-line argument is the information that follows the program's name on the command line of the operating system. Command-line arguments are used to pass information to the program. For example, when you use a text editor, you probably specify the name of the file you want to edit after the name of the word processing program. For example, if you use a word processor called WP, then this line causes the file TEST to be edited.

WP TEST

Here, TEST is a command-line argument. Your C programs may also utilize command-line arguments. These are passed to a C program through two arguments to the main() function. The parameters are called argc and argv. These parameters are optional and are not used when no command-line arguments are being used.

The argc parameter holds the number of arguments on the command-line and is an integer. It will always be at least 1 because the name of the program qualifies as the first argument. The argv parameter is an array of string pointers. The most common method for declaring argv is shown here.

```
char *argv[];
```

The empty brackets indicate that it is an array of undetermined length. All command-line arguments are passed to main() as strings. To access an individual string, index argv. For example, argv[0] points to the program's name and argv[1] points to the first argument. This program displays all the command-line arguments that it is called with.

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

```
void main(int argc, char *argv[])
{
    int i;

    for (i=1; i<argc; i++) printf("%s",argv[i]);
}
```

The ANSI C standard does not specify what constitutes a command-line argument, because operating systems vary considerably on this point. However, the most common convention is as follows:

Each command-line argument must be separated by a space or a tab character. Commas, semicolons, and the like are not considered separators. For example:

This is a test

is made up of four strings, but

this,that,and,another

is one string. If you need to pass a command-line argument that does, in fact contain spaces, you must place it between quotes, as shown in this example:

```
"this is a test"
```

A further example of the use of argc and argv now follows:

```
void main(int argc, char *argv[])
{
    if (argc !=2) {
        printf("Specify a password");
        exit(1);
    }
    if (!strcmp(argv[1], "password"))
        printf("Access Permitted");
    else
    {
        printf("Access denied");
        exit(1);
    }
    program code here .....
}
```

This program only allows access to its code if the correct password is entered as a command-line argument. There are many uses for command-line arguments and they can be a powerful tool.

My final example program takes two command-line arguments. The first is the name of a file, the second is a character. The program searches the specified file, looking for the character. If

the file contains at least one of these characters, it reports this fact. This program uses argv to access the file name and the character for which to search.

```
/*Search specified file for specified character. */

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

void main(int argc, char *argv[])
{
    FILE *fp; /* file pointer */
    char ch;

    /* see if correct number of command line arguments */
    if(argc !=3) {
        printf("Usage: find <filename> <ch>\n");
        exit(1);
    }

    /* open file for input */
    if ((fp = fopen(argv[1], "r"))==NULL) {
        printf("Cannot open file \n");
        exit(1);
    }

    /* look for character */
    while ((ch = getc(fp)) !=EOF) /* where getc() is a */
        if (ch== *argv[2]) { /*function to get one char*/
            printf("%c found",ch); /* from the file */
            break;
        }
    fclose(fp);
}
```

The names of argv and argc are arbitrary - you can use any names you like. However, argc and argv have traditionally been used since C's origin. It is a good idea to use these names so that anyone reading your program can quickly identify them as command-line parameters.