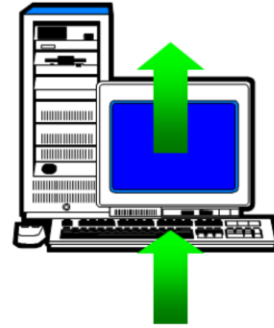


### Input and Output

- **Objective**
  - To design and implement programs which communicate both with the console and with files using standard functions
- **Contents**
  - Standard library support
  - Console I/O
  - File I/O
  - Character and Line I/O
  - Reading and Writing block structures
  - Passing arguments to programs
- **Summary**
- **Practical**
  - To write word count and file pager utilities
  - Persisting structures



---

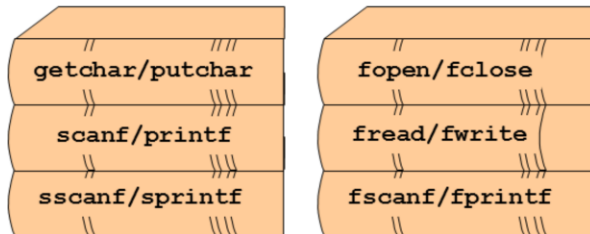
The objective of this chapter is to introduce standard input and output. The chapter covers console I/O, file I/O and input into programs from the command line.

## Input and Output Facilities

- Input and output facilities are not part of the C language
  - They form a 'standard I/O library'

```
#include <stdio.h>

int main(void)
{
    ...
    printf(...);
    ...
    ch = getchar();
    ...
}
```



All input and output, whether using console, files, communication ports, etc. is performed by library routines. This chapter covers some of the standard ones. The examples shown above are standard and are categorised as follows:

Routine	Function or Macro?	From -> To	Category Name
printf	Function	Program -> stdout	type based
scanf	Function	stdin -> Program	type based
putchar	Macro	Program -> stdout	character based
getchar	Macro	stdin -> Program	character based
fprintf/fwrite	Functions	Program-> File	type based
fscanf/fread	Functions	File -> Program	type based
sprintf	Function	Program -> String	type based
sscanf	Function	String -> Program	type based
fopen/fclose	Functions	N/A	file handlers
fseek	Function	N/A	buffer handling

stdin and stdout are two of the three standard I/O streams.

### The Standard I/O Library

- **The I/O routines of the standard C library allow you to read and write data to and from files and devices**
- **In C, there are no predefined file structures; all data is treated as sequences of bytes**
- **The following three types of I/O functions are available...**
  - Stream I/O
  - Low-level I/O
  - Console and port I/O
- **C standard only uses stream I/O**

---

Input and output processes are notoriously difficult to standardise. C has achieved dizzy heights in comparison to nearly all of its potential rivals, since it has standardised a high-level scheme for I/O. This is achieved by having a very simplistic view of files; files are treated as a sequence of bytes. Both disk files and some special device files adhere to this rule. The communication is achieved using the concept of streams.

Other forms are supported, including low-level I/O, console, and port I/O. The routines provided are not standard. They make heavy use of operating system calls, hardware interrupts, etc.

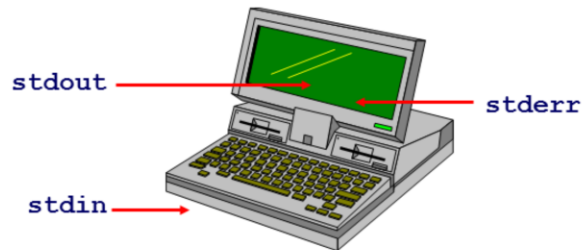
On Microsoft Windows, C allows two interpretations of its byte files: text and binary – this is not the case for other operating systems.

A Microsoft Windows text file has any disk CR/LF ("`\r\n`") combinations translated to LF in memory (and vice versa). The end-of-file character (ctrl-Z) is translated to EOF in memory (and vice versa).

In binary mode (which is Microsoft specific), no translations are performed.

### Standard Streams

- When a C program runs, three 'streams' are opened



- The standard I/O library provides routines that talk directly to the standard I/O files for...
  - Single-character I/O
  - Formatted I/O
  - String I/O

---

The special device files that are supported by the standard are the 'console' streams, `stdin`, `stdout` and `stderr`. These are opened automatically, usually inherited from the parent process, and are available throughout the execution of all C programs. There are many routines that specifically use them, notably:

```
printf, sprintf, vprintf, scanf and sscanf.  
getchar, gets, putchar and puts.  
perror and assert (use stderr).
```

By default, the streams are attached to the screen and keyboard. Redirection is possible using operating system specific techniques outside the scope of this chapter.

## getchar () and putchar ()

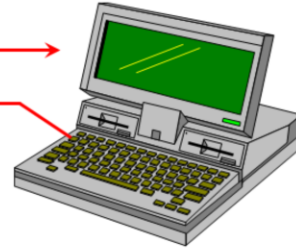
- Using help or otherwise, describe the functionality of the echo function

```
#include <stdio.h>
void echo(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
    {
        putchar(ch);
    }
}
```

```
ch = getchar();
```

```
putchar('X');
```



- Why use an int to represent a char?

Both `getchar` and `putchar` are macros are #defined in terms of `getc` and `putc` and use the struct `_iobuf`, which is mentioned later in the chapter. They are supported by buffers, defined for you through the struct `_iobuf`. The internal buffers use unsigned chars, and all integers are cast into unsigned chars within the macros.

Note that the return from `getchar` and the argument to `putchar` are ints. The standard defines the interface using integers.

EOF is #defined in `stdio.h`. It is usually `-1`.

### File I/O

- **Special I/O library routines exist that enable access to files other than `stdin`, `stdout`, `stderr`, etc.**
- **Several files may be in use at the same time, if required**

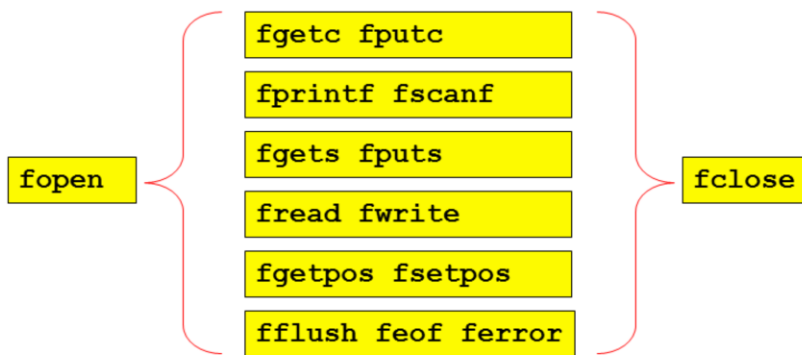


---

The routines mentioned on the previous page are special cases of general file I/O. Indeed, all functions that use the file I/O functions mentioned in this section can be used with the standard streams.

The number of files open at any one time is implementation dependent. It relies on the way in which the environment is set up.

## Buffered File Access



### Chronological overview

- There are also low-level I/O calls that do not buffer or format data

All files need to be opened. The high-level function `fopen` performs this task. The file processing is then performed using a variety of routines, and the file is closed with `fclose`. All these routines take a file handle as an argument. The full list of standard high-level file-handling functions is as follows:

#### `fopen` and `fclose`

<code>freopen</code>	As <code>fopen</code> , but with user-supplied file handle
<code>fflush</code>	Forces data to be written to the output stream and pulls in and dumps characters from the input stream
<code>fscanf</code> and <code>fprintf</code>	File versions of <code>scanf</code> and <code>printf</code>
<code>fgetc</code> and <code>fputc</code>	File versions of <code>getchar</code> and <code>putchar</code>
<code>getc</code> and <code>putc</code>	File versions of <code>getchar</code> and <code>putchar</code> (macro versions of above)
<code>fgets</code> and <code>fputs</code>	File versions of <code>gets</code> and <code>puts</code>
<code>ungetc</code>	Pushes back into buffer a character obtained by <code>getc</code>
<code>fread</code> and <code>fwrite</code>	I/O of arrays
<code>fseek</code> and <code>ftell</code>	Sets and returns internal file buffer (binary/text)
<code>rewind</code>	Initialises internal file buffer (binary/text)
<code>fgetpos</code> and <code>fputpos</code>	Saves and restores buffer pointers in special pointer type

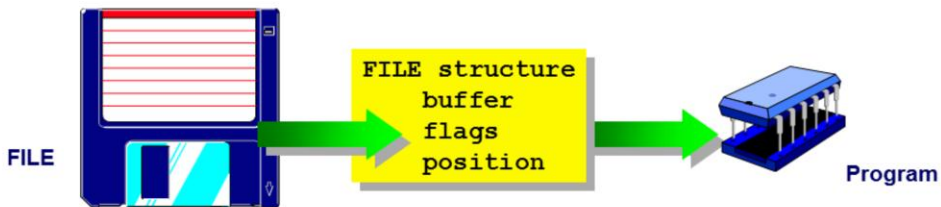
#### Also

<code>clearerr</code> , <code>feof</code> , <code>ferror</code>	Rudimentary error handling using <code>error.h</code>
<code>remove</code> and <code>rename</code>	cf. operating system's <code>del/rm</code> and <code>ren/mv</code>
<code>tmpfile</code> and <code>tmpname</code>	Creation of temporary file and filename
<code>setvbuf</code> and <code>setbuf</code>	Provides some programmer-controlled stream buffering



## Opening a File

- When a file is opened, a stream or channel is established between the file and the program



- A FILE structure is created to contain information about the file and I/O buffer
- `fopen()` returns a pointer to the FILE struct
- The FILE structure template is defined in `stdio.h`

The file handle is of type `struct _iobuf`. The template for this structure is defined in `stdio.h`. The standard has typedefd this struct to be `FILE`. `typedef` is covered in the *Further Data Types* chapter.

When a file is opened, an `_iobuf` struct is created automatically. The struct, whose internal members contain file status, buffers, etc., is initialised by `fopen` and maintained throughout using a pointer containing its address. The pointer is returned by `fopen`. The programmer assigns this value to a `FILE *`. It is this pointer that is used as the file handle.

The below fragment is an example of a possible implementation of the `_iobuf` structure

```

struct _iobuf
{
    char *_ptr;
    int _cnt;
    char *_base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char *_tmpfname;
};
  
```

`FILE` is defined thus:

```
typedef struct _iobuf FILE;
```

## fopen() - An Example

```
#include <stdio.h> /* Defines FILE */  
  
int main(void)  
{  
    FILE * file_ptr;  
    ...  
    file_ptr = fopen("myfile.ext", "r");  
    ...  
}
```

Filename      Access Mode

- The basic access modes are...
  - "r" - open file for reading
  - "w" - open file for writing
  - "a" - open file for appending at end of file
- Error handling is helped using library utilities such as `assert`, `errno` and `perror` (see notes)

The two arguments to `fopen` are two strings, i.e. two `char *`s. The first is a filename, which could either be in the current working directory or in another directory if a full or relative pathname is supplied. The second string contains up to three characters that represent the access mode. On Microsoft Windows, if the string contains the character 'b', then the files are treated as binary files; otherwise they are treated as text files ('t' for text may also be specified)...

`assert` is a macro (defined in `assert.h`) that writes to the `stderr` stream whenever the argument is false. It is usually used as pre and post conditions before and after an important process.

`errno` (`errno.h`) is a (global) `int` set by some library functions to be specific values, each with a simple specific error message.

`perror()` (`stdio.h`) is used to display `errno` using a more friendly message.

## Character I/O – fgetc () and fputc ()

```
#include <stdio.h>
int main(void)
{
    char in_name[25], out_name[25];
    FILE * in_file;

    puts("Enter name of file to be copied.");
    scanf("%24s", in_name);
    puts("Enter name of output file.");
    scanf("%24s", out_name);

    in_file = fopen(in_name, "r");
    if (!in_file)
        fprintf(stderr, "Can't open %s for reading.\n", in_name);
    else
    {
        FILE * out_file = fopen(out_name, "w");
        if (!out_file)
            fprintf(stderr, "Can't open %s for writing.\n", out_name);
        else
        {
            int ch;
            while ((ch = fgetc(in_file)) != EOF)
                fputc(ch, out_file);
            printf("File has been copied.\n");
            fclose(out_file);
        }
        fclose(in_file);
    }
    return 0;
}
```



The while loop is reminiscent of the loop in the echo function of Exercise 1, but using file I/O in place of the console I/O. The code below performs the same task (using macro versions)

```
int fcopy(char * in, char * out)
{
    FILE * in_file, * out_file;
    int ch;

    in_file = fopen(in, "r");
    if (in_file == NULL)
    {
        fprintf(stderr, "Can't open %s for reading .\n", in);
        return 1;
    }
    else
    {
        out_file = fopen(out, "w");
        if (out_file == NULL)
        {
            fprintf(stderr, "Can't open %s for writing.\n", out);
            return 2;
        }
        else
        {
            while ((ch = getc(in_file)) != EOF)
                putc(ch, out_file);
            printf("File has been copied.\n");
            fclose(out_file);
        }
        fclose(in_file);
        return 0;
    }
}
```

Note that we could replace fgetc () and fputc () by their macro equivalents getc () and putc ().

## Command-Line Arguments

- It is possible to pass arguments to a program when it begins to execute
- To access these arguments as variables, main should be defined with the following argument declarations
- The first argument is usually called argc
  - An int to hold the number of arguments
- The second argument is usually called argv
  - An array of pointers to argument strings

Could be  
char \*\* argv

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

This section describes program input. The argc and argv parameters are flexible enough to cope with all forms of input from the command line. The argv pointer accesses the command-line arguments as strings. It is the programmer's responsibility to provide the parsing and validation of these strings.

It is usual practice to make a copy of these two arguments to ensure that their values are secured.

A third argument, usually called envp, is often available although it is not part of Standard C. It has the same structure as argv, but contains strings that hold the values of the environment variables. For example, one of the strings could be:

"BUFFERS=40"

Note that the second argument is a pointer to a 'pointer to char' and can be written as char \*\*argv. Naturally, this is also true for envp.

Exiting and carrying information from a program can be performed either by a return from main (with or without an explicit return value) or from anywhere in the program by using one of the following:

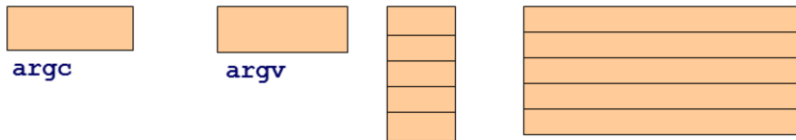
raise	Raises a signal in this process.
abort	Terminates abnormally (also raise(SIGABRT)).
exit	Terminates normally and closes open files, etc. (also raise(SIGTERM)).
atexit	Terminates normally by calling a user-defined function, which is supplied as an argument.

Note that signal handling is operating system dependant. The C standard supports very basic signal handling, which is obsolete on Unix, and does not fit into Microsoft Windows architecture very well.

## Command-Line Argument Example

```
C:\ >myecho Hi There!  
C:\MYECHO.EXE Hi There!  
C:\ >
```

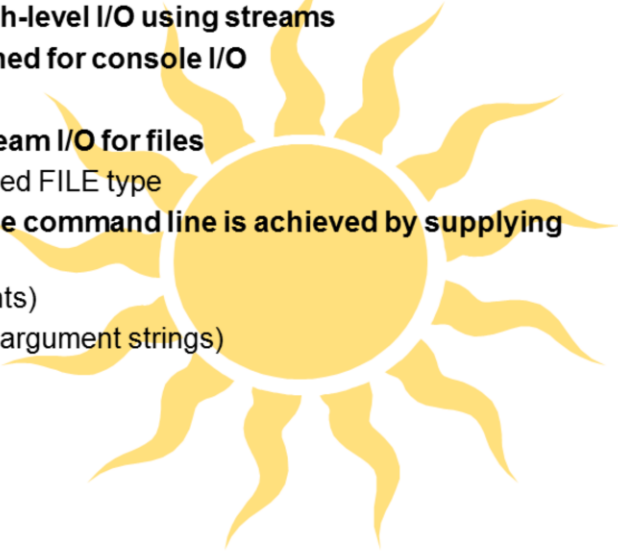
```
#include <stdio.h>  
/* Program to echo command line */  
/* arguments to standard output */  
  
int main(int argc, char * argv[])  
{  
    int i;  
    for (i = 0; i < argc; i++)  
    {  
        printf("%s%c", argv[i],  
              (i < argc - 1) ? ' ' : '\n');  
    }  
    return 0;  
}
```



The way in which `argv` is set up implies that `argc` is superfluous - the presence of the `NULL` pointer after the last pointer to argument string. It provides a choice, i.e. check with `argc` value or with the `argv` final pointer.

`argv[0]` contains a pointer to the program name, usually the name of the executable. Some operating systems support aliases (symbolic links), so strictly speaking this is the name by which the program was called.

### Summary

- **Input and output is achieved using library routines**
  - **The standard supports high-level I/O using streams**
  - **Standard streams are opened for console I/O**
    - `stdin`, `stdout` and `stderr`
  - **The standard supports stream I/O for files**
    - Handled using a predefined `FILE` type
  - **Input to a C program via the command line is achieved by supplying parameters to `main`**
    - `argc` (number of arguments)
    - `argv` (array of pointers to argument strings)
- 

---

Input and output are inherently non-portable. If portability is an issue, use the standard library routines mentioned in this chapter. If speed or hardware access is required, check the low-level functions.

`FILE` is an excellent example of data abstraction. If we are using the high-level functions, we need not concern ourselves with the way `FILE` is represented. This includes many forms of I/O including character, formatted and block reading and writing.

All of the routines mentioned in this chapter return information. Most input routines return values that are used directly in the program. Most of the other routines, including the output ones, provide useful return information that supplies status information. Using these return codes could improve the reliability of your program.

Input to a program through command line arguments and returning information from a program using the `return`/`exit`/`abort` mechanisms are both relatively straightforward. Programs return codes to the operating system. These codes provide some error-checking information.