Input and Output - Chapter Summary

1 Introduction

Input refers to the process of supplying information to a program during execution. Output refers to the process of sending information from the program to a file or device while the program is running.

As originally defined, C did not include any facilities for input and output, or I/O, as they are collectively called. A versatile range of functions for performing I/O was soon developed, however, and a wide selection of these is normally provided with each implementation of C, forming what has become a standard I/O library.

The prototypes and declarations for many of these functions are given in the header file `stdio.h`, which stands for Standard I/O Header. This file should therefore be included in any program using these I/O functions by giving the preprocessor directive:

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

2 Buffered and Unbuffered File Systems

Historically, there have been two rival methods of I/O available in C: the buffered file system, which stores information to or from a file in an area of temporary storage known as a buffer, and the unbuffered, or UNIX-like system, so-called because it was originally developed for use with the UNIX operating system and does not use ready-made file buffers.

The unbuffered file system provides low-level access to file data, and may, on some compilers, provide slightly faster access time for file operations. However, whereas the buffered system allows the programmer to handle data in whatever sized blocks are appropriate to the programming task, from single characters to large data structures, the unbuffered system passes data in blocks whose size is determined solely by the needs of the operating system. The unbuffered system also requires that the programmer provide and maintain the buffers to hold these blocks of data, and this inevitably makes the programmer's task more complicated.

For the moment, most C compilers continue to offer routines for both buffered and unbuffered I/O, but it should be noted that the ISO standard for C does not support the UNIX-like system, but defines an enlarged buffered file system.

3 Files and Devices

The I/O routines in the standard C library allow programs to read and write data to and from files and devices.

In general programming usage, a file is a named unit of storage that may be used to contain a program or information relevant to a program.

A device may be almost anything that can be connected to a computer! Usually, a computer system consists of a number of different physical devices in addition to the central processing unit, or CPU. Depending on the system configuration, these will typically include a console that consists of a screen and a keyboard, one or more disk drives of possibly different types and sizes, and probably a printer. There may, of course, be any number of other devices included.

In practice, the details of the way in which the CPU communicates with each physical device may be very different, but the buffered file system for C allows the programmer to ignore most of these distinctions and treat all physical devices and files in a consistent way, as what are termed logical files. In fact, C treats anything that is capable of I/O as a file. The only practical difference that remains is that some files allow random access, while others, such as the screen or the keyboard, do not.

This consistent approach is possible because, apart from the standard operations concerned with opening and closing logical files, C programs deal, not with the files or physical devices themselves, but with the sequences of bytes of data that may be written to or read from them.

4 Streams

A stream is a sequence of individual bytes of data. There are two types of stream.

Text streams are sequences of bytes that represent characters and formatted text. When outputted to screen or printer, the numbers held in the bytes are translated into characters according to the character convention used on the system, typically ASCII or EBCDIC. These characters will usually include escape sequences and formatting codes, so not all the characters in the stream will appear on screen or in the printout. Put another way, there may be more characters in a text stream than in the I/O data.

Binary streams are sequences of bytes that contain numbers. Typically, the numbers represent compiled machine code. These numbers are not translated into characters when outputted, and there are the same number of numbers in the stream as in the I/O data, although the ISO standard does allow the addition of null bytes to the end of the stream if these are required. This last provision is to meet the need in some systems for the stream to be padded so that it exactly fills a physical unit of storage.

5 I/O Functions

The standard C I/O library makes three types of function available: stream I/O functions, low-level I/O functions, and console and port I/O functions.

Stream functions use the buffered file system and provide either formatted or unformatted I/O. Their declarations are contained in `stdio.h`, which must be specifically included in the program file if they are to be used.

The low-level functions, such as `read`, `write`, `open`, `close` and `lseek`, use the un-buffered file system already described and cannot provide formatted I/O. They do not need the `stdio.h` file, although they may refer to standard constants defined in `stdio.h`, which should therefore be included if required.

Console and port functions write and read data to and from the screen, the keyboard and the numbered ports that provide interfaces between the CPU and specific physical devices, such as printers. Their declarations are in the console I/O header file `conio.h`, which should be specifically included if they are to be used, *and are not ISO standard*.

It is important to note that while an implementation of C offers all three kinds of I/O routines, they use very different methods to access data, and these three methods are not compatible with each other and are not covered by the ISO standard. Mixing stream and low-level functions in the same program may result in data held in buffers being lost. Console and port functions should not be used in conjunction with either stream or low-level functions, as they use calls to the operating system in order to carry out their tasks, and these calls may disrupt the operation of functions of either of the other two types.

6 The Standard Streams

Whatever the device or file that generates or stores them, all streams of the same type may be treated identically, and the same I/O functions can be applied to them.

However, whereas most files have to be explicitly opened before they can send or receive streams, there are three streams that are opened automatically when a C program begins execution: `stdin`, `stdout` and `stderr`.

Generally, the standard input stream, `stdin`, links the program and the keyboard. The standard output stream, `stdout`, links the program with the screen, as does the standard error stream, `stderr`.

Depending on the system configuration, two further streams may be opened when a C program begins execution: `stdprn`, which allows the program to send output to the standard printer, and `stdaux`, which connects the program to some other physical device, such as a mouse. *Despite their names, these streams are not ISO standard*.

The standard streams are closed automatically when execution of a program ends.

7 Single-Character Input from the Standard Devices

7.1. Buffered Input: `getchar`

`getchar` is a stream routine that provides buffered input from the standard input device (the keyboard). The character entered at the keyboard is first held in a buffer and displayed on the standard output device (the screen). While in the buffer, the character may be deleted by use of the [delete] key. The character is made available to the program when the [return] key is pressed. e.g.:

```
printf("\nPlease enter a value for 'letter': ");
letter = getchar();
```

The `printf` statement prompts the user, and the value typed in at the keyboard is assigned to the char variable `letter`.

7.2 Un-buffered Input: `getch` and `getche`

`getch` is the un-buffered console function equivalent to `getchar`. The character is made available to the program as soon as the key is depressed, without waiting for a [RETURN] first, and the character does not appear on the screen.

`getche` is an un-buffered console function similar to `getch`, but the character is echoed on the screen when the key is depressed, although it cannot, of course, be deleted with the [DELETE] key.

Both routines are not ISO standard.

8 Single-Character Output to the Standard Devices

8.1 Buffered Output: `putchar`

`putchar` is a stream routine that writes a character from the program to the standard output device (the screen). The character to be written is placed as an argument between the function's brackets, and character constants are written enclosed in single quotes:

```
putchar('Y');
printf("\nPlease press a key on the keyboard: ");
putchar(getchar());
```

The first `putchar` statement prints a capital `'Y'` to the screen (without the single quotes, of course). The second `putchar` contains `getchar` as its argument, which will

input the character entered by the user in response to the `printf` prompt in the line above when the [return] key is pressed. The `putchar` routine will then output this character to the screen. The result is to echo the user's input on a new line (because the [RETURN] key was pressed to enter the original data).

8.2 Un-buffered Output: `putch`

`putch` is the console routine equivalent to `putchar`; it writes a single character from the program to the screen. Again, the character to be written is enclosed as an argument in the function's brackets. This routine, like `getch` and `getche`, is not ISO standard.

9 Formatted I/0 to and from the Standard Devices

The most commonly-used formatted I/O functions are the stream functions `printf` and `scanf`, as described in previous chapters.

Note that `printf` is a function of type `int` and returns a positive integer value equal to the number of characters actually printed, or a negative value if an error has occurred.

`scanf` is also a function of type `int` and returns a positive integer equal to the number of fields that were successfully assigned values, or zero, if no fields were assigned.

The other functions available for formatted I/O to and from the standard devices are:

`sprintf` writes formatted data to a string buffer

`sscanf` reads formatted data from a string buffer

`vprintf` writes formatted data to the standard output device

`vsprintf` writes formatted data to a string buffer

Further details of these functions are given in Section 14 at the end of this summary.

10 Unformatted String I/O to and from the Standard Devices

`gets` is a buffered stream function that reads a line from the standard input device (the keyboard). That is to say, it reads a string of characters terminated by the [RETURN] key, which adds the null character `'\0'` to mark the end of the string. The input for `gets` appears on the screen and can be edited with the [DELETE] key until a [return] is pressed. This mechanism has the side effect that `gets` cannot be used to read a carriage-return character.

The input from `gets` is assigned the address of the character pointer or array that it takes as its argument.

The `puts` function takes a string as its argument and writes it to the standard output device (the screen). `puts` can use the same escape sequences as `printf`, but cannot write formatted data. It is a much simpler function than `printf` and therefore runs faster and takes up less space in memory.

The following program uses `puts`, `gets` and `printf` to achieve simple interaction with the user:

```
#include <stdio.h>
int main(void)
{
     char name[30];

     puts("\nTell me, what is your name?");
     scanf("%29s", name);
     printf("\nHello, %s!", name);

     return 0;
}
```

11 File Access and Stream Functions

11.1 Opening and Closing Files: `fopen` and `fclose`

Files other than the standard files must be explicitly opened before they can send or receive streams, and must then be closed again after use.

The buffered file system uses the `fopen` and `fclose` functions respectively to open and close files for use.

The general form of the `fopen` function is:

```
fopen(file name, mode)
```

The three principal modes available to `fopen` are:

`"r"`	Opens the file to be read. If the file does not exist or cannot be found, the function call will fail.
`"w"`	Opens a new, blank file to be written. Note that any existing file of the same name will be destroyed.
`"a"`	Opens the file to be appended to; new data is going to be added at the end of what is already there. The file is created if it does not already exist.

Modified versions of these three basic modes are also available:

"r+" Opens an existing file for both reading and writing.
"w+" Opens a new, blank file for both reading and writing. Note again that any existing file of the same name will be destroyed.
"a+" Opens the file for both reading and appending, creating it if it does not already exist. An existing file is opened at the end of what is already there.

fopen returns a pointer to a FILE (see 11.2 below), and this pointer will have the value NULL, as defined in stdio.h, if the operation fails because the file cannot be opened for any reason.

Note that when the modified modes "r+", "w+" and "a+" are used, although reading and writing are both allowed, the position of the file pointer must be reset before changing from one operation to the other by using a call to one of the following functions: fsetpos, which sets the position indicator of a stream, fseek, which repositions the file pointer to a given byte in the file, as described in Section 15.9 below, or rewind, which sets the file pointer at the very beginning of the stream.

Note also that some compilers offer mode modifiers, which allow text streams to be read as binary streams and vice versa.

More than one file may be open at the same time if required, up to a limit dependent on the system being used.

Files should always be closed after use. The fclose function has the general form:

```
fclose(file pointer)
```

11.2 Pointers to FILE

C's definition of a file is represented by a structure whose template is included in stdio.h. A #define command in stdio.h specifies that the name of this structure type may be represented symbolically by the word FILE. A pointer to a file is therefore declared as a pointer to type FILE.

Many programmers use the mnemonic convention of calling the input stream in and the output stream out, so file pointers returned by fopen are typically assigned as follows:

```
FILE * in;
FILE * out;

in = fopen("source.txt", "r");
out = fopen("target.txt", "w");
```

In this instance, `in` is the input stream from the file `source.txt`, which has been opened for reading, and `out` is the output stream to the file `target.txt`, which has been opened for writing.

11.3 EOF

Just as C uses the convention that the end of a string is indicated by the null character `'\0'`, so the end of a file is marked by a character that will not be used anywhere in the body of a text file; typically the character whose value is `-1`. The actual value of this character is given in the `stdio.h` file, but a `#define` command in that file also specifies that this value may be represented by the mnemonic `EOF`, for End Of File, and this is the value that is actually used in a program.

Using the symbolic constant `EOF` not only makes the program more readable, it also ensures that the program is portable to other systems where the end-of-file character may be defined to have a different value.

Note that the value of `EOF` may be a valid character in a binary file. There is a specific function called `feof` available for testing for the end of a file, however, and this should be used when checking for the end of a binary file. `feof` takes a file pointer as its argument and returns zero unless the end of a file has been reached.

11.4 File I/O Functions

The standard C library offers a wide variety of buffered I/O functions for moving through, reading from and writing to files once they have been opened by a call to `fopen`.

Details of the following commonly-used functions are given in Section 15 below:

`fgetc`, `fputc`, `fprintf`, `fscanf`, `fgets`, `fputs`, `fread`, `fwrite`, `fseek`

11.5 Additional File I/O Functions

Many other functions are available for buffered I/O with files, and programmers should consult their compiler manuals for details.

Note that the common routines `getc` and `putc` are functionally identical to `fgetc` and `fputc` respectively. Many implementations of C actually use a `#define` directive in `stdio.h` to replace `getc` and `putc` by their equivalent 'f for file' functions.

Formatted input with variable arguments is also possible with files using the `vfprintf` function, which is similar to the `vprintf` function described in 14.3 below.

11.6 Loops and Streams

The `while` loop provides a convenient and powerful mechanism for operations involving streams. Although `getc` or `fgetc` read only a single character from a stream,

a call to either function from within the brackets of a `while` statement results in a whole sequence of characters being read until the end condition is met.

This allows operations on whole files to be carried out, with appropriate changes at the desired file positions. For example, the following `while` loop copies the file generating the input stream to the file written to by the output stream, replacing all dollar `'$'` characters by sterling `'#'` signs:

```
while ((ch = fgetc(in)) != EOF)     /* Note the precedence */
{
     if (ch == '$')
          fputc('#', out);
     else
          fputc(ch, out);
}
```

11.7 Low-Level Access to Files

Most implementations of C offer a range of library functions to carry out low-level I/O on files, and programmers should consult their compiler manuals for details. Note that these functions do not buffer the data, nor do they provide the option of formatted data.

12 Error Trapping

I/O operations involving files typically employ a large number of library functions to open, read, move through, write and close the files. Each of these many functions offers the opportunity for error or failure, so that it is often difficult to trace the exact source of a fault in a file-handling program.

It is therefore essential that output statements should be included in the program to confirm the success or failure of each individual function and the result of the complete operation. Experience shows that the additional time that this takes the programmer is more than compensated for by the time saved in debugging the program!

Note that failure to open a file for reading may not be the result of a programming error at all. The file may not exist or may not be on the disk currently in the disk drive. A suitable **failure to open** message may save the programmer searching through the program for a code error that is not there.

13 Command-Line Arguments: `argc` and `argv`

Normally, a compiled program is run by typing in its name and a [return] at the system prompt. C provides the opportunity for additional parameters to be included in this command line, however, by allowing two optional arguments for the `main` function, known by convention as `argc` and `argv`.

The first argument, `argc`, is the argument count; that is, the number of arguments pointed to by the second argument, `argv`, which is the list of argument values. `argc` is therefore an `int`, and `argv` is a pointer to an array of `chars`, or a pointer to a pointer to `char`. Many implementations of C use the convention that the first argument in the argument list, `argv[0]`, is the name of the program itself.

The following program is an adaptation of the file-copying program given in the Input and Output chapter in the C notebook:

```
#include <stdio.h>

int main(int argc, char * argv[])
{
     FILE * in;

     if (argc <3)
     {
          puts("\nPlease give names of file to be read ")
               "and file to be appended to.");
          puts("The command line syntax is:"
               "\n append read_f_name appended_to_f_name");
          exit(1);
     }

     in = fopen(argv[1], "r");
     if (in == NULL)
          printf("\nCannot open %s for reading.", argv[1]);
     else
     {
          FILE * out = fopen(argv[2], "a");
          if (out == NULL)
               printf("\nCannot open %s for appending to.",
                         argv[2]);
          else
          {
               int ch;
               while ((ch = fgetc(in)) != EOF)
                    fputc(ch, out);
               printf("\nContents of %s have been appended "
                      "to %s.", argv[1], argv[2]);
```

```
                fclose(out);
            }
            fclose(in);
        }
        return 0;
}
```

The name of the program is `append`, and it is run by entering a command line consisting of the name of the program, followed by first the name of the file to be read, then the name of the file to which the data is to be appended.

Some programmers use the convention that a command-line argument that indicates a user option begins with a hyphen, so that a further adaptation of the above `append` program called `optapnd`, with an argument count of `4`, offering the option of either overwriting a file or appending to it, might have a command line:

```
optapnd source.txt target.txt -w
```

or:

```
optapnd source.txt target.txt –a
```

14 Further Examples of Formatted I/O Functions

14.1 `sprintf`

`sprintf` is so-called because it is a string version of `printf`. It writes formatted data to a character array. `printf` itself can print strings within the control string by use of the `%s` conversion specification, but these strings cannot be assigned to storage by the `printf` statement, which returns an integer value.

The general form of the `sprintf` statement is:

```
sprintf(buffer, control string, argument list);
```

The `buffer` is the address of a character array, represented either by the name of the array or by a pointer.

`sprintf` is a function of type `int` that returns a value equal to the number of characters successfully placed in the buffer array.

```
char next_train[30];
sprintf(next_train, "\nThe %.2f is the %s.",
          9.15, "Red Dragon");
```

In this instance, the array `next_train` would be passed the string:

```
9.15 Red Dragon
```

14.2 sscanf

sscanf is a string version of scanf, which reads formatted data into a program from a character array. It has the general form:

```
sscanf(buffer, control string, argument list);
```

As with sprintf, buffer is the name of, or a pointer to, an array of characters.

The following program fragment uses sscanf to assign the contents of the buffer to a floating-point variable and two character arrays, called due, name and transport respectively:

```
float due;
char name[20];
char transport[20];
sscanf("9.15 Smith train","%f%s%s",
          &due, name, transport);
printf("\nIt's %.2f!  Hurry Mr %s, "
       "or you'll miss the %s!.", due, name, train);
```

When executed, the above fragment will print out:

```
It's 9.15!  Hurry Mr Smith, or you'll miss the train!"
```

14.3. vprintf and vsprintf

The functions vprintf and vsprintf are similar to printf and sprintf; they output formatted data to the standard output device and to a character array, or buffer, respectively. Instead of an argument list, however, they each take a pointer to an argument list. This mechanism allows the creation of generalised formatted output functions, which can take varying numbers of arguments with varying names.

The general forms of vprintf and vsprintf are:

```
vprintf(control string, pointer to arg list)

vsprintf(buffer, control string, pointer to arg list)
```

The position of the quotes around the control-string arguments of these two functions is critical. Each is a single string with the logical arguments contained within this string without any other quotes to indicate where the control string ends. The logical arguments within the control string are simply separated by commas.

The pointer to the argument list should be of the type va_list, as defined in the standard argument header file stdarg.h, which must be included in the program file if either of these functions is to be used. The macros va_start and va_end must also be used to create and clear respectively the variable-length pointer to the argument list.

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

void describe(char *, …);

int main(void)
{
     long big = 32777L;
     char * fil1name = "read.me";
     char * fil1kind = "text";
     char * fil2name = "valuable";

     describe("\n%s is a %s file of %ld bytes.",
                         fil1name, fil1kind, big");
     describe("\nError.  File '%s' does not exist.",
                                        fil2name");
     return 0;
}

void describe(char * info, …)
{
     va_list argpoint;

     va_start(argpoint, info);
     vprintf(info, argpoint);
     va_end(argpoint);

}
```

15 Details of Buffered File I/O Functions

15.1 fgetc

`fgetc` reads the next single character from a stream and returns `EOF` if an error or the end of the file is encountered. `fgetc` takes a file pointer (that is, the name of the stream) as its argument, thus:

```
ch = fgetc(in);
```

15.2 fputc

`fputc` writes a single character to a stream and then increments the file-position indicator. `fputc` takes two arguments: first the character that it is going to output, then the stream to which the character will be sent, as in:

```
fputc('C', out);
```

It is a common construction in C to use a call to `fgetc` as an argument to `fputc`:

```
fputc(fgetc(in), out);
```

The value returned by `fputc` is the value of the character written, or `EOF` if an error occurs.

15.3 fprintf

`fprintf` writes formatted data to a stream. The function has the general form:

```
fprintf(stream, control string, argument list)
```

As its name suggests, `fprintf` acts exactly like a file version of `printf`, and the control string takes the same conversion specifications:

```
char io[4] = "I/O";
int number = 2;

fprintf(out, "\n%s, %s, it's off %d work we go!",
        io, io, number);
```

15.4 fscanf

`fscanf` reads formatted data from a stream and has the general form:

```
fscanf(stream, control string, argument list)
```

This is the file counterpart to `scanf`, which it closely resembles. `fscanf` returns an integer value equal to the number of arguments that are assigned values, or `EOF` if an attempt was made to read past the end of a file.

The following example assigns from the stream `in`: a floating-point number to a `float` variable `fraction`, an integer to an `int` variable `wholenum` and a character to a `char` variable `letter`:

```
fscanf(in, "%f%d%c", &fraction, &wholenum, &letter);
```

15.5 fgets

`fgets` reads a string from a stream and has the general form:

```
fgets(pointer to char array, number, stream)
```

`fgets` continues to read characters from the stream specified until it encounters a new-line character, an `EOF` or until it has read number-minus-one characters. Characters are placed in the array pointed to by the first argument, and a null character `'\0'` is added to the end of the string after the last character has been read. `fgets` returns the address of the array where the string is stored, or a null pointer if an error has occurred or if the end of the file has been reached.

15.6 fputs

`fputs` writes a string to a stream and has the general form:

```
fputs(string, stream)
```

`fputs` returns a zero value if successful or a non-zero value if an error has occurred. Note that the null character `'\0'` at the end of the string is not written to the stream. The following example writes the string literal to the `out` stream:

```
fputs("How long is a piece of string?", out);
```

15.7 fread

`fread` reads a specified number of data objects of a specified size from a stream to an array. `fread` has the general form:

```
fread(pointer to array, data size, count, stream)
```

15.8 fwrite

`fwrite` is the output counterpart to `fread`. It writes a specified number of data objects of a specified size from an array to a stream. `fwrite` has the general form:

```
fwrite(pointer to array, data size, count, stream)
```

15.9 fseek

`fseek` repositions a file-position indicator to a given byte in a file. It has the general form:

```
fseek(stream, offset, origin)
```

The `offset` is the number of bytes the position indicator has to move from the origin to reach the target position. The origin may be `0` for the start of the file, `1` for the current position or 2 for the end of the file. `fseek` returns zero if successful and non-zero on failure.

Note that the variable used for the offset must be of type `long int` in order to support the use of files larger than 64k.

Note also that the ISO standard implementation of `fseek` should only be used with binary files, since the number of characters in a text file may not be the same as the number of characters in its associated stream, as explained in Section 4 above, and this may cause `fseek` to generate positional errors.

The function `ftell` may be used in conjunction with `fseek` as it returns the current value of the file position indicator, and may therefore be used to determine the offset of those positions in a file to which `fseek` is required to move the position indicator