Subsections

- Reporting Errors
  - `perror()`
  - errno
  - `exit()`
- Streams
  - Predefined Streams
    - Redirection
- Basic I/O
- Formatted I/O
  - Printf
- scanf
- Files
  - Reading and writing files
- sprintf and sscanf
  - Stream Status Enquiries
- Low Level I/O
- Exercises

Input and Output (I/O):`stdio.h`

This chapter will look at many forms of I/O. We have briefly mentioned some forms before will look at these in much more detail here.

Your programs will need to include the standard I/O ***header*** file so do:

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

Reporting Errors

Many times it is useful to report errors in a C program. The standard library `perror()` is an easy to use and convenient function. It is used in conjunction with `errno` and frequently on encountering an error you may wish to terminate your program early. Whilst not strictly part of the `stdio.h` library we introduce the concept of `errno` and the function `exit()` here. We will meet these concepts in other parts of the Standard Library also.

`perror()`

The function `perror()` is prototyped by:

```
void perror(const char *message);
```

perror() produces a message (on standard error output -- see Section 17.2.1), describing the last error encountered, returned to `errno` (see below) during a call to a system or library function. The argument string `message` is printed first, then a colon and a blank, then the message and a newline. If `message` is a NULL pointer or points to a null string, the colon is not printed.

errno

`errno` is a special system variable that is set if a system call cannot perform its set task. It is defined in `#include <errno.h>`.

To use `errno` in a C program it must be declared via:

```
extern int errno;
```

It can be manually reset within a C program (although this is uncommon practice) otherwise it simply retains its last value returned by a system call or library function.

`exit()`

The function `exit()` is prototyped in `#include <stdlib>` by:

```
void exit(int status)
```

Exit simply terminates the execution of a program and returns the exit `status` value to the operating system. The `status` value is used to indicate if the program has terminated properly:

- it exist with a `EXIT_SUCCESS` value on successful termination
- it exist with a `EXIT_FAILURE` value on unsuccessful termination.

On encountering an error you may frequently call an `exit(EXIT_FAILURE)` to terminate an errant program.

Streams

Streams are a portable way of reading and writing data. They provide a flexible and efficient means of I/O.

A Stream is a file or a physical device (***e.g.*** printer or monitor) which is manipulated with a **pointer** to the stream.

There exists an internal C data structure, `FILE`, which represents all streams and is defined in `stdio.h`. We simply need to refer to the `FILE` structure in C programs when performing I/O with streams.

We just need to declare a variable or pointer of this type in our programs.

We do not need to know any more specifics about this definition.

We must open a stream before doing any I/O,

then access it

and then close it.

Stream I/O is **BUFFERED**: That is to say a fixed ``chunk'' is read from or written to a file via some temporary storage area (the buffer). This is illustrated in Fig. 17.1. NOTE the file pointer actually points to this buffer.

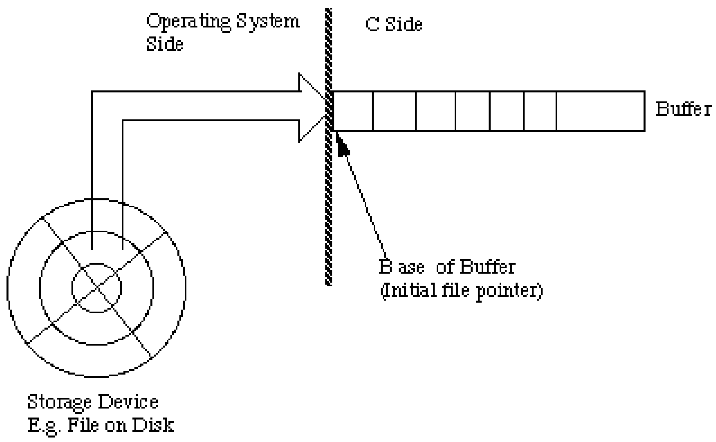

Fig. **Stream I/O Model** This leads to efficient I/O but **beware**: data written to a buffer does not appear in a file (or device) until the buffer is flushed or written out. (\n does this). Any abnormal exit of code can cause problems.

Predefined Streams

UNIX defines 3 predefined streams (in `stdio.h`):

```
stdin,  stdout,  stderr
```

They all use text a the method of I/O.

`stdin` and `stdout` can be used with files, programs, I/O devices such as keyboard, console, ***etc.***. `stderr` always goes to the console or screen.

The console is the default for `stdout` and `stderr`. The keyboard is the default for `stdin`.

Predefined stream are automatically open.

Redirection

This how we override the UNIX default predefined I/O defaults.

This is not part of C but operating system dependent. We will do redirection from the command line.

> -- redirect `stdout` to a file.

So if we have a program, `out`, that usually prints to the screen then

```
out > file1
```

will send the output to a file, `file1`.

< -- redirect `stdin` from a file to a program.

So if we are expecting input from the keyboard for a program, `in` we can read similar input from a file

```
in < file2.
```

| -- ***pipe***: puts `stdout` from one program to `stdin` of another

```
prog1 | prog2
```

e.g. Sent output (usually to console) of a program direct to printer:

```
out | lpr
```

Basic I/O

There are a couple of function that provide basic I/O facilities.

probably the most common are: `getchar()` and `putchar()`. They are defined and used as follows:

- `int getchar(void)` -- reads a char from `stdin`
- `int putchar(char ch)` -- writes a char to `stdout`, returns character written.

```
int ch;

                ch = getchar();
                (void) putchar((char) ch);
```

Related Functions:

```
  int getc(FILE *stream),
int putc(char ch,FILE *stream)
```

Formatted I/O

We have seen examples of how C uses formatted I/O already. Let's look at this in more detail.

Printf

The function is defined as follows:

`int printf(char *format, arg list ...)` --
prints to `stdout` the list of arguments according specified format string. Returns number of characters printed.

The **format string** has 2 types of object:

- ***ordinary characters*** -- these are copied to output.
- ***conversion specifications*** -- denoted by % and listed in Table 17.1.

Table: Printf/scanf format characters

Format Spec (%)	Type	Result
c	char	single character
i,d	int	decimal number
o	int	octal number
x,X	int	hexadecimal number
		lower/uppercase notation
u	int	unsigned int
s	char *	print string
		terminated by \0
f	double/float	format -m.ddd...
e,E	"	Scientific Format
		-1.23e002
g,G	"	e or f whichever
		is most compact
%	-	print % character

Between % and format char we can put:

- (minus sign)
-- left justify.

integer number
-- field width.

m.d
-- m = field width, d = precision of number of digits after decimal point <u>or</u> number of chars from a string.

So:

```
printf("%-2.3f\n",17.23478);
```

The output on the screen is:

```
17.235
```

`and:`

```
printf("VAT=17.5%%\n");
```

`...outputs:`

```
VAT=17.5%
```

scanf

This function is defined as follows:

`int scanf(char *format, args....)` -- reads from stdin and puts input in address of variables specified in `args` list. Returns number of chars read.

Format control string similar to `printf`

Note: The ADDRESS of variable or a pointer to one is required by `scanf`.

```
scanf(``%d'',&i);
```

We can just give the name of an array or string to scanf since this corresponds to the start address of the array/string.

```
char string[80];
              scanf(``%s'',string);
```

Files

Files are the most common form of a stream.

The first thing we must do is ***open*** a file. The function `fopen()` does this:

```
FILE *fopen(char *name, char *mode)
```

`fopen` returns a pointer to a FILE. The `name` string is the name of the file on disc that we wish to access. The `mode` string controls our type of access. If a file cannot be accessed for any reason a `NULL` pointer is returned.

```
Modes include: ``r'' -- read,
               ``w'' -- write and
               ``a'' -- append.
```

To open a file we must have a stream (file pointer) that ***points*** to a `FILE` structure.

So to open a file, called ***myfile.dat*** for reading we would do:

```
  FILE *stream, *fopen();
                 /* declare a stream and prototype fopen */

                 stream = fopen(``myfile.dat'',``r'');
```

```
it is good practice to to check file is opened
correctly:

                 if ( (stream = fopen( ``myfile.dat'',
                                                ``r'')) == NULL)
                                 {  printf(``Can't open %s\n'',
                                                 ``myfile.dat'');
                                  exit(1);
                                 }
                 ......
```

Reading and writing files

The functions `fprintf` and `fscanf` a commonly used to access files.

```
int fprintf(FILE *stream, char *format, args..)
                 int fscanf(FILE *stream, char *format, args..)
```

These are similar to `printf and scanf except that data is read from the` ***`stream`*** `that must have been opened with fopen().`

`The stream pointer is automatically incremented with` <u>`ALL`</u> `file read/write functions. We` **`do not`** `have to worry about doing this.`

```
  char *string[80]
                 FILE *stream, *fopen();

                 if ( (stream = fopen(...)) != NULL)
                                 fscanf(stream,``%s'', string);
```

```
Other functions for files:

int getc(FILE *stream), int fgetc(FILE *stream)

int putc(char ch, FILE *s), int fputc(char ch, FILE *s)

These are like getchar, putchar.

getc is defined as preprocessor MACRO in stdio.h. fgetc is a C library function.
Both achieve the same result!!

   fflush(FILE *stream) -- flushes a stream.

   fclose(FILE *stream) -- closes a stream.
```

`We can access predefined streams with fprintf` ***`etc.`***

```
   fprintf(stderr,``Cannot Compute!!\n'');

    fscanf(stdin,``%s'',string);
```

sprintf and sscanf

These are like `fprintf` and `fscanf` except they read/write to a string.

`int sprintf(char *string, char *format, args..)`

`int sscanf(char *string, char *format, args..)`

For Example:

```
  float full_tank = 47.0; /* litres */
                float miles = 300;
                char miles_per_litre[80];

                sprintf( miles_per_litre,``Miles per litre
                               = %2.3f'', miles/full_tank);
```

Stream Status Enquiries

There are a few useful stream enquiry functions, prototyped as follows:

```
 int feof(FILE *stream);
 int ferror(FILE *stream);
 void clearerr(FILE *stream);
 int fileno(FILE *stream);
```

Their use is relatively simple:

`feof()`
: -- returns true if the stream is currently at the end of the file. So to read a stream,`fp`, line by line you could do:

```
while ( !feof(fp) )
  fscanf(fp,"%s",line);
```

`ferror()`
: -- reports on the error state of the stream and returns true if an error has occurred.

`clearerr()`
: -- resets the error indication for a given stream.

`fileno()`
: -- returns the integer file descriptor associated with the named stream.

Low Level I/O

This form of I/O is UNBUFFERED -- each read/write request results in accessing disk (or device) directly to fetch/put a specific number of **bytes**.

There are no formatting facilities -- we are dealing with bytes of information.

This means we are now using binary (and not text) files.

Instead of file pointers we use ***low level*** `file handle or file descriptors` which give a unique integer number to identify each file.

To Open a file use:

`int open(char *filename, int flag, int perms)` -- this returns a file descriptor or -1 for a **fail**.

The `flag` controls file access and has the following predefined in `fcntl.h`:

`O_APPEND, O_CREAT, O_EXCL, O_RDONLY, O_RDWR, O_WRONLY` + others see online `man` pages or reference manuals.

`perms` -- best set to 0 for most of our applications.

The function:

```
creat(char *filename, int perms)
```

can also be used to create a file.

`int close(int handle)` -- close a file

```
int read(int handle, char *buffer,
unsigned length)

int write(int handle, char *buffer, unsigned length)
```

are used to read/write a specific number of bytes from/to a file (handle) stored or to be put in the memory location specified by `buffer`.

The `sizeof()` function is commonly used to specify the `length`.

read and write return the number of bytes read/written or -1 if they fail.

```
/* program to  read a list of floats from a binary file */
/* first byte of file is an integer saying how many  */
/* floats in file. Floats follow after it, File name got from */
/* command line */

#include<stdio.h>
#include<fcntl.h>

float bigbuff[1000];

main(int argc, char **argv)

{  int fd;
                 int bytes_read;
                 int file_length;

                 if ( (fd = open(argv[1],O_RDONLY)) = -1)
                                { /* error file not open */....
                                                perror("Datafile");
                                                exit(1);
                                }
                 if ( (bytes_read = read(fd,&file_length,
                                                sizeof(int))) == -1)
                                { /* error reading file */...
                                                exit(1);
                                }
                 if ( file_length > 999 )  {/* file too big */ ....}
                 if ( (bytes_read = read(fd,bigbuff,
                                               file_length*sizeof(float))) == -1)
                                { /* error reading open */...
                                                exit(1);
                                }
}
```

Exercises

Exercise 12573

Write a program to copy one named file into another named file. The two file names are given as the first two arguments to the program.

Copy the file a block (512 bytes) at a time.

```
Check:  that the program has two arguments
```

```
  or print "Program need two arguments"
that the first name file is readable
  or print "Cannot open file .... for reading"
 that the second file is writable
   or print "Cannot open file .... for writing"
```

Exercise 12577

Write a program last that prints the last *n* lines of a text file, by *n* and the file name should be specified form command line input. By default *n* should be 5, but your program should allow an optional argument so that

```
last -n file.txt
```

prints out the last n lines, where n is any integer. Your program should make the best use of available storage.

Exercise 12578

Write a program to compare two files and print out the lines where they differ. Hint: look up appropriate string and file handling library routines. This should not be a very long program.

Dave Marshall
1/5/1999