5

Standard I/O Library

5.1 Introduction

In this chapter, we describe the standard I/O library. This library is specified by the ISO C standard because it has been implemented on many operating systems other than the UNIX System. Additional interfaces are defined as extensions to the ISO C standard by the Single UNIX Specification.

The standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size (as in Section 3.9). This makes the library easy to use, but at the same time introduces another set of problems if we're not cognizant of what's going on.

> The standard I/O library was written by Dennis Ritchie around 1975. It was a major revision of the Portable I/O library written by Mike Lesk. Surprisingly little has changed in the standard I/O library after more than 35 years.

5.2 Streams and `FILE` Objects

In Chapter 3, all the I/O routines centered on file descriptors. When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operations. With the standard I/O library, the discussion centers on *streams*. (Do not confuse the standard I/O term *stream* with the STREAMS I/O system that is part of System V and was standardized in the XSI STREAMS option in the Single UNIX Specification, but is now marked obsolescent in SUSv4.) When we open or create a file with the standard I/O library, we say that we have associated a stream with the file.

With the ASCII character set, a single character is represented by a single byte. With international character sets, a character can be represented by more than one byte.

Standard I/O file streams can be used with both single-byte and multibyte ("wide") character sets. A stream's orientation determines whether the characters that are read and written are single byte or multibyte. Initially, when a stream is created, it has no orientation. If a multibyte I/O function (see `<wchar.h>`) is used on a stream without orientation, the stream's orientation is set to wide oriented. If a byte I/O function is used on a stream without orientation, the stream's orientation is set to byte oriented. Only two functions can change the orientation once set. The `freopen` function (discussed shortly) will clear a stream's orientation; the `fwide` function can be used to set a stream's orientation.

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

int fwide(FILE *fp, int mode);
```

Returns: positive if stream is wide oriented,
negative if stream is byte oriented,
or 0 if stream has no orientation

The `fwide` function performs different tasks, depending on the value of the *mode* argument.

- If the *mode* argument is negative, `fwide` will try to make the specified stream byte oriented.
- If the *mode* argument is positive, `fwide` will try to make the specified stream wide oriented.
- If the *mode* argument is zero, `fwide` will not try to set the orientation, but will still return a value identifying the stream's orientation.

Note that `fwide` will not change the orientation of a stream that is already oriented. Also note that there is no error return. Consider what would happen if the stream is invalid. The only recourse we have is to clear `errno` before calling `fwide` and check the value of `errno` when we return. Throughout the rest of this book, we will deal only with byte-oriented streams.

When we open a stream, the standard I/O function `fopen` (Section 5.5) returns a pointer to a `FILE` object. This object is normally a structure that contains all the information required by the standard I/O library to manage the stream: the file descriptor used for actual I/O, a pointer to a buffer for the stream, the size of the buffer, a count of the number of characters currently in the buffer, an error flag, and the like.

Application software should never need to examine a `FILE` object. To reference the stream, we pass its `FILE` pointer as an argument to each standard I/O function. Throughout this text, we'll refer to a pointer to a `FILE` object, the type `FILE *`, as a *file pointer*.

Throughout this chapter, we describe the standard I/O library in the context of a UNIX system. As we mentioned, this library has been ported to a wide variety of other operating systems. To provide some insight about how this library can be implemented, we will talk about its typical implementation on a UNIX system.

5.3 Standard Input, Standard Output, and Standard Error

Three streams are predefined and automatically available to a process: standard input, standard output, and standard error. These streams refer to the same files as the file descriptors `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`, respectively, which we mentioned in Section 3.2.

These three standard I/O streams are referenced through the predefined file pointers `stdin`, `stdout`, and `stderr`. The file pointers are defined in the `<stdio.h>` header.

5.4 Buffering

The goal of the buffering provided by the standard I/O library is to use the minimum number of `read` and `write` calls. (Recall Figure 3.6, which showed the amount of CPU time required to perform I/O using various buffer sizes.) Also, this library tries to do its buffering automatically for each I/O stream, obviating the need for the application to worry about it. Unfortunately, the single aspect of the standard I/O library that generates the most confusion is its buffering.

Three types of buffering are provided:

1. Fully buffered. In this case, actual I/O takes place when the standard I/O buffer is filled. Files residing on disk are normally fully buffered by the standard I/O library. The buffer used is usually obtained by one of the standard I/O functions calling `malloc` (Section 7.8) the first time I/O is performed on a stream.

   The term *flush* describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines, such as when a buffer fills, or we can call the function `fflush` to flush a stream. Unfortunately, in the UNIX environment, *flush* means two different things. In terms of the standard I/O library, it means writing out the contents of a buffer, which may be partially filled. In terms of the terminal driver, such as the `tcflush` function in Chapter 18, it means to discard the data that's already stored in a buffer.

2. Line buffered. In this case, the standard I/O library performs I/O when a newline character is encountered on input or output. This allows us to output a single character at a time (with the standard I/O `fputc` function), knowing that actual I/O will take place only when we finish writing each line. Line buffering is typically used on a stream when it refers to a terminal—standard input and standard output, for example.

   Line buffering comes with two caveats. First, the size of the buffer that the standard I/O library uses to collect each line is fixed, so I/O might take place if we fill this buffer before writing a newline. Second, whenever input is requested through the standard I/O library from either (a) an unbuffered stream

or (b) a line-buffered stream (that requires data to be requested from the kernel), *all* line-buffered output streams are flushed. The reason for the qualifier on (b) is that the requested data may already be in the buffer, which doesn't require data to be read from the kernel. Obviously, any input from an unbuffered stream, item (a), requires data to be obtained from the kernel.

3. Unbuffered. The standard I/O library does not buffer the characters. If we write 15 characters with the standard I/O `fputs` function, for example, we expect these 15 characters to be output as soon as possible, probably with the `write` function from Section 3.8.

   The standard error stream, for example, is normally unbuffered so that any error messages are displayed as quickly as possible, regardless of whether they contain a newline.

ISO C requires the following buffering characteristics:

- Standard input and standard output are fully buffered, if and only if they do not refer to an interactive device.
- Standard error is never fully buffered.

This, however, doesn't tell us whether standard input and standard output are unbuffered or line buffered if they refer to an interactive device and whether standard error should be unbuffered or line buffered. Most implementations default to the following types of buffering:

- Standard error is always unbuffered.
- All other streams are line buffered if they refer to a terminal device; otherwise, they are fully buffered.

The four platforms discussed in this book follow these conventions for standard I/O buffering: standard error is unbuffered, streams open to terminal devices are line buffered, and all other streams are fully buffered.

We explore standard I/O buffering in more detail in Section 5.12 and Figure 5.11.

If we don't like these defaults for any given stream, we can change the buffering by calling either the `setbuf` or `setvbuf` function.

```
#include <stdio.h>

void setbuf(FILE *restrict fp, char *restrict buf);

int setvbuf(FILE *restrict fp, char *restrict buf, int mode,
            size_t size);
```

Returns: 0 if OK, nonzero on error

These functions must be called *after* the stream has been opened (obviously, since each requires a valid file pointer as its first argument) but *before* any other operation is performed on the stream.

With `setbuf`, we can turn buffering on or off. To enable buffering, *buf* must point to a buffer of length `BUFSIZ`, a constant defined in `<stdio.h>`. Normally, the stream is then fully buffered, but some systems may set line buffering if the stream is associated with a terminal device. To disable buffering, we set *buf* to `NULL`.

With `setvbuf`, we specify exactly which type of buffering we want. This is done with the *mode* argument:

`_IOFBF`	fully buffered
`_IOLBF`	line buffered
`_IONBF`	unbuffered

If we specify an unbuffered stream, the *buf* and *size* arguments are ignored. If we specify fully buffered or line buffered, *buf* and *size* can optionally specify a buffer and its size. If the stream is buffered and *buf* is `NULL`, the standard I/O library will automatically allocate its own buffer of the appropriate size for the stream. By appropriate size, we mean the value specified by the constant `BUFSIZ`.

> Some C library implementations use the value from the `st_blksize` member of the `stat` structure (see Section 4.2) to determine the optimal standard I/O buffer size. As we will see later in this chapter, the GNU C library uses this method.

Figure 5.1 summarizes the actions of these two functions and their various options.

Function	*mode*	*buf*	Buffer and length	Type of buffering
`setbuf`		non-null	user *buf* of length `BUFSIZ`	fully buffered or line buffered
		`NULL`	(no buffer)	unbuffered
`setvbuf`	`_IOFBF`	non-null	user *buf* of length *size*	fully buffered
		`NULL`	system buffer of appropriate length	
	`_IOLBF`	non-null	user *buf* of length *size*	line buffered
		`NULL`	system buffer of appropriate length	
	`_IONBF`	(ignored)	(no buffer)	unbuffered

Figure 5.1 Summary of the `setbuf` and `setvbuf` functions

Be aware that if we allocate a standard I/O buffer as an automatic variable within a function, we have to close the stream before returning from the function. (We'll discuss this point further in Section 7.8.) Also, some implementations use part of the buffer for internal bookkeeping, so the actual number of bytes of data that can be stored in the buffer can be less than *size*. In general, we should let the system choose the buffer size and automatically allocate the buffer. When we do this, the standard I/O library automatically releases the buffer when we close the stream.

At any time, we can force a stream to be flushed.

```
#include <stdio.h>

int fflush(FILE *fp);
```

Returns: 0 if OK, `EOF` on error

The `fflush` function causes any unwritten data for the stream to be passed to the kernel. As a special case, if *fp* is `NULL`, `fflush` causes all output streams to be flushed.

5.5 Opening a Stream

The fopen, freopen, and fdopen functions open a standard I/O stream.

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int fd, const char *type);
```

All three return: file pointer if OK, NULL on error

The differences in these three functions are as follows:

1. The fopen function opens a specified file.
2. The freopen function opens a specified file on a specified stream, closing the stream first if it is already open. If the stream previously had an orientation, freopen clears it. This function is typically used to open a specified file as one of the predefined streams: standard input, standard output, or standard error.
3. The fdopen function takes an existing file descriptor, which we could obtain from the open, dup, dup2, fcntl, pipe, socket, socketpair, or accept functions, and associates a standard I/O stream with the descriptor. This function is often used with descriptors that are returned by the functions that create pipes and network communication channels. Because these special types of files cannot be opened with the standard I/O fopen function, we have to call the device-specific function to obtain a file descriptor, and then associate this descriptor with a standard I/O stream using fdopen.

Both fopen and freopen are part of ISO C; fdopen is part of POSIX.1, since ISO C doesn't deal with file descriptors.

type	Description	open(2) Flags
r or rb	open for reading	O_RDONLY
w or wb	truncate to 0 length or create for writing	O_WRONLY|O_CREAT|O_TRUNC
a or ab	append; open for writing at end of file, or create for writing	O_WRONLY|O_CREAT|O_APPEND
r+ or r+b or rb+	open for reading and writing	O_RDWR
w+ or w+b or wb+	truncate to 0 length or create for reading and writing	O_RDWR|O_CREAT|O_TRUNC
a+ or a+b or ab+	open or create for reading and writing at end of file	O_RDWR|O_CREAT|O_APPEND

Figure 5.2 The *type* argument for opening a standard I/O stream

ISO C specifies 15 values for the *type* argument, shown in Figure 5.2. Using the character b as part of the *type* allows the standard I/O system to differentiate between a

text file and a binary file. Since the UNIX kernel doesn't differentiate between these types of files, specifying the character `b` as part of the *type* has no effect.

With `fdopen`, the meanings of the *type* argument differ slightly. The descriptor has already been opened, so opening for writing does not truncate the file. (If the descriptor was created by the `open` function, for example, and the file already existed, the `O_TRUNC` flag would control whether the file was truncated. The `fdopen` function cannot simply truncate any file it opens for writing.) Also, the standard I/O append mode cannot create the file (since the file has to exist if a descriptor refers to it).

When a file is opened with a type of append, each write will take place at the then current end of file. If multiple processes open the same file with the standard I/O append mode, the data from each process will be correctly written to the file.

> Versions of `fopen` from Berkeley before 4.4BSD and the simple version shown on page 177 of Kernighan and Ritchie [1988] do not handle the append mode correctly. These versions do an `lseek` to the end of file when the stream is opened. To correctly support the append mode when multiple processes are involved, the file must be opened with the `O_APPEND` flag, which we discussed in Section 3.3. Doing an `lseek` before each write won't work either, as we discussed in Section 3.11.

When a file is opened for reading and writing (the plus sign in the *type*), two restrictions apply.

- Output cannot be directly followed by input without an intervening `fflush`, `fseek`, `fsetpos`, or `rewind`.
- Input cannot be directly followed by output without an intervening `fseek`, `fsetpos`, or `rewind`, or an input operation that encounters an end of file.

We can summarize the six ways to open a stream from Figure 5.2 in Figure 5.3.

Restriction	r	w	a	r+	w+	a+
file must already exist	•			•		
previous contents of file discarded		•			•	
stream can be read	•			•	•	•
stream can be written		•	•	•	•	•
stream can be written only at end			•			•

Figure 5.3 Six ways to open a standard I/O stream

Note that if a new file is created by specifying a *type* of either `w` or `a`, we are not able to specify the file's access permission bits, as we were able to do with the `open` function and the `creat` function in Chapter 3. POSIX.1 requires implementations to create the file with the following permissions bit set:

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH
```

Recall from Section 4.8, however, that we can restrict these permissions by adjusting our `umask` value.

By default, the stream that is opened is fully buffered, unless it refers to a terminal device, in which case it is line buffered. Once the stream is opened, but before we do

any other operation on the stream, we can change the buffering if we want to, with the `setbuf` or `setvbuf` functions from the previous section.

An open stream is closed by calling `fclose`.

```
#include <stdio.h>
int fclose(FILE *fp);
```

Returns: 0 if OK, `EOF` on error

Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded. If the standard I/O library had automatically allocated a buffer for the stream, that buffer is released.

When a process terminates normally, either by calling the `exit` function directly or by returning from the `main` function, all standard I/O streams with unwritten buffered data are flushed and all open standard I/O streams are closed.

5.6 Reading and Writing a Stream

Once we open a stream, we can choose from among three types of unformatted I/O:

1. Character-at-a-time I/O. We can read or write one character at a time, with the standard I/O functions handling all the buffering, if the stream is buffered.
2. Line-at-a-time I/O. If we want to read or write a line at a time, we use `fgets` and `fputs`. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call `fgets`. We describe these two functions in Section 5.7.
3. Direct I/O. This type of I/O is supported by the `fread` and `fwrite` functions. For each I/O operation, we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation. We describe these two functions in Section 5.9.

   The term *direct I/O*, from the ISO C standard, is known by many names: binary I/O, object-at-a-time I/O, record-oriented I/O, or structure-oriented I/O. Don't confuse this feature with the `O_DIRECT` open flag supported by FreeBSD and Linux—they are unrelated.

(We describe the formatted I/O functions, such as `printf` and `scanf`, in Section 5.11.)

Input Functions

Three functions allow us to read one character at a time.

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

All three return: next character if OK, `EOF` on end of file or error

The function `getchar` is defined to be equivalent to `getc(stdin)`. The difference between `getc` and `fgetc` is that `getc` can be implemented as a macro, whereas `fgetc` cannot be implemented as a macro. This means three things.

1. The argument to `getc` should not be an expression with side effects, because it could be evaluated more than once.
2. Since `fgetc` is guaranteed to be a function, we can take its address. This allows us to pass the address of `fgetc` as an argument to another function.
3. Calls to `fgetc` probably take longer than calls to `getc`, as it usually takes more time to call a function.

These three functions return the next character as an `unsigned char` converted to an `int`. The reason for specifying unsigned is so that the high-order bit, if set, doesn't cause the return value to be negative. The reason for requiring an integer return value is so that all possible character values can be returned, along with an indication that either an error occurred or the end of file has been encountered. The constant `EOF` in `<stdio.h>` is required to be a negative value. Its value is often –1. This representation also means that we cannot store the return value from these three functions in a character variable and later compare this value with the constant `EOF`.

Note that these functions return the same value whether an error occurs or the end of file is reached. To distinguish between the two, we must call either `ferror` or `feof`.

```
#include <stdio.h>

int ferror(FILE *fp);

int feof(FILE *fp);
```

Both return: nonzero (true) if condition is true, 0 (false) otherwise

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

In most implementations, two flags are maintained for each stream in the `FILE` object:

- An error flag
- An end-of-file flag

Both flags are cleared by calling `clearerr`.

After reading from a stream, we can push back characters by calling `ungetc`.

```
#include <stdio.h>

int ungetc(int c, FILE *fp);
```

Returns: *c* if OK, `EOF` on error

The characters that are pushed back are returned by subsequent reads on the stream in reverse order of their pushing. Be aware, however, that although ISO C allows an implementation to support any amount of pushback, an implementation is required to provide only a single character of pushback. We should not count on more than a single character.

The character that we push back does not have to be the same character that was read. We are not able to push back `EOF`. When we reach the end of file, however, we can push back a character. The next read will return that character, and the read after that will return `EOF`. This works because a successful call to `ungetc` clears the end-of-file indication for the stream.

Pushback is often used when we're reading an input stream and breaking the input into words or tokens of some form. Sometimes we need to peek at the next character to determine how to handle the current character. It's then easy to push back the character that we peeked at, for the next call to `getc` to return. If the standard I/O library didn't provide this pushback capability, we would have to store the character in a variable of our own, along with a flag telling us to use this character instead of calling `getc` the next time we need a character.

When we push characters back with `ungetc`, they are not written back to the underlying file or device. Instead, they are kept incore in the standard I/O library's buffer for the stream.

Output Functions

Output functions are available that correspond to each of the input functions we've already described.

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

int putchar(int c);
```

All three return: *c* if OK, `EOF` on error

As with the input functions, `putchar(c)` is equivalent to `putc(c, stdout)`, and `putc` can be implemented as a macro, whereas `fputc` cannot be implemented as a macro.

5.7 Line-at-a-Time I/O

Line-at-a-time input is provided by the two functions, `fgets` and `gets`.

```
#include <stdio.h>

char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);
```

Both return: `buf` if OK, `NULL` on end of file or error

Both specify the address of the buffer to read the line into. The `gets` function reads from standard input, whereas `fgets` reads from the specified stream.

With `fgets`, we have to specify the size of the buffer, n. This function reads up through and including the next newline, but no more than $n-1$ characters, into the

buffer. The buffer is terminated with a null byte. If the line, including the terminating newline, is longer than $n-1$, only a partial line is returned, but the buffer is always null terminated. Another call to `fgets` will read what follows on the line.

The `gets` function should never be used. The problem is that it doesn't allow the caller to specify the buffer size. This allows the buffer to overflow if the line is longer than the buffer, writing over whatever happens to follow the buffer in memory. For a description of how this flaw was used as part of the Internet worm of 1988, see the June 1989 issue (vol. 32, no. 6) of *Communications of the ACM*. An additional difference with `gets` is that it doesn't store the newline in the buffer, as `fgets` does.

> This difference in newline handling between the two functions goes way back in the evolution of the UNIX System. Even the Version 7 manual (1979) states "`gets` deletes a newline, `fgets` keeps it, all in the name of backward compatibility."

Even though ISO C requires an implementation to provide `gets`, you should use `fgets` instead. In fact, `gets` is marked as an obsolescent interface in SUSv4 and has been omitted from the latest version of the ISO C standard (ISO/IEC 9899:2011).

Line-at-a-time output is provided by `fputs` and `puts`.

```
#include <stdio.h>

int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);
```

Both return: non-negative value if OK, `EOF` on error

The function `fputs` writes the null-terminated string to the specified stream. The null byte at the end is not written. Note that this need not be line-at-a-time output, since the string need not contain a newline as the last non-null character. Usually, this is the case—the last non-null character is a newline—but it's not required.

The `puts` function writes the null-terminated string to the standard output, without writing the null byte. But `puts` then writes a newline character to the standard output.

The `puts` function is not unsafe, like its counterpart `gets`. Nevertheless, we'll avoid using it, to prevent having to remember whether it appends a newline. If we always use `fgets` and `fputs`, we know that we always have to deal with the newline character at the end of each line.

5.8 Standard I/O Efficiency

Using the functions from the previous section, we can get an idea of the efficiency of the standard I/O system. The program in Figure 5.4 is like the one in Figure 3.5: it simply copies standard input to standard output, using `getc` and `putc`. These two routines can be implemented as macros.

```
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 5.4 Copy standard input to standard output using `getc` and `putc`

We can make another version of this program that uses `fgetc` and `fputc`, which should be functions, not macros. (We don't show this trivial change to the source code.)

Finally, we have a version that reads and writes lines, shown in Figure 5.5.

```
#include "apue.h"

int
main(void)
{
    char    buf[MAXLINE];

    while (fgets(buf, MAXLINE, stdin) != NULL)
        if (fputs(buf, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Figure 5.5 Copy standard input to standard output using `fgets` and `fputs`

Note that we do not close the standard I/O streams explicitly in either Figure 5.4 or Figure 5.5. Instead, we know that the `exit` function will flush any unwritten data and then close all open streams. (We'll discuss this in Section 8.5.) It is interesting to compare the timing of these three programs with the timing data from Figure 3.6. We show this data when operating on the same file (492.65 MB with 15.7 million lines) in Figure 5.6.

For each of the three standard I/O versions, the user CPU time is larger than the best `read` version from Figure 3.6, because the character-at-a-time standard I/O versions have a loop that is executed 500 million times, and the loop in the line-at-a-time version is executed 15.7 million times. In the `read` version, its loop is executed

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.6	0.01	0.52	6.45	
`fgets, fputs`	2.03	0.27	6.70	143
`getc, putc`	7.33	0.24	8.37	114
`fgetc, fputc`	7.48	0.23	8.53	114
single byte time from Figure 3.6	17.54	114.03	131.82	

Figure 5.6 Timing results using standard I/O routines

only 3,942 times (for a buffer size of 131,072 bytes). The difference in clock times stems from the difference in user times and the difference in the times spent waiting for I/O to complete, as the system times are roughly comparable.

The system CPU time is less with the standard I/O library, which is a little surprising, because it is using a 4 KB buffer. The difference is likely because of timing, because any time not spent executing user or system code is spent waiting for disk reads to complete. One advantage of using the standard I/O routines is that we don't have to worry about buffering or choosing the optimal I/O size. We do have to determine the maximum line size for the version that uses `fgets`, but that's easier than trying to choose the optimal I/O size.

The final column in Figure 5.6 is the number of bytes of text space—the machine instructions generated by the C compiler—for each of the `main` functions. We can see that the version using `getc` and `putc` takes the same amount of space as the one using the `fgetc` and `fputc` functions. Usually, `getc` and `putc` are implemented as macros, but in the GNU C library implementation the macro simply expands to a function call.

The version using line-at-a-time I/O is about 20% faster than the version using character-at-a-time I/O. If the `fgets` and `fputs` functions are implemented using `getc` and `putc` (see Section 7.7 of Kernighan and Ritchie [1988], for example), then we would expect the timing to be similar to the `getc` version. Actually, we might expect the line-at-a-time version to take longer, since we would be adding the overhead of 1 billion extra function calls to the existing 31 million ones. What is happening with this example is that the line-at-a-time functions are implemented using `memccpy`(3). Often, the `memccpy` function is implemented in assembly language instead of C, for efficiency.

The last point of interest with these timing numbers is that the `fgetc` version is so much faster than the `BUFFSIZE=1` version from Figure 3.6. Both involve the same number of function calls—about 1 billion—yet the `fgetc` version is more than 2 times faster in terms of user CPU time and more than 15 times faster in terms of clock time. The difference is that the version using `read` executes 1 billion function calls, which in turn execute 1 billion system calls. With the `fgetc` version, we still execute 1 billion function calls, but this translates into only 250,000 system calls. System calls are usually much more expensive than ordinary function calls.

As a disclaimer, you should be aware that these timing results are valid only on the single system they were run on. The results depend on many implementation features that aren't the same on every UNIX system. Nevertheless, having a set of numbers such as these, and explaining why the various versions differ, helps us understand the

system better. From this section and Section 3.9, we've learned that the standard I/O library is not much slower than calling the `read` and `write` functions directly. For most nontrivial applications, the largest amount of user CPU time is taken by the application, not by the standard I/O routines.

5.9 Binary I/O

The functions from Section 5.6 operated with one character at a time, and the functions from Section 5.7 operated with one line at a time. If we're doing binary I/O, we often would like to read or write an entire structure at a time. To do this using `getc` or `putc`, we have to loop through the entire structure, one byte at a time, reading or writing each byte. We can't use the line-at-a-time functions, since `fputs` stops writing when it hits a null byte, and there might be null bytes within the structure. Similarly, `fgets` won't work correctly on input if any of the data bytes are nulls or newlines. Therefore, the following two functions are provided for binary I/O.

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```

Both return: number of objects read or written

These functions have two common uses:

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating-point array, we could write

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

   Here, we specify *size* as the size of each element of the array and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

   Here, we specify *size* as the size of structure and *nobj* as 1 (the number of objects to write).

The obvious generalization of these two cases is to read or write an array of structures. To do this, *size* would be the `sizeof` the structure, and *nobj* would be the number of elements in the array.

Both `fread` and `fwrite` return the number of objects read or written. For the read case, this number can be less than *nobj* if an error occurs or if the end of file is encountered. In this situation, `ferror` or `feof` must be called. For the write case, if the return value is less than the requested *nobj*, an error has occurred.

A fundamental problem with binary I/O is that it can be used to read only data that has been written on the same system. This was OK many years ago, when all the UNIX systems were PDP-11s, but the norm today is to have heterogeneous systems connected together with networks. It is common to want to write data on one system and process it on another. These two functions won't work, for two reasons.

1. The offset of a member within a structure can differ between compilers and systems because of different alignment requirements. Indeed, some compilers have an option allowing structures to be packed tightly, to save space with a possible runtime performance penalty, or aligned accurately, to optimize runtime access of each member. This means that even on a single system, the binary layout of a structure can differ, depending on compiler options.

2. The binary formats used to store multibyte integers and floating-point values differ among machine architectures.

We'll touch on some of these issues when we discuss sockets in Chapter 16. The real solution for exchanging binary data among different systems is to use an agreed-upon canonical format. Refer to Section 8.2 of Rago [1993] or Section 5.18 of Stevens, Fenner, & Rudoff [2004] for a description of some techniques various network protocols use to exchange binary data.

We'll return to the `fread` function in Section 8.14 when we use it to read a binary structure, the UNIX process accounting records.

5.10 Positioning a Stream

There are three ways to position a standard I/O stream:

1. The two functions `ftell` and `fseek`. They have been around since Version 7, but they assume that a file's position can be stored in a long integer.

2. The two functions `ftello` and `fseeko`. They were introduced in the Single UNIX Specification to allow for file offsets that might not fit in a long integer. They replace the long integer with the `off_t` data type.

3. The two functions `fgetpos` and `fsetpos`. They were introduced by ISO C. They use an abstract data type, `fpos_t`, that records a file's position. This data type can be made as big as necessary to record a file's position.

When porting applications to non-UNIX systems, use `fgetpos` and `fsetpos`.

```
#include <stdio.h>

long ftell(FILE *fp);
                        Returns: current file position indicator if OK, -1L on error

int fseek(FILE *fp, long offset, int whence);
                                        Returns: 0 if OK, -1 on error

void rewind(FILE *fp);
```

For a binary file, a file's position indicator is measured in bytes from the beginning of the file. The value returned by `ftell` for a binary file is this byte position. To position a binary file using `fseek`, we must specify a byte *offset* and indicate how that offset is interpreted. The values for *whence* are the same as for the `lseek` function from Section 3.6: `SEEK_SET` means from the beginning of the file, `SEEK_CUR` means from the current file position, and `SEEK_END` means from the end of file. ISO C doesn't require an implementation to support the `SEEK_END` specification for a binary file, as some systems require a binary file to be padded at the end with zeros to make the file size a multiple of some magic number. Under the UNIX System, however, `SEEK_END` is supported for binary files.

For text files, the file's current position may not be measurable as a simple byte offset. Again, this is mainly under non-UNIX systems that might store text files in a different format. To position a text file, *whence* has to be `SEEK_SET`, and only two values for *offset* are allowed: 0—meaning rewind the file to its beginning—or a value that was returned by `ftell` for that file. A stream can also be set to the beginning of the file with the `rewind` function.

The `ftello` function is the same as `ftell`, and the `fseeko` function is the same as `fseek`, except that the type of the offset is `off_t` instead of `long`.

```
#include <stdio.h>

off_t ftello(FILE *fp);
                        Returns: current file position indicator if OK, (off_t)-1 on error

int fseeko(FILE *fp, off_t offset, int whence);
                                        Returns: 0 if OK, -1 on error
```

Recall the discussion of the `off_t` data type in Section 3.6. Implementations can define the `off_t` type to be larger than 32 bits.

As we mentioned earlier, the `fgetpos` and `fsetpos` functions were introduced by the ISO C standard.

```
#include <stdio.h>

int fgetpos(FILE *restrict fp, fpos_t *restrict pos);

int fsetpos(FILE *fp, const fpos_t *pos);
                                        Both return: 0 if OK, nonzero on error
```

The `fgetpos` function stores the current value of the file's position indicator in the object pointed to by *pos*. This value can be used in a later call to `fsetpos` to reposition the stream to that location.

5.11 Formatted I/O

Formatted Output

Formatted output is handled by the five `printf` functions.

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);

int dprintf(int fd, const char *restrict format, ...);
```

All three return: number of characters output if OK, negative value if output error

```
int sprintf(char *restrict buf, const char *restrict format, ...);
```

Returns: number of characters stored in array if OK, negative value if encoding error

```
int snprintf(char *restrict buf, size_t n,
             const char *restrict format, ...);
```

Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

The `printf` function writes to the standard output, `fprintf` writes to the specified stream, `dprintf` writes to the specified file descriptor, and `sprintf` places the formatted characters in the array *buf*. The `sprintf` function automatically appends a null byte at the end of the array, but this null byte is not included in the return value.

Note that it's possible for `sprintf` to overflow the buffer pointed to by *buf*. The caller is responsible for ensuring that the buffer is large enough. Because buffer overflows can lead to program instability and even security violations, `snprintf` was introduced. With it, the size of the buffer is an explicit parameter; any characters that would have been written past the end of the buffer are discarded instead. The `snprintf` function returns the number of characters that would have been written to the buffer had it been big enough. As with `sprintf`, the return value doesn't include the terminating null byte. If `snprintf` returns a positive value less than the buffer size *n*, then the output was not truncated. If an encoding error occurs, `snprintf` returns a negative value.

Although `dprintf` doesn't deal with a file pointer, we include it with the rest of the related functions that handle formatted output. Note that using `dprintf` removes the need to call `fdopen` to convert a file descriptor into a file pointer for use with `fprintf`.

The format specification controls how the remainder of the arguments will be encoded and ultimately displayed. Each argument is encoded according to a conversion specification that starts with a percent sign (%). Except for the conversion

specifications, other characters in the format are copied unmodified. A conversion specification has four optional components, shown in square brackets below:

```
%[flags][fldwidth][precision][lenmodifier]convtype
```

The flags are summarized in Figure 5.7.

Flag	Description
'	(apostrophe) format integer with thousands grouping characters
-	left-justify the output in the field
+	always display sign of a signed conversion
(space)	prefix by a space if no sign is generated
#	convert using alternative form (include 0x prefix for hexadecimal format, for example)
0	prefix with leading zeros instead of padding with spaces

Figure 5.7 The flags component of a conversion specification

The `fldwidth` component specifies a minimum field width for the conversion. If the conversion results in fewer characters, it is padded with spaces. The field width is a non-negative decimal integer or an asterisk.

The `precision` component specifies the minimum number of digits to appear for integer conversions, the minimum number of digits to appear to the right of the decimal point for floating-point conversions, or the maximum number of bytes for string conversions. The precision is a period (.) followed by a optional non-negative decimal integer or an asterisk.

Either the field width or precision (or both) can be an asterisk. In this case, an integer argument specifies the value to be used. The argument appears directly before the argument to be converted.

The `lenmodifier` component specifies the size of the argument. Possible values are summarized in Figure 5.8.

Length modifier	Description
hh	signed or unsigned `char`
h	signed or unsigned `short`
l	signed or unsigned `long` or wide character
ll	signed or unsigned `long long`
j	`intmax_t` or `uintmax_t`
z	`size_t`
t	`ptrdiff_t`
L	`long double`

Figure 5.8 The length modifier component of a conversion specification

The `convtype` component is not optional. It controls how the argument is interpreted. The various conversion types are summarized in Figure 5.9.

With the normal conversion specification, conversions are applied to the arguments in the order they appear after the *format* argument. An alternative conversion specification syntax allows the arguments to be named explicitly with the sequence *%n$*

Conversion type	Description
d,i	signed decimal
o	unsigned octal
u	unsigned decimal
x,X	unsigned hexadecimal
f,F	double floating-point number
e,E	double floating-point number in exponential format
g,G	interpreted as f, F, e, or E, depending on value converted
a,A	double floating-point number in hexadecimal exponential format
c	character (with l length modifier, wide character)
s	string (with l length modifier, wide character string)
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters written so far
%	a % character
C	wide character (XSI option, equivalent to lc)
S	wide character string (XSI option, equivalent to ls)

Figure 5.9 The conversion type component of a conversion specification

representing the *n*th argument. Note, however, that the two syntaxes can't be mixed in the same format specification. With the alternative syntax, arguments are numbered starting at one. If either the field width or precision is to be supplied by an argument, the asterisk syntax is modified to **m$*, where *m* specifies the position of the argument supplying the value.

The following five variants of the `printf` family are similar to the previous five, but the variable argument list (`...`) is replaced with *arg*.

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

int vprintf(const char *restrict format, va_list arg);

int vfprintf(FILE *restrict fp, const char *restrict format,
             va_list arg);

int vdprintf(int fd, const char *restrict format, va_list arg);
```

All three return: number of characters output if OK, negative value if output error

```
int vsprintf(char *restrict buf, const char *restrict format,
             va_list arg);
```

Returns: number of characters stored in array if OK, negative value if encoding error

```
int vsnprintf(char *restrict buf, size_t n,
              const char *restrict format, va_list arg);
```

Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

We use the `vsnprintf` function in the error routines in Appendix B.

Refer to Section 7.3 of Kernighan and Ritchie [1988] for additional details on handling variable-length argument lists with ISO Standard C. Be aware that the variable-length argument list routines provided with ISO C—the `<stdarg.h>` header and its associated routines—differ from the `<varargs.h>` routines that were provided with older UNIX systems.

Formatted Input

Formatted input is handled by the three `scanf` functions.

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format, ...);
```

All three return: number of input items assigned, `EOF` if input error or end of file before any conversion

The `scanf` family is used to parse an input string and convert character sequences into variables of specified types. The arguments following the format contain the addresses of the variables to initialize with the results of the conversions.

The format specification controls how the arguments are converted for assignment. The percent sign (%) indicates the beginning of a conversion specification. Except for the conversion specifications and white space, other characters in the format have to match the input. If a character doesn't match, processing stops, leaving the remainder of the input unread.

There are three optional components to a conversion specification, shown in square brackets below:

```
%[*][fldwidth][m][lenmodifier]convtype
```

The optional leading asterisk is used to suppress conversion. Input is converted as specified by the rest of the conversion specification, but the result is not stored in an argument.

The `fldwidth` component specifies the maximum field width in characters. The `lenmodifier` component specifies the size of the argument to be initialized with the result of the conversion. The same length modifiers supported by the `printf` family of functions are supported by the `scanf` family of functions (see Figure 5.8 for a list of the length modifiers).

The `convtype` field is similar to the conversion type field used by the `printf` family, but there are some differences. One difference is that results that are stored in unsigned types can optionally be signed on input. For example, –1 will scan as 4294967295 into an unsigned integer. Figure 5.10 summarizes the conversion types supported by the `scanf` family of functions.

The optional `m` character between the field width and the length modifier is called the *assignment-allocation character*. It can be used with the `%c`, `%s`, and `%[` conversion

Conversion type	Description
d	signed decimal, base 10
i	signed decimal, base determined by format of input
o	unsigned octal (input optionally signed)
u	unsigned decimal, base 10 (input optionally signed)
x,X	unsigned hexadecimal (input optionally signed)
a,A,e,E,f,F,g,G	floating-point number
c	character (with l length modifier, wide character)
s	string (with l length modifier, wide character string)
[	matches a sequence of listed characters, ending with]
[^	matches all characters except the ones listed, ending with]
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters read so far
%	a % character
C	wide character (XSI option, equivalent to lc)
S	wide character string (XSI option, equivalent to ls)

Figure 5.10 The conversion type component of a conversion specification

specifiers to force a memory buffer to be allocated to hold the converted string. In this case, the corresponding argument should be the address of a pointer to which the address of the allocated buffer will be copied. If the call succeeds, the caller is responsible for freeing the buffer by calling the `free` function when the buffer is no longer needed.

The `scanf` family of functions also supports the alternative conversion specification syntax allowing the arguments to be named explicitly: the sequence *%n$* represents the *n*th argument. With the `printf` family of functions, the same numbered argument can be referenced in the format string more than once. In this case, however, the Single UNIX Specification states that the behavior is undefined with the `scanf` family of functions.

Like the `printf` family, the `scanf` family supports functions that use variable argument lists as specified by `<stdarg.h>`.

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

int vscanf(const char *restrict format, va_list arg);

int vfscanf(FILE *restrict fp, const char *restrict format,
            va_list arg);

int vsscanf(const char *restrict buf, const char *restrict format,
            va_list arg);
```

All three return: number of input items assigned, `EOF` if input error or end of file before any conversion

Refer to your UNIX system manual for additional details on the `scanf` family of functions.

5.12 Implementation Details

As we've mentioned, under the UNIX System, the standard I/O library ends up calling the I/O routines that we described in Chapter 3. Each standard I/O stream has an associated file descriptor, and we can obtain the descriptor for a stream by calling `fileno`.

Note that `fileno` is not part of the ISO C standard, but rather an extension supported by POSIX.1.

```
#include <stdio.h>

int fileno(FILE *fp);
```

Returns: the file descriptor associated with the stream

We need this function if we want to call the `dup` or `fcntl` functions, for example.

To look at the implementation of the standard I/O library on your system, start with the header `<stdio.h>`. This will show how the `FILE` object is defined, the definitions of the per-stream flags, and any standard I/O routines, such as `getc`, that are defined as macros. Section 8.5 of Kernighan and Ritchie [1988] has a sample implementation that shows the flavor of many implementations on UNIX systems. Chapter 12 of Plauger [1992] provides the complete source code for an implementation of the standard I/O library. The implementation of the GNU standard I/O library is also publicly available.

Example

The program in Figure 5.11 prints the buffering for the three standard streams and for a stream that is associated with a regular file.

```
#include "apue.h"

void    pr_stdio(const char *, FILE *);
int     is_unbuffered(FILE *);
int     is_linebuffered(FILE *);
int     buffer_size(FILE *);

int
main(void)
{
    FILE    *fp;

    fputs("enter any character\n", stdout);
    if (getchar() == EOF)
        err_sys("getchar error");
    fputs("one line to standard error\n", stderr);

    pr_stdio("stdin",  stdin);
    pr_stdio("stdout", stdout);
    pr_stdio("stderr", stderr);
```

```
    if ((fp = fopen("/etc/passwd", "r")) == NULL)
        err_sys("fopen error");
    if (getc(fp) == EOF)
        err_sys("getc error");
    pr_stdio("/etc/passwd", fp);
    exit(0);
}

void
pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);
    if (is_unbuffered(fp))
        printf("unbuffered");
    else if (is_linebuffered(fp))
        printf("line buffered");
    else /* if neither of above */
        printf("fully buffered");
    printf(", buffer size = %d\n", buffer_size(fp));
}

/*
 * The following is nonportable.
 */

#if defined(_IO_UNBUFFERED)

int
is_unbuffered(FILE *fp)
{
    return(fp->_flags & _IO_UNBUFFERED);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flags & _IO_LINE_BUF);
}

int
buffer_size(FILE *fp)
{
    return(fp->_IO_buf_end - fp->_IO_buf_base);
}

#elif defined(__SNBF)

int
is_unbuffered(FILE *fp)
{
    return(fp->_flags & __SNBF);
}
```

```
int
is_linebuffered(FILE *fp)
{
    return(fp->_flags & __SLBF);
}

int
buffer_size(FILE *fp)
{
    return(fp->_bf._size);
}

#elif defined(_IONBF)

#ifdef _LP64
#define _flag __pad[4]
#define _ptr __pad[1]
#define _base __pad[2]
#endif

int
is_unbuffered(FILE *fp)
{
    return(fp->_flag & _IONBF);
}

int
is_linebuffered(FILE *fp)
{
    return(fp->_flag & _IOLBF);
}

int
buffer_size(FILE *fp)
{
#ifdef _LP64
    return(fp->_base - fp->_ptr);
#else
    return(BUFSIZ); /* just a guess */
#endif
}

#else

#error unknown stdio implementation!

#endif
```

Figure 5.11 Print buffering for various standard I/O streams

Note that we perform I/O on each stream before printing its buffering status, since the first I/O operation usually causes the buffers to be allocated for a stream. The structure members and the constants used in this example are defined by the implementations of

the standard I/O library used on the four platforms described in this book. Be aware that implementations of the standard I/O library vary, and programs like this example are nonportable, since they embed knowledge specific to particular implementations.

If we run the program in Figure 5.11 twice, once with the three standard streams connected to the terminal and once with the three standard streams redirected to files, we get the following result:

```
$ ./a.out                                   stdin, stdout, and stderr connected to terminal
enter any character
                                            we type a newline
one line to standard error
stream = stdin, line buffered, buffer size = 1024
stream = stdout, line buffered, buffer size = 1024
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
$ ./a.out < /etc/group > std.out 2> std.err
                                            run it again with all three streams redirected
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
```

We can see that the default for this system is to have standard input and standard output line buffered when they're connected to a terminal. The line buffer is 1,024 bytes. Note that this doesn't restrict us to 1,024-byte input and output lines; that's just the size of the buffer. Writing a 2,048-byte line to standard output will require two `write` system calls. When we redirect these two streams to regular files, they become fully buffered, with buffer sizes equal to the preferred I/O size—the `st_blksize` value from the `stat` structure—for the file system. We also see that the standard error is always unbuffered, as it should be, and that a regular file defaults to fully buffered. □

5.13 Temporary Files

The ISO C standard defines two functions that are provided by the standard I/O library to assist in creating temporary files.

```
#include <stdio.h>

char *tmpnam(char *ptr);
                                        Returns: pointer to unique pathname

FILE *tmpfile(void);
                                        Returns: file pointer if OK, NULL on error
```

The tmpnam function generates a string that is a valid pathname and that does not match the name of any existing file. This function generates a different pathname each time it is called, up to TMP_MAX times. TMP_MAX is defined in <stdio.h>.

> Although ISO C defines TMP_MAX, the C standard requires only that its value be at least 25. The Single UNIX Specification, however, requires that XSI-conforming systems support a value of at least 10,000. This minimum value allows an implementation to use four digits (0000–9999), although most implementations on UNIX systems use alphanumeric characters.
>
> The tmpnam function is marked obsolescent in SUSv4, but the ISO C standard continues to support it.

If *ptr* is NULL, the generated pathname is stored in a static area, and a pointer to this area is returned as the value of the function. Subsequent calls to tmpnam can overwrite this static area. (Thus, if we call this function more than once and we want to save the pathname, we have to save a copy of the pathname, not a copy of the pointer.) If *ptr* is not NULL, it is assumed that it points to an array of at least L_tmpnam characters. (The constant L_tmpnam is defined in <stdio.h>.) The generated pathname is stored in this array, and *ptr* is returned as the value of the function.

The tmpfile function creates a temporary binary file (type wb+) that is automatically removed when it is closed or on program termination. Under the UNIX System, it makes no difference that this file is a binary file.

Example

The program in Figure 5.12 demonstrates these two functions.

```
#include "apue.h"

int
main(void)
{
    char    name[L_tmpnam], line[MAXLINE];
    FILE    *fp;

    printf("%s\n", tmpnam(NULL));       /* first temp name */

    tmpnam(name);                       /* second temp name */
    printf("%s\n", name);

    if ((fp = tmpfile()) == NULL)       /* create temp file */
        err_sys("tmpfile error");
    fputs("one line of output\n", fp);  /* write to temp file */
    rewind(fp);                         /* then read it back */
    if (fgets(line, sizeof(line), fp) == NULL)
        err_sys("fgets error");
    fputs(line, stdout);                /* print the line we wrote */

    exit(0);
}
```

Figure 5.12 Demonstrate tmpnam and tmpfile functions

If we execute the program in Figure 5.12, we get

```
$ ./a.out
/tmp/fileT0Hsu6
/tmp/filekmAsYQ
one line of output
```

□

The standard technique often used by the `tmpfile` function is to create a unique pathname by calling `tmpnam`, then create the file, and immediately `unlink` it. Recall from Section 4.15 that unlinking a file does not delete its contents until the file is closed. This way, when the file is closed, either explicitly or on program termination, the contents of the file are deleted.

The Single UNIX Specification defines two additional functions as part of the XSI option for dealing with temporary files: `mkdtemp` and `mkstemp`.

> Older versions of the Single UNIX Specification defined the `tempnam` function as a way to create a temporary file in a caller-specified location. It is marked obsolescent in SUSv4.

```
#include <stdlib.h>

char *mkdtemp(char *template);
                                Returns: pointer to directory name if OK, NULL on error

int mkstemp(char *template);
                                Returns: file descriptor if OK, –1 on error
```

The `mkdtemp` function creates a directory with a unique name, and the `mkstemp` function creates a regular file with a unique name. The name is selected using the *template* string. This string is a pathname whose last six characters are set to `XXXXXX`. The function replaces these placeholders with different characters to create a unique pathname. If successful, these functions modify the *template* string to reflect the name of the temporary file.

The directory created by `mkdtemp` is created with the following access permission bits set: `S_IRUSR | S_IWUSR | S_IXUSR`. Note that the file mode creation mask of the calling process can restrict these permissions further. If directory creation is successful, `mkdtemp` returns the name of the new directory.

The `mkstemp` function creates a regular file with a unique name and opens it. The file descriptor returned by `mkstemp` is open for reading and writing. The file created by `mkstemp` is created with access permissions `S_IRUSR | S_IWUSR`.

Unlike `tmpfile`, the temporary file created by `mkstemp` is not removed automatically for us. If we want to remove it from the file system namespace, we need to unlink it ourselves.

Use of `tmpnam` and `tempnam` does have at least one drawback: a window exists between the time that the unique pathname is returned and the time that an application creates a file with that name. During this timing window, another process can create a file of the same name. The `tmpfile` and `mkstemp` functions should be used instead, as they don't suffer from this problem.

Example

The program in Figure 5.13 shows how to use (and how not to use) the mkstemp function.

```
#include "apue.h"
#include <errno.h>

void make_temp(char *template);

int
main()
{
    char    good_template[] = "/tmp/dirXXXXXX"; /* right way */
    char    *bad_template = "/tmp/dirXXXXXX";   /* wrong way*/

    printf("trying to create first temp file...\n");
    make_temp(good_template);
    printf("trying to create second temp file...\n");
    make_temp(bad_template);
    exit(0);
}

void
make_temp(char *template)
{
    int         fd;
    struct stat sbuf;

    if ((fd = mkstemp(template)) < 0)
        err_sys("can't create temp file");
    printf("temp name = %s\n", template);
    close(fd);
    if (stat(template, &sbuf) < 0) {
        if (errno == ENOENT)
            printf("file doesn't exist\n");
        else
            err_sys("stat failed");
    } else {
        printf("file exists\n");
        unlink(template);
    }
}
```

Figure 5.13 Demonstrate mkstemp function

If we execute the program in Figure 5.13, we get

```
$ ./a.out
trying to create first temp file...
temp name = /tmp/dirUmBT7h
file exists
```

```
trying to create second temp file...
Segmentation fault
```

The difference in behavior comes from the way the two template strings are declared. For the first template, the name is allocated on the stack, because we use an array variable. For the second name, however, we use a pointer. In this case, only the memory for the pointer itself resides on the stack; the compiler arranges for the string to be stored in the read-only segment of the executable. When the `mkstemp` function tries to modify the string, a segmentation fault occurs. □

5.14 Memory Streams

As we've seen, the standard I/O library buffers data in memory, so operations such as character-at-a-time I/O and line-at-a-time I/O are more efficient. We've also seen that we can provide our own buffer for the library to use by calling `setbuf` or `setvbuf`. In Version 4, the Single UNIX Specification added support for *memory streams*. These are standard I/O streams for which there are no underlying files, although they are still accessed with `FILE` pointers. All I/O is done by transferring bytes to and from buffers in main memory. As we shall see, even though these streams look like file streams, several features make them more suited for manipulating character strings.

Three functions are available to create memory streams. The first is `fmemopen`.

```
#include <stdio.h>

FILE *fmemopen(void *restrict buf, size_t size,
               const char *restrict type);
```

Returns: stream pointer if OK, `NULL` on error

The `fmemopen` function allows the caller to provide a buffer to be used for the memory stream: the *buf* argument points to the beginning of the buffer and the *size* argument specifies the size of the buffer in bytes. If the *buf* argument is null, then the `fmemopen` function allocates a buffer of *size* bytes. In this case, the buffer will be freed when the stream is closed.

The *type* argument controls how the stream can be used. The possible values for *type* are summarized in Figure 5.14.

type	Description
`r` or `rb`	open for reading
`w` or `wb`	open for writing
`a` or `ab`	append; open for writing at first null byte
`r+` or `r+b` or `rb+`	open for reading and writing
`w+` or `w+b` or `wb+`	truncate to 0 length and open for reading and writing
`a+` or `a+b` or `ab+`	append; open for reading and writing at first null byte

Figure 5.14 The *type* argument for opening a memory stream

Note that these values correspond to the ones for file-based standard I/O streams, but there are some subtle differences. First, whenever a memory stream is opened for

append, the current file position is set to the first null byte in the buffer. If the buffer contains no null bytes, then the current position is set to one byte past the end of the buffer. When a stream is not opened for append, the current position is set to the beginning of the buffer. Because the append mode determines the end of the data by the first null byte, memory streams aren't well suited for storing binary data (which might contain null bytes before the end of the data).

Second, if the *buf* argument is a null pointer, it makes no sense to open the stream for only reading or only writing. Because the buffer is allocated by `fmemopen` in this case, there is no way to find the buffer's address, so to open the stream only for writing means we could never read what we've written. Similarly, to open the stream only for reading means we can only read the contents of a buffer into which we can never write.

Third, a null byte is written at the current position in the stream whenever we increase the amount of data in the stream's buffer and call `fclose`, `fflush`, `fseek`, `fseeko`, or `fsetpos`.

Example

It's instructive to look at how writes to a memory stream operate on a buffer we provide. Figure 5.15 shows a sample program that seeds the buffer with a known pattern to see how writes to the stream behave.

```
#include "apue.h"

#define BSZ 48

int
main()
{
    FILE *fp;
    char buf[BSZ];

    memset(buf, 'a', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    if ((fp = fmemopen(buf, BSZ, "w+")) == NULL)
        err_sys("fmemopen failed");
    printf("initial buffer contents: %s\n", buf);
    fprintf(fp, "hello, world");
    printf("before flush: %s\n", buf);
    fflush(fp);
    printf("after fflush: %s\n", buf);
    printf("len of string in buf = %ld\n", (long)strlen(buf));

    memset(buf, 'b', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "hello, world");
    fseek(fp, 0, SEEK_SET);
    printf("after  fseek: %s\n", buf);
    printf("len of string in buf = %ld\n", (long)strlen(buf));
```

```
    memset(buf, 'c', BSZ-2);
    buf[BSZ-2] = '\0';
    buf[BSZ-1] = 'X';
    fprintf(fp, "hello, world");
    fclose(fp);
    printf("after fclose: %s\n", buf);
    printf("len of string in buf = %ld\n", (long)strlen(buf));

    return(0);
}
```

Figure 5.15 Investigate memory stream write behavior

When we run the program on Linux, we get the following:

```
$ ./a.out
                                    overwrite the buffer with a's
initial buffer contents:            fmemopen places a null byte at beginning of buffer
before flush:                       buffer is unchanged until stream is flushed
after fflush: hello, world
len of string in buf = 12           null byte added to end of string
                                    now overwrite the buffer with b's
after  fseek: bbbbbbbbbbbbhello, world        fseek causes flush
len of string in buf = 24           null byte appended again
                                    now overwrite the buffer with c's
after fclose: hello, worldccccccccccccccccccccccccccccccc
len of string in buf = 46           no null byte appended
```

This example shows the policy for flushing memory streams and appending null bytes. A null byte is appended automatically whenever we write to a memory stream and advance the stream's notion of the size of the stream's contents (as opposed to the size of the buffer, which is fixed). The size of the stream's contents is determined by how much we write to it.

> Of the four platforms covered in this book, only Linux 3.2.0 provides support for memory streams. This is a case of the implementations not having caught up yet with the latest standards, and will change with time. □

The other two functions that can be used to create a memory stream are `open_memstream` and `open_wmemstream`.

```
#include <stdio.h>

FILE *open_memstream(char **bufp, size_t *sizep);

#include <wchar.h>

FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

Both return: stream pointer if OK, NULL on error

The `open_memstream` function creates a stream that is byte oriented, and the `open_wmemstream` function creates a stream that is wide oriented (recall the discussion of multibyte characters in Section 5.2). These two functions differ from `fmemopen` in several ways:

- The stream created is only open for writing.
- We can't specify our own buffer, but we can get access to the buffer's address and size through the *bufp* and *sizep* arguments, respectively.
- We need to free the buffer ourselves after closing the stream.
- The buffer will grow as we add bytes to the stream.

We must follow some rules, however, regarding the use of the buffer address and its length. First, the buffer address and length are only valid after a call to `fclose` or `fflush`. Second, these values are only valid until the next write to the stream or a call to `fclose`. Because the buffer can grow, it may need to be reallocated. If this happens, then we will find that the value of the buffer's memory address will change the next time we call `fclose` or `fflush`.

Memory streams are well suited for creating strings, because they prevent buffer overflows. They can also provide a performance boost for functions that take standard I/O stream arguments used for temporary files, because memory streams access only main memory instead of a file stored on disk.

5.15 Alternatives to Standard I/O

The standard I/O library is not perfect. Korn and Vo [1991] list numerous defects—some in the basic design, but most in the various implementations.

One inefficiency inherent in the standard I/O library is the amount of data copying that takes place. When we use the line-at-a-time functions, `fgets` and `fputs`, the data is usually copied twice: once between the kernel and the standard I/O buffer (when the corresponding `read` or `write` is issued) and again between the standard I/O buffer and our line buffer. The Fast I/O library [`fio`(3) in AT&T 1990a] gets around this by having the function that reads a line return a pointer to the line instead of copying the line into another buffer. Hume [1988] reports a threefold increase in the speed of a version of the `grep`(1) utility simply by making this change.

Korn and Vo [1991] describe another replacement for the standard I/O library: *sfio*. This package is similar in speed to the *fio* library and normally faster than the standard I/O library. The *sfio* package also provides some new features that aren't found in most other packages: I/O streams generalized to represent both files and regions of memory, processing modules that can be written and stacked on an I/O stream to change the operation of a stream, and better exception handling.

Krieger, Stumm, and Unrau [1992] describe another alternative that uses mapped files—the `mmap` function that we describe in Section 14.8. This new package is called ASI, the Alloc Stream Interface. The programming interface resembles the UNIX System memory allocation functions (`malloc`, `realloc`, and `free`, described in

Section 7.8). As with the *sfio* package, ASI attempts to minimize the amount of data copying by using pointers.

Several implementations of the standard I/O library are available in C libraries that were designed for systems with small memory footprints, such as embedded systems. These implementations emphasize modest memory requirements over portability, speed, or functionality. Two such implementations are the uClibc C library (see `http://www.uclibc.org` for more information) and the Newlib C library (`http://sources.redhat.com/newlib`).

5.16 Summary

The standard I/O library is used by most UNIX applications. In this chapter, we looked at many of the functions provided by this library, as well as at some implementation details and efficiency considerations. Be aware of the buffering that takes place with this library, as this is the area that generates the most problems and confusion.

Exercises

5.1 Implement `setbuf` using `setvbuf`.

5.2 Type in the program that copies a file using line-at-a-time I/O (`fgets` and `fputs`) from Figure 5.5, but use a `MAXLINE` of 4. What happens if you copy lines that exceed this length? Explain what is happening.

5.3 What does a return value of 0 from `printf` mean?

5.4 The following code works correctly on some machines, but not on others. What could be the problem?

```
#include    <stdio.h>

int
main(void)
{
    char    c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

5.5 How would you use the `fsync` function (Section 3.13) with a standard I/O stream?

5.6 In the programs in Figures 1.7 and 1.10, the prompt that is printed does not contain a newline, and we don't call `fflush`. What causes the prompt to be output?

5.7 BSD-based systems provide a function called `funopen` that allows us to intercept read, write, seek, and close calls on a stream. Use this function to implement `fmemopen` for FreeBSD and Mac OS X.

This page intentionally left blank