Advanced Programming in the UNIX Environment — *Standard I/O Library*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

Table of Contents I

Streams and FILE Objects
STDIN, STDOUT, and STDERR
Buffering
Opening a Stream
Reading and Writing a Stream
- Character-at-a-time I/O Input Functions
- Character-at-a-time I/O Output Functions
- Line-at-a-Time I/O
- Standard I/O Efficiency
- Binary I/O
- Positioning a Stream
- Formatted I/O
  - Formatted Output
  - Formatted Input

Implementation Detail

Table of Contents II

Temporary Files

Memory Streams

Alternatives to Standard I/O

Summary

Streams and FILE Objects I

- In Chapter 3 all the I/O routines centered around file descriptors.
- With the standard I/O library the discussion centers around **streams**. When we open or create a file with standard I/O library we say that we have associated a stream with the file.
- When we open a stream, the standard I/O function `fopen` returns a pointer to a `FILE` object.
- A `FILE` object is normally a structure that contains all the information required by the standard I/O library to manage the stream:
  - the fd used for actual I/O,
  - a pointer to a buffer for the stream,
  - the size of the buffer,

Streams and FILE Objects II

    - a count of the number of characters currently in the buffer,
    - an error flag,
    - and so on.

- Throughout this text, we'll refer to a pointer to a `FILE` object, the type `FILE *`, as a **file pointer**.
- 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.
- The `freopen` function (discussed shortly) will clear a stream's orientation; the `fwide` function can be used to set a stream's orientation.

Streams and FILE Objects III

```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *fp, int mode);
```

- The `fwide` function performs different tasks, depending on the value of the *mode* argument.
  - If the *mode* is negative, `fwide` will try to make the specified stream **byte oriented**.
  - If the *mode* is positive, `fwide` will try to make the specified stream **wide oriented**.
  - If the *mode* is zero, `fwide` will not try to set the orientation, but will return a value identifying the stream's orientation.
- Note that `fwide` will not change the orientation of a stream that is already oriented.

Streams and FILE Objects IV

- Also note that there is no error return. We should 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.

stdin, stdout, and stderr

- Three streams are predefined and automatically available to a process: standard input, standard output, and standard error.
- These three standard I/O streams are referenced through the predefined file pointers stdin, stdout, and stderr. They are defined in stdio.h.

Buffering I

- The goal of the buffering provided by the standard I/O library is to use the minimum number of `read` and `write` system calls.
- There are three types of buffering provided:
  - Fully buffered. For this case actual I/O takes place when the standard I/O buffer is filled.
  - Line buffered. In this case the standard I/O library perform I/O when a new-line character is encountered on input or output.
  - Unbuffered. The standard I/O library does not buffer the characters.
- `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.

Buffering II

- 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.
- If we don't like these default settings for any given stream, we can change the buffering by calling either of the following two functions:

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
```

- At any time we can force a stream to be flushed:

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

Buffering III

- If *fp* is `NULL`, this function causes all output streams to be flushed.

Opening a Stream I

- The following three 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);
```

- `fopen` opens a specified file
- `freopen` 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.

Opening a Stream II

- `fdopen` takes an existing file descriptor as its parameter, and associated a standard I/O stream with the descriptor.
- Parameter *type*:

`r, rb`	open for reading
`w, wb`	truncate to 0 or create for writing
`a, ab`	append or create for writing
`r+, r+b, rb+`	open for r/w
`w+, w+b, wb+`	truncate to 0 or create for r/w
`a+, a+b, ab+`	open or create for r/a

- Using the character `b` as part of the type allows the functions to differentiate between *text* file and *binary* file.
- An open stream is closed by calling `fclose`:

Opening a Stream III

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

- Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded.

Reading and Writing a Stream

- Once we open a stream we can choose among three different types of unformatted I/O:
  - Character-at-a-time I/O
  - Line-at-a-time I/O
  - Direct I/O

Character-at-a-time I/O Input Functions I

- Three functions allow us to read one character at a time:

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

- The function getchar is defined to be equivalent to getc(stdin).
- The difference between the first two functions is that getc can be implemented as a macro when fgetc must be implemented as a function.
- These three functions return the next character as an unsigned char converted to an int.

Character-at-a-time I/O Input Functions II

- These three functions return the same value `EOF` whether an error occurs or the end of file is reached. So we need call either `ferror` or `feof` to distinguish these two cases.

```
#include <stdio.h>
int ferror(FILE *fp);
int feof(FILE *fp);
void clearerr(FILE *fp);
```

- After reading from a stream we can push back characters by calling `ungetc`:

```
#include <stdio.h>
int ungetc (int c, FILE *fp);
```

Character-at-a-time I/O Input Functions III

- 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.
- A successful call to `ungetc` clears the end-of-file indication for the stream.

Character-at-a-time I/O Output Functions

- We will find an output function that corresponds to each of the input functions above.

```c
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

- Like the input functions, `putchar(c)` is equivalent to `putc(c, stdout)`, and `putc` can be implemented as a macro while `fputc` can NOT be implemented as a macro.

Line-at-a-Time I/O I

- Both of function bellow specify the address of the buffer to read the line into. `gets` reads from `stdin` while `fgets` reads from the specified stream.

```c
#include <stdio.h>
char *fgets(char *restrict buf, int n,
            FILE *restrict fp);
char *gets(char *buf);
```

- With `fgets` we have to specify the size of the buffer *n* while `gets` need not.
- `fgets` reads $n - 1$ characters into the buffer and terminated with a null byte, next read will follows the line. The last read of a line will include the newline character.

Line-at-a-Time I/O II

- Another difference with gets is that it does not store the newline in the buffer, as fgets does.
- Even though ISO C requires an implementation to provide gets, it should never be used.

```c
#include <stdio.h>
int fputs(const char *restrict str,
          FILE *restrict fp);
int puts(const char *str);
```

- fputs will chop the last null byte before writes to stdout, no mater what the characters are.
- puts will chop the last null byte before writes to stdout, then writes a newline character.

Standard I/O Efficiency

- Example (Figure 5.4, `stdio/getcputc.c`);
- Example (Figure 5.5, `stdio/fgetsfputs.c`);
- Example (Figure 5.6).

Binary I/O

- The following two functions are provided for binary I/O:

```c
#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);
```

- There are two common usage for these functions: r/w a binary array, or r/w a structure.

Positioning a Stream I

- There are three ways to position a standard I/O stream:
  - ftell and fseek;
  - ftello and fseeko;
  - fgetpos and fsetpos.

```
#include <stdio.h>
long ftell(FILE *fp);
int fseek(FILE *fp, long offset, int whence);
void rewind(FILE *fp);
off_t ftello(FILE *fp);
int fseeko(FILE *fp, off_t offset, int whence);
int fgetpos(FILE *restrict fp,
            fpos_t *restrict pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

Positioning a Stream II

- Implementations can define the `off_t` type to be larger than 32 bits.
- `fgetpos` 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 re-position the stream to that location.
- When porting applications to non-UNIX systems, use `fgetpos` and `fsetpos`.

Formatted Output Functions I

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

- printf writes to the stdout, fprintf writes to the specified stream, and sprintf places the formatted characters in the array *buf*.
- sprintf automatically places a null byte at the end of the array, but this null byte is not included in the return value.

Formatted Output Functions II

- The following three variants of the `printf` family are similar to the previous three:

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

Formatted Output Functions III

- 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.

Formatted Input

- Formatted input is handled by the three scanf functions:

```c
#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, ... );
```

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

```c
#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);
```

Implementation Detail I

- Under UNIX, 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.

```
#include <stdio.h>
int fileno(FILE *fp);
#define NULL 0
#define EOF (-1)
#define BUFSIZ 1024
#define OPEN_MAX 20 /* max #files open at once */
typedef struct _iobuf {
  int cnt;     /* characters left */
  char *ptr;   /* next character position */
  char *base;  /* location of buffer */
  int flag;    /* mode of file access */
  int fd;      /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];
```

Implementation Detail II

```c
#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])
enum _flags {
  _READ = 01,  /* file open for reading */
  _WRITE = 02, /* file open for writing */
  _UNBUF = 04, /* file is unbuffered */
  _EOF = 010,  /* EOF has occurred on this file */
  _ERR = 020   /* error occurred on this file */
};
int _fillbuf(FILE *);
int _flushbuf(int, FILE *);
#define feof(p) ((p)->flag & _EOF) != 0)
#define ferror(p) ((p)->flag & _ERR) != 0)
#define fileno(p) ((p)->fd)
#define getc(p) (--(p)->cnt >= 0 ? \
    (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 ? \
    *(p)->ptr++ = (x) : _flushbuf((x),p))
#define getchar() getc(stdin)
```

Implementation Detail III

```
#define putchar(x) putc((x), stdout)
```

- Example (Figure 5.11, `stdio/buf.c`).

Temporary Files I

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

```c
#include <stdio.h>
char *tmpnam(char *ptr);
FILE tmpfile(void);
```

- `tmpnam` generates a string that is a valid pathname and that is not the same as any existing file. It generates a different pathname each time it is called, up to `TMP_MAX` times, which is defined in `stdio.h`.
- `tmpfile` creates a temporary binary file (type `wb+`) that is automatically removed when it is closed or on program termination.

Temporary Files II

- Example (Figure 5.12, stdio/tempfiles.c).
- 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.
- tmpnam is dangerous under Linux OS, use mkstemp or tmpfile instead.

```c
#include <stdlib.h>
char *mkdtemp(char *template);
int mkstemp(char *template);
```

- The mkdtemp function creates a directory and returns the pointer to the directory name if OK, NULL on error

Temporary Files III

- The `mkstemp` function creates a regular file with a unique name and returns the *fd* of the temporary file or -1 on error.
- The last six characters of template must be `XXXXXX` and these are replaced with a string that makes the filename unique in both functions.
- The directory is created with permissions 0700.
- The file is created with permissions 0600 and opened with the `O_EXCL` flag, guaranteeing that when `mkstemp` returns successfully we are the only user.
- Example (Figure 5.13, `stdio/mkstemp.c`).

Memory Streams I

- The SUS v4 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);
```

Memory Streams II

- 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* 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. They are similar to *type* of function fopen.
- Example (Figure 5.15, stdio/memstr.c).
- The other two functions that can be used to create a memory stream are open_memstream and open_wmemstream.

Memory Streams III

```c
#include <stdio.h>
FILE *open_memstream(char **bufp, size_t *sizep);
#include <wchar.h>
FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

- The `open_memstream` function creates a stream that is byte oriented, and the `open_wmemstream` creates a stream that is wide oriented. 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.

Memory Streams IV

- 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.

Alternatives to Standard I/O I

- The standard I/O library is not perfect.
- One inefficiency inherent in the standard I/O library is the amount of data copying that takes place. The fast I/O library (**fio**) 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.

Alternatives to Standard I/O II

- Another replacement for the standard I/O library: **sfio** 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.
- Another alternative that uses mapped files 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.

Alternatives to Standard I/O III

- 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]http://www.uclibc.org and the [Newlib C library]http://sources.redhat.com/newlib.

Summary

- The standard I/O library is used by most UNIX applications.
- Be aware of the buffering that takes place with this library, as this is the area that generates the most **problems** and **confusion**.

The End

The End of Chapter 5.