Standard I/O Library

Contents

- Standard I/O Library
  - Specified by the ISO C standard
  - Handles buffer allocation
    → Makes it easy to use
    → So as not to recognize what's going on
  - Majorly written by Dennis Ritchie (1975)

FILE Objects

- FILE object
  - Stream based
    - Byte orientation vs. multi-byte(wide) orientation
  - A structure containing all the information needed for the standard I/O library
    - The file descriptor
    - A pointer to a buffer for the stream
    - The size of the buffer
    - The number of characters currently in the buffer
    - An error flag,
    - And so on.
  - Three default streams (defined in <stdio.h>)
    - stdin, stdout, and stderr corresponding to STDIN_FILENO(0), STDOUT_FILENO(1), STDERR_FILENO(2), respectively

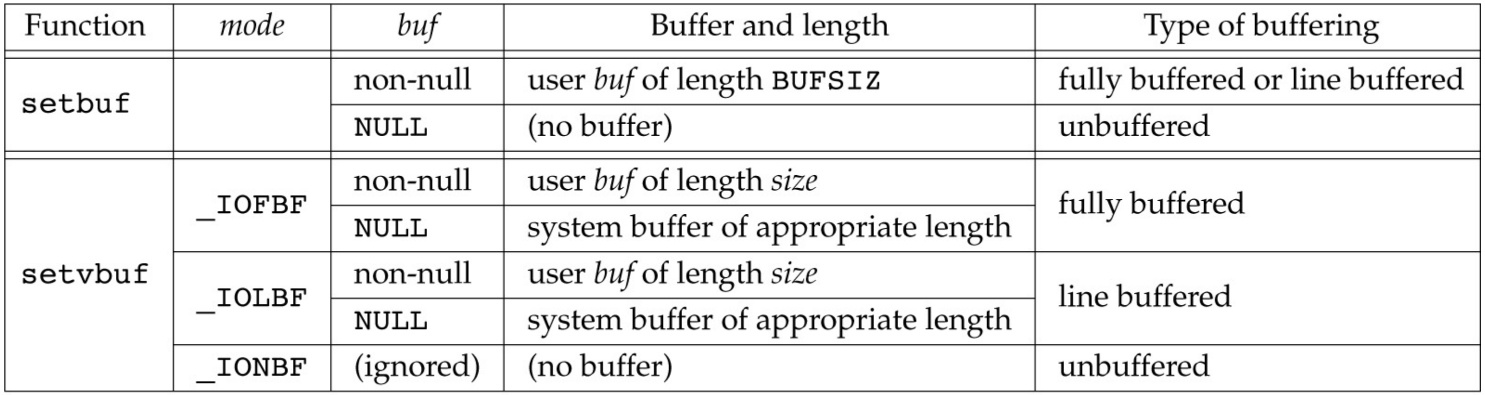
Buffering

- Goal of buffering
  - Minimize the number of read() and write() calls
- 3 types
  - Fully buffering
    - I/O timing : Buffer is filled
    - Ex: File stream
  - Line buffering
    - I/O timing : a newline character is encountered
    - Ex: Terminal stream
  - Unbuffered
    - I/O timing : immediately
    - Ex: Standard error stream

setbuf( )

```
#include <stdio.h>

void setbuf(FILE *fp, char *buf);
int setvbuf(FILE *fp, char *buf, int mode, size_t size);
                                        Return: 0 if OK, nonzero on error
```

- Change the buffering
- Called after the stream has been opened, but before any other operation is called



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

fflush( )

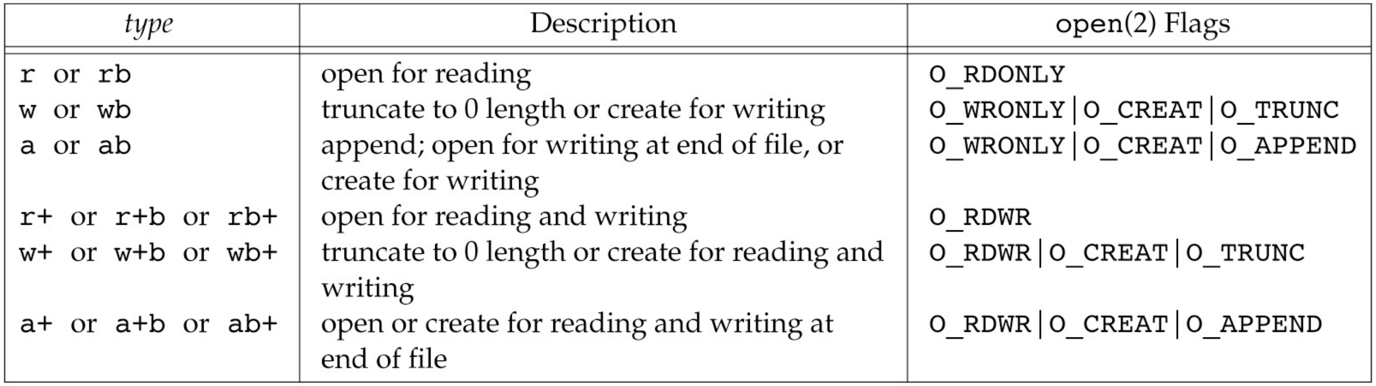
```
#include <stdio.h>

int fflush(FILE *fp);
                                        Return: 0 if OK, EOF on error
```

- Force any unwritten data for the stream to be passed to the kernel
- If *fp* is NULL, it causes all output streams to be flushed.

fopen( )

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);
FILE *freopen(const char *pathname, const char *type, FILE *fp);
FILE *fdopen(int filedes, const char *type);
                              All Return: file pointer if OK, NULL on error
```

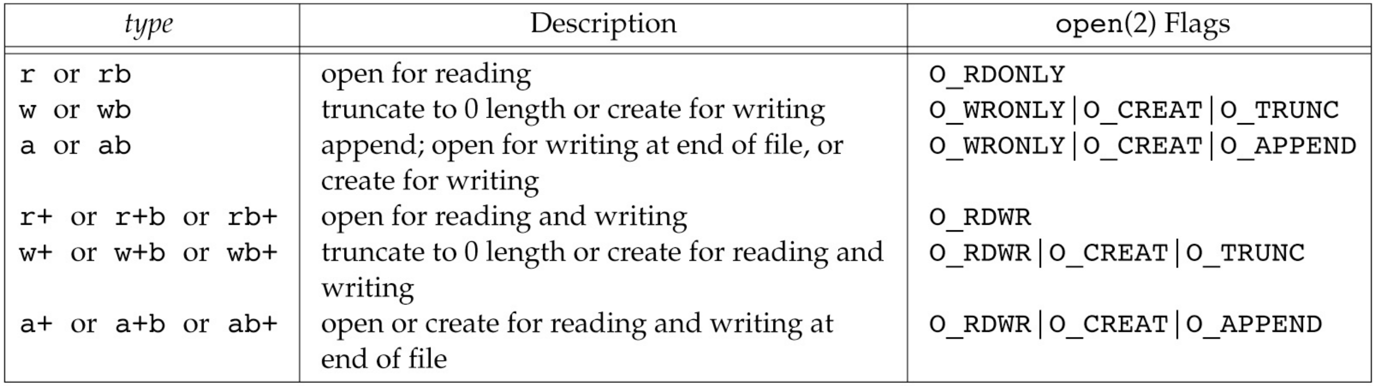
- freopen
  - Close the file and reopen with orientation cleared
- fdopen
  - Often used for pipes and device files (after obtaining a file descriptor by calling the device-specific function)



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

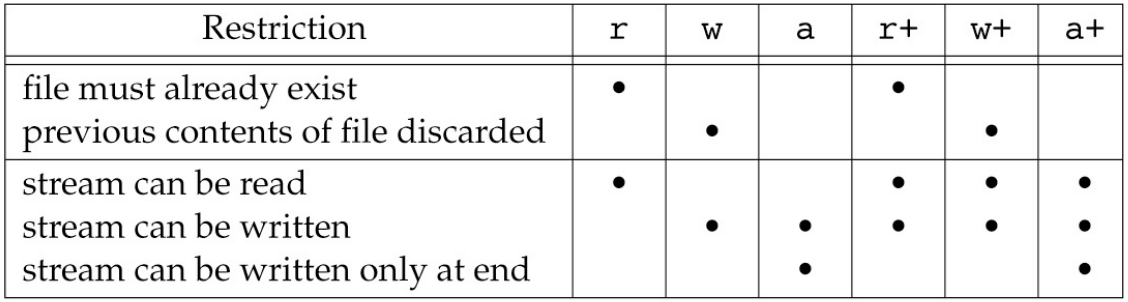
Figure 5.2 The *type* argument for opening a standard I/O stream

fopen( )



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



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

fclose( )

```
#include <stdio.h>

int fclose(FILE *fp);
                                        All Return: 0 if OK, EOF on error
```

Read and Write

- 3 types of I/O
  - Character-at-a-time I/O
  - Line-at-a-time I/O
  - Direct (Binary) I/O

Character-at-a-time

Input functions

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
                    All Return: next character if OK, EOF on end of file or error
```

Output functions

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
                                       All Return: c if OK, EOF on error
```

- getc()/putc() is macro vs. fgetc()/fputc() is not macro
- getchar() = getc(stdin), putchar() = putc(stdout)

```
int ungetc(int c, FILE *fp);
                                       All Return: c if OK, EOF on error
```

Line-at-a-time

Input functions

```
#include <stdio.h>
char *fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
                                   Both return: buf if OK, NULL on end of file or error
```

- fgets() : Null-byte terminated & no more than *n*-1 characters are read
- gets() : Never used.

Output functions

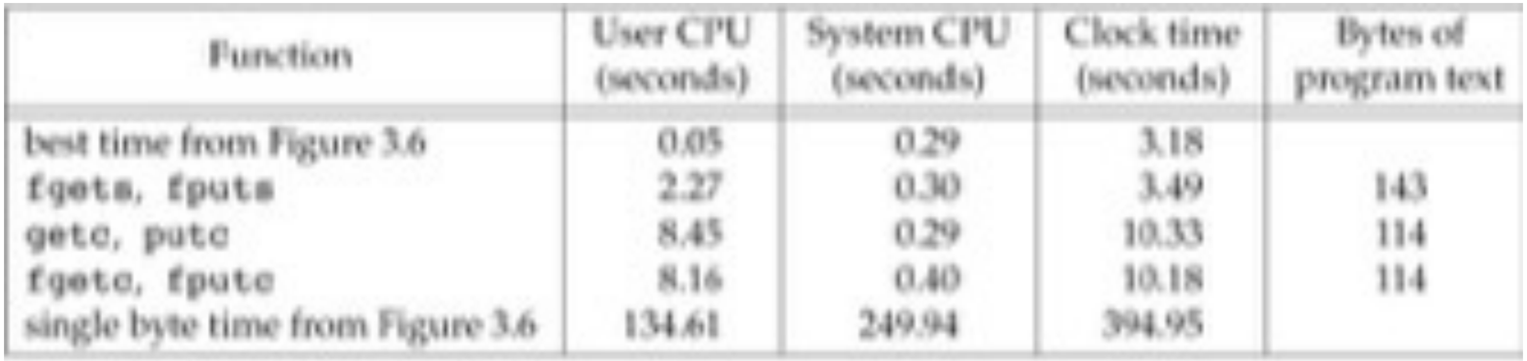
```
#include <stdio.h>
int fputs(const char *str, FILE *fp);
int puts(const char *str);
                                   Both return: non-negative value if OK, EOF on error
```

Standard I/O Efficiency

```
#include "apue.h"

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

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

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

    exit(0);
}
```

```
#include "apue.h"
#define MAXLINE 4096

int main(int argc, char *argv[]) {
    char    buf[MAXLINE];

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

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

    exit(0);
}
```

Standard I/O Efficiency



Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.6	0.05	0.29	3.18	
`fgets, fputs`	2.27	0.30	3.49	143
`getc, putc`	8.45	0.29	10.33	114
`fgetc, fputc`	8.16	0.40	10.18	114
single byte time from Figure 3.6	134.61	249.94	394.95	

Figure 5.6 Timing results using standard I/O routines

Binary I/O

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
                                    Both return: number of objects read or written
```

- Limitation
  - read() & write() should be used on the same system
- Example

```
float data[10];
if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");

struct {
    short count;
    long total;
    char name[NAMESIZE];
} item;
if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

Positioning a Stream

```
#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, nonzero on error
void rewind(FILE *fp);
```

- whence
  - SEEK_SET, SEEK_CUR, SEEK_END

Formatted I/O

Output

```
#include <stdio.h>
int printf(const char *format, ...);
int fprintf(FILE *fp, const char *format, ...);
            Both return: number of characters output if OK, negative value if output error
int sprintf(char *buf, const char *format, ...);
int snprintf(char *buf, size_t n, const char *format, ...);
                    Both return: number of characters if OK, negative value if encoding error
```

Input

```
#include <stdio.h>
int scanf(const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
int sscanf(const char *buf, const char *format, ...);
                                        All three return: number of input items assigned,
                                   EOF if input error or end of file before any conversion
```

Example

```
#include "apue.h"
void pr_stdio(const char *, 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/motd", "r")) == NULL) err_sys("fopen error");
    if (getc(fp) == EOF) err_sys("getc error");
    pr_stdio("/etc/motd", fp);
    exit(0);
}

void pr_stdio(const char *name, FILE *fp)
{
    printf("stream = %s, ", name);
    /* The following is nonportable. */
    if (fp->_IO_file_flags & _IO_UNBUFFERED) printf("unbuffered");
    else if (fp->_IO_file_flags & _IO_LINE_BUF) printf("line buffered");
    else /* if neither of above */ printf("fully buffered");
    printf(", buffer size = %d\n", fp->_IO_buf_end - fp->_IO_buf_base);
}
```

Execution

```
$ ./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/motd, fully buffered, buffer size = 4096
$ ./a.out < /etc/termcap > 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/motd, fully buffered, buffer size = 4096
```

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
```

- tmpfile
  - Creates a temporary binary file (type wb+) that is automatically removed when it is closed or on program termination. (cf. unlink())
- Avoid use of tmpnam
  - The deficiency is "Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name."

Example

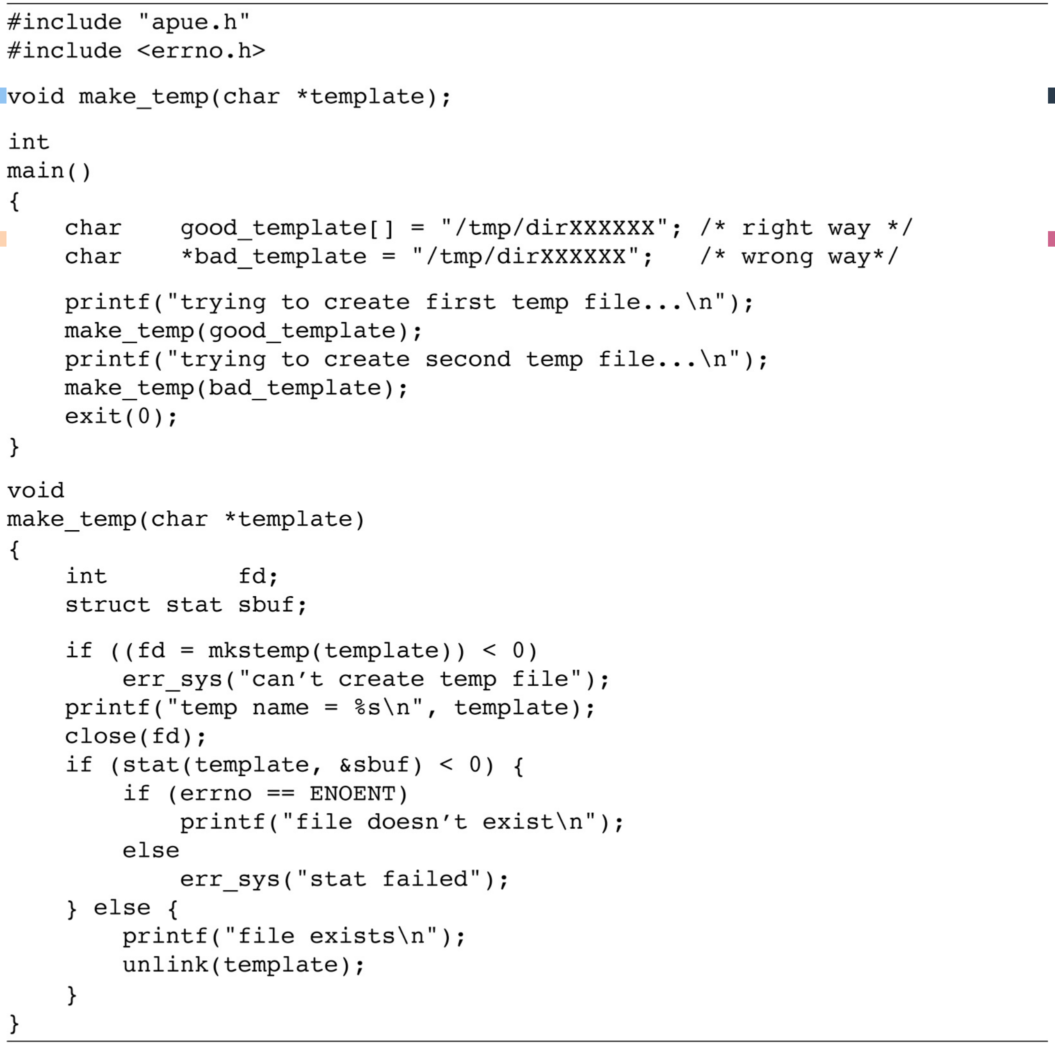
```
#include "apue.h"
int main(void) {
        char name[L_tmpnam], line[MAXLINE];
        FILE *fp;
        printf("%s\n", tmpnam(NULL));                   /* first temp name */
        tmpnam(name);                                   /* second temp name */
        printf("%s\n", name);
        if ((fp = 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);
}
```

```
$ ./a.out
/tmp/fileC1Icwc
/tmp/filemSkHSe
one line of output
```

Temporary Files

```
#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
```

- mkdtemp
  - Creates a temporary directory
  - template : last 6 characters are set to XXXXXX, which will be set on success
  - Permission = S_IRUSR | S_IWUSR | S_IXUSR
- mkstemp
  - Creates a temporary regular file
  - template : last 6 characters are set to XXXXXX , which will be set on success
  - Permission = S_IRUSR | S_IWUSR
  - Not automatically removed.



```c
#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

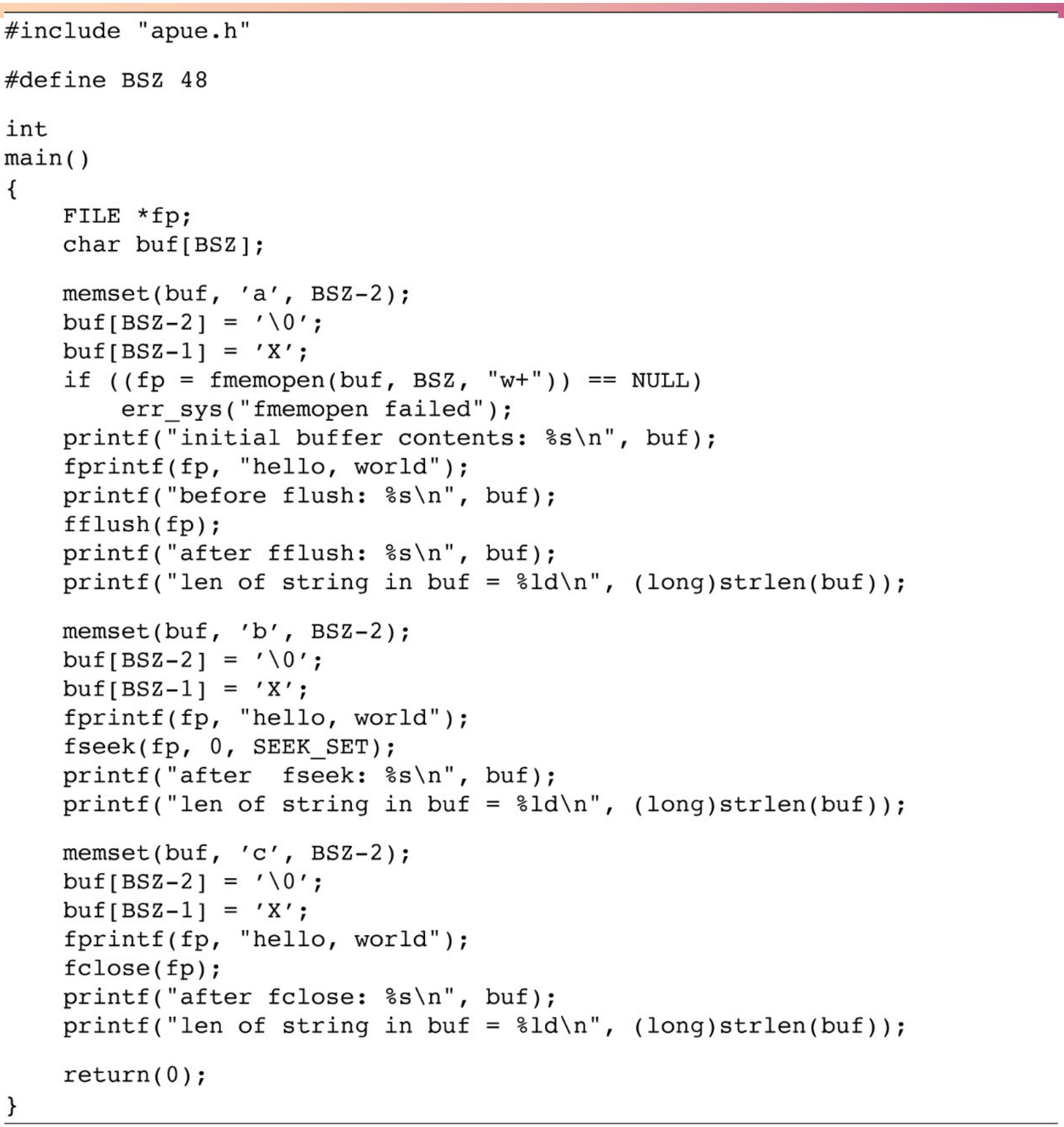
```
$ ./a.out
trying to create first temp file...
temp name = /tmp/dirvqyjMo
file exists
trying to create second temp file...
Segmentation fault (core dumped)
```

Memory Streams

```
#include <stdio.h>
FILE *fmemopen(void *restrict buf, size_t size,
                         const char *restrict type);
                                        Returns: stream pointer if OK, NULL on error
```

- Create memory streams.

Example



```
#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

Execution

```
$ ./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, worldcccccccccccccccccccccccccccccccccc
len of string in buf = 46               no null byte appended
```