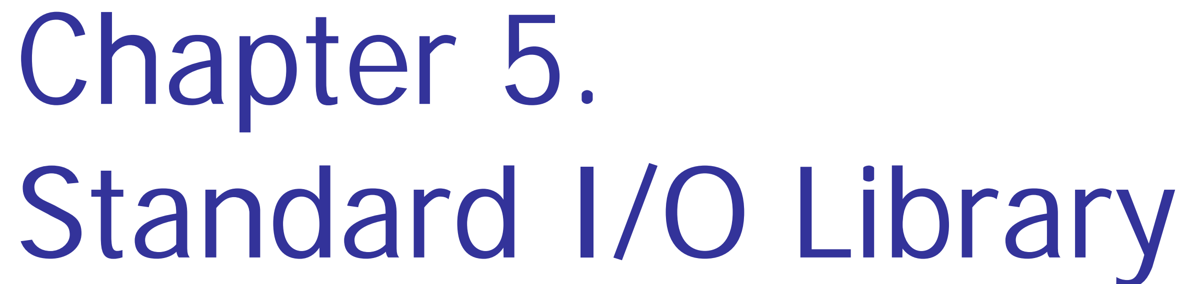


Chapter 5. Standard I/O Library

System Programming

http://www.cs.ccu.edu.tw/~pahsiung/courses/sp/

熊博安

國立中正大學資訊工程學系

pahsiung@cs.ccu.edu.tw

(05)2720411 ext. 33119

Class: EA-104

Office: EA-512

Introduction

Standard I/O Library

- Written by Dennis Ritchie in 1975
- Implemented on many OS
- ISO C standard
- Buffer allocation
- Perform I/O in optimal-sized chunks
- Easy to use, but introduces other problems

Streams and FILE objects

- A stream is associated with a file when we open or create a file using std I/O lib
- ASCII character set
  - Single character represented by a byte
- International character set
  - Single character represented by one or more bytes (multibyte)
  - "wide" character sets

Streams and FILE objects

- Initially
  - A stream has no orientation
- Multibyte I/O function
  - Stream's orientation = wide-oriented
- A byte I/O function
  - Stream's orientation = byte-oriented
- Change stream orientation
  - freopen (to clear), fwide (to set)

Streams and FILE objects

- #include <stdio.h>
- #include <wchar.h>
- int fwide(FILE *fp, int mode);
- Returns:
  - +ve if stream is wide-oriented,
  - -ve if stream is byte-oriented,
  - 0 if stream has no orientation

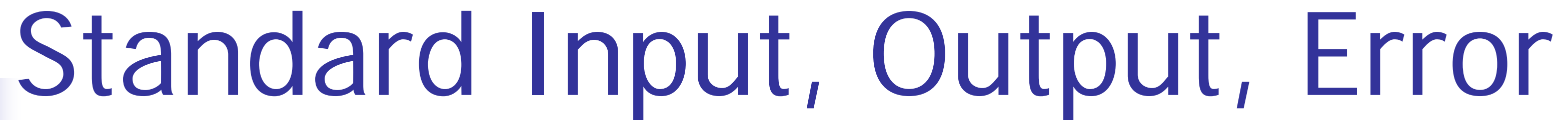
+ve set stream wide-oriented,
-ve set stream byte-oriented,
0 no change in orientation

Streams and FILE objects

- fopen() returns a FILE object
  - file descriptor
  - pointer to stream buffer
  - buffer size
  - #chars in buffer
  - error flag
  - …

Streams and FILE objects

- Applications need not examine FILE obj
- Pass FILE pointer as argument to standard I/O functions
- FILE * is called file pointer



Standard Input, Output, Error

- 3 streams are automatically created for a process
  - stdin
  - stdout
  - stderr
  - Defined in <stdio.h>

Buffering

3 types of buffering

- Fully Buffered
- Line Buffered
- Unbuffered

Fully Buffered I/O

- Actual I/O takes place when std I/O buffer is FILLED!
- Disk files are fully buffered
- A buffer can be flushed (write out contents of buffer to disk, …, etc.)
  - fflush(): write out buffer contents (std I/O)
  - tcflush(): discard buffer data (terminal driver)

Slides©2006 Pao-Ann Hsiung, Dept of CSIE, National Chung Cheng University, Taiwan

Line Buffered I/O

- Actual I/O is performed only when a newline character is encountered
- fputc(): can input/output a single char
- Line buffering is used on a stream associated with a terminal
- Fixed buffer size → I/O takes place before newline when buffer is full

Unbuffered I/O

- Characters unbuffered
- fputs()
- stderr is unbuffered
- (hence, error messages are always output very quickly!)

Buffering

- ISO C requirements on buffering:
  - stdin and stdout are fully buffered (except in an interactive device)
  - stderr is never fully buffered
- Most implementations of buffering:
  - stderr is always unbuffered
  - All other streams are line buffered for terminal device; o/w fully buffered

Buffering

- Type of buffering can be changed:

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

- Returns 0 if OK, nonzero on error

setbuf(): turns buffering on or off

- char *buf:
  - a buffer of length BUFSIZ (<stdio.h>)

    (Terminal device: line buffered
    otherwise: fully buffered)
  - NULL (buffering turned off)

setvbuf(): set buffering type

- int mode
  - _IOFBF: fully buffered
  - _IOLBF: line buffered
  - _IONBF: unbuffered
- char *buf: buffer

  (NULL➔ automatic buffer allocation

  Files: buffer size = st_blksize in stat

  Pipes: buffer size = BUFSIZ)
- size_t size: buffer size

setbuf() v/s setvbuf() (Fig. 5.1)

Function	*mode*	*buf*	Buffer & length	Type of buffering
setbuf		nonnull	user *buf* of length BUFSIZ	fully buffered or line buffered
		NULL	(no buffer)	unbuffered
setvbuf	_IOFBF	nonnull	user *buf* of length *size*	fully buffered
		NULL	system buffer of appropriate length	
	_IOLBF	nonnull	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.

Stream Flushing

- A stream can be flushed:

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

- Returns: 0 if OK, EOF on error
- Unwritten data passed to kernel
- fp = NULL → ALL output streams flushed!

Opening a Stream

```
#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     );
```

- Return: file pointer if OK, NULL on error

fopen, freopen, fdopen

ISO C
- fopen opens a specified file
- freopen opens a specified file on a specified stream (to open a specified file as stdin, stdout, or stderr)

POSIX.1
- fdopen associates a stream with a file descriptor (for pipes, network comm channels, which cannot be opened by fopen)

Slides©2006 Pao-Ann Hsiung, Dept of CSIE, National Chung Cheng University, Taiwan

type argument (Fig. 5.2)

type	Description
`r` or `rb`	open for reading
`w` or `wb`	truncate to 0 length or create for writing
`a` or `ab`	append; open for writing at end of file, or create for writing
`r+` or `r+b` or `rb+`	open for reading and writing
`w+` or `w+b` or `wb+`	truncate to 0 length or create for reading and writing
`a+` or `a+b` or `ab+`	open or create for reading and writing at end of file

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

- b ➔ binary (for some non-UNIX OS)
- UNIX doesn’t distinguish between text and binary, so no difference in UNIX

6 ways to open stream (Fig. 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 different ways to open a standard I/O stream.

To close an open stream

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

- Returns: 0 if OK, EOF one error
- Before closing:
  - Output buffer data is flushed
  - Input buffer data is discarded
  - Automatically allocated buffer is released

Reading / Writing a Stream

3 types of Unformatted I/O:

- Character-at-a-time I/O: getc(), fgetc(), getchar()
- Line-at-a-time I/O: fgets(), fputs()
- Direct I/O: fread(), fwrite() (read/write some number of objects)

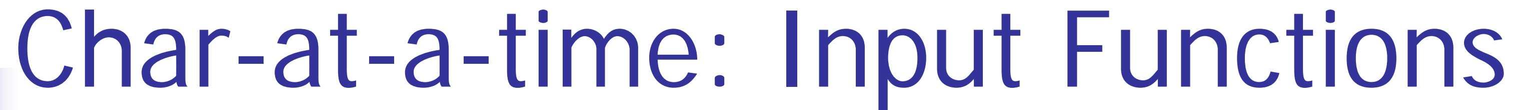
Binary I/O

Object-at-a-time I/O

Record-oriented I/O

Structure-oriented I/O

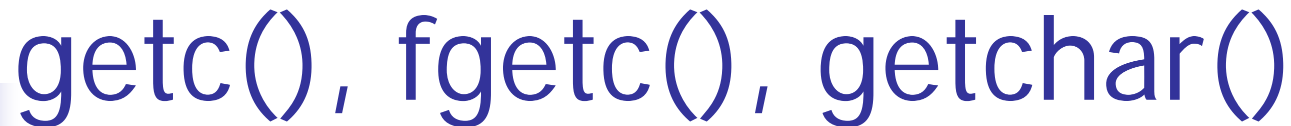
Char-at-a-time: Input Functions



```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

int getchar(void);   (== getc(stdin))
```

- Return: next char if OK, EOF on EOF or error

getc(), fgetc(), getchar()



- getc() is a macro
  - arguments without side effects
- fgetc() is a function
  - func addr can be passed as arguments
  - It is slower than getc()
- EOF=-1 ➔ int return value
- No distinction between:
  - file error
  - end of file

File Error or EOF?

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

- Return: nonzero if true, 0 otherwise
- void clearerr(FILE *fp);
  - Clears 2 flags in FILE object:
    - an error flag
    - an EOF flag

ungetc()

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

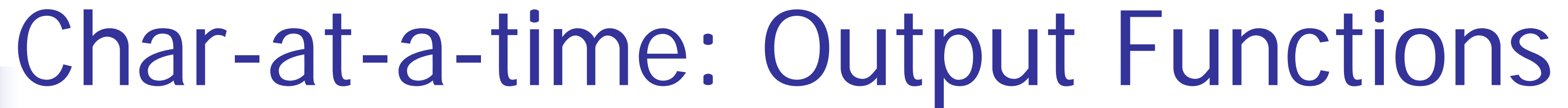
- Returns: c if OK, EOF on error
- Chars are pushed back into stream
  - Need not push back same char as read
  - Cannot push EOF

Slides©2006 Pao-Ann Hsiung, Dept of CSIE, National Chung Cheng University, Taiwan

ungetc()

- At EOF, we can push back a character
  - Because ungetc clears EOF indication for the stream
- Generally, used for breaking input stream into words or tokens
  - peek next char to determine how to handle current character

Char-at-a-time: Output Functions



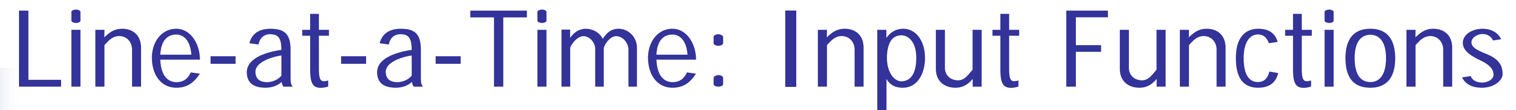
#include <stdio.h>

macro → int putc(int c, FILE *fp);

function → int fputc(int c, FILE *fp);

int putchar(int c); (== putc(c, stdout))

- Return: c if OK, EOF on error



Line-at-a-Time: Input Functions

```
#include <stdio.h>

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

(stores: line + \n + NULL)

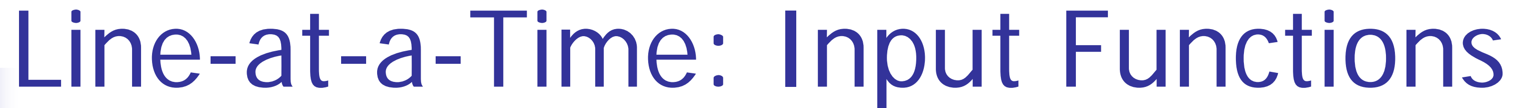
deprecated →

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

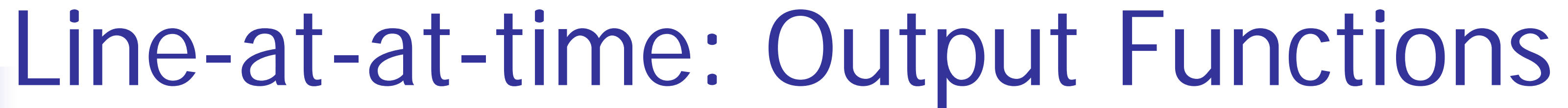
(stores line only, without \n and NULL)

- Return: buf if OK, NULL on EOF or error

Line-at-a-Time: Input Functions

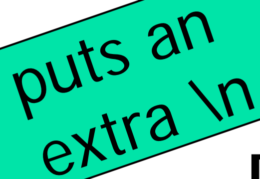


- Never use gets()
  - It doesn't allow caller to specify buffer size
  - Allows buffer to overflow
    - writing over whatever happens to follow the buffer in memory
    - flaw used as part of the Internet worm in 1988
      - **Communications of the ACM, Vol. 32, No. 6, 1989**



Line-at-at-time: Output Functions

```
#include <stdio.h>

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

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



- Return: nonnegative value if OK,
  EOF on error

Line-at-a-time I/O

Suggestion:

- Use:
  - fgets() and fputs()
- Remember:
  - we always have to deal with the newline character at the end of each line!

Standard I/O Efficiency

- How does the three types of unformatted I/O compare in efficiency?
- How does unformatted I/O compare with direct read() and write()?
- Are macros more efficient than functions? Why?
- What about Program Text size?

stdin → stdout (getc, putc) Figure 5.4

```
#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);
}
```

stdin → stdout (fgets(), fputs) Figure 5.5

```
#include    "ourhdr.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);
}
```

- flush data
- close streams

Timing Results using std I/O (Fig. 5.4) (textbook edition 1)

Function	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Bytes of program text
best time from Figure 3.1	0.0	0.3	0.3	
`fgets, fputs`	2.2	0.3	2.6	184
`getc, putc`	4.3	0.3	4.8	384
`fgetc, fputc`	4.6	0.3	5.0	152
single byte time from Figure 3.1	23.8	397.9	423.4	

Figure 5.4 Timing results using standard I/O routines.

Timing results using std I/O (Fig. 5.6)

Function	User CPU (s)	System CPU (s)	Clock time (s)	Program text (byte)
best time (Fig. 3.5)	0.01	0.18	6.67	
fgets, fputs	2.59	0.19	7.15	139
getc, putc	10.84	0.27	12.07	120
fgetc, fputc	10.44	0.27	11.42	120
single byte (Fig. 3.5)	124.89	161.65	288.64	120

Standard I/O Efficiency

\# Loop Iterations:

- Char-at-a-time: **100 million times**
- Line-at-a-time: **3,144,984 times**
- read() version: **12,611 times** (BUFFSIZE=8,192)
- Thus, different user CPU times!

Program Text Size and Execution Time

- getc(), putc(): inline macros
- ➔ more instructions, larger text size
- GNU C library: macro = function call
  - program texts are all same size
- Execution Time:
- Char-at-a-time = 2 * (Line-at-a-time)
- Line-at-a-time: implemented using memccpy(), which is in assembly.

fgetc() and BUFFSIZE=1 prog

- Compared to single-byte read, fgetc() is
  - 12 times faster in user CPU time
  - 25 times faster in clock time
- fgetc() is much faster! Why?
- Both: 200 million function calls
- read(): 200 million system calls!!
- fgetc(): 25,222 system calls!!

Slides©2006 Pao-Ann Hsiung, Dept of CSIE, National Chung Cheng University, Taiwan

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);
```

- Return: #objects read or written

Binary I/O

- Read or write an entire structure!
- Cannot use fgets() or fputs()
  - There may be NULLs or newlines in a structure
- Hence, fread() and fwrite() are required!

Binary I/O: (1/2) Read/Write Binary Array

- Write elements 2 through 5 of a floating point array:

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

Binary I/O: (2/2) Read/Write a Structure

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

Binary Data Exchange

May be incompatible among different machine architectures because:

- Offset of a member in a structure can differ between compilers and systems
- Binary formats of int and float may differ between different machines

So use a higher level protocol instead!

Slides©2006 Pao-Ann Hsiung, Dept of CSIE, National Chung Cheng University, Taiwan

Positioning a Stream

Three ways:

- ftell and fseek (UNIX Version 7)
  - assumes filepos as long integer data type
- ftello and fseeko (Single UNIX Specification)
  - assumes filepos as off_t data type ← replace long int
- fgetpos and fsetpos (ISO C)
  - assumes filepos as fpos_t data type ← as big as necessary
  - more portable (UNIX, non-UNIX, …)

ftell() and fseek()

```
#include <stdio.h>
long ftell (FILE *fp);
```

- Returns: curr filepos 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);
```

ftello() and fseeko()

```
#include <stdio.h>
off_t ftello(FILE *fp);
```

- Returns curr filepos indicator if OK, (off_t)-1 on error

```
int fseeko(FILE *fp, off_t offset,
                int whence);
```

- Returns 0 if OK, nonzero on error

fgetpos() and fsetpos()

```
#include <stdio.h>

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

int fsetpos(FILE *fp, const fpos_t *pos);
```

Return: 0 if OK, nonzero on error

Can reposition to a previous position obtained by fgetpos()

Formatted Output

```c
#include <stdio.h>

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

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

- Return: #char output if OK, <0 on error

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

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

- Return: #char stored in array (without counting NULL)
- NULL appended to buf

Formatted Output (variants)

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
int vfprintf(FILE *fp, const char *format,
        va_list arg);
```

- Return: #char output if OK, <0 on error

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

- Return: #char stored in array if OK, <0 if error

Formatted Input

```
#include <stdio.h>

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

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

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

Return: #items assigned, EOF on error or on EOF

Implementation Details

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

- Returns: file descriptor from stream
- Needed by dup() and fcntl() functions

Print stdio buffering (Fig. 5.11)

```
#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);
```

Print stdio buffering (Fig. 5.11)

```
   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);
   /* following is nonportable */
   if   (fp->_IO_file_flags & _IO_UNBUFFERED)
        printf("unbuffered");
   else if (fp->_IO_file_flag & _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);
}
```

Program 5.3: Output (1)

$ a.out

enter any character

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

Program 5.3: Output (2)

```
$ a.out < /etc/termcap > std.out 2> std.err
$ 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);
```

- NULL→store in static area
- O/W ptr = user-given buffer

- Returns: pointer to unique pathname

```
FILE *tmpfile(void);
```

- Returns: file pointer if OK, NULL on error

wb+ (create for read/write)

Fig. 5.12: tmpnam, tmpfile

```
#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); }
```

Fig. 5.12: Output

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

tmpfile()

UNIX technique for tmpfile():

- Call tmpnam() to create a unique pathname
- Create the file
- Unlink it immediately

(unlinking is not deleting immediately)

tempnam()

- #include <stdio.h>
- char *tempnam(const char *directory, const char *prefix);
- Returns: pointer to unique pathname

tempnam()

Directory choices:

- `TMPDIR` (environment variable)
- *directory* (non-NULL argument)
- `P_tmpdir` defined in `<stdio.h>`
- `/tmp` (a local directory)

Prefix: up to 5 chars as prefix of filename

Fig. 5.13: tempnam()

```
#include   "apue.h"

int
main(int argc, char *argv[])
{
  if (argc != 3)
      err_quit("usage: a.out <directory>
  <prefix>");

  printf("%s\n", tempnam(
      argv[1][0] != ' ' ? argv[1] : NULL,
      argv[2][0] != ' ' ? argv[2] : NULL) );
  exit(0);
}
```

Fig. 5.13: Outputs

$ ./a.out /home/sar TEMP

/home/sar/TEMPsf00zi

$ ./a.out “ “ PFX (P_tmpdir=“/tmp”)

/tmp/PFXfBw7Gi

$ TMPDIR=/var/tmp ./a.out /usr/tmp “ “

/var/tmp/file8fVYNi

$ TMPDIR=/no/such/dir ./a.out /home/sar/tmp QQQ

/home/sar/tmp/QQQ98s8Ui

mkstemp()

- #include <stdlib.h>
- int mkstemp(char *template);
- Returns: file descriptor if OK, -1 on error
- filename of temporary file
  - template: last 6 chars set to XXXXXX
- Unlike tmpfile(), temporary file is not unlinked automatically by mkstemp()

Which to use?

- tmpnam, tempnam
  - Window between the unique pathname generation and actual file creation
    - During this time window, another process can create a file of the same name
- tempfile, mkstemp
  - No such time window
  - Use these functions

Alternatives to Std I/O

- Korn and Vo [1991]: list numerous defects
- Inefficiency in standard I/O
  - too much data copying
    - Eg.: line-at-a-time functions (fgets, fputs)
      - between kernel and std I/O buffer
      - between std I/O buffer and line buffer

Alternatives to Std I/O

- Alternatives
  - Fast I/O library (fio): returns pointer to the line
    - grep 3 times faster
  - sfio: generalized to files and memory regions, processing modules on stacked on I/O stream, better exception handling
  - Alloc Stream Interface (ASI): uses mmap
  - Embedded Systems
    - uClibc C library (http://www.uclibc.org)
    - newlibc C library (http://sources.redhat.com/newlib)