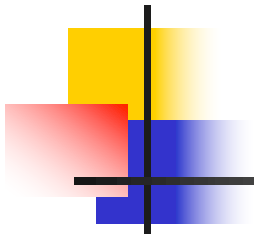
Chapter 5. Standard I/O Library

朱金辉

华南理工大学软件学院

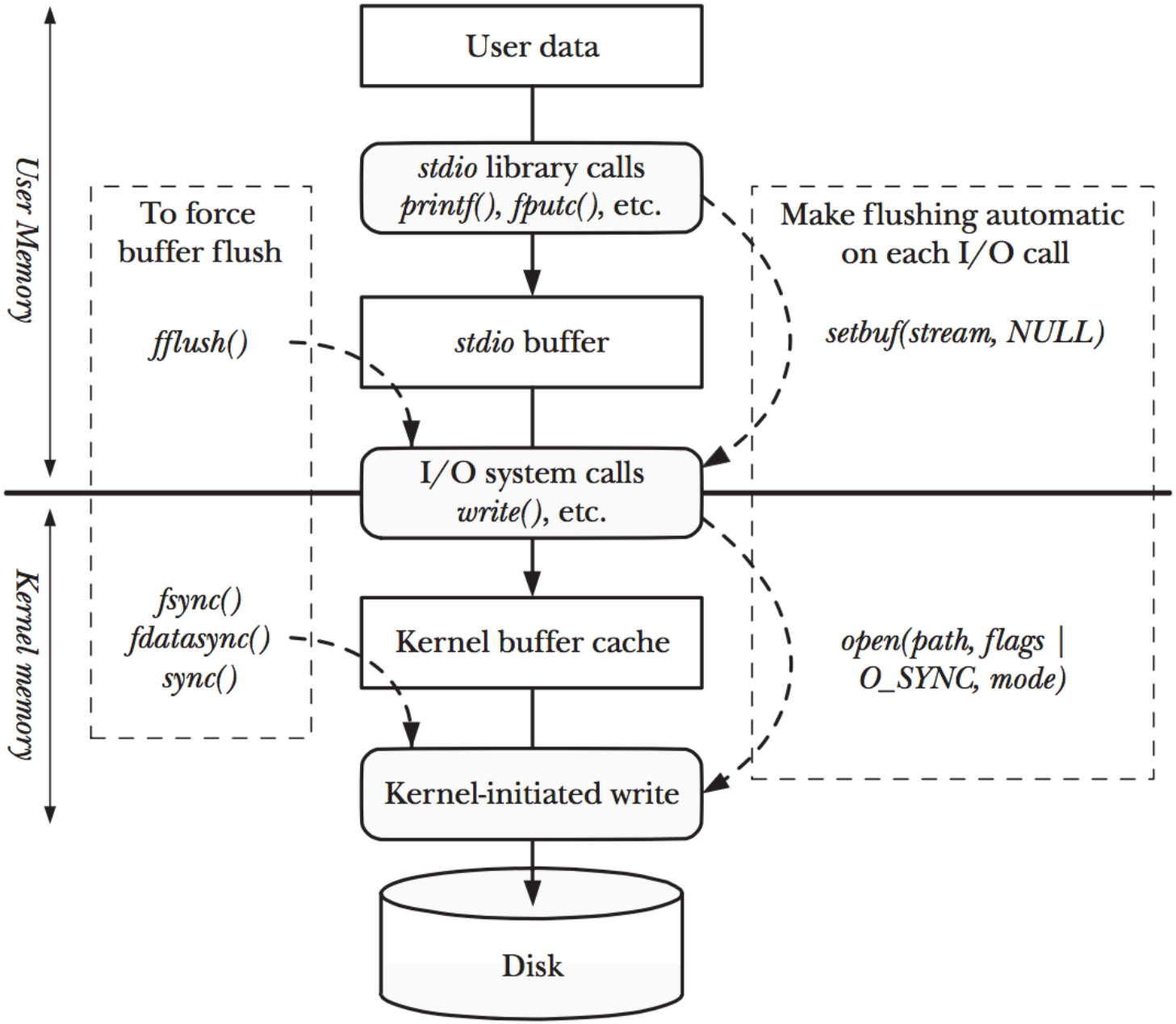


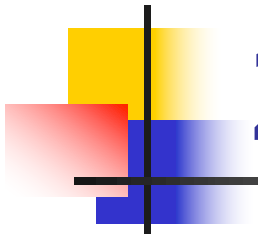
1. Introduction

Standard I/O Library

- Written by Dennis Ritchie in 1975
- Implemented on many OS
- ANSI C standard
- Buffer allocation
- Perform I/O in optimal-sized chunks
- Streams

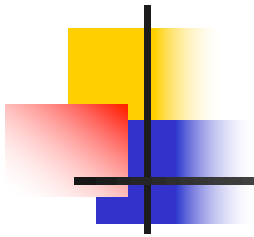
Summary of I/O buffering





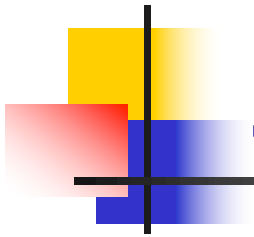
2. Streams

- A stream is associated with a file when we open or create a file using std I/O lib
- fopen() returns a FILE object
  - file descriptor
  - pointer to stream buffer
  - buffer size
  - #chars in buffer
  - error flag
  - …



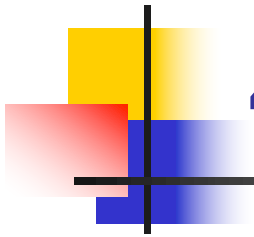
Streams

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



3. Standard Input, Output, Error

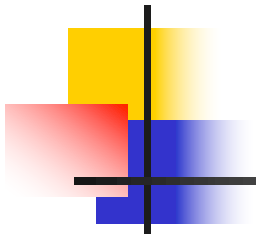
- 3 streams are automatically created for a process
  - stdin
  - stdout
  - stderr
  - Defined in <stdio.h>



4. Buffering

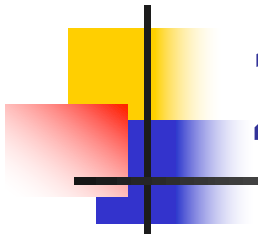
3 types of buffering

- Fully Buffered
- Line Buffered
- Unbuffered



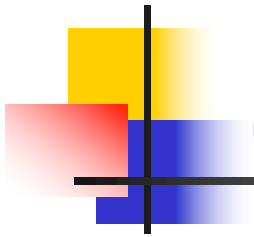
1) Fully Buffered I/O

- 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)



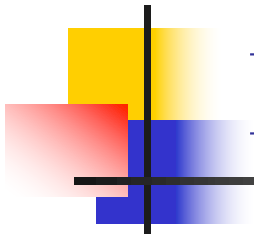
2) Line Buffered I/O

- Actual I/O is performed only when a newline character is encountered
- Line buffering is used on a stream associated with a terminal
- Fixed buffer size → I/O takes place before newline when buffer is full



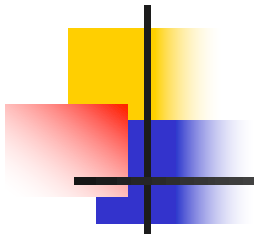
3) Unbuffered I/O

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



Buffering

- ANSI C requirements on buffering:
  - stdin and stdout are fully buffered (if and only if do not refer to an interactive device)
  - stderr is never fully buffered
- SVR4 and 4.3+BSD buffering:
  - stderr is always unbuffered
  - All other streams are line buffered for terminal device; otherwise fully buffered

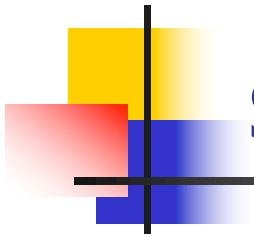


Buffering

- 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:

  (NULL➔ automatic buffer allocation

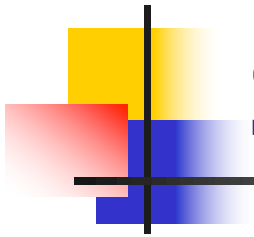
  Files: buffer size = st_blksize in stat

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

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

Function	*mode*	*buf*	Buffer 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



Stream Flushing

- A stream can be flushed:

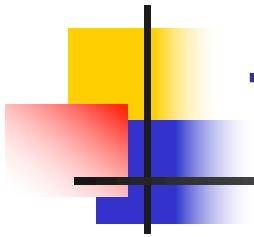
```
#include <stdio.h>

int fflush(FILE *fp);
```

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

5. 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

- fopen opens a specified file
- freopen opens a specified file on a specified stream (to open a specified file as stdin, stdout, or stderr)
- fdopen associates a stream with a file descriptor (for pipes, network comm channels, which cannot be opened by fopen)

type argument (Fig. 5.2)

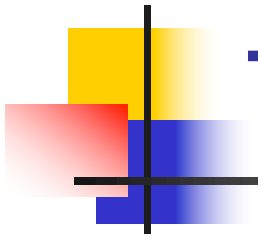
type	Description	open(2) Flags
`r` or `rb`	open for reading	`O_RDONLY`
`w` or `wb`	truncate to 0 length or create for writing	`O_WRONLY|O_CREAT|O_TRUNC`
`a` or `ab`	append; open for writing at end of file, or create for writing	`O_WRONLY|O_CREAT|O_APPEND`
`r+` or `r+b` or `rb+`	open for reading and writing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

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 ways to open a standard I/O stream

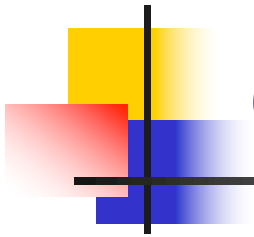


To close an open stream

#include <stdio.h>

int fclose(FILE *fp);

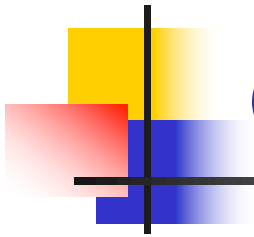
- Returns: 0 if OK, EOF one error
- Before closing:
  - Output buffer is flushed
  - Input buffer is discarded



6. Reading / Writing a Stream

3 types of Unformatted I/O:

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

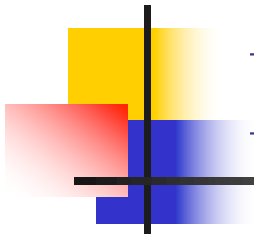


Char-at-a-time: Input Functions

```
#include <stdio.h>

int getc(FILE *fp);

int fgetc(FILE *fp);

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

Usually, getc are implemented as macros, but in the GNU C library implementation the macro simply expands to a function call.

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



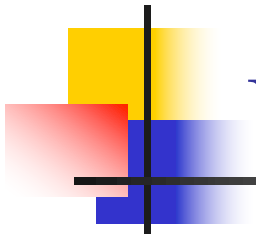
File Error or EOF?

- #include <stdio.h>

  int ferror(FILE *fp);

  int feof(FILE *fp);
- Return: nonzero if true, 0 otherwise

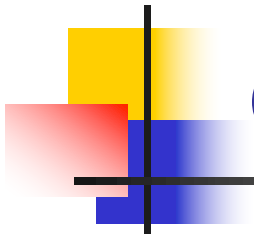
- void clearerr(FILE *fp);
- Clears 2 flags in FILE object:
  - an error flag
  - an EOF flag



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



Char-at-a-time: Output Functions

```
#include <stdio.h>

int putc(int c, FILE *fp);

int fputc(int c, FILE *fp);

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

- Return: c if OK, EOF on error

7. Line-at-a-Time: Input Functions

```
#include <stdio.h>

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

(stores: line + \n + NULL)

deprecated

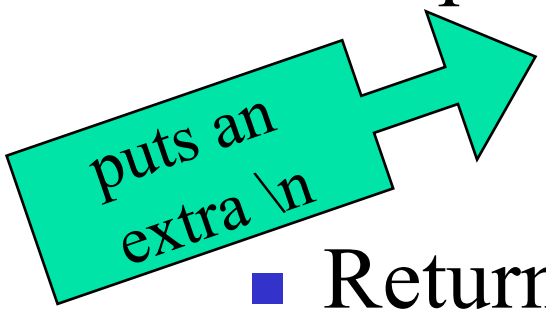
```
char *gets(char *buf);
```

(stores line only, without \n and NULL)

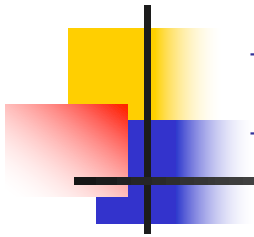
- Return: buf if OK, NULL on EOF or error

Line-at-at-time: Output Functions

```
#include <stdio.h>

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

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



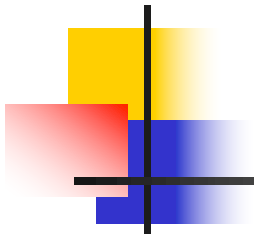
- Return: nonnegative value if OK,
  EOF on error



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!



8. 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?

stdin → stdout (getc, putc) Program 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) Program 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);
}
```

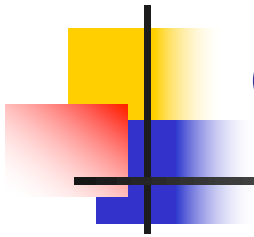
Timing Results using std I/O

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

```
→  fileio git:(master) ✗ time ./test_fileio 1 <~/test.data >/dev/null
./test_fileio 1 < ~/test.data > /dev/null  18.84s user 139.50s system 98% cpu 2:40.01 total
→  fileio git:(master) ✗ time ./test_fileio 1024 <~/test.data >/dev/null
./test_fileio 1024 < ~/test.data > /dev/null  0.02s user 0.21s system 84% cpu 0.275 total
→  fileio git:(master) ✗ time ./test_fileio 4096 <~/test.data >/dev/null
./test_fileio 4096 < ~/test.data > /dev/null  0.00s user 0.11s system 70% cpu 0.158 total
→  fileio git:(master) ✗ time ./test_fileio 524288 <~/test.data >/dev/null
./test_fileio 524288 < ~/test.data > /dev/null  0.00s user 0.08s system 60% cpu 0.139 total
```

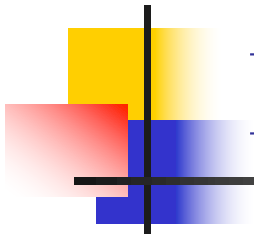
```
→  stdio git:(master) ✗ time ./getcputc < ~/test.data > /dev/null
./getcputc < ~/test.data > /dev/null  6.79s user 0.14s system 93% cpu 7.378 total
→  stdio git:(master) ✗ time ./fgetsfputs < ~/test.data > /dev/null
./fgetsfputs < ~/test.data > /dev/null  0.09s user 0.11s system 75% cpu 0.260 total
```



9. 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



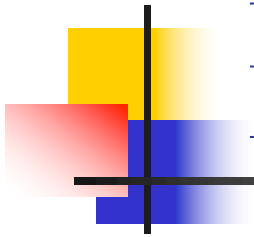
Binary I/O

- 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: Read/Write Binary Array

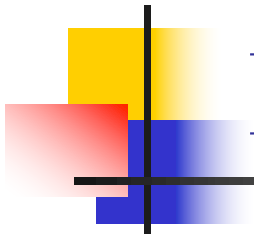
- Write elements 2 through 5 of a floating point array:

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



Binary I/O: 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" );
```

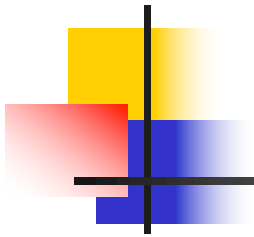


Binary Data Exchange

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

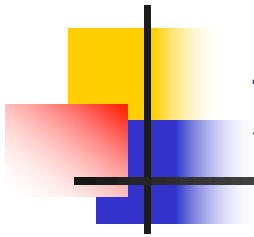
So use a higher level protocol instead!



10. Positioning a Stream

Three ways:

1. ftell and fseek
2. ftello and fseeko
3. fgetpos and fsetpos
   - more portable (ANSI)
   - new data type: fpos_t



ftell() and fseek()

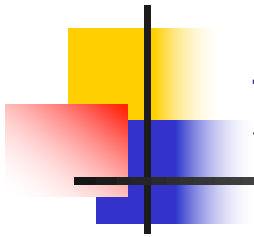
```
#include <stdio.h>

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

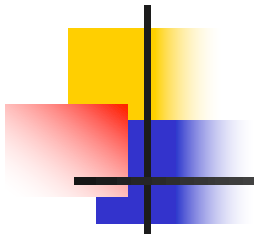
```
void rewind (FILE *fp);
```



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



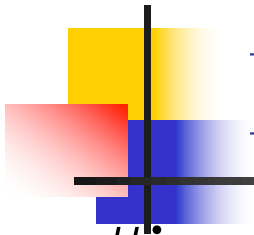
11. Formatted Output

```
#include <stdio.h>
int printf(const char *format, …);
int fprintf(FILE *fp, const char *format, …);
```

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

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

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



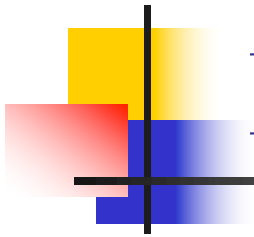
Formatted Output (variants)

```
#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);
```

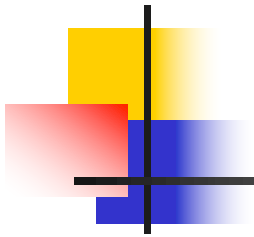
- Return: #char stored in array



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



12. Implementation Details

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

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

Print stdio buffering (Program 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);
```

```
    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->_flag & _IONBF)        printf("unbuffered");
    else if (fp->_flag & _IOLBF)        printf("line buffered");
    else /* if neither of above */      printf("fully buffered");
    printf(", buffer size = %d\n", fp->_bufsiz);
}
```

Program 5.11: Output (1)

$ a.out

enter any character

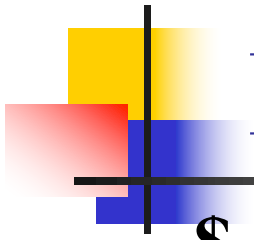
one line to standard error

stream = stdin, line buffered, buffer size = 128

stream = stdout, line buffered, buffer size = 128

stream = stderr, unbuffered, buffer size = 8

stream = /etc/motd, fully buffered, buffer size=8192



Program 5.11: 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=8192

stream=stdout, fully buffered, buffer size=8192

stream=stderr, unbuffered, buffer size=8

stream=/etc/motd, fully buffered, buffer size=8192

13. Temporary Files

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

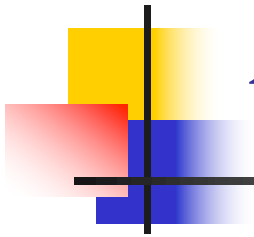
```
#include <stdio.h>
char *tmpnam(char *ptr);
```

- Returns: pointer to unique pathname

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

- Returns: file pointer if OK, NULL on error

wb+
(create for read/write)



tmpfile()

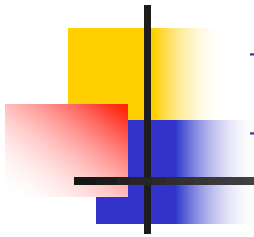
UNIX technique for tmpfile():

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

(unlinking is not deleting immediately)

Program 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);
}
```



Program 5.12: Output

$ a.out

/usr/tmp/fileC1Icwc

/usr/tmp/filemSkHSe

one line of output