CSC 4304 - Systems Programming
Fall 2008

# Lecture - V
# File I/O - II

Tevfik Koşar

Louisiana State University
September 16th, 2008

## Summary of Last Class

- Advanced Structures in C
  - Local vs Global Variables
  - Dynamic Memory Management
- File I/O
  - opening and closing files
  - reading from / writing to files
  - seeking files

## In Today's Class

- Buffered File I/O
  - opening and closing streams
  - reading from / writing to streams
  - Binary I/O
  - Formatted I/O

## Buffered I/O

- Unbuffered I/O: each read write invokes a system call in the kernel.
  - read, write, open, close, lseek
- Buffered I/O: data is read/written in optimal-sized chunks from/to disk --> streams
  - standard I/O library written by Dennis Ritchie

## Standard I/O Library

- Difference from File I/O
  - File Pointers vs File Descriptors
  - fopen vs open
    - When a file is opened/created, a *stream* is associated with the file.
    - FILE object
      - File descriptor, buffer size, # of remaining chars, an error flag, and the like.
  - stdin, sdtout, stderr defined in <stdio.h>
    - STDIO_FILENO, STDOUT_FILENO,…

## Buffering

- Goal
  - Use the minimum number of read and write calls.
- Types
  - Fully Buffered
    - Actual I/O occurs when the buffer is filled up.
    - A buffer is automatically allocated when the first-time I/O is performed on a stream.
    - flush: standard I/O lib vs terminal driver

## Buffering

- Line Buffered
  - Perform I/O when a newline char is encountered! – usually for terminals.
  - Caveats
    - The filling of a fixed buffer could trigger I/O.
    - The flushing of all line-buffered outputs if input is requested.
- Unbuffered
  - Expect to output asap, e.g. using write()
  - E.g., stderr

## Buffering

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

- Full/line buffering if buf is not NULL (BUFSIZ)
  - Terminals
- mode: _IOFBF, IOLBF, _IONBF (<stdio.h>)
  - Optional size → st_blksize (stat())
- #define BUFSIZ 1024 (<stdio.h>)
- They must be called before any op is performed on the streams!

## Buffering

- ANSI C Requirements
  - Fully buffered for stdin and stdout unless interactive devices are referred to.
    - SVR4/4.3+BSD – line buffered
  - Standard error is never fully buffered.

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

- All output streams are flushed if fp == NULL

## Opening a Stream

- #include <stdio.h>
- FILE *fopen(const char *pathname, const char *type);

- opens a specified file
- types:
  - r : open for reading
  - w : create for writing or truncate to 0
  - a : open or create for writing at the end of file
  - r+ : open for reading and writing
  - w+: create for reading and writing or truncate to 0
  - a+ :open or create for reading and writing at the end of file

  - use b to differentiate text vs binary , e.g. rb, wb ..etc

## Restrictions

| Type | r | w | a | r+ | w+ | a+ |
|---|---|---|---|---|---|---|
| File exists? | Y | | | Y | | |
| Truncate | | Y | | | Y | |
| R | Y | | | Y | Y | Y |
| W | | Y | Y | Y | Y | Y |
| W only at end | | | Y | | | Y |

* When a file is opened for reading and writing:

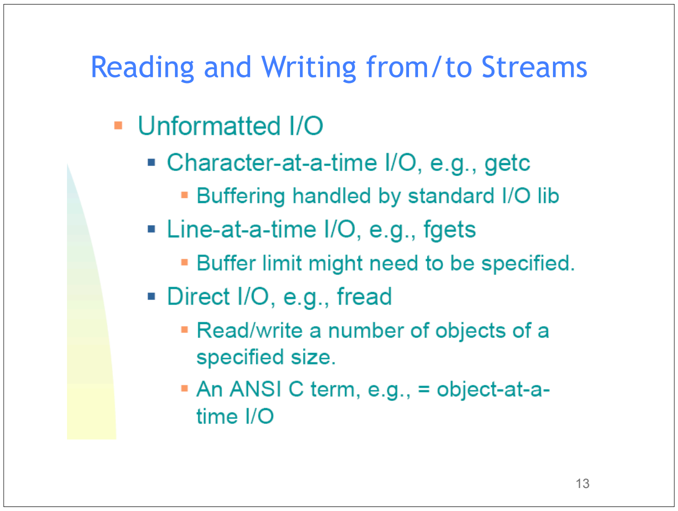
- Output cannot be directly followed by input without an intervening *fseek, fsetpos, or rewind*
- Input cannot be directly followed by output without an intervening *fseek, fsetpos, or rewind*

## Closing a Stream

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

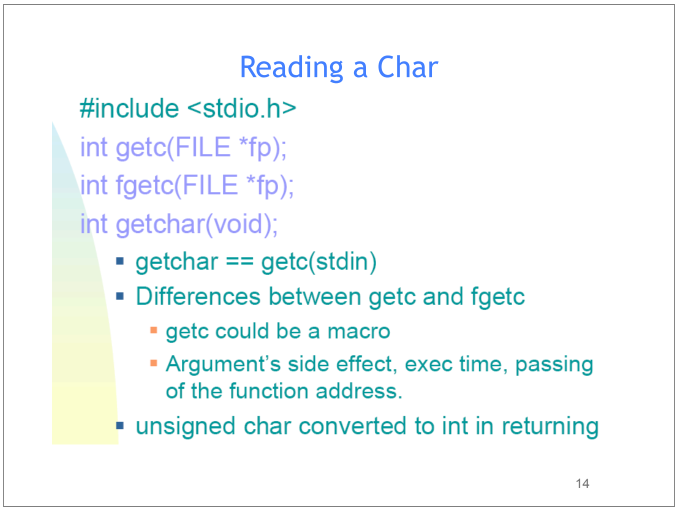
- Flush buffered output
- Discard buffered input
- All I/O streams are closed after the process exits.

- setbuf or setvbuf to change the buffering of a file before any operation on the stream.



## Reading and Writing from/to Streams

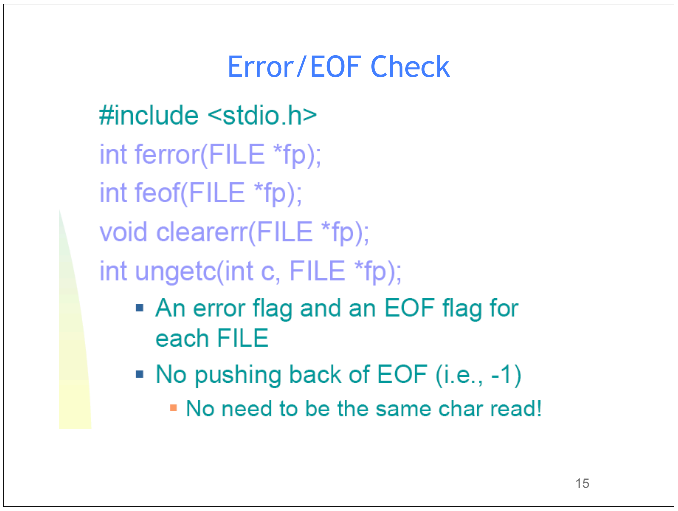
- Unformatted I/O
  - Character-at-a-time I/O, e.g., getc
    - Buffering handled by standard I/O lib
  - Line-at-a-time I/O, e.g., fgets
    - Buffer limit might need to be specified.
  - Direct I/O, e.g., fread
    - Read/write a number of objects of a specified size.
    - An ANSI C term, e.g., = object-at-a-time I/O



## Reading a Char

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

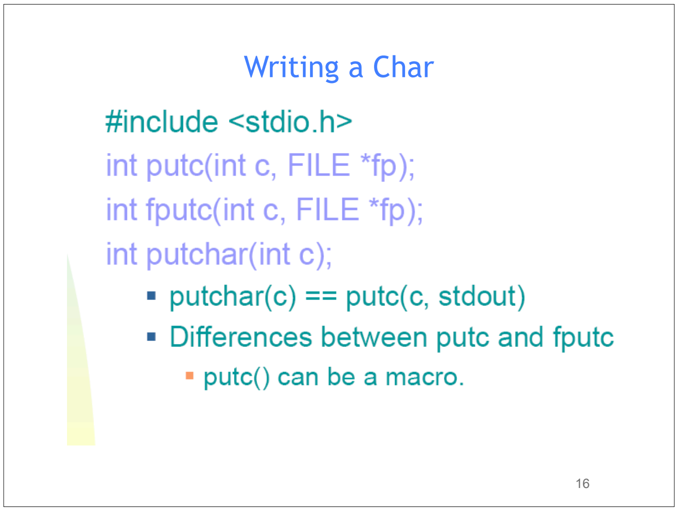
- getchar == getc(stdin)
- Differences between getc and fgetc
  - getc could be a macro
  - Argument’s side effect, exec time, passing of the function address.
- unsigned char converted to int in returning



## Error/EOF Check

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

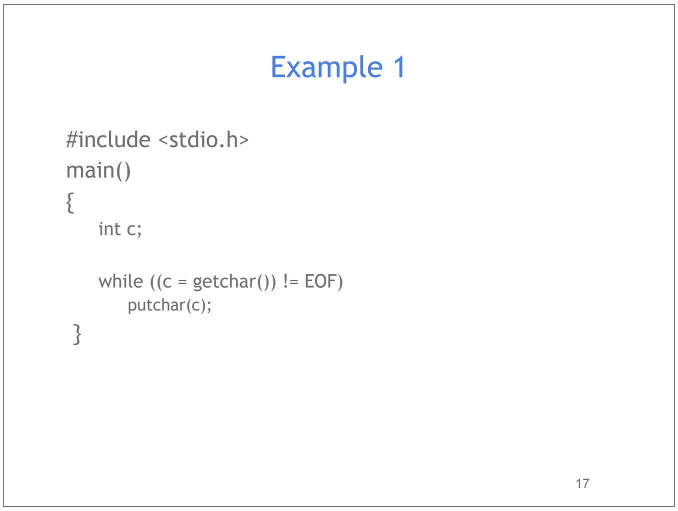
- An error flag and an EOF flag for each FILE
- No pushing back of EOF (i.e., -1)
  - No need to be the same char read!



## Writing a Char

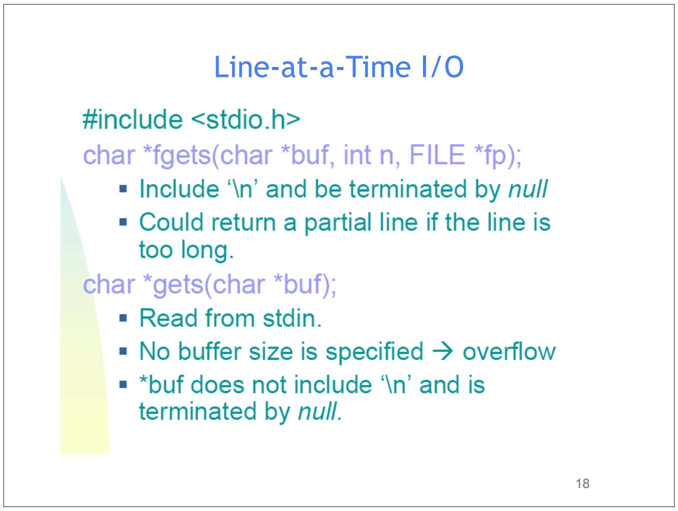
```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

- putchar(c) == putc(c, stdout)
- Differences between putc and fputc
  - putc() can be a macro.



## Example 1

```
#include <stdio.h>
main()
{
    int c;

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



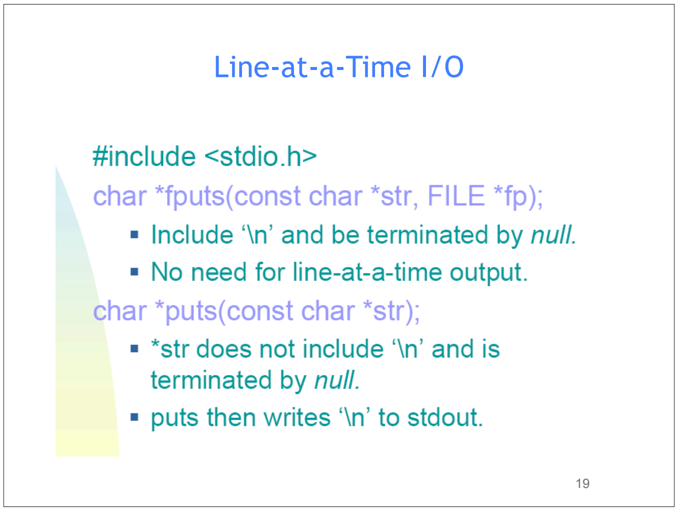
## Line-at-a-Time I/O

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

- Include ‘\n’ and be terminated by *null*
- Could return a partial line if the line is too long.

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

- Read from stdin.
- No buffer size is specified → overflow
- *buf does not include ‘\n’ and is terminated by *null*.



## Line-at-a-Time I/O

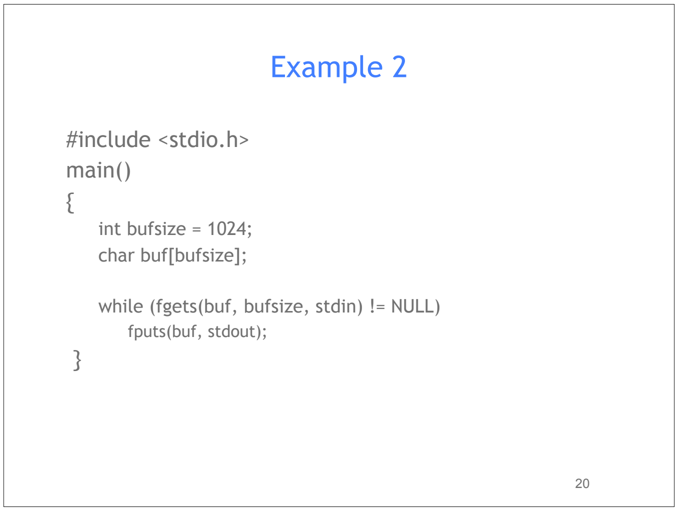
#include <stdio.h>

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

- Include ‘\n’ and be terminated by *null.*
- No need for line-at-a-time output.

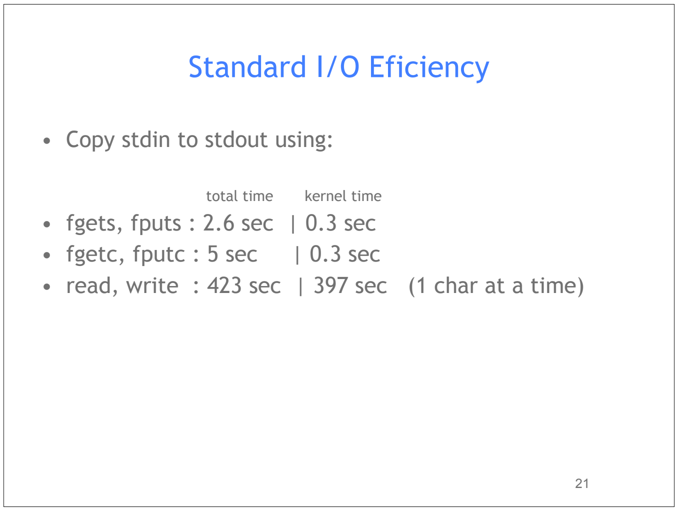
char *puts(const char *str);

- *str does not include ‘\n’ and is terminated by *null.*
- puts then writes ‘\n’ to stdout.



## Example 2

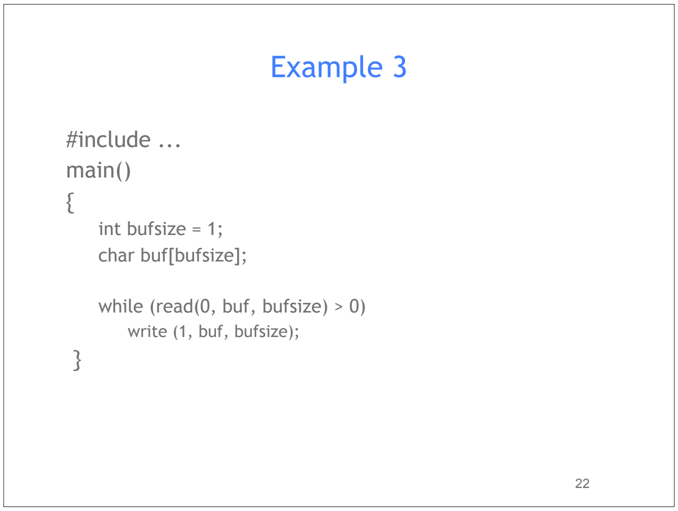
```
#include <stdio.h>
main()
{
    int bufsize = 1024;
    char buf[bufsize];

    while (fgets(buf, bufsize, stdin) != NULL)
        fputs(buf, stdout);
 }
```



## Standard I/O Eficiency

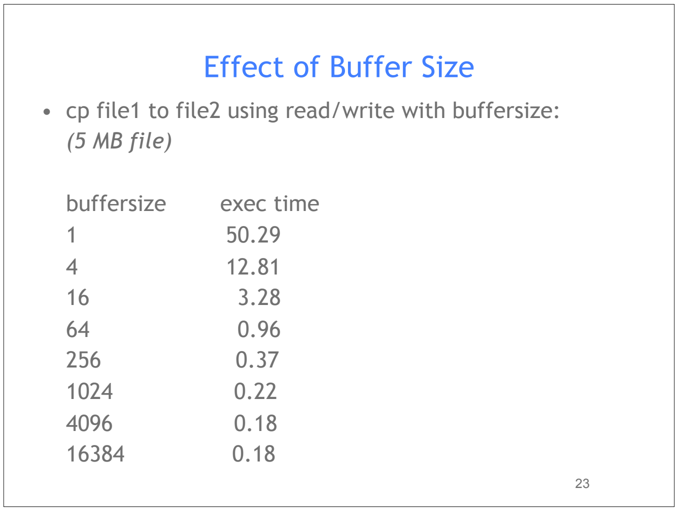
- Copy stdin to stdout using:

|  | total time | kernel time |  |
|---|---|---|---|
| fgets, fputs : | 2.6 sec | 0.3 sec |  |
| fgetc, fputc : | 5 sec | 0.3 sec |  |
| read, write : | 423 sec | 397 sec | (1 char at a time) |



## Example 3

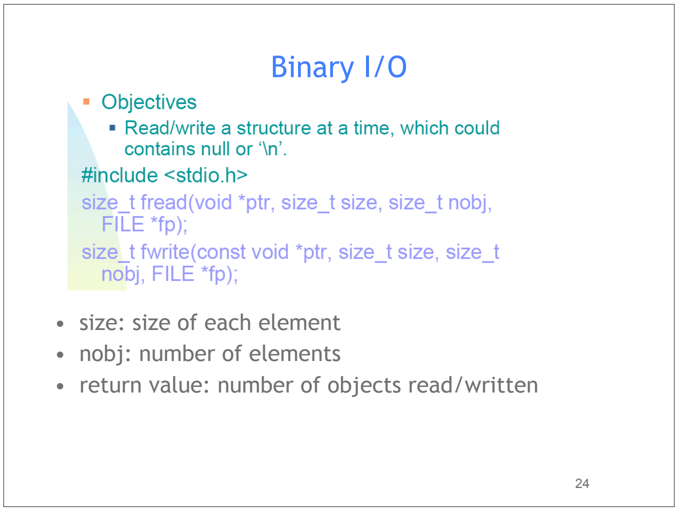
```
#include …
main()
{
    int bufsize = 1;
    char buf[bufsize];

    while (read(0, buf, bufsize) > 0)
        write (1, buf, bufsize);
 }
```



## Effect of Buffer Size

- cp file1 to file2 using read/write with buffersize:
  *(5 MB file)*

| buffersize | exec time |
|---|---|
| 1 | 50.29 |
| 4 | 12.81 |
| 16 | 3.28 |
| 64 | 0.96 |
| 256 | 0.37 |
| 1024 | 0.22 |
| 4096 | 0.18 |
| 16384 | 0.18 |



## Binary I/O

- Objectives
  - Read/write a structure at a time, which could contains null or ‘\n’.

#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);

- size: size of each element
- nobj: number of elements
- return value: number of objects read/written

## Binary I/O

- Not portable for programs using fread and fwrite
  1. The offset of a member in a structure can differ between compilers and systems (due to alignment).
  2. The binary formats for various data types, such as integers, could be different over different machines.

## Binary I/O

- Example 1

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

- Example 2

```
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);
int fseek(FILE *fp, long offset, int whence);
void rewind(FILE *fp);
```

- Assumption: a file’s position can be stored in a long (since Version 7)
- whence: same as lseek
  - Binary files: No requirements for SEEK_END under ANSI C (good under Unix, possible padding for other systems).
  - Text files: SEEK_SET only – 0 or returned value by ftell (different formats for some sys).

## Positioning a Stream

```
#include <stdio.h>
long fgetpos(FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos);
```

- ANSI C standard
- Good for non-Unix systems
- A new data type fpos_t

## Formatted I/O

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

- Overflow is possible for sprintf() – ‘\0’ appended at the end of the string.

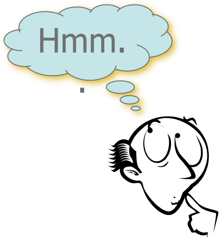
```
int vprintf(const char *format, var_list arg);
int vfprintf(FILE *fp, const char *format, var_list
   arg);
int vsprintf(char *buf, const char *format,
   var_list arg);
```

## Formatted I/O

```
#include <stdio.h>
int scanf(const char *format, …);
int fscanf(FILE *fp, const char
   *format, …);
int sscanf(char *buf, const char
   *format, …);
```

## Summary

- Buffered File I/O
  - opening and closing streams
  - reading from / writing to streams
  - Binary I/O
  - Formatted I/O



- Next Lecture: Files & Directories

- Read Ch.5 from Stevens

## Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), and B. Knicki (WPI).