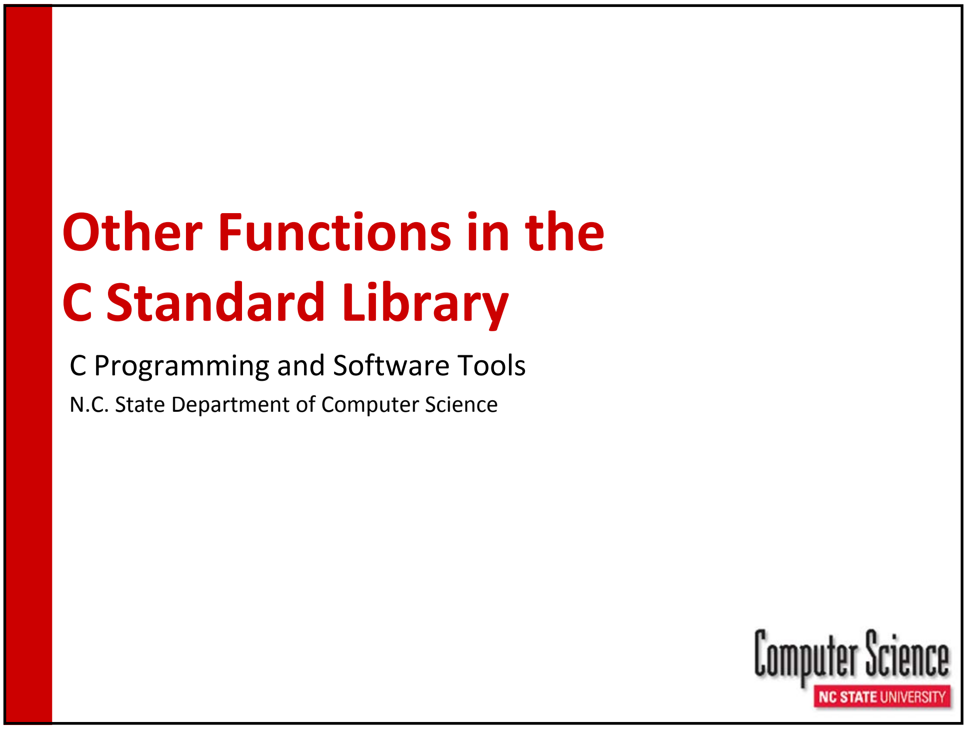
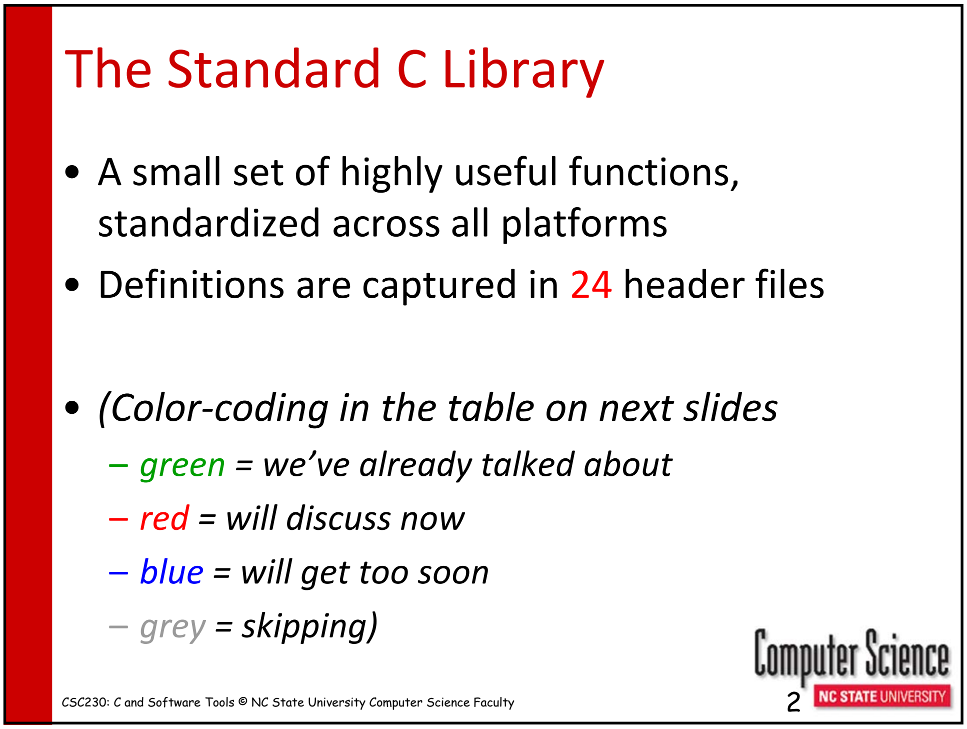


Other Functions in the C Standard Library

C Programming and Software Tools

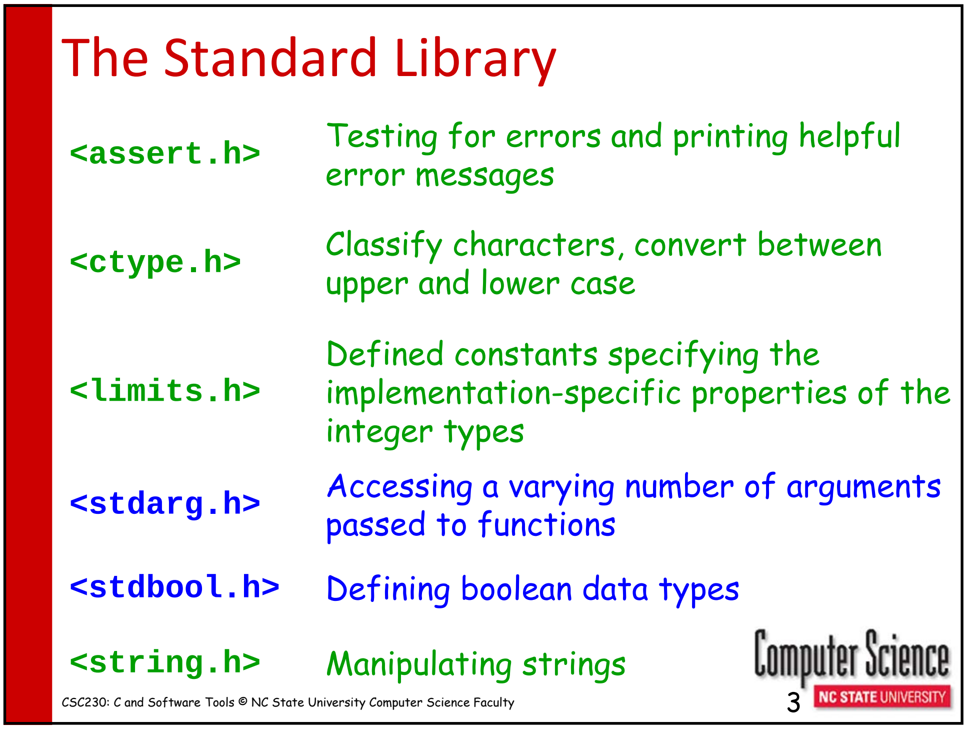
N.C. State Department of Computer Science



The Standard C Library

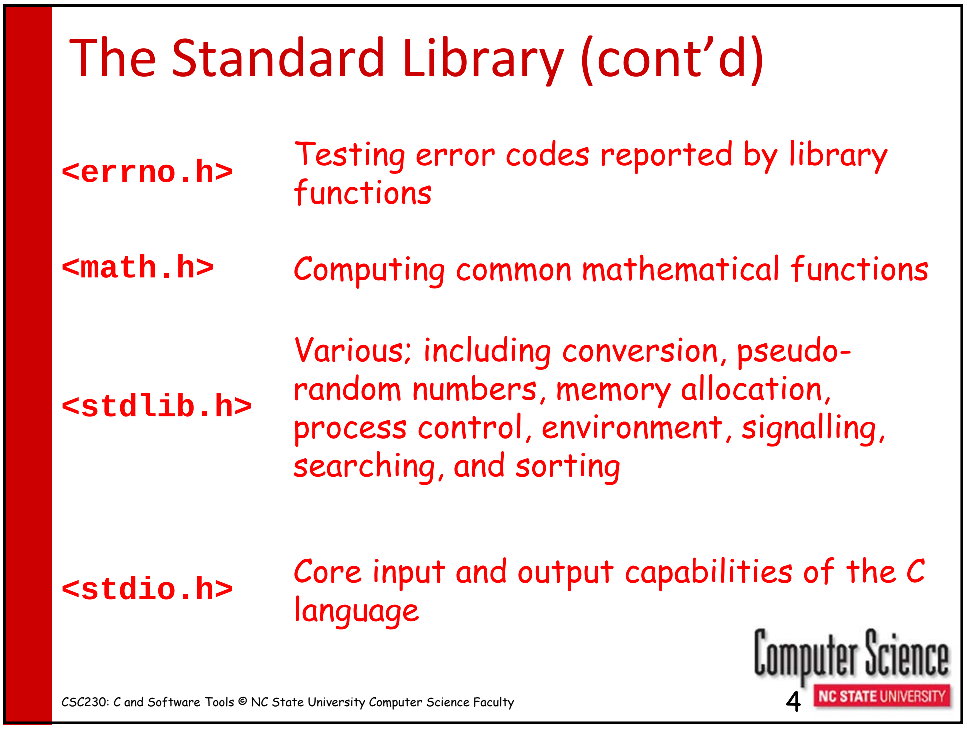
- A small set of highly useful functions, standardized across all platforms
- Definitions are captured in 24 header files

- *(Color-coding in the table on next slides*
  - *green = we've already talked about*
  - *red = will discuss now*
  - *blue = will get too soon*
  - *grey = skipping)*



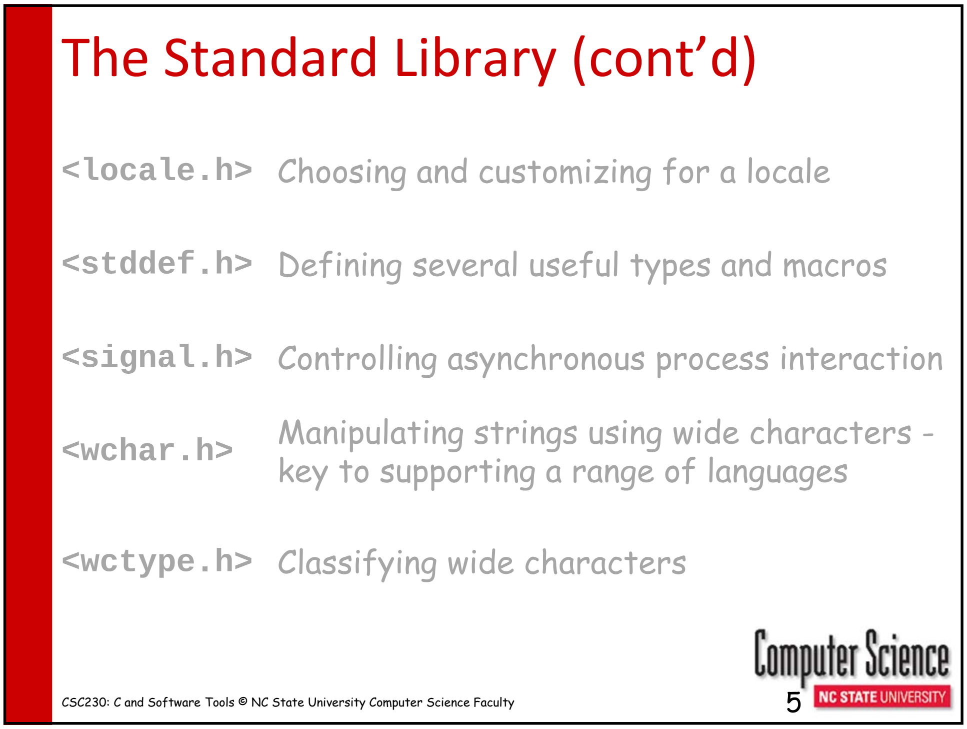
The Standard Library

`<assert.h>`	Testing for errors and printing helpful error messages
`<ctype.h>`	Classify characters, convert between upper and lower case
`<limits.h>`	Defined constants specifying the implementation-specific properties of the integer types
`<stdarg.h>`	Accessing a varying number of arguments passed to functions
`<stdbool.h>`	Defining boolean data types
`<string.h>`	Manipulating strings



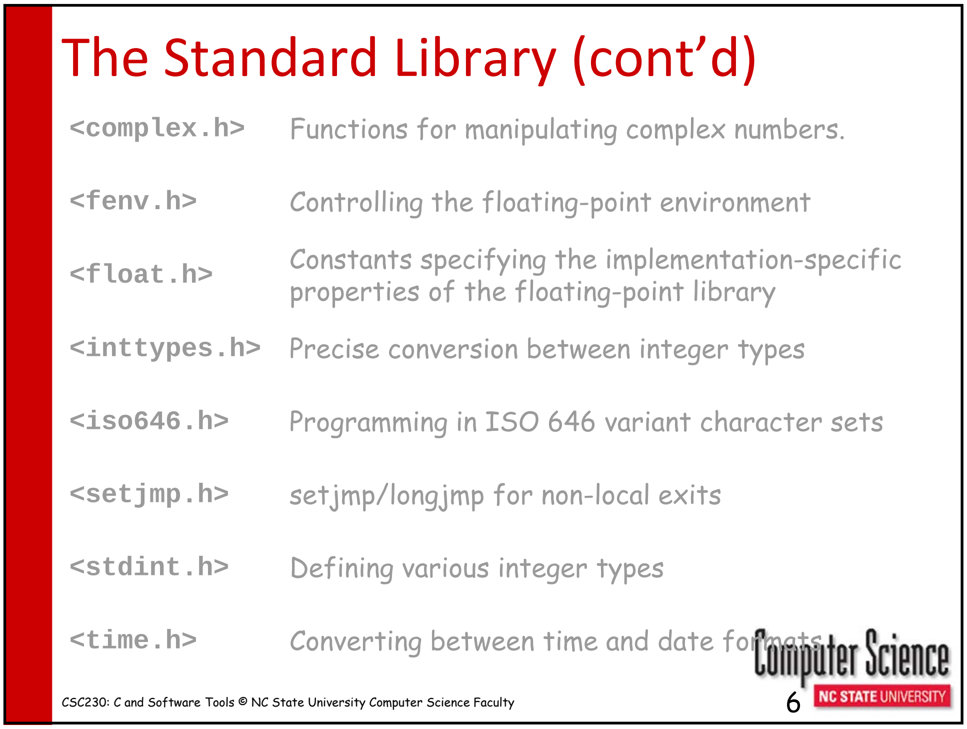
The Standard Library (cont'd)

`<errno.h>`	Testing error codes reported by library functions
`<math.h>`	Computing common mathematical functions
`<stdlib.h>`	Various; including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting
`<stdio.h>`	Core input and output capabilities of the C language



The Standard Library (cont’d)

`<locale.h>`	Choosing and customizing for a locale
`<stddef.h>`	Defining several useful types and macros
`<signal.h>`	Controlling asynchronous process interaction
`<wchar.h>`	Manipulating strings using wide characters - key to supporting a range of languages
`<wctype.h>`	Classifying wide characters



The Standard Library (cont’d)

`<complex.h>`	Functions for manipulating complex numbers.
`<fenv.h>`	Controlling the floating-point environment
`<float.h>`	Constants specifying the implementation-specific properties of the floating-point library
`<inttypes.h>`	Precise conversion between integer types
`<iso646.h>`	Programming in ISO 646 variant character sets
`<setjmp.h>`	setjmp/longjmp for non-local exits
`<stdint.h>`	Defining various integer types
`<time.h>`	Converting between time and date formats

<math.h>: Using the Math Library

- Note: to use math functions, be sure you specify **-lm** to gcc (after the source file names)

```
> gcc pgmx.c -lm -o pgmx
```

- Some constants

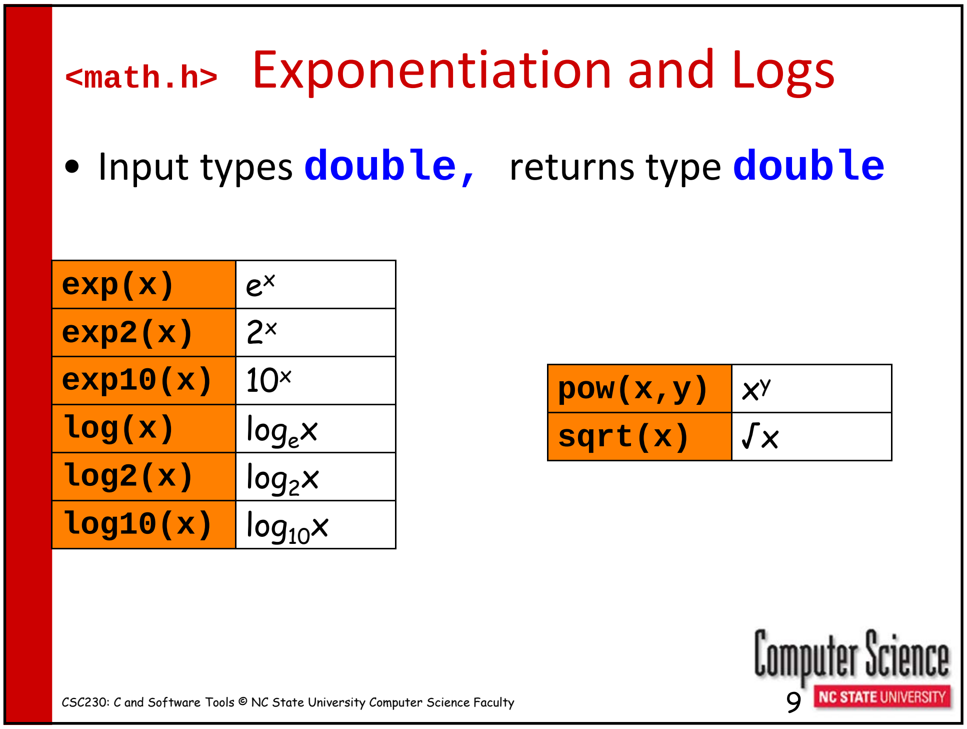
M_E	The base of natural logarithms
M_PI	Value of π
M_SQRT2	Square root of 2

- See King, Chapter 23.4 for additional resources

<math.h> Trigonometry

- Input type is a **double**, returns type **double**

cos(), **sin(),** **tan()**	Input in radians Return value in [-1.0, 1.0]
acos(), **asin(),** **atan()**	Input in [-1.0, 1.0] Return value in [-π, π] radians

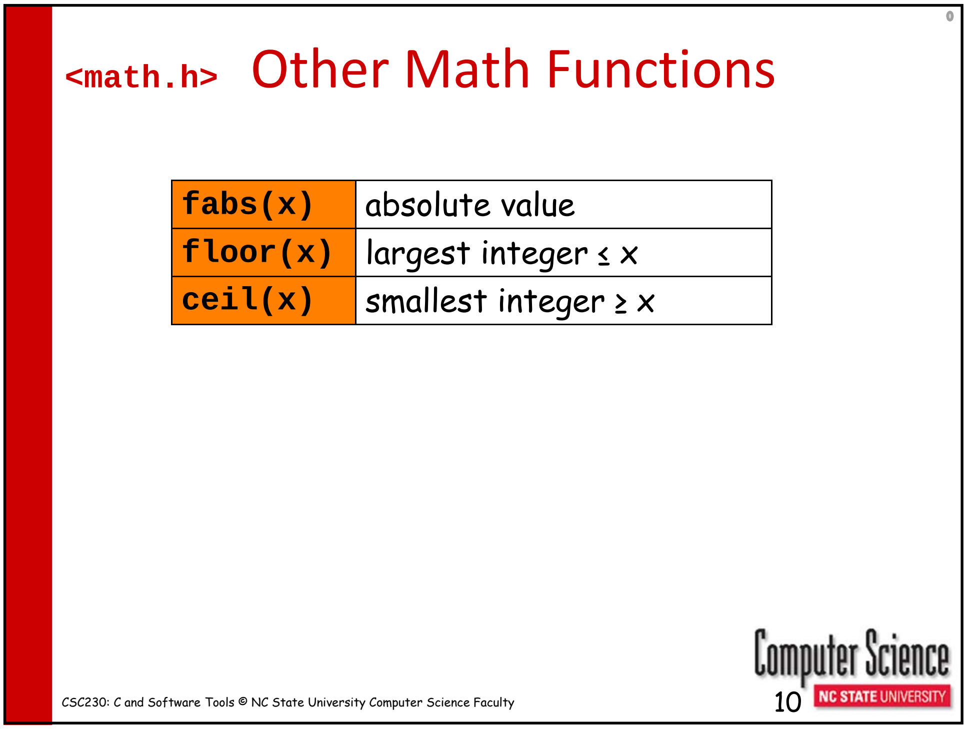


`<math.h>` Exponentiation and Logs

- Input types **`double`,** returns type **`double`**

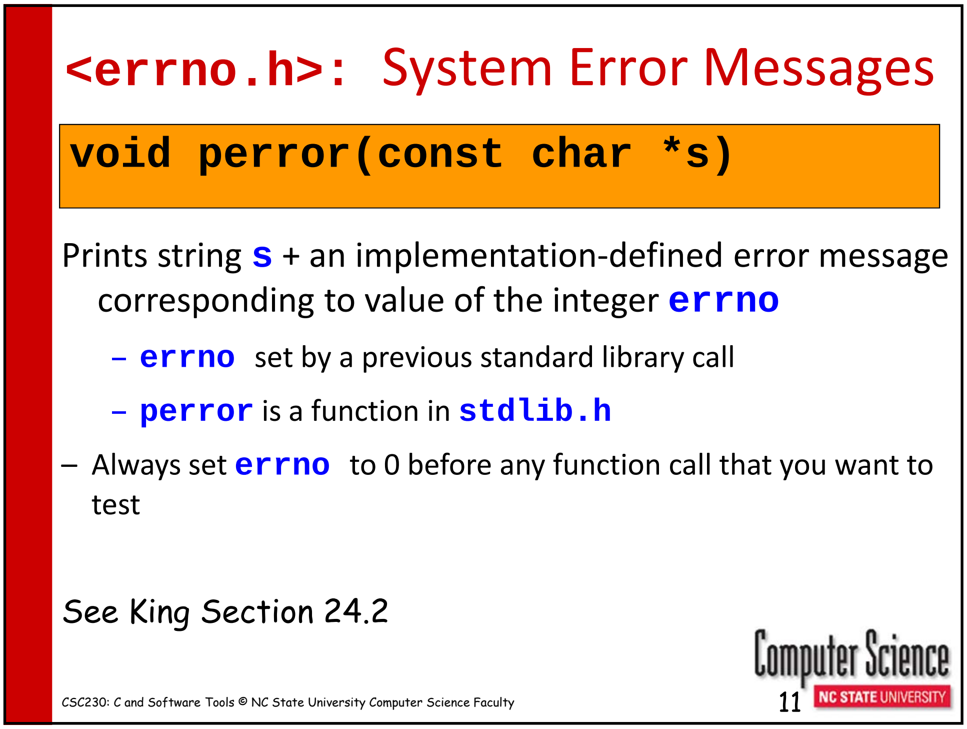
`exp(x)`	e^x
`exp2(x)`	2^x
`exp10(x)`	10^x
`log(x)`	$\log_e x$
`log2(x)`	$\log_2 x$
`log10(x)`	$\log_{10} x$

`pow(x,y)`	x^y
`sqrt(x)`	$\sqrt{x}$



`<math.h>` Other Math Functions

`fabs(x)`	absolute value
`floor(x)`	largest integer ≤ x
`ceil(x)`	smallest integer ≥ x



<errno.h>: System Error Messages

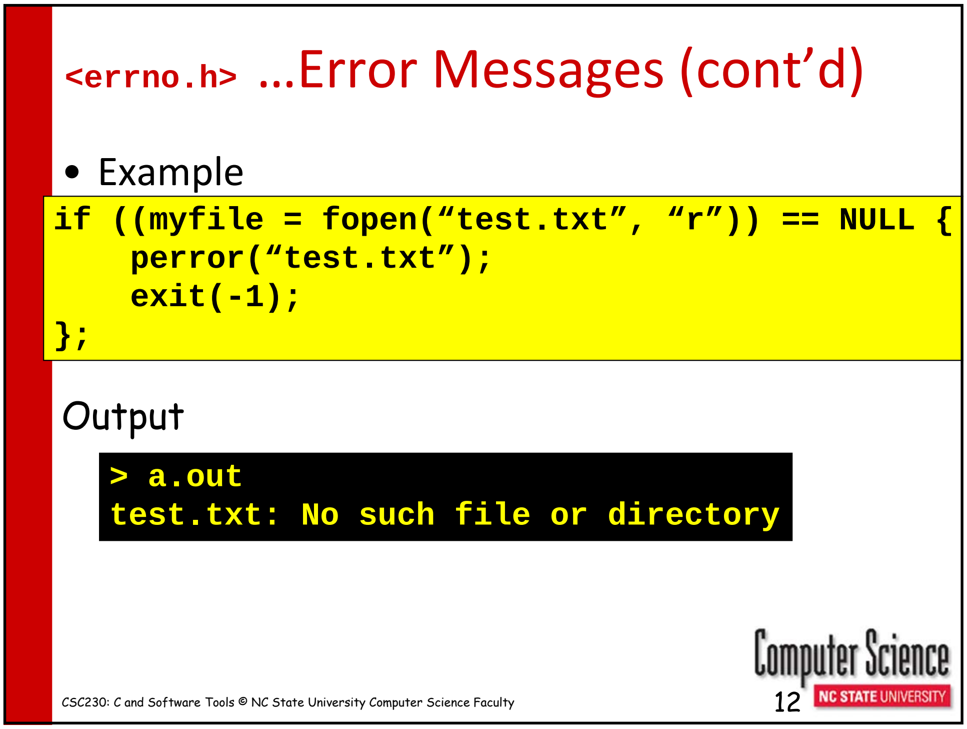
```
void perror(const char *s)
```

Prints string **s** + an implementation-defined error message corresponding to value of the integer **errno**

- **errno** set by a previous standard library call
- **perror** is a function in **stdlib.h**

– Always set **errno** to 0 before any function call that you want to test

See King Section 24.2



<errno.h> ...Error Messages (cont'd)

- Example

```
if ((myfile = fopen(“test.txt”, “r”)) == NULL {
    perror(“test.txt”);
    exit(-1);
};
```

Output

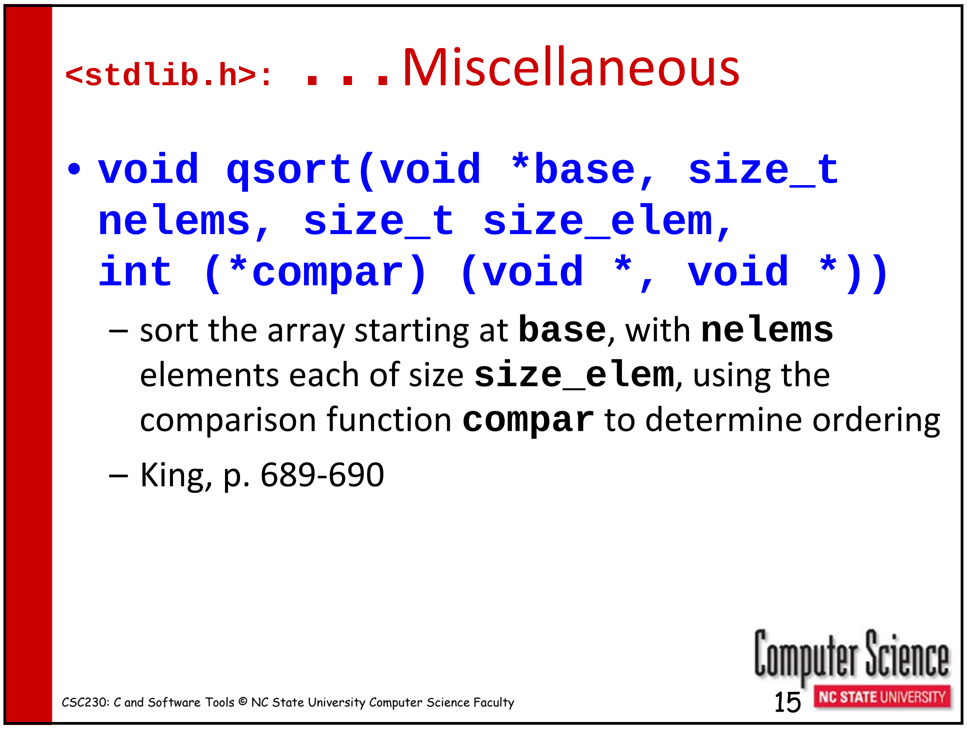
```
> a.out
test.txt: No such file or directory
```

`<math.h>` Errors

- Domain Error: argument outside domain
  - EDOM is stored in errno
  - Function may return NaN, but return value is implementation defined
- Range Error: result is outside range of double
  - ERANGE is stored in errno if overflow (may be stored for underflow)
  - Function returns + or – HUGE_VAL if overflow
  - Function returns 0 for underflow

`<stdlib.h>:` Miscellaneous

- `void abort(void)`, `void exit(int status)`
  - terminate execution, terminate with a non-zero return code
- `void * bsearch(void * key, void *base, size_t nelems, size_t size_elem, int (*compar) (void *, void *))`
  - binary search in a sorted array starting at **`base`**, with **`nelems`** elements each of size **`size_elem`**, looking for the item with value **`key`**, using the comparison function **`compar`** to determine equality
  - King, p. 689-690

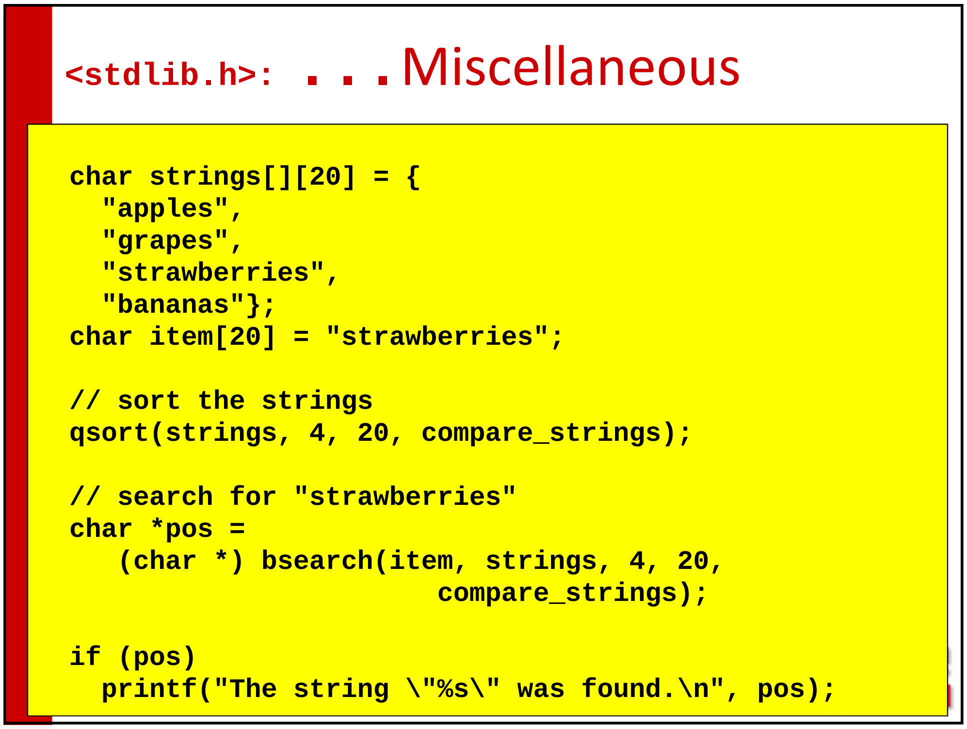


<stdlib.h>: . . . Miscellaneous

- **void qsort(void *base, size_t nelems, size_t size_elem, int (*compar) (void *, void *))**
  - sort the array starting at **base**, with **nelems** elements each of size **size_elem**, using the comparison function **compar** to determine ordering
  - King, p. 689-690

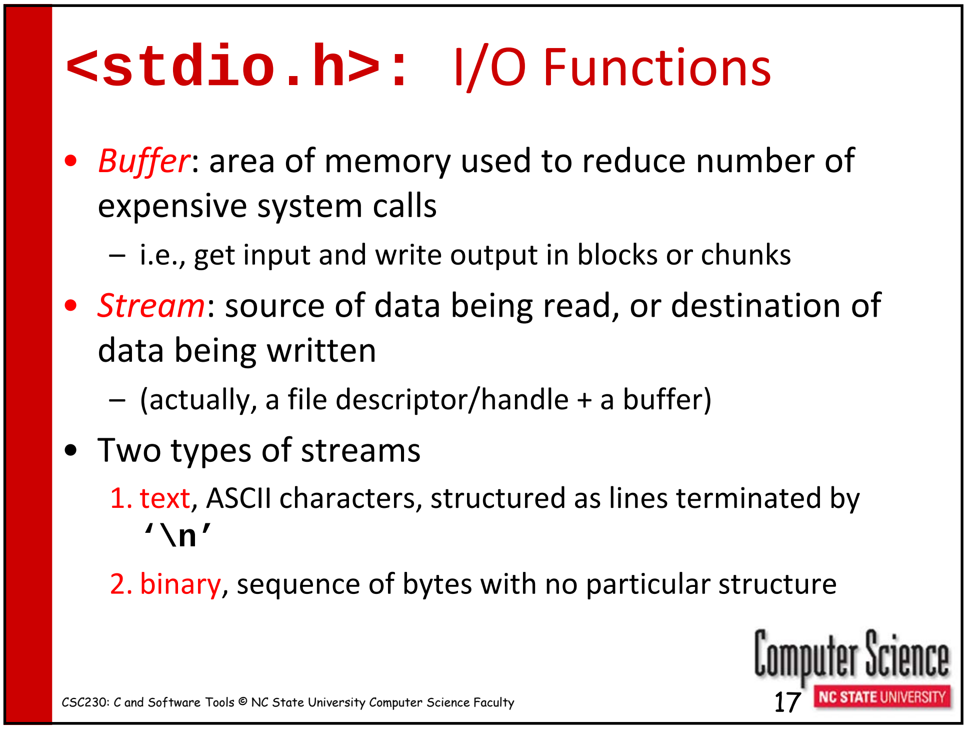
CSC230: C and Software Tools © NC State University Computer Science Faculty

15



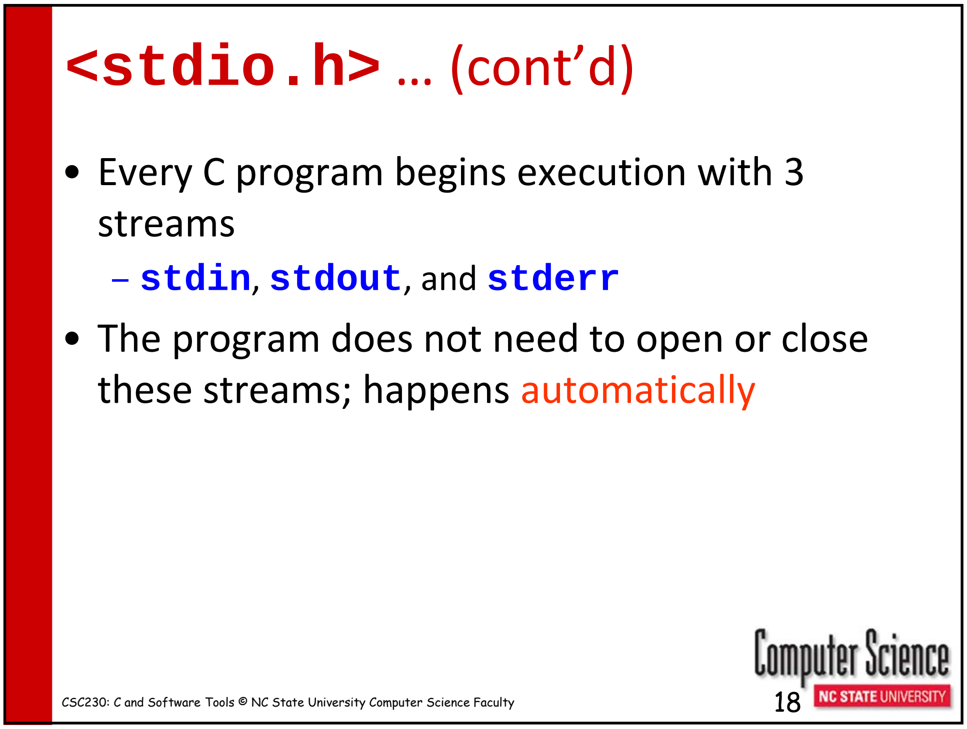
<stdlib.h>: . . . Miscellaneous

```
char strings[][20] = {
  "apples",
  "grapes",
  "strawberries",
  "bananas"};
char item[20] = "strawberries";

// sort the strings
qsort(strings, 4, 20, compare_strings);

// search for "strawberries"
char *pos =
   (char *) bsearch(item, strings, 4, 20,
                       compare_strings);

if (pos)
  printf("The string \"%s\" was found.\n", pos);
```



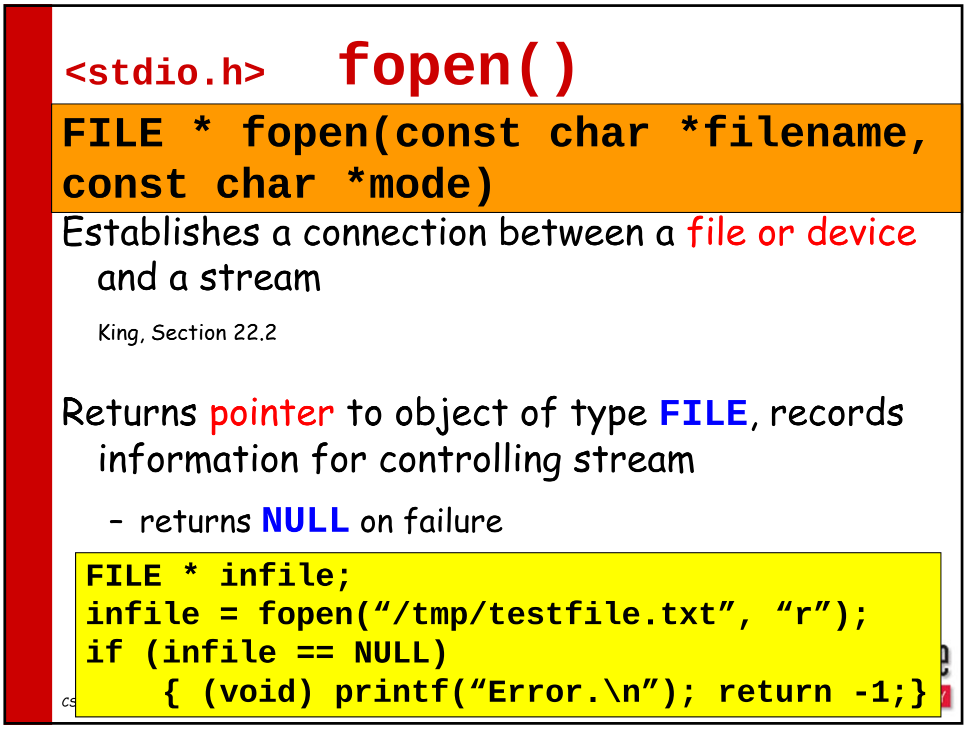
`<stdio.h>:` I/O Functions

- *Buffer*: area of memory used to reduce number of expensive system calls
  - i.e., get input and write output in blocks or chunks
- *Stream*: source of data being read, or destination of data being written
  - (actually, a file descriptor/handle + a buffer)
- Two types of streams
  1. text, ASCII characters, structured as lines terminated by `'\n'`
  2. binary, sequence of bytes with no particular structure



`<stdio.h>` … (cont'd)

- Every C program begins execution with 3 streams
  - `stdin`, `stdout`, and `stderr`
- The program does not need to open or close these streams; happens automatically



<stdio.h> fopen()

FILE * fopen(const char *filename, const char *mode)

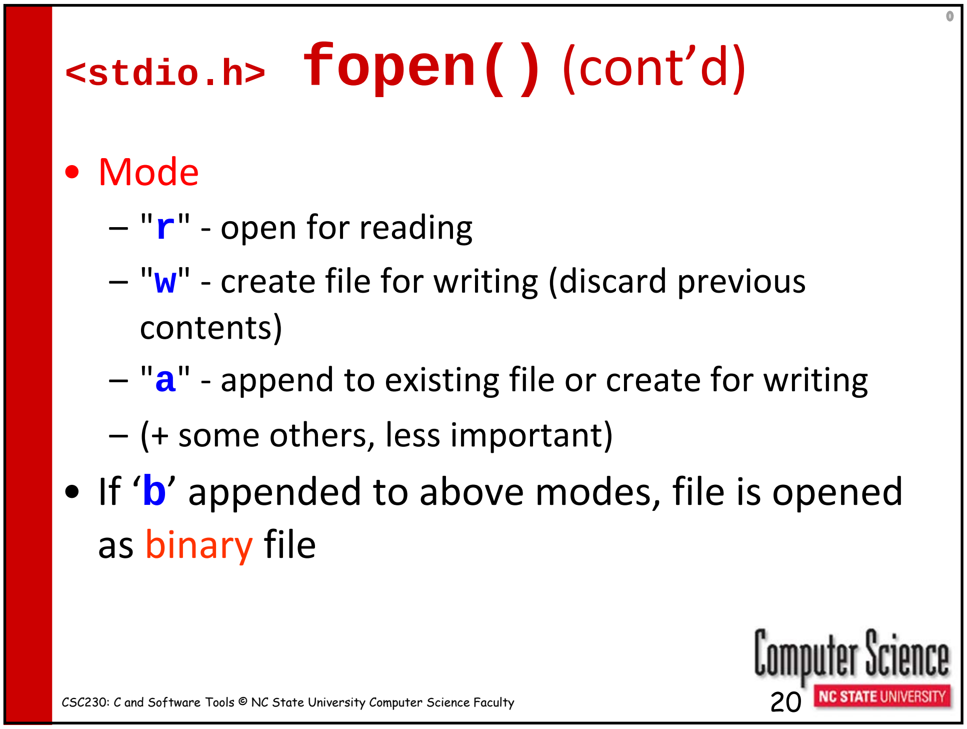
Establishes a connection between a file or device and a stream

King, Section 22.2

Returns pointer to object of type **FILE**, records information for controlling stream

- returns **NULL** on failure

```
FILE * infile;
infile = fopen(“/tmp/testfile.txt”, “r”);
if (infile == NULL)
    { (void) printf(“Error.\n”); return -1;}
```



<stdio.h> fopen() (cont’d)

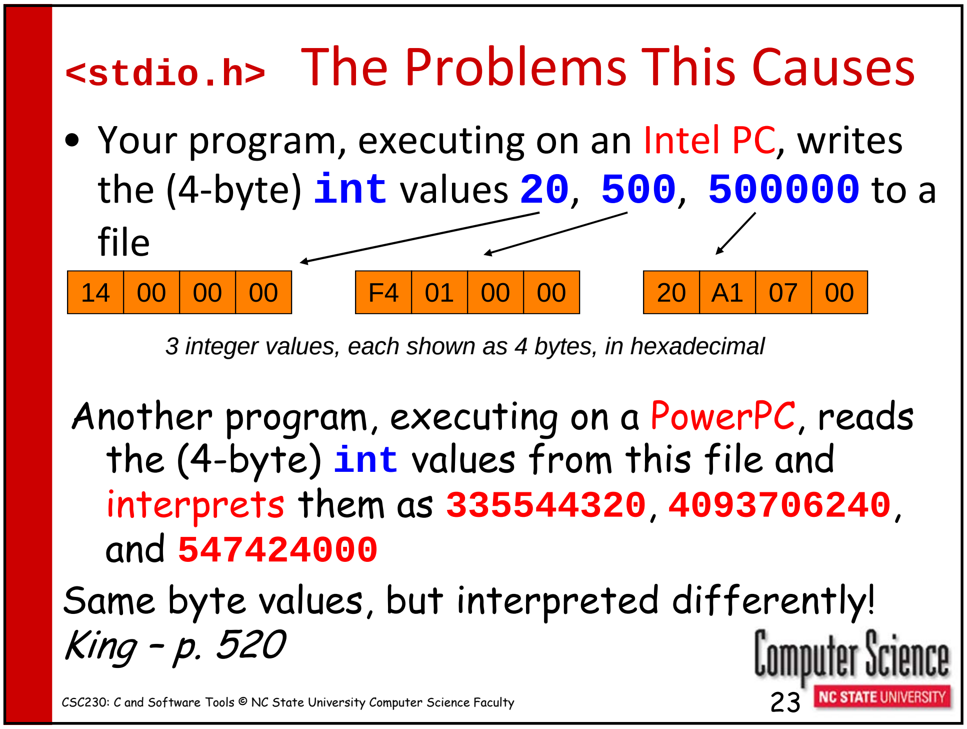
- Mode
  - "**r**" - open for reading
  - "**w**" - create file for writing (discard previous contents)
  - "**a**" - append to existing file or create for writing
  - (+ some others, less important)
- If ‘**b**’ appended to above modes, file is opened as binary file

`<stdio.h>` Binary Files

- Needed if
  - non-ASCII data, or
  - need to handle differences between outputs produced by different platforms (e.g., Windows ↔ Linux)
- Examples of binary files
  - images: .bmp, .gif, .jpg, .tif
  - audio: .wav, .ac3
  - video: .avi
  - word processing: .rtf
  - encrypted files
  - etc.

`<stdio.h>` Byte-Ordering

- Different architectures store the bytes of a word in different orders
- What's an *architecture*? Type of processor
  - Ex.: Intel, PowerPC, ARM, VIA, CELL, etc.
- What's a *word*? Primitive datatypes of a language
  - Ex.: **`int`**, **`short int`**, **`float`**, **`double`**, …



<stdio.h> The Problems This Causes

- Your program, executing on an Intel PC, writes the (4-byte) `int` values **20**, **500**, **500000** to a file

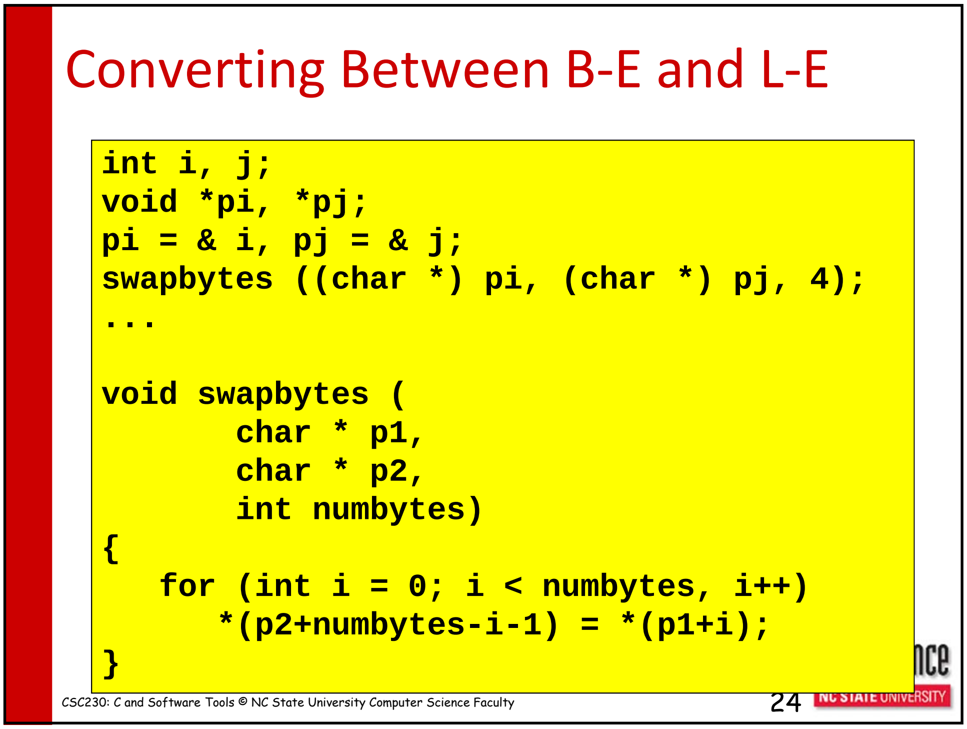
14	00	00	00	F4	01	00	00	20	A1	07	00

3 integer values, each shown as 4 bytes, in hexadecimal

Another program, executing on a PowerPC, reads the (4-byte) `int` values from this file and interprets them as **335544320**, **4093706240**, and **547424000**

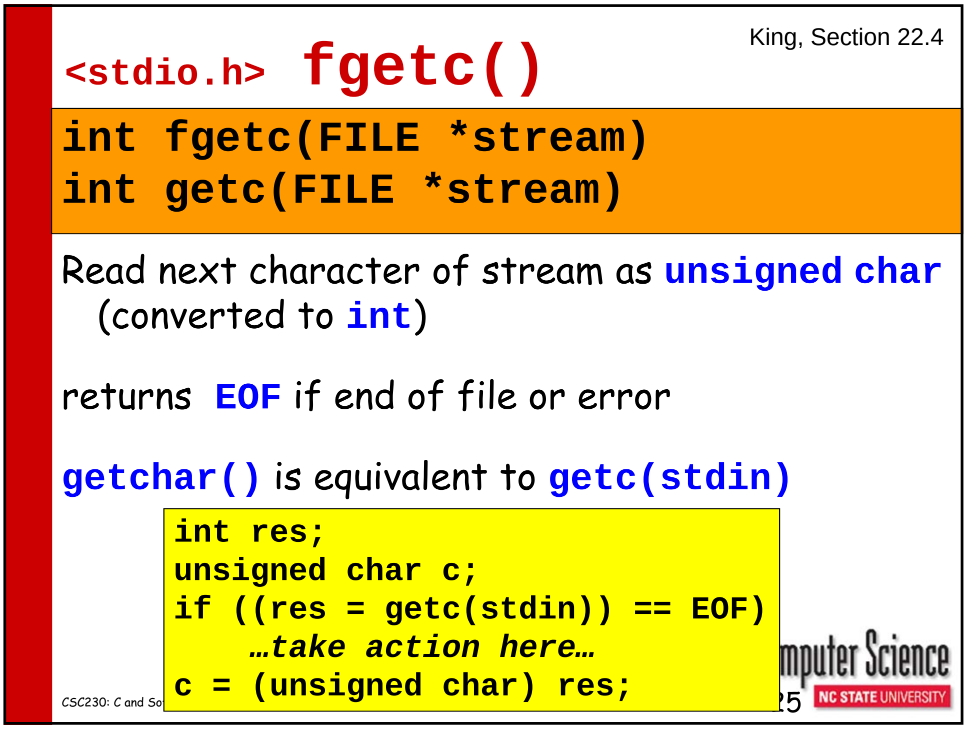
Same byte values, but interpreted differently!

King – p. 520



Converting Between B-E and L-E

```
int i, j;
void *pi, *pj;
pi = & i, pj = & j;
swapbytes ((char *) pi, (char *) pj, 4);
...

void swapbytes (
       char * p1,
       char * p2,
       int numbytes)
{
   for (int i = 0; i < numbytes, i++)
      *(p2+numbytes-i-1) = *(p1+i);
}
```



King, Section 22.4

<stdio.h> fgetc()

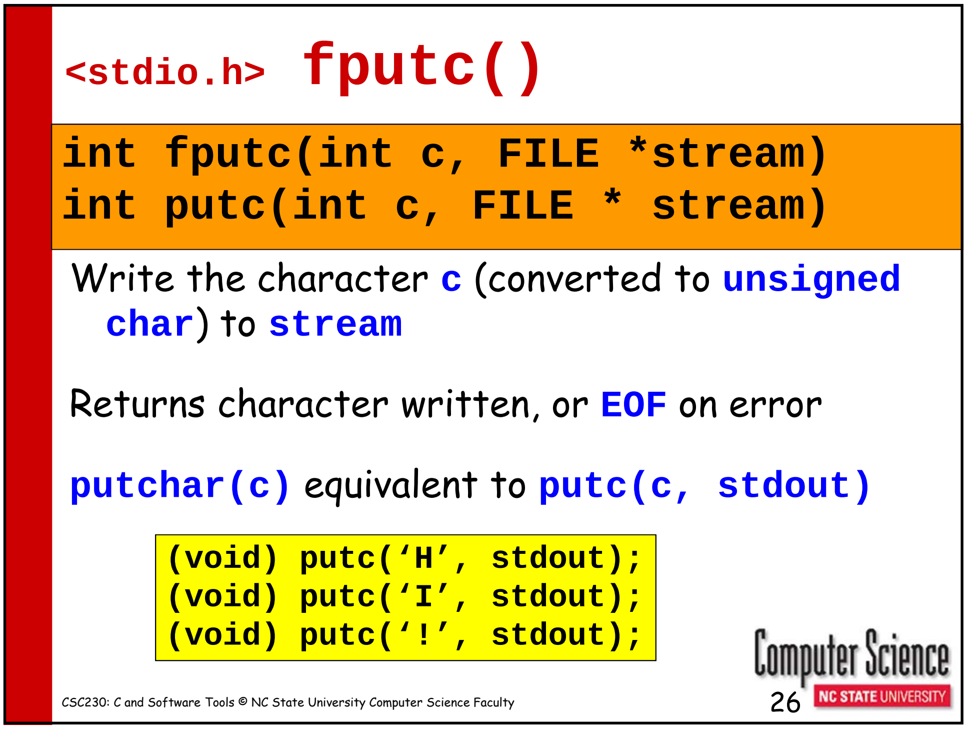
```
int fgetc(FILE *stream)
int getc(FILE *stream)
```

Read next character of stream as **unsigned char** (converted to **int**)

returns **EOF** if end of file or error

getchar() is equivalent to **getc(stdin)**

```
int res;
unsigned char c;
if ((res = getc(stdin)) == EOF)
    …take action here…
c = (unsigned char) res;
```



<stdio.h> fputc()

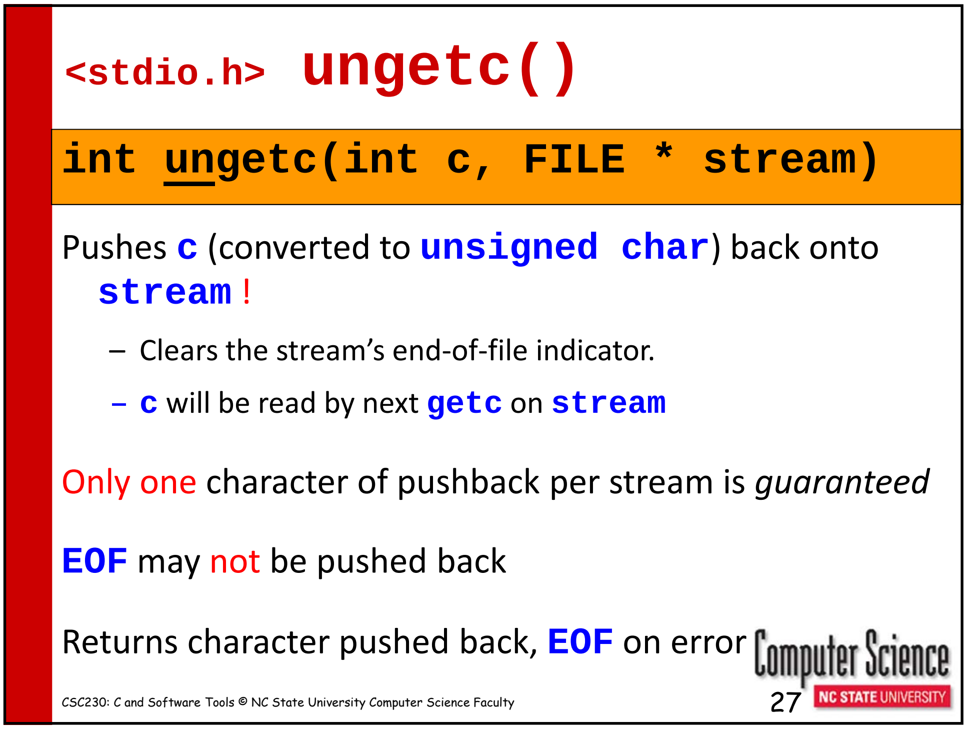
```
int fputc(int c, FILE *stream)
int putc(int c, FILE * stream)
```

Write the character **c** (converted to **unsigned char**) to **stream**

Returns character written, or **EOF** on error

putchar(c) equivalent to **putc(c, stdout)**

```
(void) putc(‘H’, stdout);
(void) putc(‘I’, stdout);
(void) putc(‘!’, stdout);
```



<stdio.h> ungetc()

```
int ungetc(int c, FILE * stream)
```

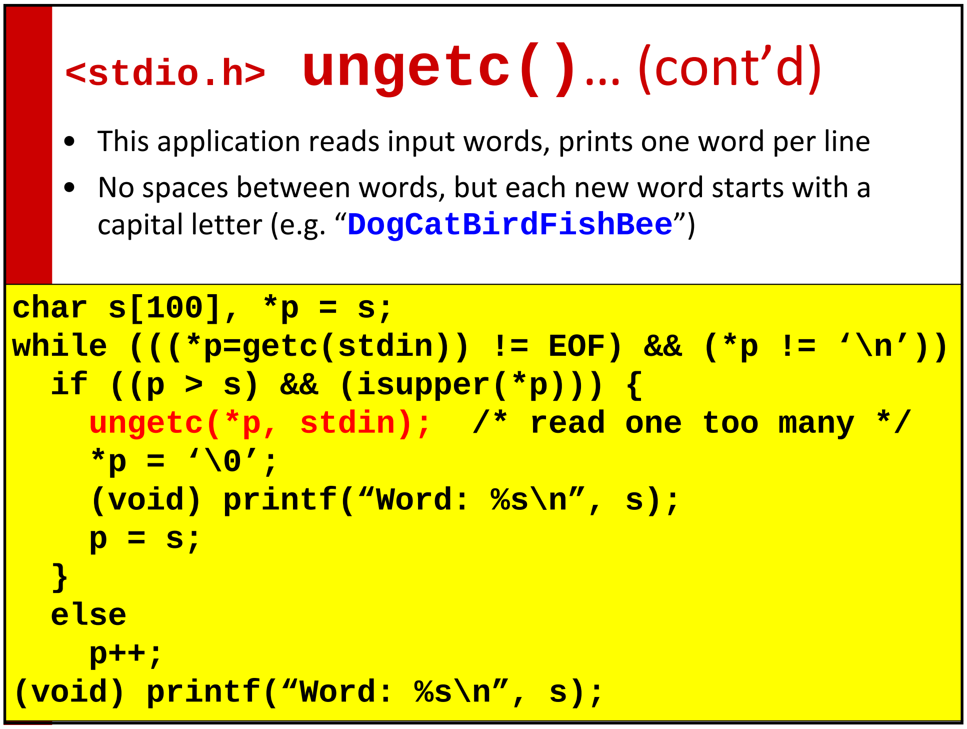
Pushes **c** (converted to **unsigned char**) back onto **stream**!

- Clears the stream’s end-of-file indicator.
- **c** will be read by next **getc** on **stream**

Only one character of pushback per stream is *guaranteed*

EOF may not be pushed back

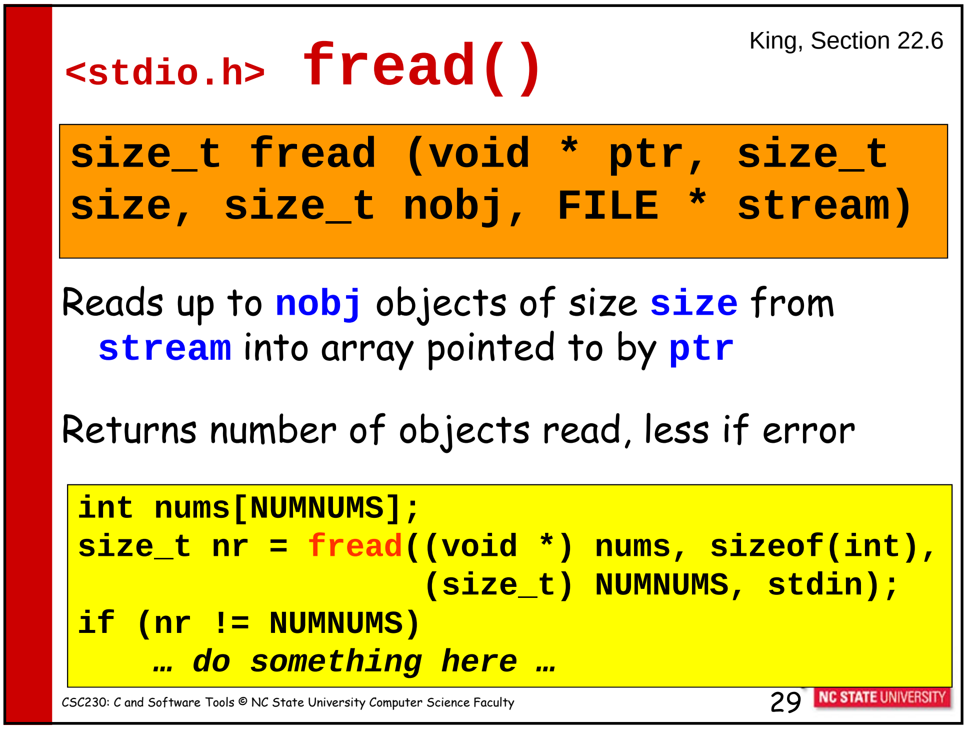
Returns character pushed back, **EOF** on error



<stdio.h> ungetc()… (cont’d)

- This application reads input words, prints one word per line
- No spaces between words, but each new word starts with a capital letter (e.g. “**DogCatBirdFishBee**”)

```
char s[100], *p = s;
while (((*p=getc(stdin)) != EOF) && (*p != ‘\n’))
  if ((p > s) && (isupper(*p))) {
    ungetc(*p, stdin);  /* read one too many */
    *p = ‘\0’;
    (void) printf(“Word: %s\n”, s);
    p = s;
  }
  else
    p++;
(void) printf(“Word: %s\n”, s);
```



King, Section 22.6

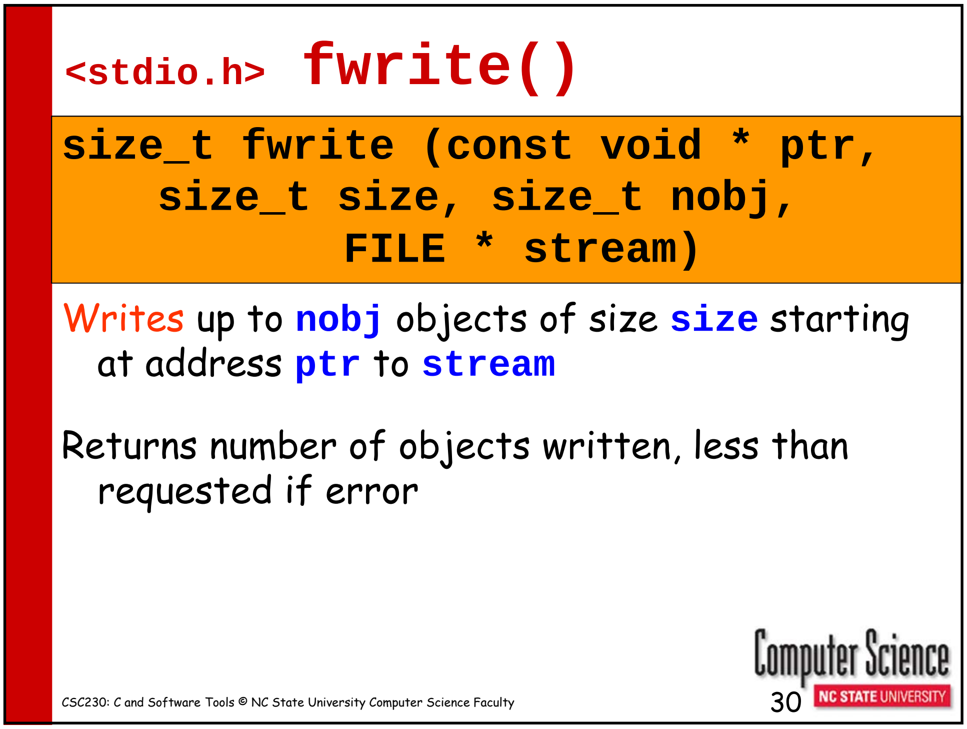
<stdio.h> fread()

```
size_t fread (void * ptr, size_t
size, size_t nobj, FILE * stream)
```

Reads up to **nobj** objects of size **size** from **stream** into array pointed to by **ptr**

Returns number of objects read, less if error

```
int nums[NUMNUMS];
size_t nr = fread((void *) nums, sizeof(int),
                  (size_t) NUMNUMS, stdin);
if (nr != NUMNUMS)
    … do something here …
```

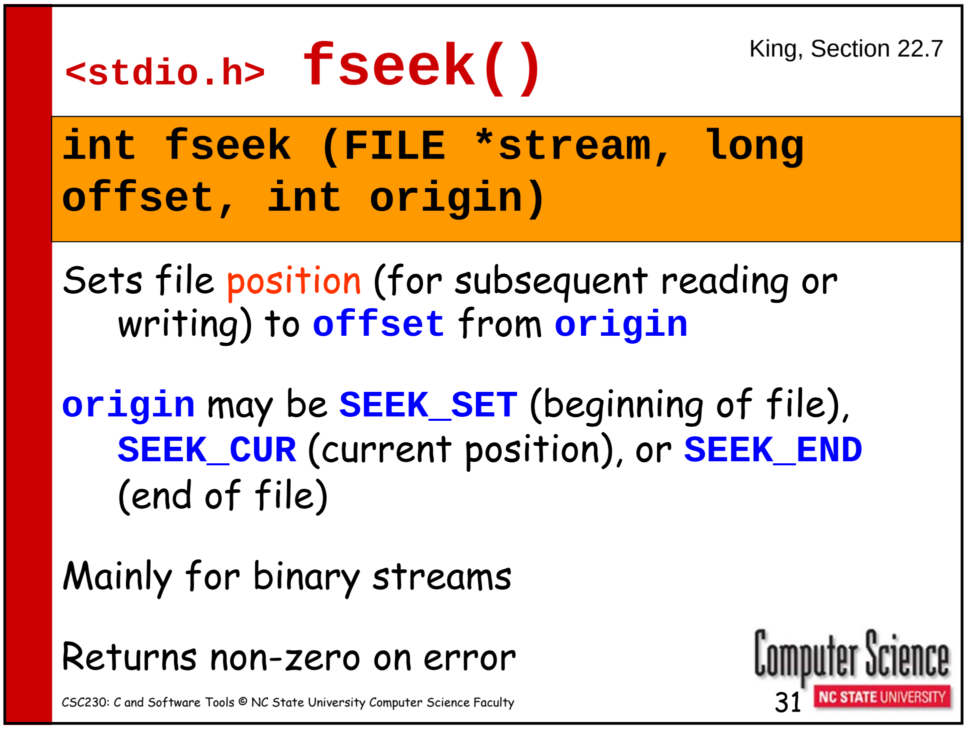


<stdio.h> fwrite()

```
size_t fwrite (const void * ptr,
     size_t size, size_t nobj,
               FILE * stream)
```

Writes up to **nobj** objects of size **size** starting at address **ptr** to **stream**

Returns number of objects written, less than requested if error



King, Section 22.7

<stdio.h> fseek()

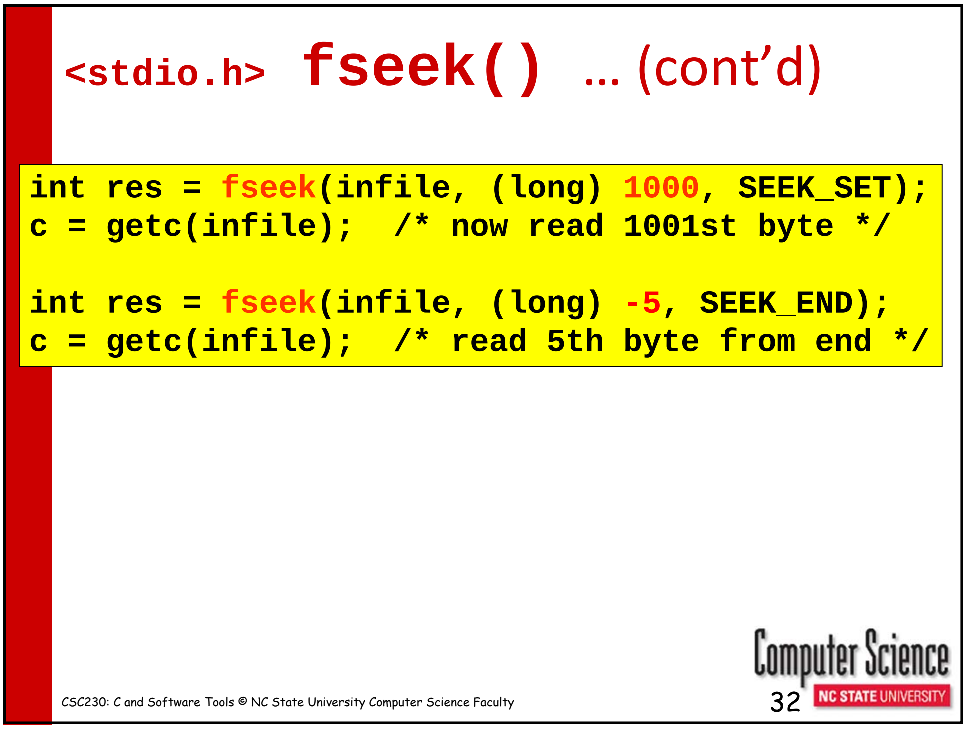
```
int fseek (FILE *stream, long
offset, int origin)
```

Sets file position (for subsequent reading or writing) to offset from origin

origin may be SEEK_SET (beginning of file), SEEK_CUR (current position), or SEEK_END (end of file)

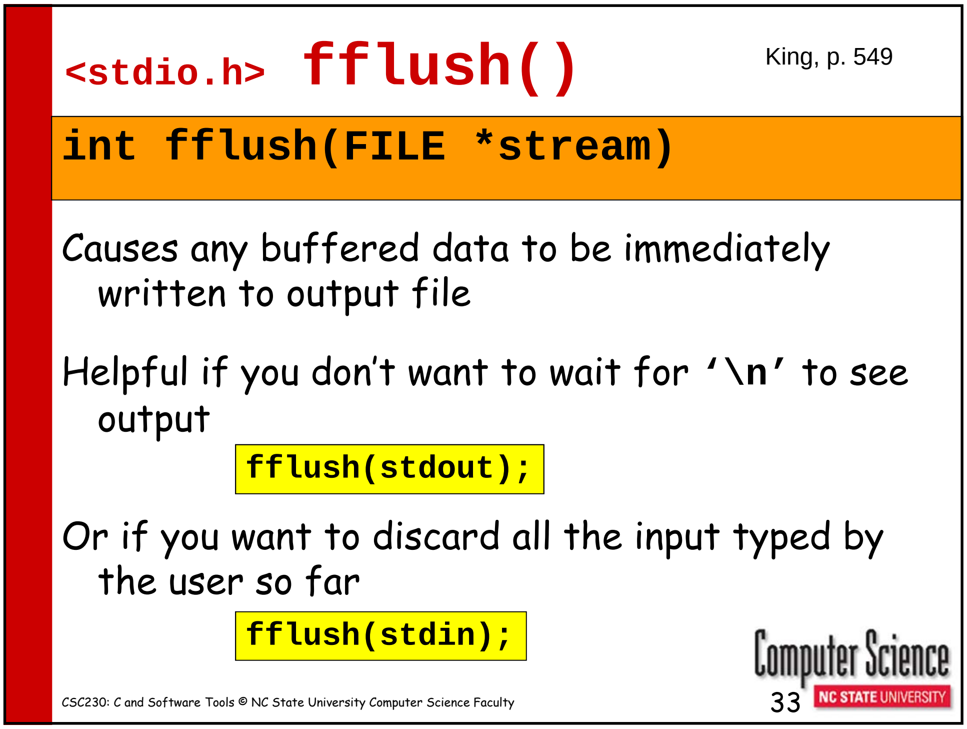
Mainly for binary streams

Returns non-zero on error



<stdio.h> fseek() … (cont'd)

```
int res = fseek(infile, (long) 1000, SEEK_SET);
c = getc(infile);  /* now read 1001st byte */

int res = fseek(infile, (long) -5, SEEK_END);
c = getc(infile);  /* read 5th byte from end */
```



<stdio.h> fflush()

King, p. 549

```
int fflush(FILE *stream)
```

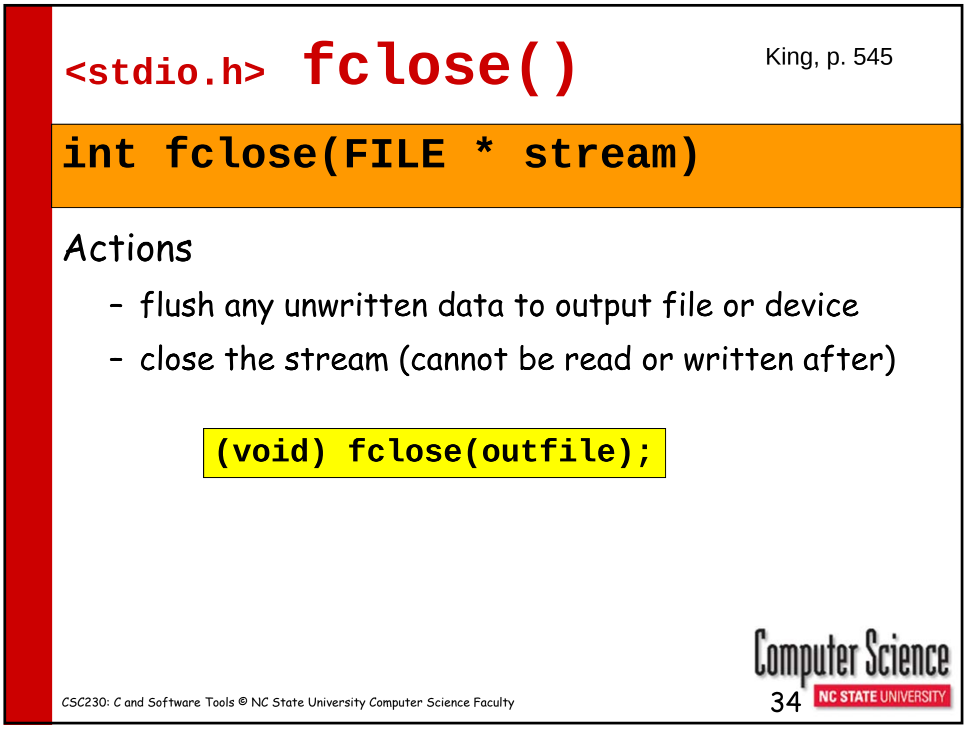
Causes any buffered data to be immediately written to output file

Helpful if you don't want to wait for '**\n**' to see output

```
fflush(stdout);
```

Or if you want to discard all the input typed by the user so far

```
fflush(stdin);
```



<stdio.h> fclose()

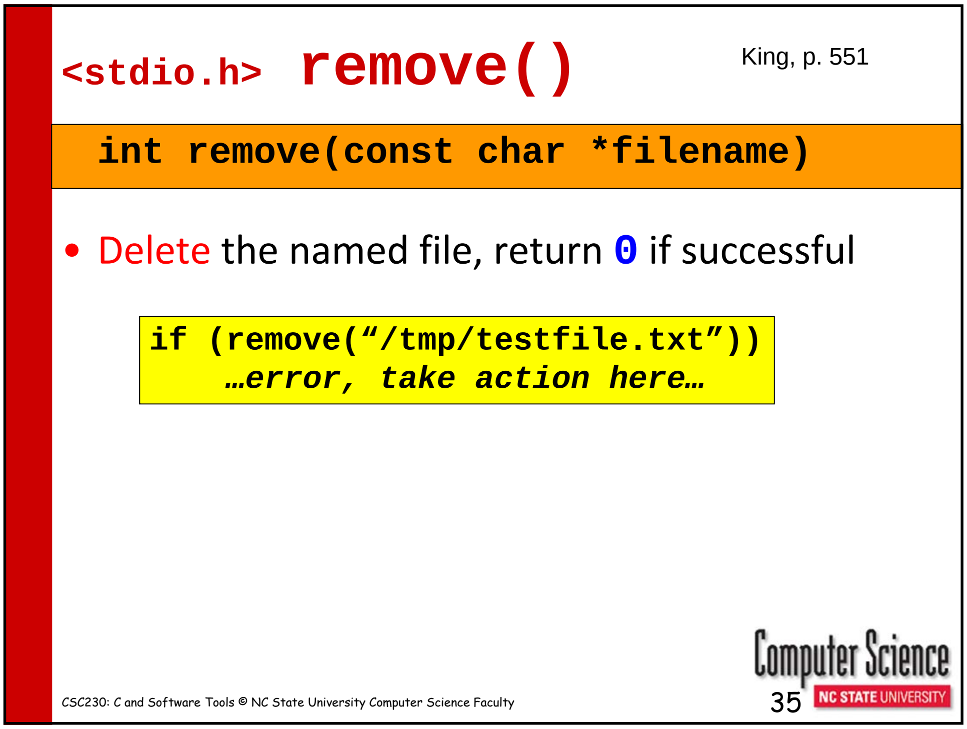
King, p. 545

```
int fclose(FILE * stream)
```

Actions

- flush any unwritten data to output file or device
- close the stream (cannot be read or written after)

```
(void) fclose(outfile);
```



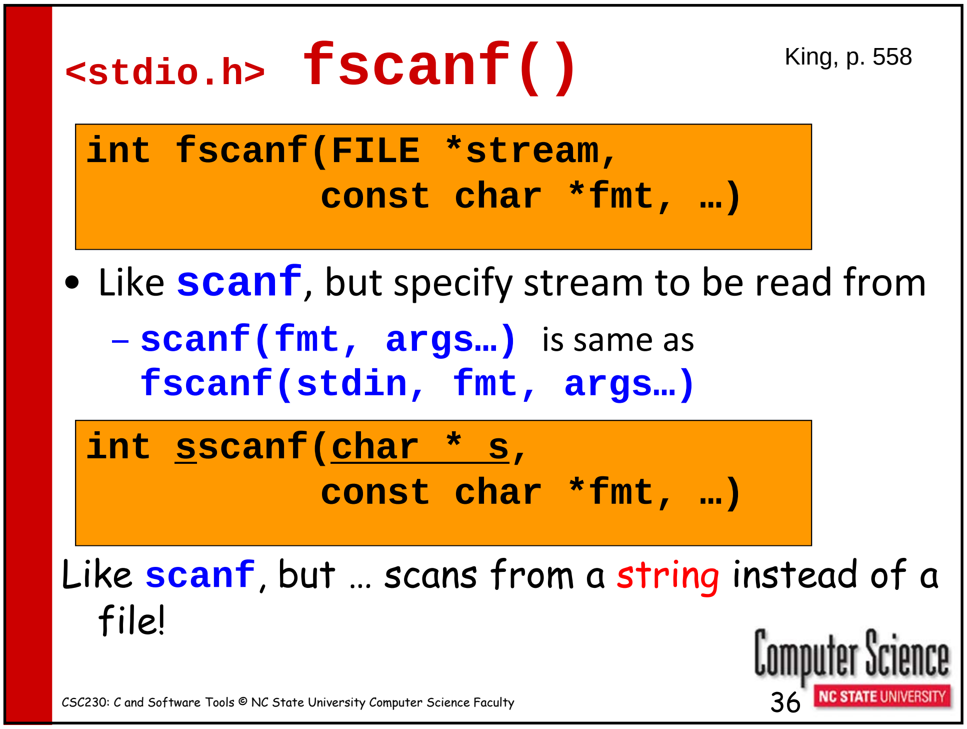
<stdio.h> remove()

King, p. 551

```
int remove(const char *filename)
```

- Delete the named file, return **0** if successful

```
if (remove("/tmp/testfile.txt"))
    …error, take action here…
```



<stdio.h> fscanf()

King, p. 558

```
int fscanf(FILE *stream,
           const char *fmt, …)
```

- Like **scanf**, but specify stream to be read from
  - **scanf(fmt, args…)** is same as **fscanf(stdin, fmt, args…)**

```
int sscanf(char * s,
           const char *fmt, …)
```

Like **scanf**, but … scans from a string instead of a file!

<stdio.h> fprintf()

King, p. 552

```
int fprintf(FILE *stream,
            const char *fmt, …)
```

- Like **printf**, but specify stream to be written to
  - **printf(fmt, args…)** is same as **frintf(stdin, fmt, args…)**

```
int sprintf(char * s, FILE *stream,
            const char *fmt, …)
```

Like **printf**, but … prints to a string instead of a file!

<stdio.h> I/O Error Functions

King, p. 564

```
int feof(FILE *stream)
```

Returns non-zero if **EOF** for **stream** has been reached

```
int ferror(FILE *stream)
```

Returns non-zero if error indicator for **stream** is set

```
void clearerr(FILE *stream)
```

Clears previously set error indicator for **stream**

- errors are not cleared unless programmer explicitly uses **clearerr**