C Standard Library

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- The C Standard Library
- More I/O functions
- Binary I/O
- Memory copy / initialization functions
- Math functions
- Math constants / functions
- Error handling

The C Standard Library

- A small set of useful functions, standardized across all platforms
- Function declarations, macros and type definitions spread across **24 Header Files**
  - You've already seen several of these:

Header	Description
limits.h	Constants for implementation-specific properties of integer types
stdbool.h	Short names for the bool type.
assert.h	Error checking mechanism for commenting and debugging

The C Standard Library

- You've seen these, but you'll learn more about them today.

stdio.h	Text and binary input/output to the terminal and files
stdlib.h	Various utilities: memory allocation, random numbers, sorting
string.h	Manipulating strings

The C Standard Library

- You'll meet some headers for the first time.

errno.h	Error codes reported by library functions and the OS
ctype.h	Classify character types, convert uppercase ← → lowercase
math.h	Common mathematical constants and functions
stdarg.h	Working with variable numbers of arguments, like printf() does.

soon

The C Standard Library

- There are lots more that we won't get to meet.

locale.h	Choosing behaviors based on where you are
time.h	Time & date calculations and format conversions
wchar.h	Wide character support, for non-Latin character sets
wctype.h	Like ctype.h, but for wide characters.
signal.h	Asynchronous interaction between processes and the OS
stddef.h	Useful macros and types
complex.h	Functions for working with complex numbers
tgmath.h	Type-generic macros for working with either floating-point or complex numbers

The C Standard Library

fenv.h	Control over the floating-point computation environment
float.h	Constants for implementation-specific properties of floating point numbers and calculations
inttypes.h	Precise conversion between integer types
iso646.h	For programming with an ISO 646 variant character set, where some standard C symbols may not be available.
setjump.h	Support for non-local exit from a function (kind of like an exception)
stdint.h	Integer types with standard bit widths

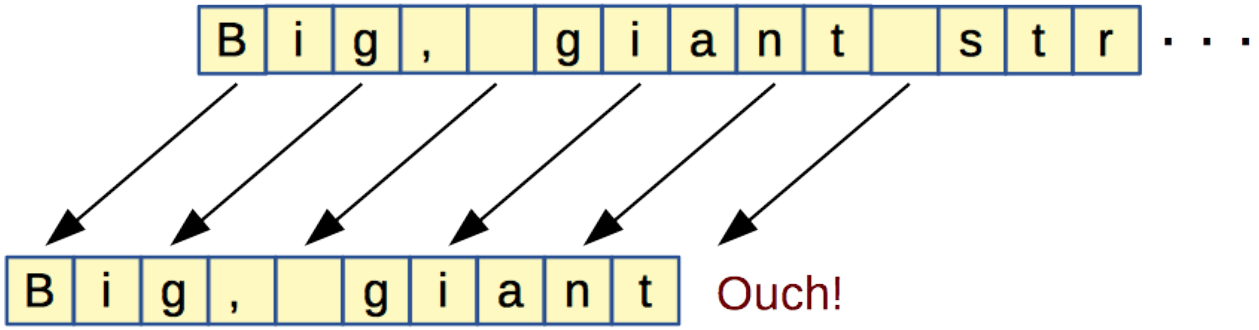
More from string.h

- You remember string.h
  - It gives functions like strcpy(), strcmp(), and atoi()
  - Let's see some of these other functions.

Preventing Buffer Overflow

- Remember, some string functions can be tricked into a buffer overflow.

```
strcpy( myString, “Big, giant string that you ”
“weren’t planning for and you don’t have enough “
“room to store” );
```



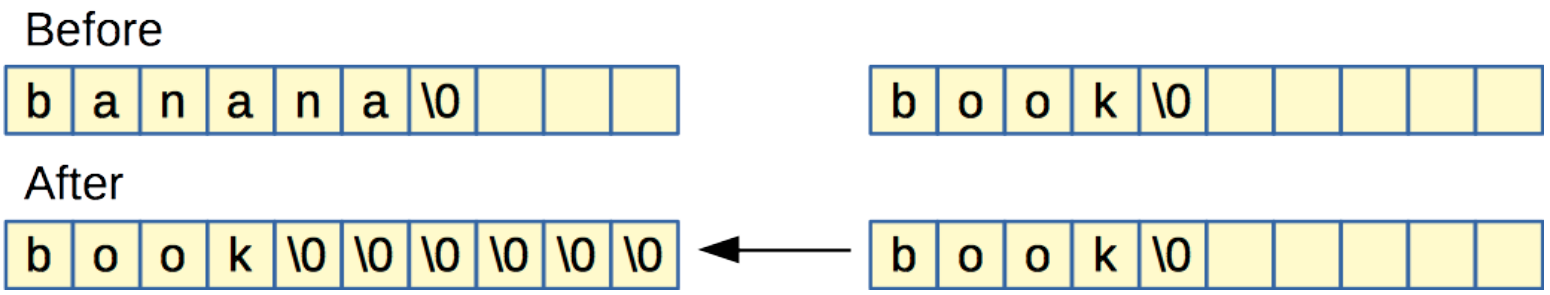
Copying Strings, with Restraint

- Its friend, strncpy(), can make it easier to prevent this.

```
char *strncpy(char *dest, const char *src, size_t n);
```

  - It takes an extra parameter, n, limiting the number of characters written to dest.
  - It pads with zeros up to n

```
strncpy( dest, src, 10 );
```

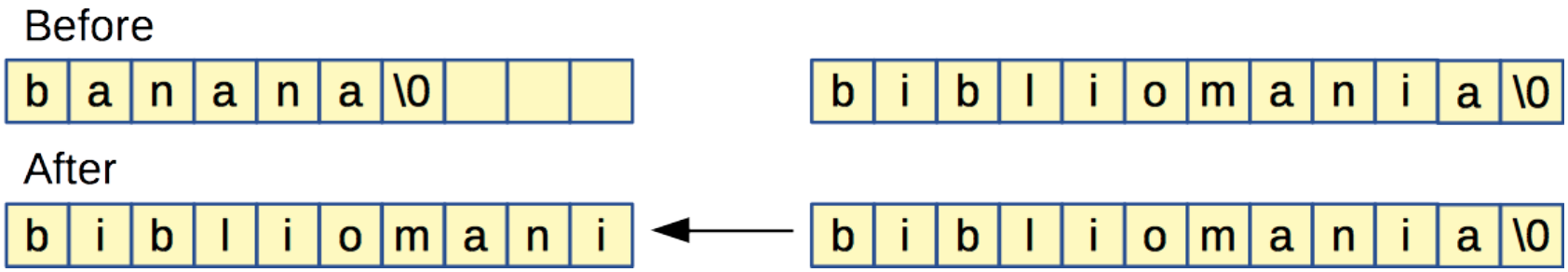


Copying Strings, with Restraint

- So, this might be typical usage, if it did the right thing … which it doesn't.

```
strncpy( dest, src, sizeof( dest ) );
```

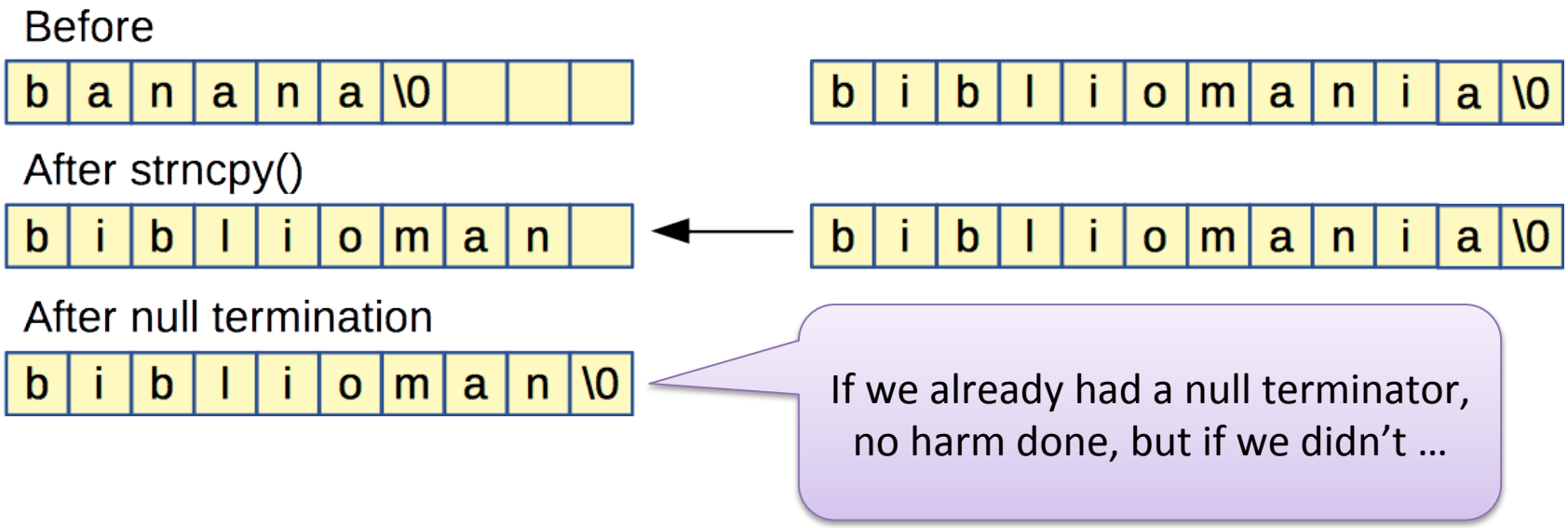
- Caution, strncpy() won't null terminate if it runs out of room.



Copying Strings, with Restraint

- If we want to guarantee null termination, we will need to do it ourselves:

```
strncpy( buffer, word, sizeof( buffer ) - 1 );
buffer[ sizeof( buffer ) – 1 ] = '\0';
```

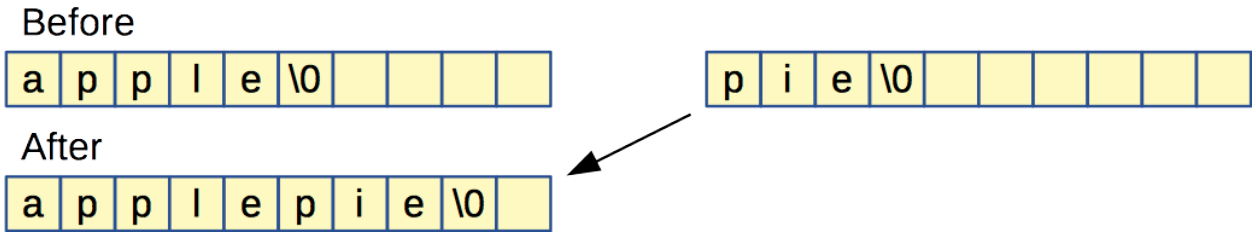


Concatenating Strings

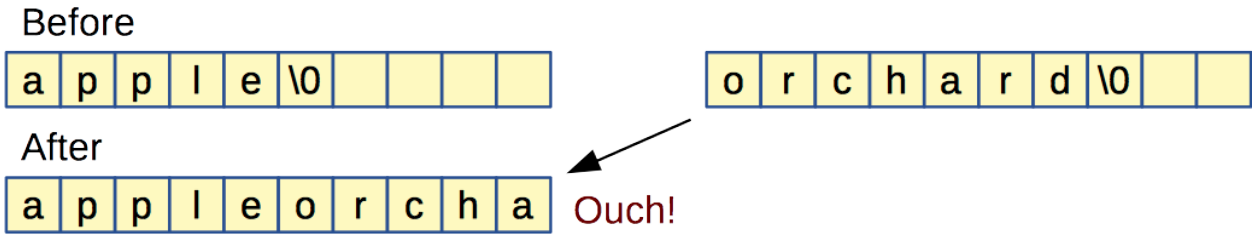
- We have two flavors of string concatenation, with and without a bound.

```
char *strcat(char *dest, const char *src);
```

  - Copies the string from source to the end of dest.
  - Overwriting the null terminator, and writing out a new one.



  - You have to consider possible buffer overflows.



Concatenating Strings

- strncat() can help.
  - It lets you give a bound on the length of the string appended.

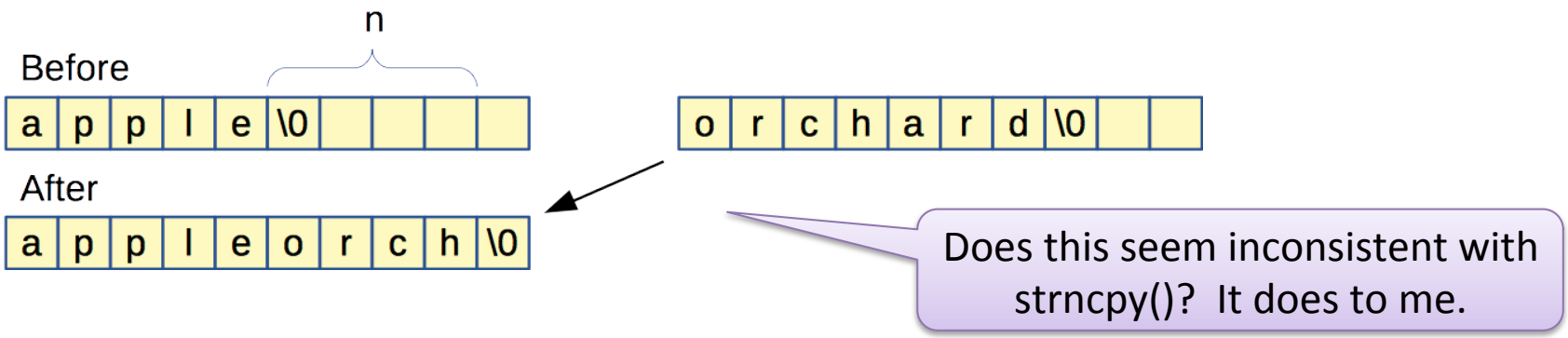
```
char *strncat(char *dest, const char *src, size_t n);
```

  - So, you might use it like:

```
strncat( dest, src, 4 );
```

  - Or, more generally:

```
strncat( dest, src, sizeof(dest) – strlen(dest) – 1 );
```

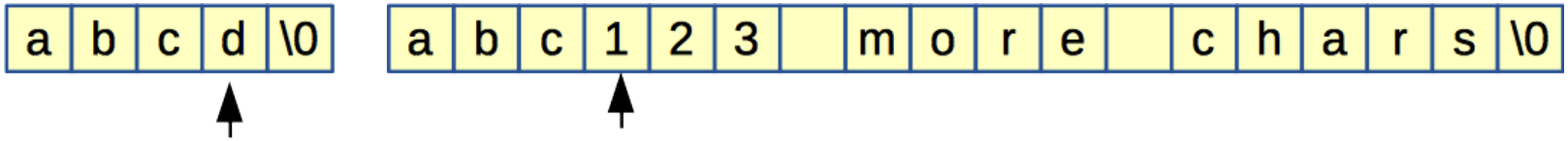


Comparing With Caution

- There are bounded versions of lots of the string handling functions.
- Worried about buffer overflow during string comparison? We've got a function for that:

```
int strncmp( const char *s1, const char *s2,
             size_t n);
```

  - Compares at most n characters in s1 and s2.
  - Seems like it's only useful if one of your strings may not be null terminated
  - … or you just want to look at the start of a string.



Formatted Input from a String

- There's a very handy function for parsing the contents of a string

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

  - Parse the contents of str according to the given format string.
  - Assign remaining arguments based on the given conversion specifiers
  - Return the number of conversion specifiers matched.
- Kind of like wrapping a Scanner around a string in Java.
- These are declared in stdio.h (not string.h)

Formatted Input from a String

- Example, parse an int and a float from the contents of a buffer:

```
char buffer[] = “25 31.2”;
int iVar;
float fVar;
sscanf( buffer, “%d%f”, &iVar, &fVar );
```

- This function is why I almost never use atoi(), atof() …
  - sscanf() can do the job of all of them.
  - And it gives you a more useful success/failure report.

```
int val = atoi( str );
```

I return zero on an error.

```
if ( sscanf( “%d”, &val ) != 1 )
    oops;
```

I save a value, and return a success/ value status.

Marking Your Spot

- This is really the function where %n is useful.
  - scanf() and fscanf() automatically move forward in the input.
  - … but not sscanf(), it just reads the string you give it from the start.
- The %n conversion specification can let you continue from where you left off.

```
sscanf( buffer, "%d%n", &value, &pos );
sscanf( buffer + pos, "%s", word );
```

Formatted Output to a String

- We have similar functions for going the other way:

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

```
int snprintf(char *str, size_t size, const char *format, ...);
```

  - Like printf() but with output going to the given string.
  - snprintf() writes at most n bytes (including a null terminator at the end)

There's a lot of inconsistency here.

- Example:

```
char buffer[ 10 ];
snprintf( buffer, sizeof(buffer), "%s", "abcdefghijklmnop" );
```

Copies just the first 9 characters.

Different Bounded Behaviors

strncpy()	n is a number of characters to write padding with zeros if possible may not null terminate if there's no extra room
strncmp()	n is a maximum number of characters to compare will stop when we reach n, when there's a difference or either string ends (at null termination)
strncat()	n is a maximum number of characters to append not counting the null terminator will always write a null terminator
snprintf()	n is a buffer capacity on the destination including the null terminator will always write a null terminator

This behavior is my favorite.

Copying Memory

- Recall, `string.h` offers functions to copy strings, and arbitrary memory.
- We can use memcpy() to copy an array of any type.

```
memcpy( someArray, otherArray, n * sizeof( int ) );
```

  - But, the memory for these arrays can't overlap.
- Meet memcpy's smarter, slightly slower friend:

```
void *memmove(void *dest, const void *src, size_t n);
```

  - It's OK for src and dest to overlap.

Initializing Memory

- There's also a handy function to set blocks of memory to a chosen value.

```
void *memset(void *s, int c, size_t n);
```

  - Sets n bytes of memory to byte value c
  - Returns a pointer to dest.

- Example:

```
memset( map, '.', n * n );
```

String Processing Zoo

- The standard library includes lots of functions for working with strings and memory
- We only need to know a few, but it's good to know what's out there.
- A function to measure string length.

Function	Description
size_t strlen(str)	Return the length of the given string.

- Functions to copy strings:

Function	Description
strcpy(dest, src)	Copy string from src to dest
strncpy(dest, src, n)	Copy at most n characters from src to dest.

String Processing Zoo

- Functions to compare:

Function	Description
strcmp(s1, s2)	Compare strings s1 and s2, returning negative, zero or positive.
strncmp(s1, s2, n)	Compare at most n characters of s1 and s2
strcoll(s1, s2)	Compare strings, locale dependent

You don't need to know the gray ones.

- Functions to concatenate:

Function	Description
strcat(dest, src)	Concatenate src to the end of dest
strncat(dest, src, n)	Concatenate, copying at most n characters

String Processing Zoo

- Functions to search:

Function	Description
char *strchr(str, c)	Return pointer to first occurrence of c in str, or NULL.
char *strstr(haystack, needle)	Return pointer to first occurrence of string needle in string haystack.
char *strpbrk(str, char *accept)	Return pointer to first occurrence of any char from accept in str.
size_t strcspn(str, char *accept)	Return length of sequence of str consisting of only chars from accept
char *strtok(str, char *delim)	Break str at any character in delim, return pointer to the next token.

- More functions in your book, Chapter 26

Working with Characters

- We also have functions for working with individual characters, declared in ctype.h

```
#include <ctype.h>
```

- Functions to classify characters

```
int isalpha( int ch );
int isspace( int ch );
int isdigit( int ch );
int isprint( int ch );
```

True if ch is a letter.

True if ch is whitespace.

True if ch is a digit.

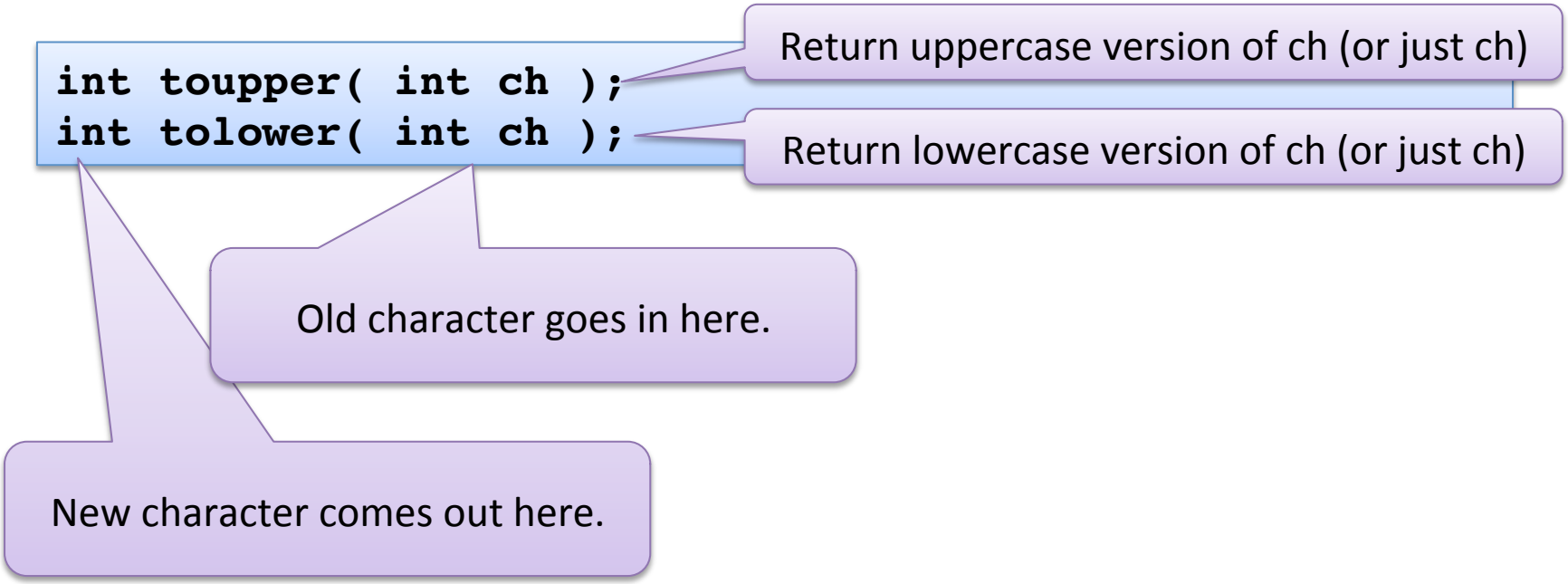
```
int isalnum( int ch );
int ispunct( int ch );
int isupper( int ch );
…
```

… you get the idea …

Lots more, but you don't need to know these.

Working with Characters

- Functions to convert characters characters



Fun with ctype.h

```
bool needSpace = false;

int ch;
while ( ( ch = getchar() ) != EOF ) {
  while ( ch != EOF && !isalpha( ch ) )
    ch = getchar();

  if ( isalpha( ch ) && needSpace )
    putchar( ' ' );

  while ( isalpha( ch ) ) {
    putchar( tolower( ch ) );
    ch = getchar();
  }

  needSpace = true;
}
```

Skip non-letters.

Print a space between words.

Print contiguous words as lowercase.

More from stdio.h

- You already know a lot about stdio.h
  - … but it can do even more for you.
- More functions for:
  - Character I/O
  - File errors and EOF
  - I/O Undo
  - Buffering, Seeking and Flushing
  - Binary I/O

Character I/O

- We have functions for character I/O on the terminal

```
int getchar(void);
```

```
int putchar(int c);
```

  - They read/write a single character
  - Return EOF on failure.

- We have similar functions for working with any stream

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

I'm a function.

I may be a macro.

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

We're the same way.

Character I/O Example

- Copying a file character-by-character:

```
FILE *src = fopen( argv[ 1 ], "r" );
FILE *dest = fopen( argv[ 2 ], "w" );

int ch;
while ( ( ch = fgetc( src ) ) != EOF )
  fputc( ch, dest );

fclose( src );
fclose( dest );
```

Open a source and destination file.

Copy characters until we hit EOF.

Clean up when we're done.

Checking for EOF

- We have a simple, general technique for detecting end-of-file
  - Call a reading function
  - See if it returns EOF.
  - Afterward, we have to remember if we've already hit the EOF.
- LibC remembers this for us.

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

  - Returns true if you've already reached the EOF.
  - Set whenever a reading function encounters EOF (even internally).

Easy Mistakes With EOF

- This gives us a new, terrible way to try to detect end-of file.

```
while ( ! feof( input ) ) {
  fscanf( input, "%d", &value );
  …;
}
```

This is bad.
Don't do this.

```
25 102EOF
```

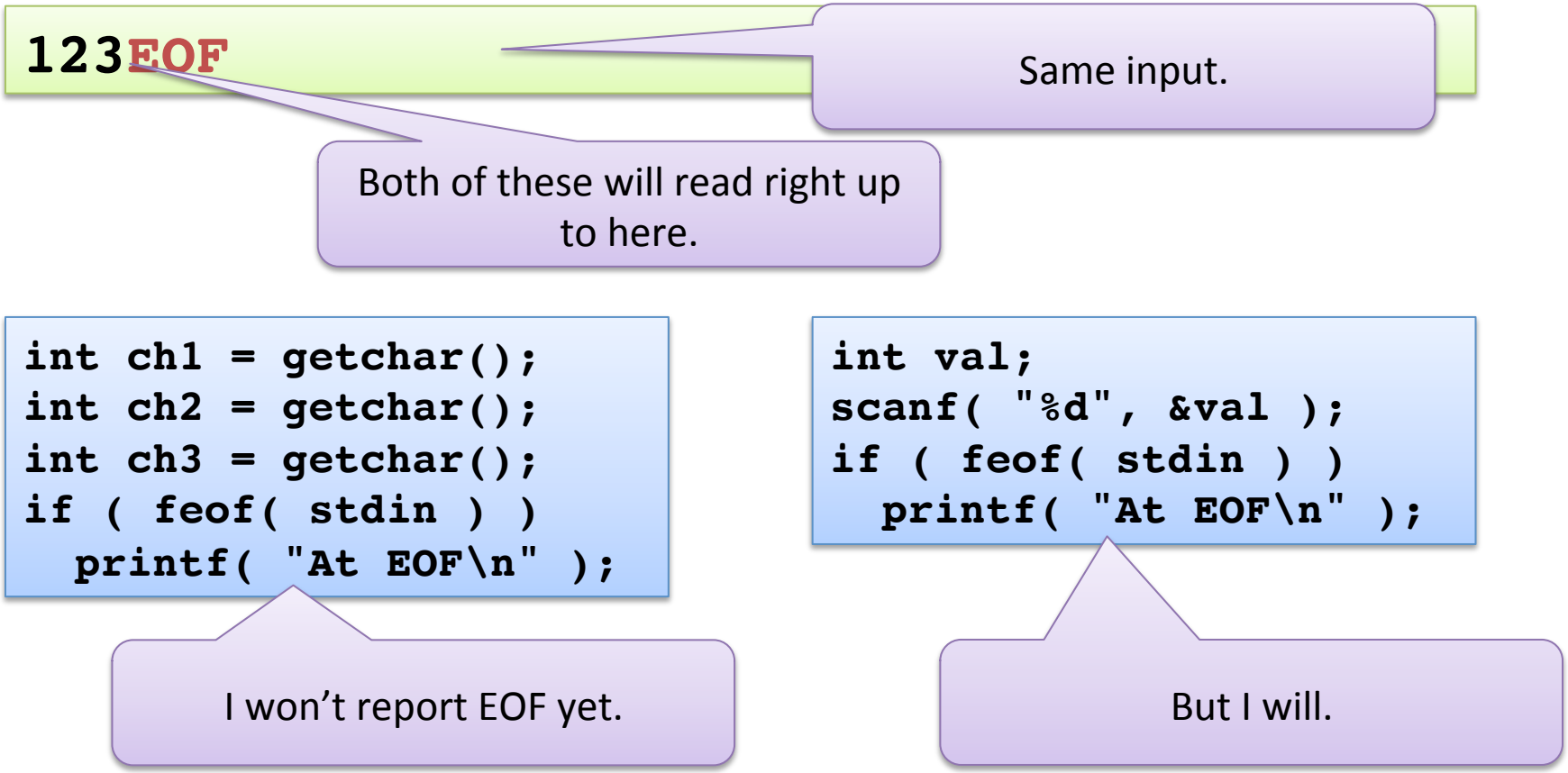
If you hit EOF right here, this will work.

```
25 102\n
```

But, if there's any whitespace, it will think it got an extra value.

Know What You're Doing

- feof() works as advertised, but you have to think about what's going on internally.



Things go Wrong

- A file can encounter an error condition.
  - If you try to write to a file that's open for reading.
  - Or the other way around
  - Or the disk is full
  - Or you reach the maximum files size.
- When this happens, the stream notices and remembers it.
  - If you know how to ask, it will tell you something when wrong:

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

Forgetting Past Mistakes

- The EOF and error conditions remain set
- … until you explicitly clear them.
- There's a function for that:

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

Take Backsies

- Streams perform buffering
- An input stream will let you return one input character to the buffer

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

  - Pushes character c back onto the input stream
  - Clears the EOF condition (if it's set)
  - Oh, and you can't push back an EOF (why would you want to?)
  - It even works with **stdin**

Flushing the Buffer

- Streams perform buffering, but sometimes you want your output **right now!**

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

  - Buffered output is written out immediately
- Normally, standard output flushes … when it sees a newline

```
printf( "You can't see me " );
sleep( 5 );
printf( "... until you write a newline\n" );
```

Flushing the Buffer

- You can take control of this with fflush()

```
printf( "But flushing writes output immediately" );
fflush( stdout );
sleep( 5 );

printf( "... even without a newline.\n" );
```

Binary I/O

- A streams can operate in two modes:
  - Text mode : tries to hide some details like line termination sequence.
  - Binary mode : shows you every character/byte exactly.
- Every stream you've used so far has been in text mode.
- On the common platform it doesn't matter.
  - They're exactly the same.
  - But, in general they may not be.

Meet More Mode Strings

- In the mode string, you can choose text or binary mode … and other things.

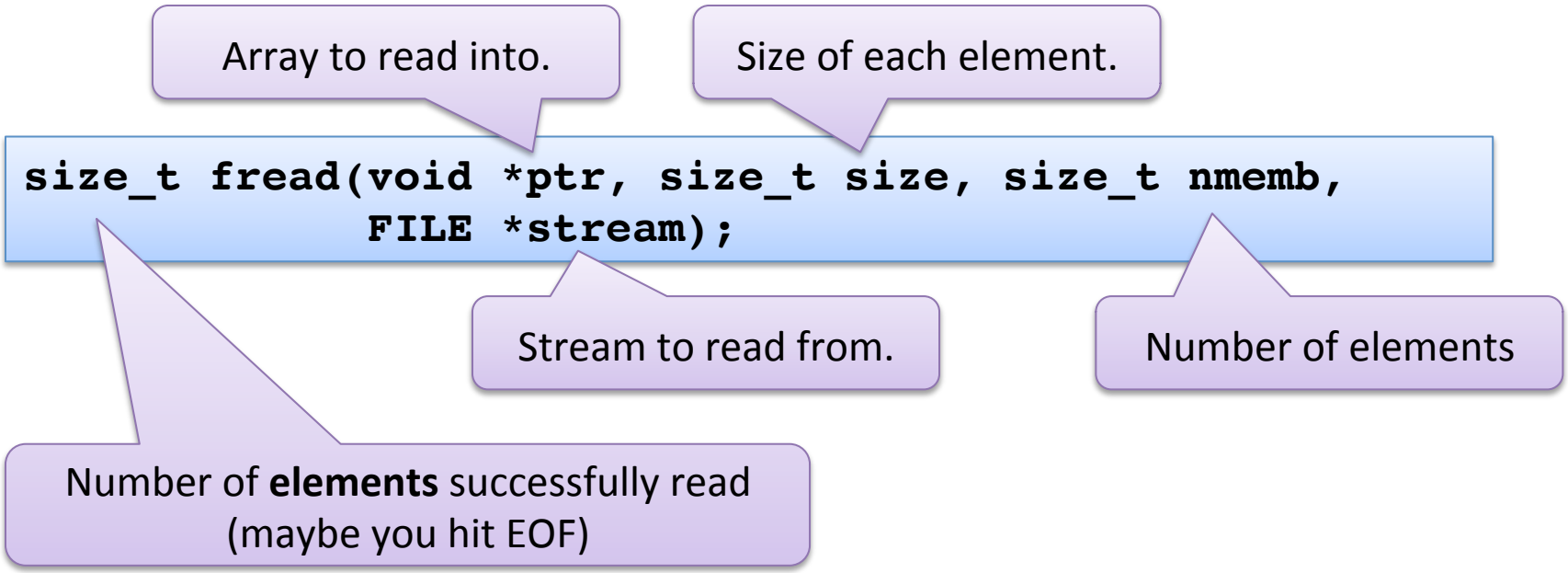
```
FILE *fopen(const char *path, const char *mode);
```

Mode String	Access Type
r	Open for reading
w	Open for writing (created and truncated)
r+	Open for read and write
a	Open for appending (created but not truncated)
rb	Open a **binary** file for reading
wb	Open a **binary** file for writing

We could invent more combinations of modes.

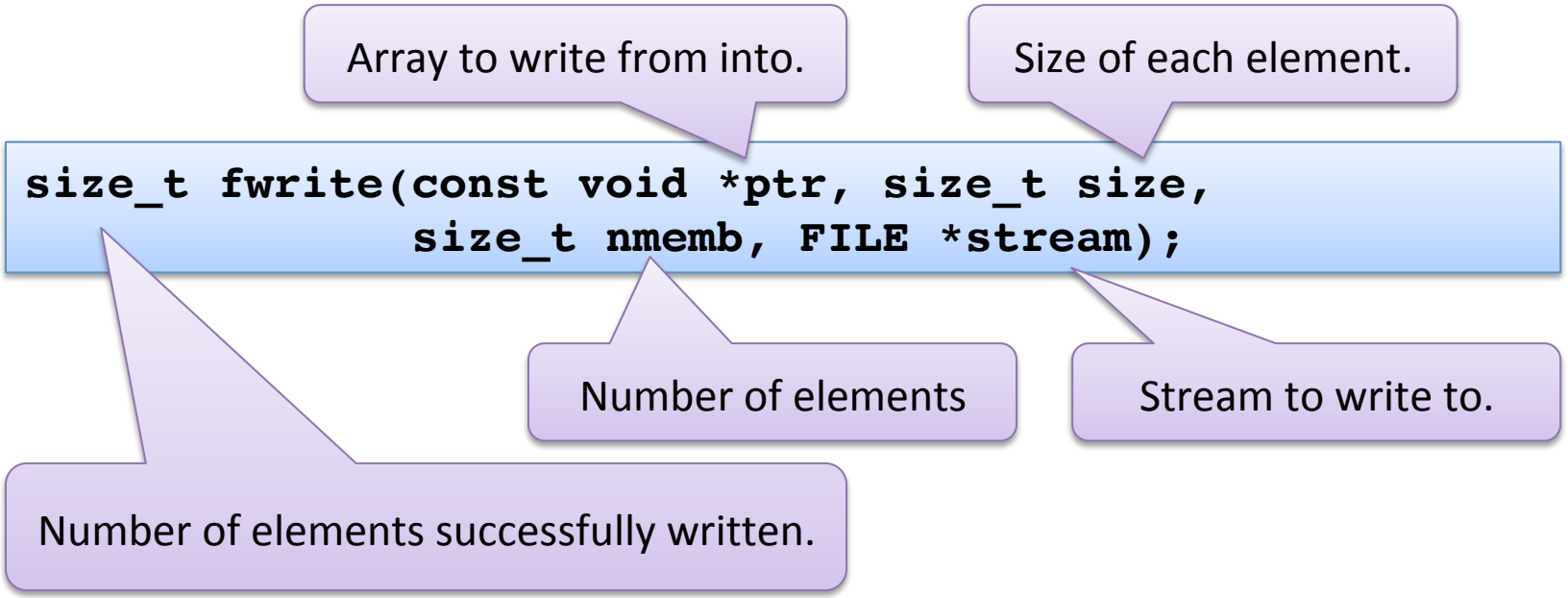
Block Read/Write Functions

- Reading and writing individual characters can be slow
- C gives us functions to read/write arrays of characters or other types.
  - If you're not reading/writing characters → you probably want a binary stream.



Block Read/Write Functions

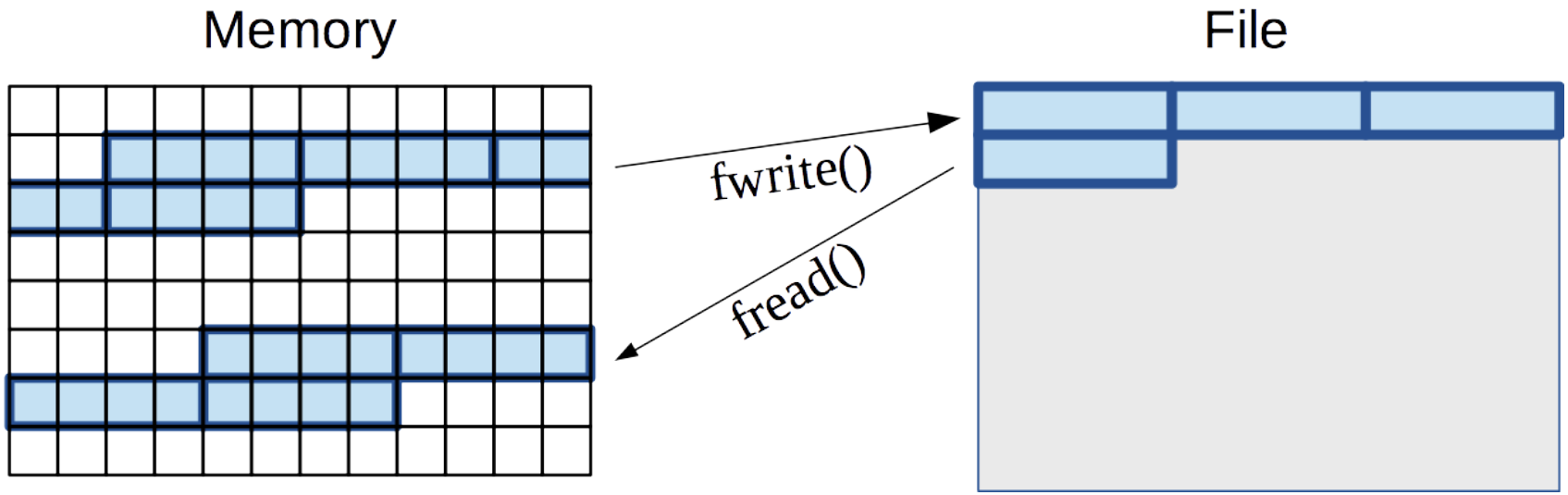
- We have a similar function for writing.



- These are a lot like the underlying calls provided by the OS

fread() and fwrite() Illustrated

- You can think of fread() and fwrite() like this.
  - They copy arrays of data between memory and files.



Block I/O Example

- We may get better performance copying a file a block at a time:

```
FILE *src = fopen( argv[ 1 ], "rb" );
FILE *dest = fopen( argv[ 2 ], "wb" );

char buffer[ 1024 ];
int len;
while ( ( len = fread( buffer, 1, sizeof(buffer),
                       src ) ) != 0 )
  fwrite( buffer, 1, len, dest );

fclose( src );
fclose( dest );
```

The **b** does nothing on the common platform.

But, it might make a difference on other platforms, so it's a good idea for portability.

Block I/O Example

- We can serialize/deserialize a whole array at a time:

```
FILE *fp = fopen( "list.bin", "wb" );

int list[] = { 3, 472, 257381, 719293945 };

fwrite( list, sizeof(int), 4, fp );
fclose( fp );
```

Write the contents of this array in memory.

```
FILE *fp = fopen( "list.bin", "rb" );

int list[ 4 ];

fread( list, sizeof(int), 4, fp );
fclose( fp );
```

Read it Back.

Binary Representation and Byte Order

- We can look at the contents of this file
  - The hexdump command can help

```
$ hexdump -C list.bin
00000000  03 00 00 00 d8 01 00 00
          65 ed 03 00 f9 8d df 2a
          |........e......*|a
00000010
```

I had to wrap these. Should be on one line.

- Here, we can see that the common platform is a LSB-First (Little-Endian) system
- But, other systems might store individual bytes in the opposite order, MSB-First (Big-Endian)

Binary I/O and Byte Order

- Differences in byte order can cause problems if we move binary files between platforms.
- It's generally the programmer's job to worry about this.
- Of course, we can write code to reorder bytes:

```
void swapBytes4( int *val )
{
  char *p = (char *)val;

  char t = p[ 0 ];
  p[ 0 ] = p[ 3 ];
  p[ 3 ] = t;

  t = p[ 1 ];
  p[ 1 ] = p[ 2 ];
  p[ 2 ] = t;
}
```

Get a pointer to the bytes in this value.

Swap the first and last.

Swap the middle two.

Taking Another Look

- Normally, files are accessed sequentially
  - After you read/write a byte you automatically get the next byte
  - Buffering done inside the library promotes sequential access performance.
- But, you you can go back to the start of a file.
  - To, re-read or re-write it from the start.

```
int rewind(FILE *stream );
```

Where to?

With respect to what?

Jumping Around

What stream?

Where to?

```
int fseek(FILE *stream, long offset, int whence);
```

- There's a more general function to seek to anywhere.
- The whence parameter tells you with-respect to what?
  - SEEK_SET : just go to this offset
  - SEEK_CUR : offset is with-respect-to the current position
  - SEEK_END : offset is from the end of the file.
- Works for binary streams, limited functionality for text

```
fseek( input, -1, SEEK_CUR );  // Backup one byte
fseek( input, 0, SEEK_END );   // Goto end of file
fseek( input, p, SEEK_SET );   // Goto position p
```

Checking Your Location

- fseek() is generally a bad idea for text files.
- While reading, a text file may lie to you about what's in the input.
  - e.g., turning a carriage-return, linefeed into just a linefeed.
  - … so, you may not know exactly how many characters you've read.
- But, a stream can tell you if you ask:

```
long ftell( FILE *stream );
```

- It's safe to seek to a position you've obtained from ftell()

Lots more Functions

- We won't learn all the I/O functions (but we've learned most of them)
- There are a few more useful ones….
  - Remove a file:

```
int remove(const char *pathname);
```

  - Rename an existing file:

```
int rename( const char *old, const char *new );
```

  - Make a temporary file that goes away when closed:

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

Random Numbers

- **`stdlib.h`** provides functions and constants for pseudo-random number generation:
  - Generate a random integer 0 … RAND_MAX

```
int rand( );
```

  - Constant for the maximum random value

```
RAND_MAX
```

  - Provide a seed for the pseudo-random sequence

```
void srand( unsigned int seed );
```

Choosing a Seed

- By default, you will get the same pseudo-random sequence at every execution.
  - Unlike Math.random() in Java.
- A common remedy, seed the sequence with the current time:

```
void srand( time( NULL ) );
```

I'm declared in time.h

Number of seconds since the start of the universe, January 1, 1970.

Example

- To generate a number in [0.0,1.0]

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double getrand() {
    int r = rand();
    return (double) r / RAND_MAX;
}

int main() {
    srand( time(NULL) );
   …
    double r = getrand();
   …
```

Random Numbers

- We could generalize this technique to get any range we need.
- Say we want a real number between **`min`** and **`max`** (inclusive)

```
double min = …, max = … ;
double range = max – min;

double r1 = (double) rand() / RAND_MAX * range;
double r2 = r1 + min;
```

Things go Wrong

- C has a simple technique for handling errors
- And a header to go along with it:

```
#include <errno.h>
```

Secretly, this is no longer really a global variable. That would be bad. But, for now, pretend it is.

  - It exposes a global variable errno
  - An integer that gets set to indicate numerous error conditions

- To use errno:

clear it

do something

check it

```
errno = 0;
do_something_dangerous;
if ( errno == EACCES ) …
```

Documentation

```
$ man fopen
FOPEN(3)                    Linux Programmer's Manual                    FOPEN(3)

NAME

       fopen, fdopen, freopen - stream open functions

SYNOPSIS

       #include <stdio.h>

       FILE *fopen(const char *path, const char *mode);

…

ERRORS
       EINVAL The  mode  provided  to  fopen(),  fdopen(),  or  freopen()  was
              invalid.

       The  fopen(),  fdopen()  and  freopen() functions may also fail and set
       errno for any of the errors specified for the routine malloc(3).

…
```

Making Sense of errno

- There are numerous values errno may be given (EACCESS, EEXIST, EFAULT, ENOENT, ENOMEM, ENOSPC …)
- There's a function that knows short messages for each of these:

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

Message of your choice, included with the error report.

Using perror()

- We use perror() like this:

```
FILE *fp = fopen( “someFile.txt”, “r” );
if ( fp == NULL ) {
  perror( “someFile.txt” );
  exit( 1 );
}
```

- And we get:

```
$ myProgram
someFile.txt: No such file or directory.
```

Math

- (Common) math support is advertised in math.h

```
#include <math.h>
```

- You also need to link with the math library:

```
gcc -Wall -std=c99 program.c -o program -lm
```

- You get some important constants

Macro	Value
M_E	Base of the natural logarithm
M_PI	You know, π
M_SQRT2	Square root of 2

Trigonometry

- You get common mathematical functions

```
double sin(double x);
```

Double goes in

Double comes out.

- And lots of friends:

Function	Description
sin(x)	Trigonometric sine, cosine and tangent, returning [-1.0, 1.0]
cos(x)	
tan(x)	
asin(x)	Trigonometric arcsine, arccosine and arctangent functions, input [-1.0, 1.0], output [-π/2, π/2]
acos(x)	
atan(x)	
atan2(y, x)	This one's my favorite, output [-π, π]

Exponents, Logs and Roots (oh my)

- But wait, there's more

Function	Description
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) … OK, $x^{1/2}$

Rounding and Friends

- Even more math functions:

Function	Description
round(x)	nearest integer (as a double)
fabs(x)	absolute value
floor(x)	largest integer no larger than x (as a double)
ceil(x)	smallest integer no less than x (as a double)