

File I/O and Strings

CSC 230 : C and Software Tools
NC State Department of Computer
Science

Topics for Today

- File I/O
- Parsing input : the rest of the story
- Working with Strings
 - Direct string manipulation
 - String length
 - Copying strings
 - Comparing strings
 - Parsing numbers from strings

File I/O

- Our good friend, stdio.h, also includes support for file I/O

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

- We just need to ask it to open a new file.

```
FILE *stream = fopen( "file-name.txt", "r" );
```

Path to the file you want.

Mode String (reading, writing, etc).

- This creates a new *stream* and returns a *file pointer* for working with it.
 - Or it returns **NULL** on failure.

File Mode

- When you open a file, the second parameter says how you want to access it.

```
FILE *fp = fopen( "file-name.txt", "r" );
```

Mode String	Access Type
r	Open for reading
w	Open for writing (created and truncated)

More mode strings later.

Reading and Writing

- We have versions of printf() for writing to any stream.

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

Stream you want
to write to.

The rest is just
like printf.

- We have a similar version of scanf()

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

Same here, just one new
parameter at the start.

Finishing I/O

- Streams perform *buffering*
 - They keep I/O characters in memory to avoid costly system calls to the OS
- You need to tell the system when you're done with a file.

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

- This will flush any buffered data.
- Forgetting to close a file is called a *file leak*

File Input Example

- An example, reading a list of numbers:

```
FILE *fp = fopen( "numbers.txt", "r" );
if ( !fp ) {
    ...
}

int total = 0;

int val;
while ( fscanf( fp, "%d", &val ) == 1 )
    total += val;

fclose( fp );

printf( "Total: %d\n", total );
```

I want to open for reading.

I want to read from a file,
instead of standard input.

I'm all done with this file.

File Output Example

- An example, writing a file of numbers.

```
FILE *fp = fopen( "output.txt", "w" );
if ( !fp ) {
    ...
}

for ( int i = 0; i < 25; i++ )
    fprintf( fp, "%f\n", i * i * 3.141592653589793 );

fclose( fp );
```

I want to open for writing.

I want to write to a file instead of standard output.

All done, flush any buffered output.

Leaking Files

- There's typically a limit on the number of open files you can have
 - That's why leaking files is bad
- I have a test for this: `openTest.c`
 - It opens files over and over until it can't create any more
 - Let's give it a try

Three Streams for Free

- When you call `fopen()`, you get back a **stream**

```
FILE *stream = fopen( "file-name.txt", "r" );
```

- Your program gets three streams at startup, you've been working with them all semester
 - stdin** : standard input (from the terminal by default)
 - stdout** : standard output (to the terminal by default)
 - stderr** : standard error (also to the terminal by default)
- These are all global variables declared in **stdio.h**

Using the Standard Streams

- With names for our standard streams
 - This gives us alternative ways to do terminal I/O.

```
fscanf( stdin, ... );
```



```
scanf( ... );
```

```
fprintf( stdout, ... );
```



```
printf( ... );
```

```
fprintf( stderr, ... );
```

Maybe the first two aren't very useful, but here's something new.

- Printing to stderr is a standard way of reporting, well, errors.
 - Why? We can redirect it differently from stdout.

Using stderr

- We can use stderr for any error reports

```
FILE *fp = fopen( "numbers.txt", "r" );
if ( !fp ) {
    fprintf( stderr, "Can't open input file\n" );
    exit( 1 );
}

int len = 0;
if ( fscanf( input, "%d", &len ) != 1 ) {
    fprintf( stderr, "Can't read input length\n" );
    exit( 1 );
}

. . .
```

Redirecting Streams

- From the shell, we can choose to redirect streams differently.

I redirect standard output.

I redirect standard error.

```
$ ./program > output.txt 2> error.txt
```

- Or the same

Send standard error to the same file as standard output.

```
$ ./program > everything.txt 2>&1
```

- This syntax is dependent on your shell, but this is what bash uses.

Reading Strings

- You already know:
 - We can use scanf() to read into strings (character arrays).

```
char str[ 100 ];  
  
scanf( "%s", str );
```

This will read a sequence
of non-whitespace
characters.

Notice, no &.

- Why no & on the str parameter?

- scanf() just needs to change the contents of the character array, not where it points.

Reading Strings Without Buffer Overflow

- You already know:
 - This can buffer overflow, if the user enters a long string of non-whitespace characters.

```
char str[ 100 ];  
  
scanf( "%s", str );
```

- Something new, input field width
 - Specify a field width to limit the number of characters read.

```
char str[ 100 ];  
  
scanf( "%99s", str );
```

Read up to 99 characters.
Good, we still have room
for the null terminator.

All of this works for scanf()
and fscanf().

String Parsing Options

- The %s format specifier reads space-delimited words.
- scanf() supports a subset of regular expression syntax.
 - Using square brackets we can define a *character class*, a set of exactly what characters to match.

```
scanf( "%[abc]", str );
```

Match a sequence of one or more a, b and c characters.

Doesn't skip spaces.

Character Class Examples

```
scanf( "%[abc]", str );
```

bcabbxaccb****

This will return zero matches.

zero

```
scanf( "%[ 0123456789]", str );
```

123 456 xyz 789

You'll get this last space.

String Parsing Options

- Inside the brackets, we can specify ranges of characters.

```
scanf( "%[A-Z]", str );
```

Match any capital letter.

- We can combine ranges and individual characters.

```
scanf( "%[a-zA-Z.]", str );
```

Match lower-case or capital letters or dot.

abc.XYZ.123

String Parsing Options

- You can negate a character class by starting it with a ^

```
scanf( "%[^A-Z]", str );
```

Match a sequence of anything but capital letters.

```
scanf( "%[^A-PR-Z]", str );
```

Same as above, but will also match Q.

abcQ123Wxyz

String Parsing Options

- Using *, we can tell scanf to match a format specifier, but not save it:

```
scanf( "%*s%s", str );
```

Skip spaces, then save the next word in str.

Match a word (but don't store it)

- Really, this works with any format specifier.

```
scanf( "%*d%d", &val );
```

```
scanf( "%*[a-z]%20[A-Z]", str );
```

Skipping Whitespace

- Any whitespace in a format string ... matches any amount of whitespace.

```
scanf( "%c", &ch );
```

I'll read the next character,
even if it's whitespace

```
scanf( " %c", &ch );
```

I'll skip all whitespace, then
read a character

```
scanf( "%[0-9]", str );
```

I'll read just digits.

```
scanf( "%[0-9]", str );
```

I'll skip whitespace, then read
digits.

Skipping Whitespace

- Be careful, stray whitespace can give seemingly strange behavior.

```
scanf( "%d%d\n", &a, &b );
```

OK. You want me to skip whitespace after reading these ints.

I better not return until I'm sure I have it all.

Matching Literal Characters

- Literal characters will match copies of themselves.

```
scanf( "%d+d", &a, &b );
```

25+72

25 + 72

I'll match the +, but I won't skip spaces before it.

```
scanf( "%d +%d", &a, &b );
```

But I will.

```
scanf( "%d %%", &a );
```

Need to match a literal %? We got that.

Marking Your Spot

- The %n conversion specification tells scanf() to report how many characters it's read so far.

```
scanf( "%d +%n%d", &a, &n, &b );
```

25+72

317 + 214

Put the report here.

- The %n doesn't count in the return value.
- This is kind of useful with scanf() and fscanf()
- It will be more useful with sscanf() ... later.

Working with Strings

- Ok. You've read in a string.
- Now, what can you do with it?
- Well, a string is an array of characters, so you could access its characters directly.

```
int spaces = 0;
for ( int i = 0; str[ i ] != '\0'; i++ ) {
    if ( str[ i ] == ' ' )
        spaces++;
}
```

Where Does it End?

- The null terminator lets us pass strings to functions
 - Without a separate parameter for their length.

```
void redact( int len, char str[] );
```

You may need this for other arrays ... but usually not with strings.

The null terminator lets you find where the string ends.

```
void redact( char str[] );
```

Where Does it End?

- The null terminator evaluates to false.

```
void redact( char str[] )
{
    for ( int i = 0; str[ i ]; i++ )
        if ( str[ i ] >= '0' &&
            str[ i ] <= 'z' )
            str[ i ] = '#';
}
```

Copying Strings

- You can't copy between strings with the assignment operator.

```
char buffer[100] = "abc123";
buffer = "Welcome to C";
```

- This doesn't work for arrays ...
and strings are stored in arrays ...
so ☹
- ... but, there are library functions to help us out
☺

This works, at initialization time, but not for assignments in general.

Help from the Standard Library

- The C standard library offers functions to work with strings.
- String functions are declared in the `string.h` header:

```
#include <string.h>
```

- All these functions expect **null terminated strings**.
- Today, we'll look at functions to:
 - Measure string length
 - Copy between strings
 - Compare strings
 - Convert strings to ints, doubles, longs, etc.

String Length

- We have `strlen()` to obtain string length:

```
size_t strlen( const char s[] );
```

- Counts characters up to (but not including) the null terminator.

- The prototype tells us a lot about the function

I take an array of
characters.

But, I promise not
to modify it.

```
size_t strlen( const char s[] );
```

I return the length, as a `size_t` (which will
convert to `int` if you want)

String Length

- It's easy to use:

```
int len = strlen( name );
```

- It's not too smart.
 - It just looks for the null terminator.
 - So, it has a cost that's linear in the string length.
- This is a bad way to iterate over a string:

```
for ( int i = 0; i < strlen( word ); i++ )  
    ...;
```

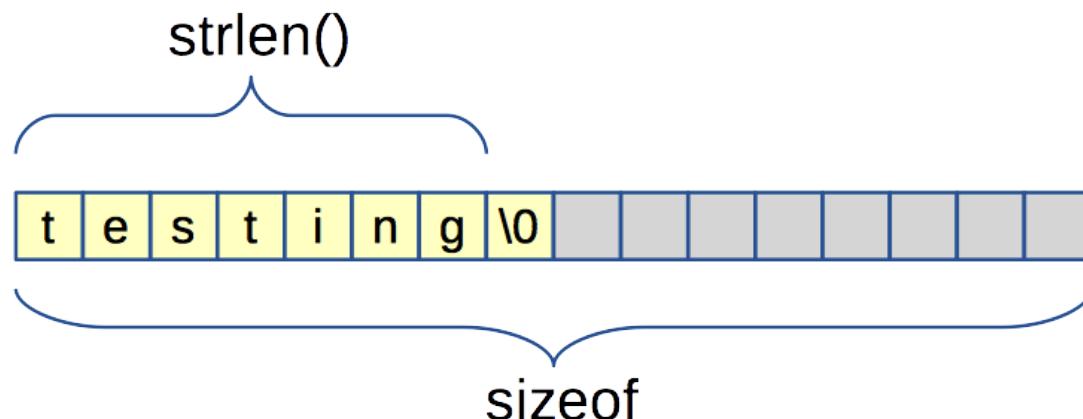
- This is much more efficient (and maybe easier)

```
for ( int i = 0; word[ i ]; i++ )  
    ...;
```

strlen() vs sizeof

- Sometimes, people have some confusion between strlen() and sizeof.

```
char buffer[ 16 ] = "testing";
```



`strlen()` vs `sizeof`

```
char buffer[ 16 ] = "testing";
```

I'm an operator, usually evaluated at compile time.

```
size_t s = sizeof buffer;
```

I don't depend on what's in the string. I'll say 16.

```
size_t n = strlen( buffer );
```

I'm a function call, executed at runtime.

```
buffer[ 1 ] = '\0';  
n = strlen( buffer );  
s = sizeof buffer;
```

I'll say 7. I depend on what's in the string.

Now, I'll say 1.

But still 16 here.

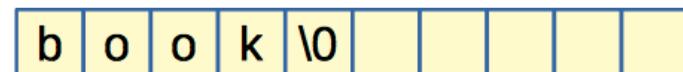
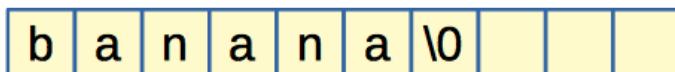
Copying Strings

- We have `strcpy()` to copy strings:

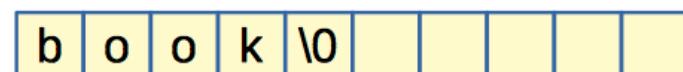
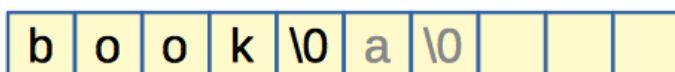
```
char *strcpy(char dest[], const char src[]);
```

- Copies from src to dest, up to and including the null terminator.
- Return a pointer to the start of the destination string (in case you want to do something else with it)

Before



After



Copying Strings

- We can copy from literal strings

```
strcpy( myString, "Greetings Program" );
```

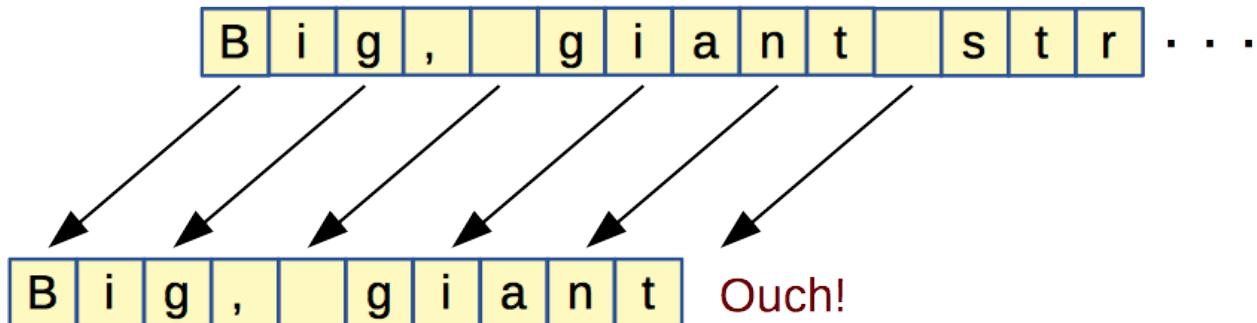
- We can copy from one char array to another.

```
strcpy( myString, name );
```

Copying Strings

- But, there's a potential for buffer overflow:

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

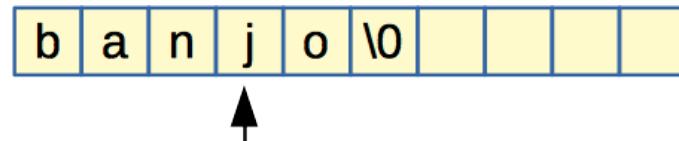
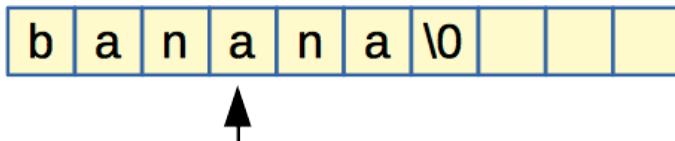


Comparing Strings

- We have `strcmp()` to compare strings:

```
int strcmp(const char s1[], const char s2[]);
```

- Lexicographically compares strings s_1 and s_2 , returning:
 - Less than zero if s_1 is before s_2
 - Zero if they are equal
 - Greater than zero if s_1 is after s_2
- Stops when it hits a null terminator or a character on which they differ.



Comparing Strings

- We can use it to compare strings for equality:

```
if ( strcmp( name1, name2 ) == 0 )  
    ...;
```

- It's easy to make a mistake if you just use `strcmp()` as a truth value.

```
if ( strcmp( passwd, "joshua" ) )  
    printf( "They're equal\n" );
```

Oops. I guess they're
not equal.

Parsing Numeric Values

- We can parse strings as numeric values
- Say you want to parse an int out of a string?
 - atoi() will do that for you (declared in **stdlib.h**)

```
int atoi(const char str[]);
```

String goes in here.

int comes out here.

Parsing Numeric Values

- atoi() has some friends, for other types of conversions
 - There's atof() for parsing strings as doubles.
`double atof(const char str[]);`
 - There's atol() for parsing strings as long ints.
`long atol(const char str[]);`
 - There's atoll() for parsing strings as long long ints.
`long long atoll(const char str[]);`
- These functions are OK (but I almost never use them)

Problems with atoi() and Friends

- Given an invalid parameter value, these functions return zero.
- So, you can't tell the difference between:

```
int x = atoi( "0" );
```

and

```
int x = atoi( "garbage" );
```

- Later, we'll see a general-purpose way to do all these conversions.