

SLOC

Source Lines of Code

What is a Line of Code?

- A way of measuring the quantity of source code
- Used in Software Engineering for Estimating Coding Projects
 - Generally useful for rough order of magnitude measurements
 - 1000 lines vs 10000 lines
 - Not very useful/accurate indicator for small differences in project size
 - 1000 lines vs 1200 lines
- Many factors affect productivity
 - Lines of Code by itself is not a good indicator of effort required
 - Primarily useful for comparing similar types of projects

How is a Line of Code Measured?

- Lots of Variation
- Common measurements:
 1. Physical lines - the number of lines in the source files
 2. Number of physical lines containing semicolons
 3. Number of statements - ignores comments and blank lines
 - For C source files this is approximately the number of semicolons in the files
 - Semicolons in comments or string literals do not count

Example

1 physical line,
1 physical line containing a semicolon
2 statements

```
welcomeDone = true; printf("hello\n"); // say hi;
```

Compared to

```
welcomeDone = true;  
// say hi;  
printf("hello\n");
```

3 physical lines,
3 physical lines containing a semicolon
2 statements

Which Measure is Best?

- Want the best representation of the amount of thought or work required to produce the code
- Want the measurement to be independent of style
- The count of the number of statements is generally preferred
- A count of the number of semicolons that are not inside a comment or string literal is a close approximation to the statement count

Limitations to This Approach

- Consider this example:

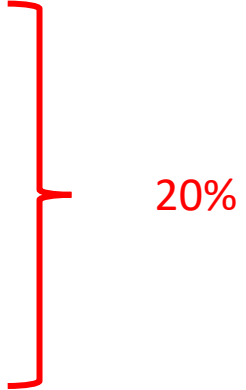

```
for (int i=0; i<10; ++i)
{
    printf("line %d\n", %d);
}
```

- This example has 2 statements but it has 3 semicolons.
- This is a limitation of the semicolon count approach to measuring lines of code

Threshold Requirements

- Build a tool to measure SLOC for a set of source files
- A list of source files can be specified on the command line
- Each source file will be analyzed and the results displayed
- The results are:
 - The number of physical lines (newline characters in the file)
 - The number of semicolons that are not part of a comment or string literal
- If more than one file is specified then a total will be provided at the end.
- If no files are specified then the tool will read from “standard input” and the results are displayed similar to how a single file is processed

Objective Requirements

- Appropriate names for variables
 - Well organized source code
 - Proper indentation
 - Variables should be local unless they need to be global
- 
- Submitted on time
- 

Bonus Objective Requirements

- Additional 10% credit for handling these additional cases:
 - Semicolon in a character literal:
`\;`
 - String literals containing a quote:
`"a \" inside a string literal"`
- Demonstrate with the bonus.c test file

Sample Input 1

A = 3; B = 5; C = 7;

- One physical line
- Three statements

Sample Input 2

```
char * str = "a little harder; ";  
/* note how the string literal  
   above contains a ;  
*/
```

- 4 physical lines
- 1 statement

This semicolon counts



These semicolons do not count



Sample Input 3

```
// this example uses the style
// of comments that start
// with two slashes “//” and end
// at the end of the line
// semicolons (;;;;;) in here don't
// count either
```

- 6 physical lines
- No statements

Bonus test case

```
// physical: 7   sloc: 3
int main()
{
    int v = 0;
    char c = ';';
    char * str = ";;;;;;;;;\";";
}
```

Example Output

- Multiple Files

```
./sloc sample_*.c
      3      1 sample_1.c
      1      4 sample_2.c
      0      6 sample_3.c
      4     11 Total
```

- One File

```
./sloc sample_2.c
      1      4 sample_2.c
```

- Standard Input

```
cat sample_2.c | ./sloc
      1      4
```

Design Approach

Solution has two parts:

- Command line processing
- Processing the content of a file

Command Line Processing

Are filenames present on the command line?

Yes:

- Open file and prepare to process it
- Call function to process the content of the file
- Display results for the file
- Display summary results if more than one filename is present

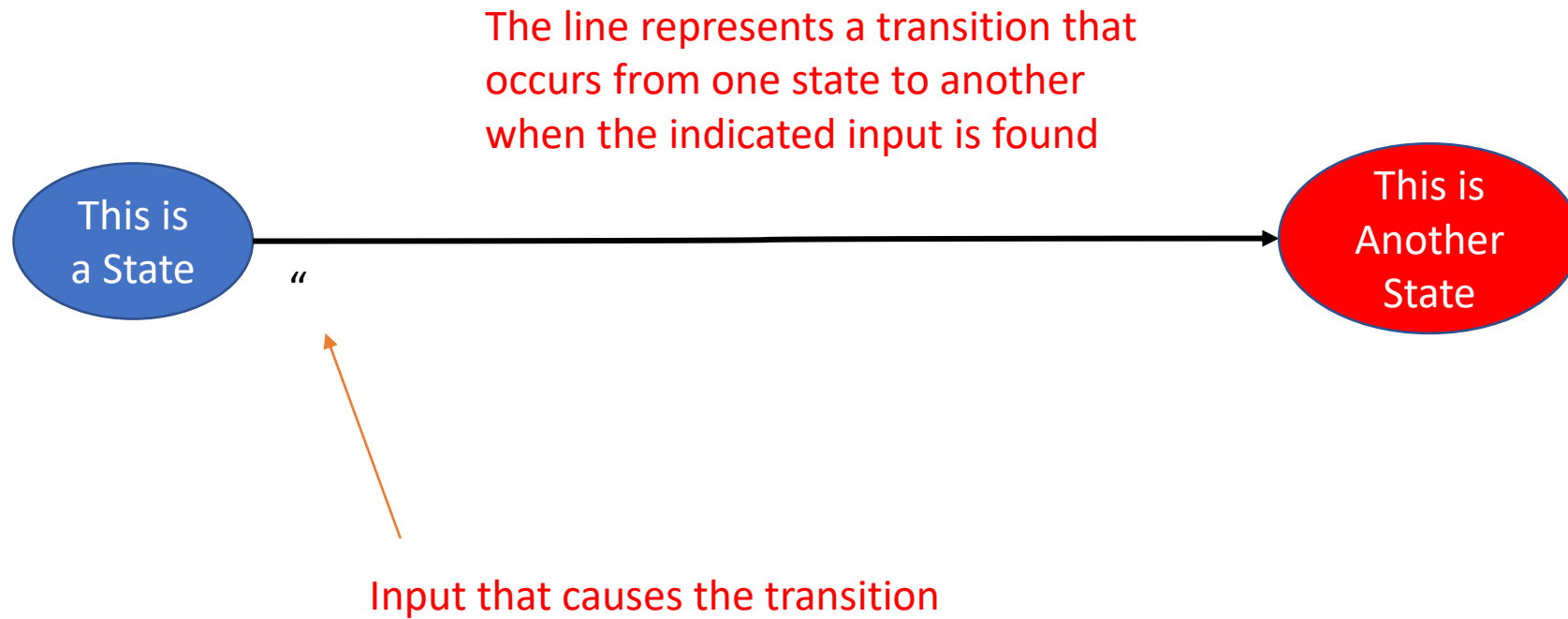
No:

- Process the content of stdin
- Display the results

Processing the Content of a File

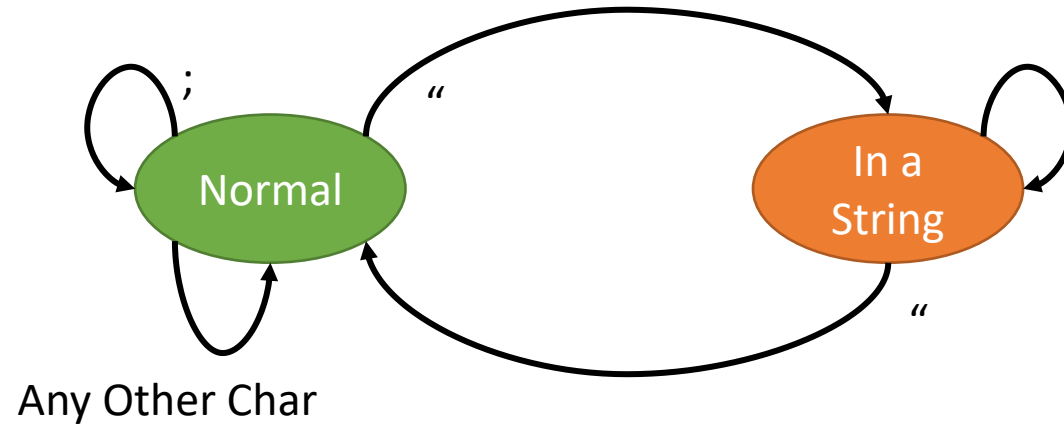
- Process the input one character at a time – use `getc` to read file
- Counting physical lines is counting each `'\n'` (newline) character
- Counting the number of statements requires keeping track of the *state* to determine which semicolons to count
 - Semicolons inside a string literal or comment do not count!
- States include:
 - Normal: ok to count any semicolon found
 - Inside a String: don't count semicolons
 - Inside a Comment: don't count semicolons
- Other states needed to determine when a comment starts/ends

State Diagram Conventions



State Diagram – Handling a String

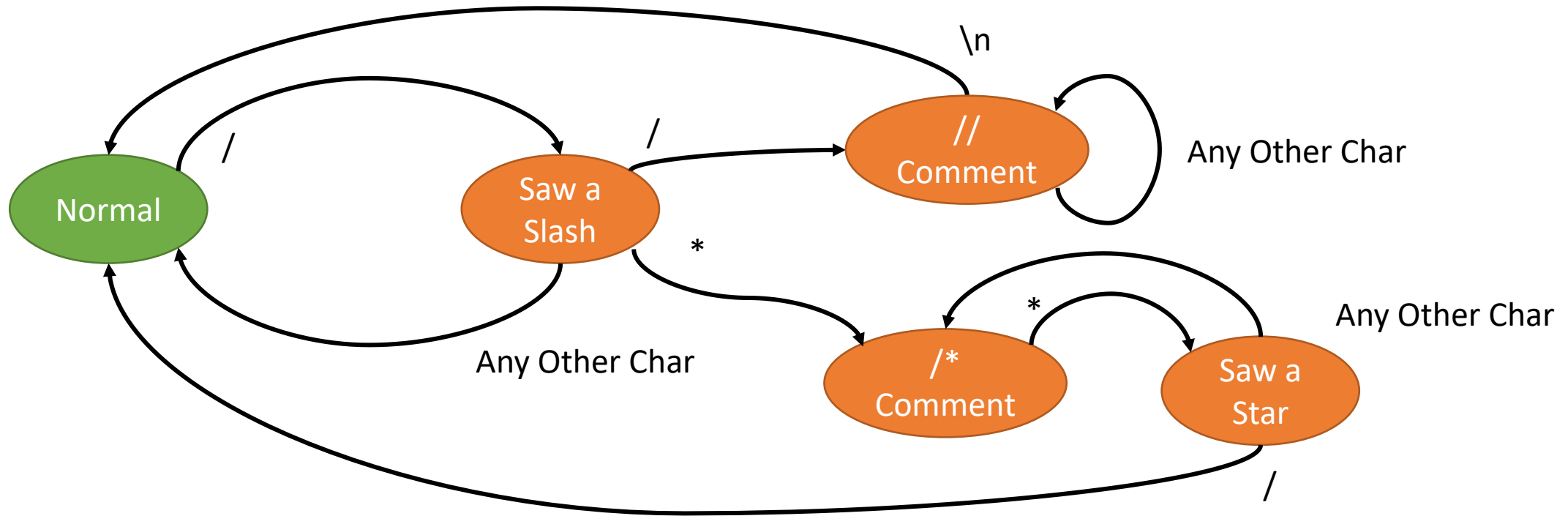
Semicolons are
counted in this
state



Any Other Char

Semicolons are
ignored in this
state

State Diagram – Handling a Comment



Real Life is More Complicated

A few things that you need not deal with in this project but would have to be dealt with in a “real” project:

- Character literals like: ‘;’ This is different than a string but the same idea applies. This should not be counted.
- Escaped characters like: “string with a quote \” inside it”
- Errors like unterminated strings and multi-line comments

Submission of Homework

- Homework due on **Wednesday, September 21**
- Submit printout of source code
- Submit screenshot of results – all three examples as shown on the next slide
- The 4 sample files are available in my directory:
 ~tburger/class/sloc_test_cases

Example Output

- Multiple Files

```
./sloc sample_*.c
      3      1 sample_1.c
      1      4 sample_2.c
      0      6 sample_3.c
      4     11 Total
```

- One File

```
./sloc sample_2.c
      1      4 sample_2.c
```

- Standard Input

```
cat sample_2.c | ./sloc
      1      4
```

Requirements

Threshold

- Produce the same results as shown in the sample output.
- Output does not have to be formatted exactly the same but the results do have to match

Objective

- Follow the style guide
- Submit on time