Console I/O and Compilation

CSC 230 : C and Software Tools

NC State Department of Computer Science

Topics for Today

- Console I/O
  - Character I/O
  - Meet printf() and scanf()
- Program Execution in Java and C
- Meet the Preprocessor
- Tokenization
- Coding Style

Console I/O In C

- In C, I/O is provided by functions in the *standard library*
  - This library is expected on all platforms
- To use the I/O parts of the standard library, you need to include the header file:

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

- We'll also get some use out of:

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

- These are *preprocessor directives*, telling the preprocessor to get these files and compile them along with our source code.

Streams

- A *stream* is a file or device we can read or write
- Just like in Java, a C program starts with three streams it can use
  - *Standard input* (input typed at the terminal)
  - *Standard output* (output to the terminal)
  - *Standard error* (more output to the terminal)
- To a program, reading and writing to the terminal looks just like reading or writing a file
  - We can even signal the End-Of-File condition on standard input:
    - Type CTRL-D if your program is running in Linux
    - Or, CTRL-Z if it's running in Windows.

Redirecting Standard Streams

- We can redirect these streams to or from actual files (without the program even noticing)
  - We won't learn about file I/O for a while, but this will let us get by without it.
- From the terminal, you can redirect standard input from a file

```
$ myProgram < input_file.txt
```

- … or standard output to a file.

```
$ myProgram > output_file.txt
```

Reading just one Character

- stdio.h provides a function
  `int getchar( void )`

It returns an int, the next character read.

It doesn't take any parameters.

- Returns the next character read (as a small non-negative integer)
- … or, if there's no more input, the constant EOF (the value -1)
- That's why it's return type is int instead of char (students seem to forget this)

Remember

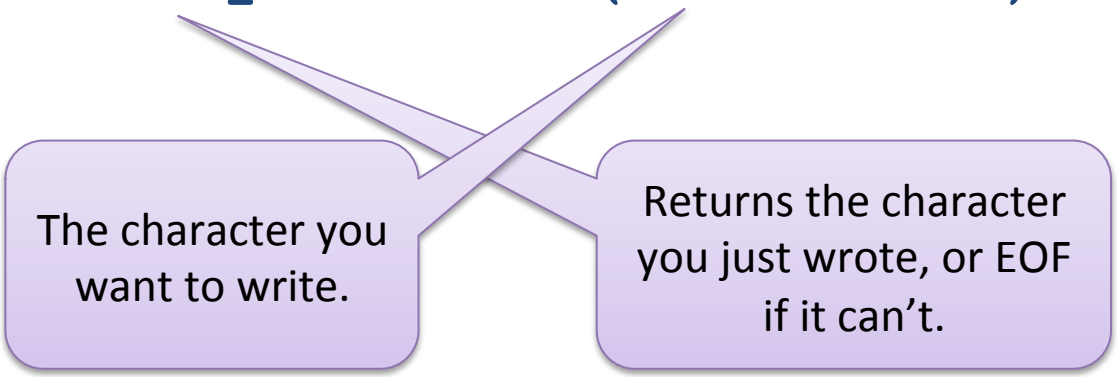
getchar()

returns **int**

Writing just one Character

- stdio.h also provides

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

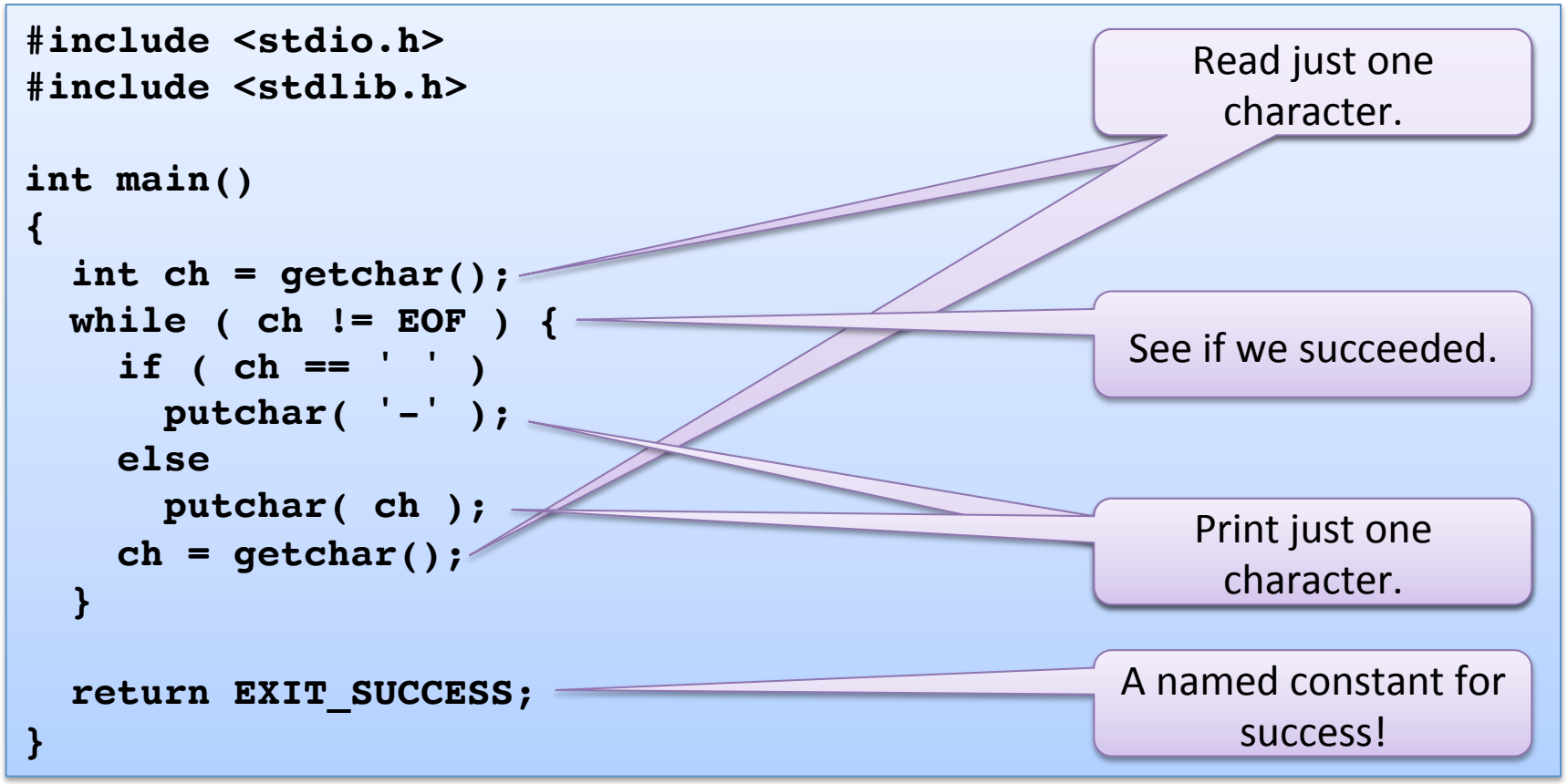


Character I/O Example

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

int main()
{
  int ch = getchar();
  while ( ch != EOF ) {
    if ( ch == ' ' )
      putchar( '-' );
    else
      putchar( ch );
    ch = getchar();
  }

  return EXIT_SUCCESS;
}
```



Formatted Output

- The printf() function is good for generating formatted output
- You probably saw a similar function in Java. It works like:

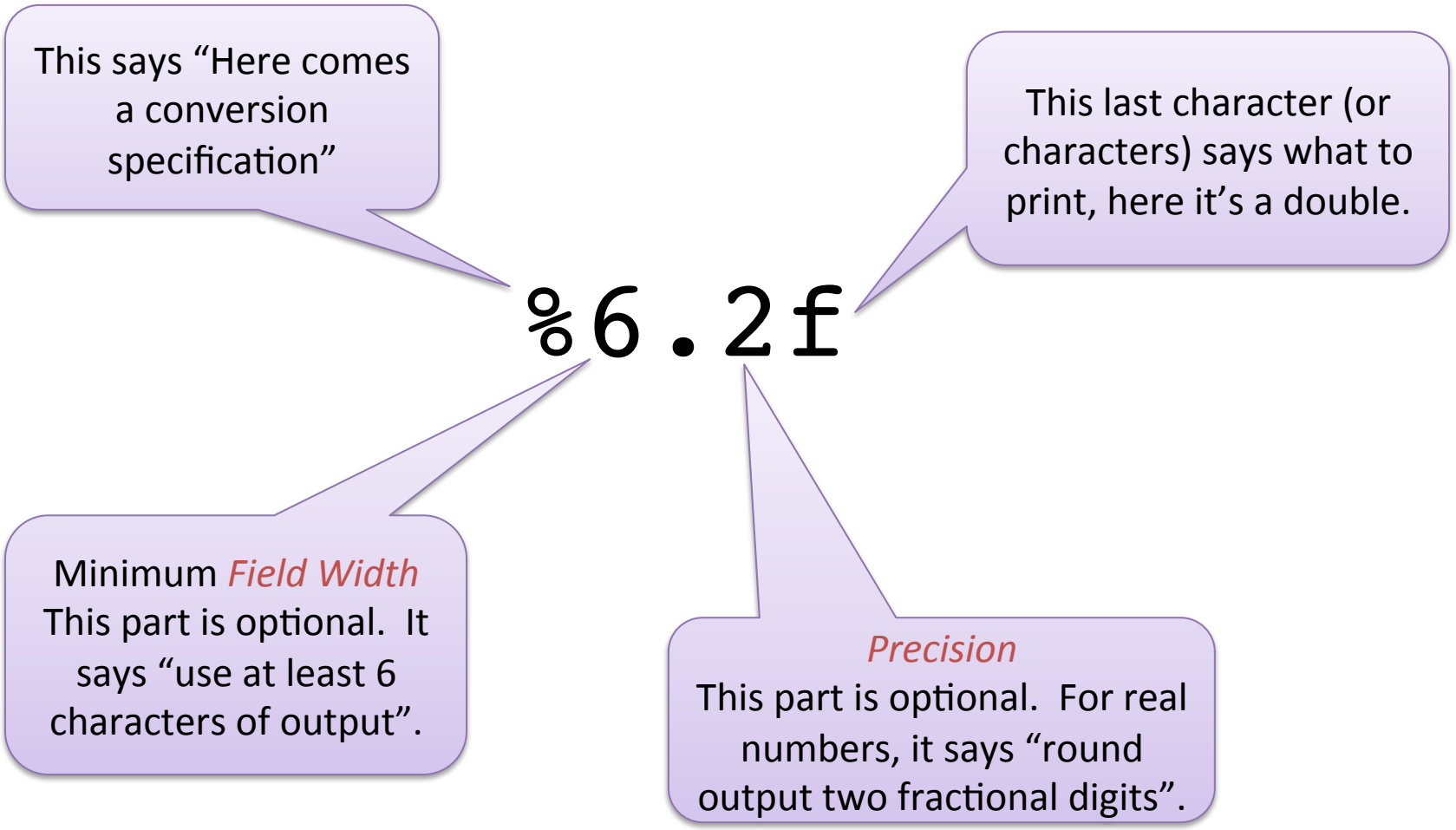
```
printf( "value: %6.2f\n", 3.1415926 );
```

This is a *format string*. Most of it gets printed literally.

… but not parts like this. This is a *conversion specification*.

Each conversion specification says how to print one of the remaining parameters.

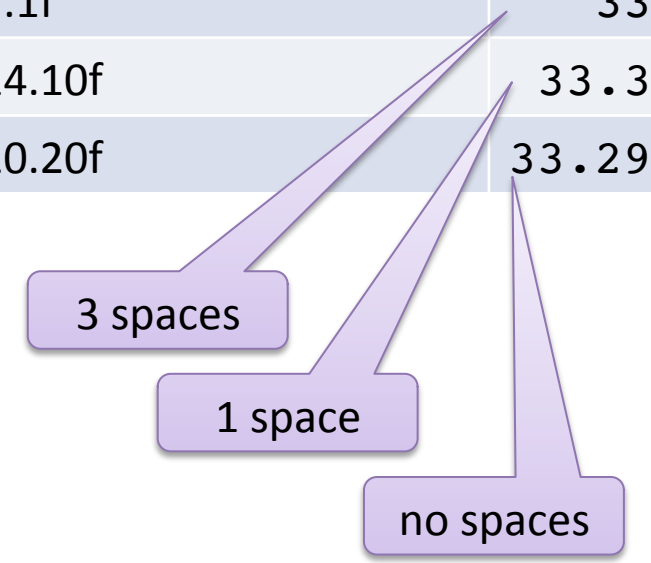
Making Sense of Conversion Specifications



Conversion Specification Examples

- There are lots of ways to print a value like 33.3:

Conversion Specification	Output
%7.1f	33.3
%14.10f	33.3000000000
%20.20f	33.29999999999999715783



Conversion Specification Overview

- printf() can output lots of different types of values.
- Here are some we'll need for now.

Format Specification	Output
%c	A single character (given its numeric character code)
%d	A decimal integer
%ld	A long integer
%f	A float or double
%s	A string
%zd	The size of some memory region (a value of type size_t)

Reading Formatted Input

- **printf()** is for creating formatted output to the terminal.
- Its friend, **scanf()**, is is for reading formatted input.
- The scanf() function converts sequences of characters into other types
  - Like int, double, string, …
  - It can skips whitespace and other input you tell it to ignore
- It's a lot like printf()

```
int scanf( const char *fmt, … )
```

  - It takes a format string, followed by a variable number of additional arguments
  - It matches characters from the input and assigns them to the subsequent arguments.

Using scanf()

- The format string can contain one or more *conversion specifications*.
- Each of these fills in the value of one of the remaining arguments.
- You can't do this with *pass-by-value*

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

This couldn't possibly work

- So, C uses a mechanism for *pass-by-reference*

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

With the &, we're passing val so scanf can change what's in it.

Using Scanf

- scanf() can match multiple conversion specifications.

```
scanf( "%f%d", &floatVal, &intVal );
```

- It returns the number of specifications matched

```
int matches = scanf( "%f%d", &floatVal, &intVal );
```

  - Fewer on bad input
  - … or EOF if it reaches the end-of-file before matching any.
  - And, it only changes a parameter if it can match it successfully.

Using Scanf

- So, you could do something like:

```
printf( "Please enter an integer: " );

int val;
int matches = scanf( "%d", &val );
if ( matches == 1 ) {
  printf( "You entered %d\n", val );
} else if ( matches == 0 ) {
  printf( "That's not an integer.\n" );
} else {
  printf( "You didn't enter anything.\n" );
}
```

Conversion Specification Zoo

- scanf() understands almost exactly the same conversion specifications as printf()
- … but, there are some differences 😐

Parse input as	Conversion Specification
int	%d (in decimal)
float	%f
double	%lf
long int	%ld
A single character	%c
A whole string	%s

Matches the longest sequence of characters that looks like an int.

… longest sequence of characters that looks like a real number.

This one is a little different.

Most of these skip leading whitespace … but not this one.

Fun with Scanf

- Consider the code:

```
float fval = 1.0f;
int ival = 2;
int matches = scanf( "%f%d", &fval, &ival );
```

- On the following input, what value will these three variables have?

```
   3      4.12
```

- How about this input?

```
37.22b5
```

Formatted I/O Example

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

int main()
{
  int val;

  while ( scanf( "%d", &val ) == 1 ) {
    printf( "%6d\n", val );
  }

  return EXIT_SUCCESS;
}
```

While we keep matching one decimal integer.

Print it in a 6-character field.

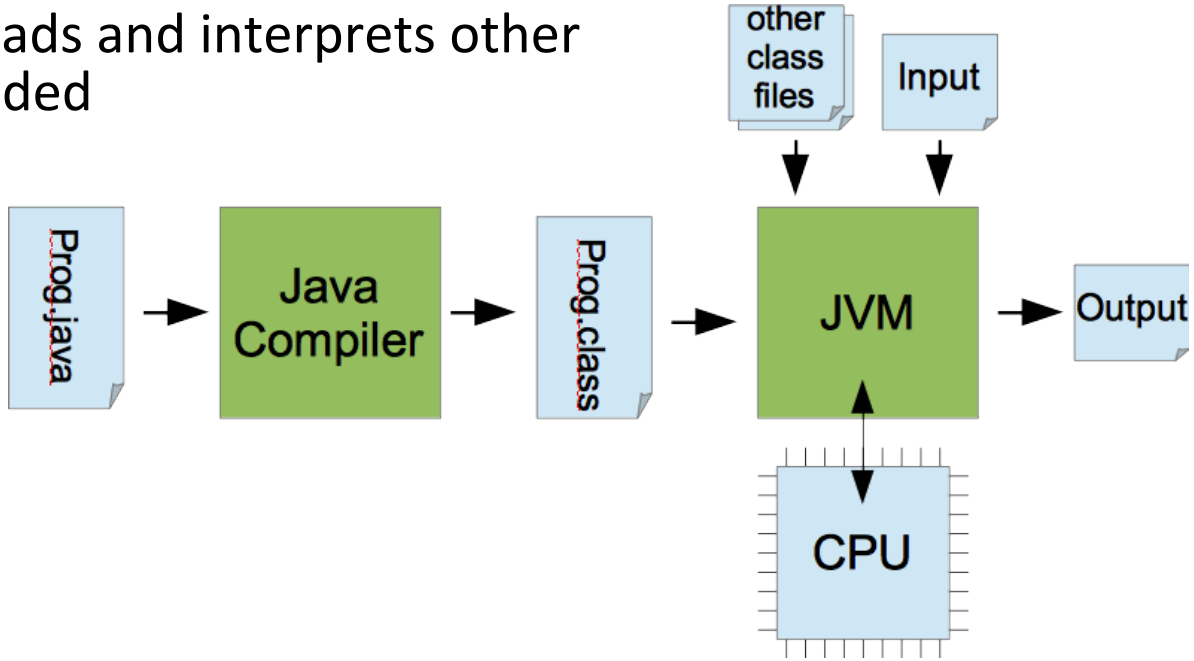
```
25
   -12   77
green
62
```

output

```
    25
   -12
    77
```

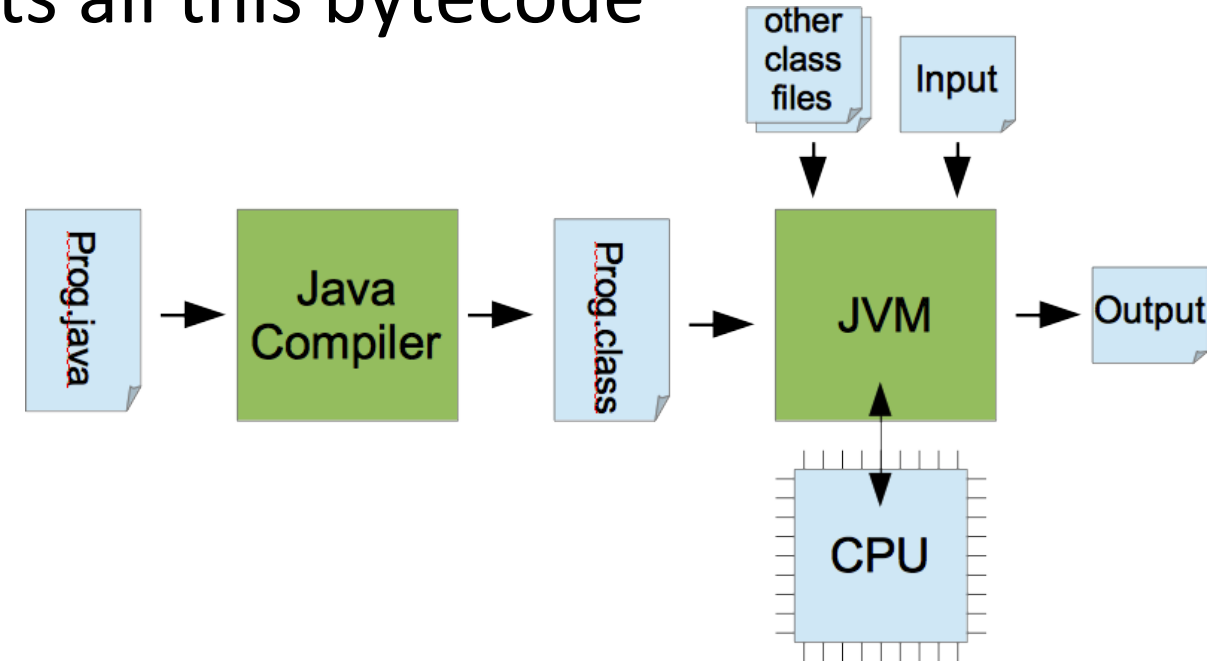
Executing Java Programs

- Java source code is *compiled* into Java class file containing *bytecode*
  - A platform-independent, intermediate representation
- To run it, we need an interpreter, the Java Virtual Machine
  - Takes a class file as input, runs native machine code to simulate each bytecode instruction
  - Automatically loads and interprets other class files as needed



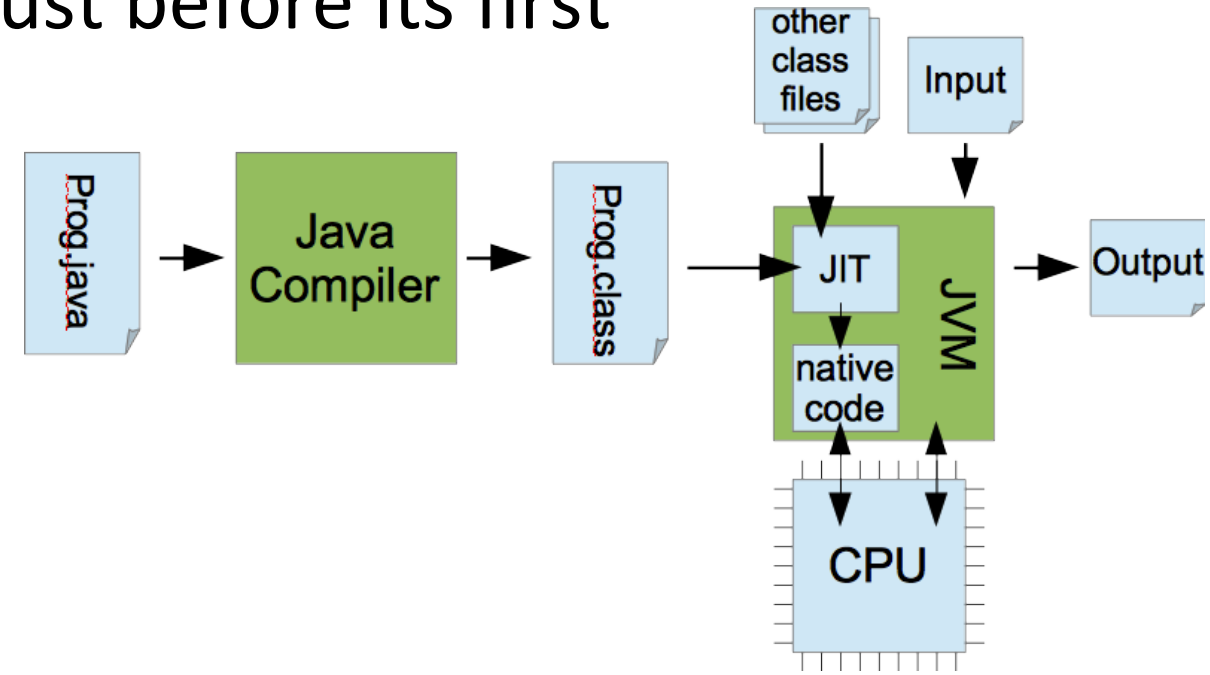
Executing Java Programs

- This is great. The class files for our compiled program are platform independent.
- But, some extra overhead is incurred as the JVM interprets all this bytecode



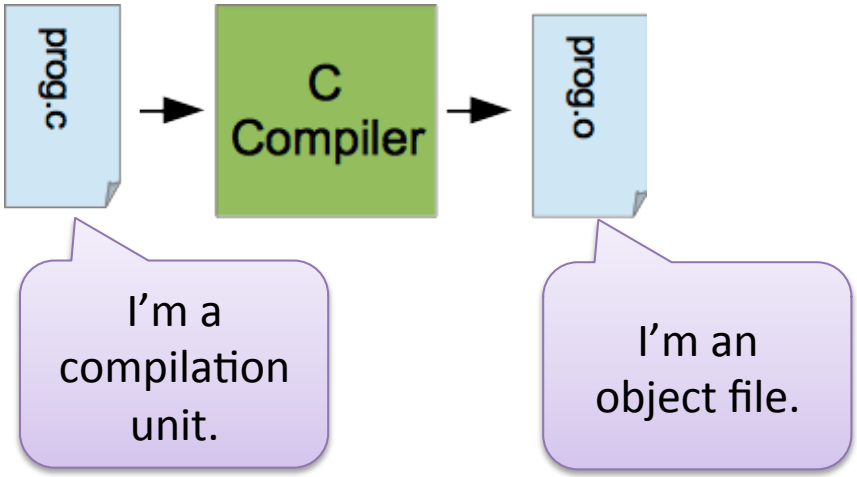
Executing Java Programs

- With Just-In-Time compilation, Java can get closer to native processor speeds
- Each method is compiled to native machine instructions just before its first execution



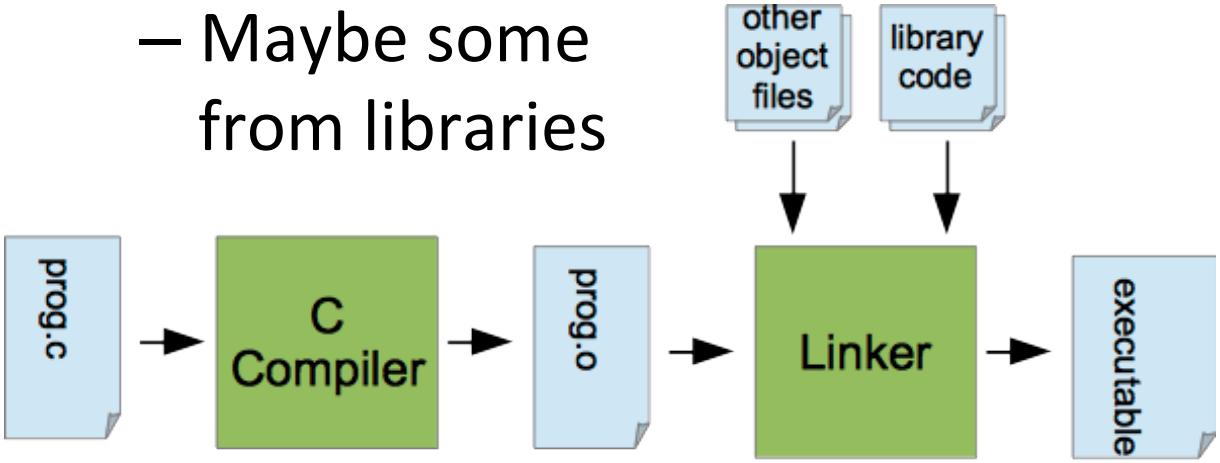
Executing C Programs

- A C Compiler generates native machine code for the target processor
- One *compilation unit* generates one *object file*



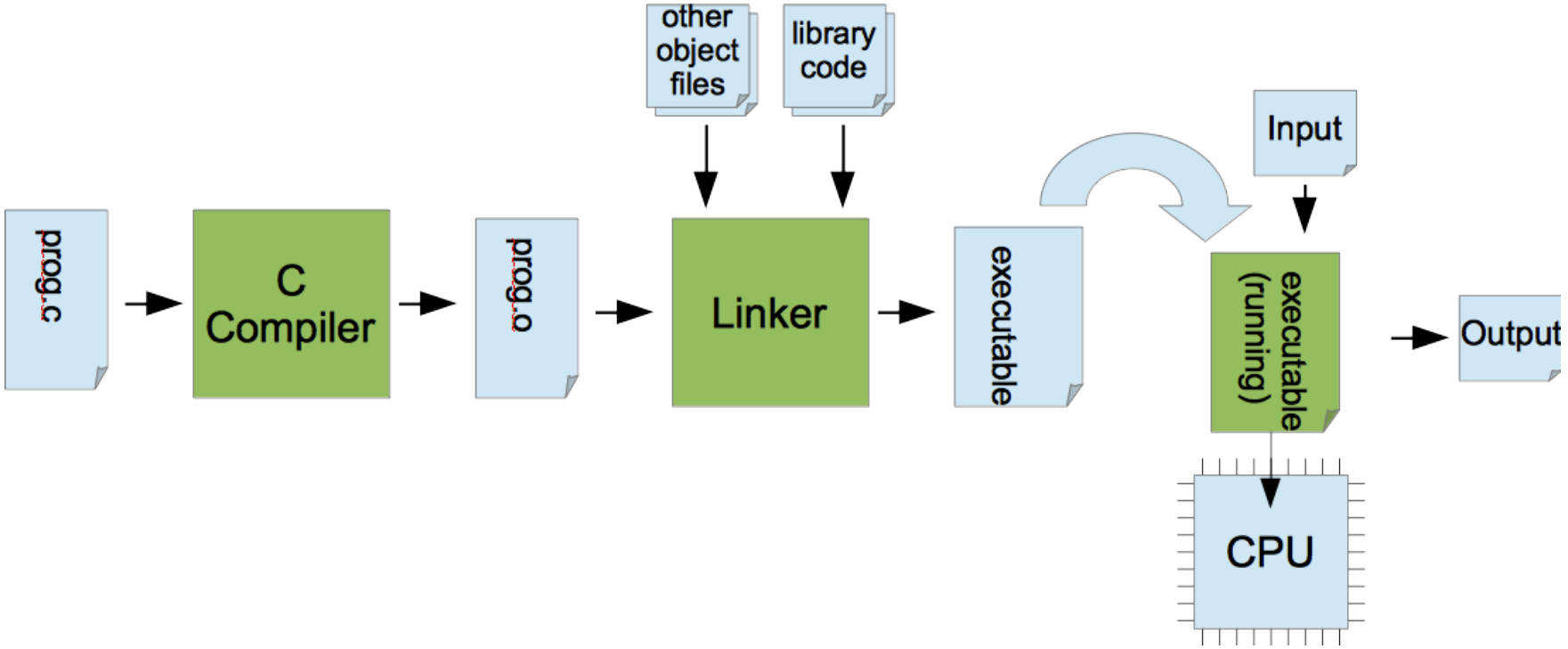
Executing C Programs

- A linker combines object files to create an executable program
  - Maybe some other objects we wrote
  - Maybe some from libraries



Executing C Programs

- The executable is ready to run
- Just load it into memory and start running at the top of main()



Compiled vs Interpreted

- Each approach has advantages.
- What do you think? Would compiled code (compiled to the native instruction set) execute faster than interpreted?
- Which would offer better support for debugging and error messages?
- Which would offer greater platform independence?
- Which would offer more opportunities for code analysis and optimization?

The Preprocessor

- The preprocessor operates on the source code before the compiler even sees it.
- Performs basic text operations
  - *Includes headers* : inserting code that enables use of code from other components
  - *Expands macros* : replacing macro names with corresponding definitions
  - More things we'll learn about later
- Lines starting with # are *preprocessor directives*
  - instructions processed (and removed) by the preprocessor

Preprocessor Constants

- Preprocessor macros give us a way to define named constants:

```
#define SIZE 25
```

Replace occurrences of this …

… with this.

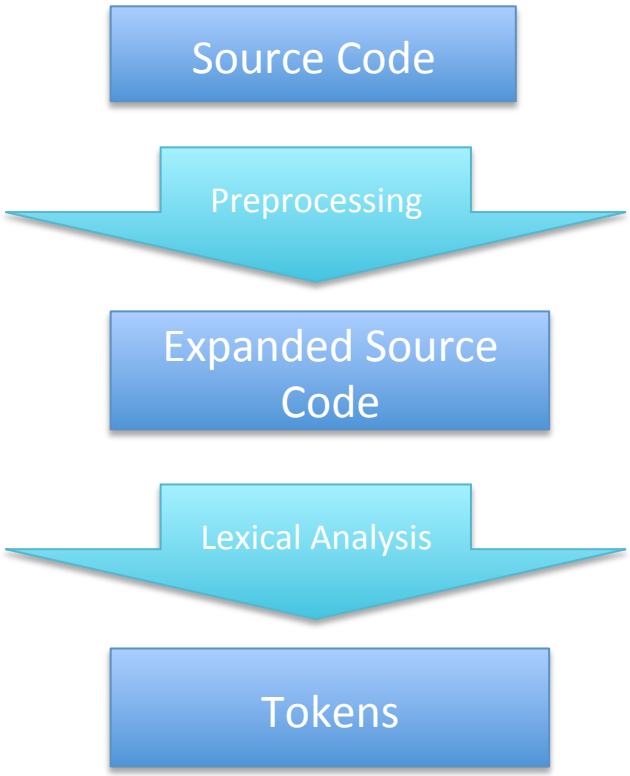
Be careful, you probably don't want a semi-colon here.

```
for ( int i = 0; i < SIZE; i++ )
  …
```

```
for ( int i = 0; i < 25; i++ )
  …
```

Steps in C Compilation

- Really, generating an executable has more steps than you might expect.

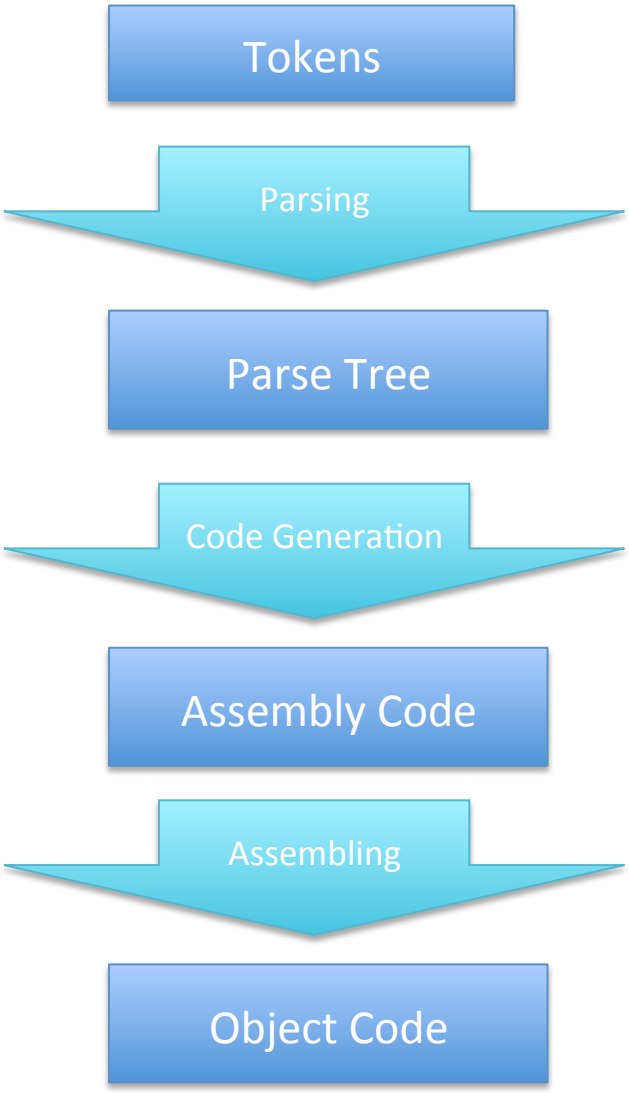



```
#define N 3
length += base * N;
```

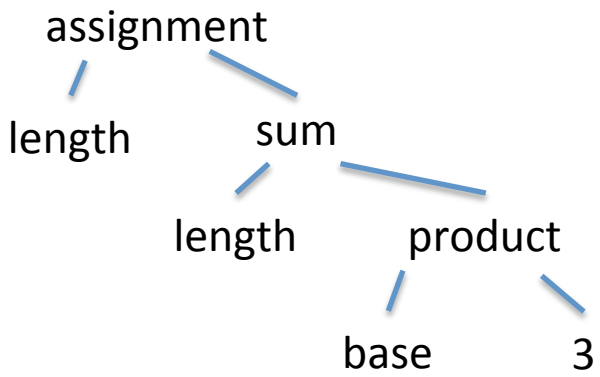
```
length += base * 3;
```

```
length   +=   base   *   3   ;
```

Steps in C Compilation



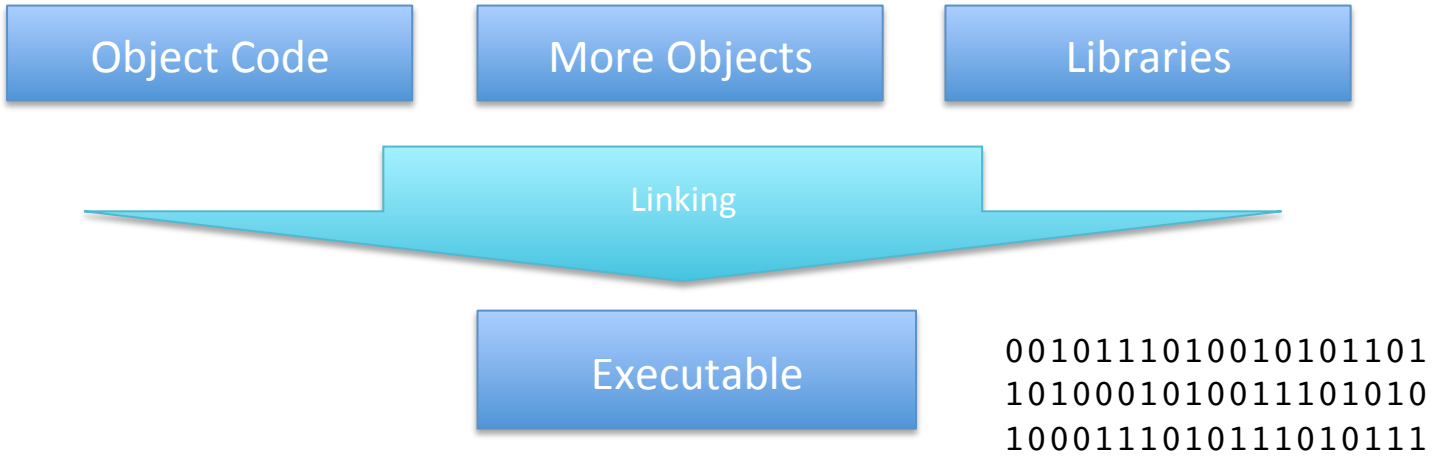
length += base * 3 ;




```
mov ebx, base
imul ebx, 3
iadd ebx, length
mov length, ebx
```

```
0010111010010101101
```

Steps in C Compilation



Looking for Tokens

- The compiler has to break the source into *tokens*
- This is called *lexical analysis*
- A token can be:
  - An *identifier* (e.g., a variable or a function name)
  - A *keyword* (e.g., void or while)
  - A *literal value* (e.g., 3.1415, or “Hello World”)
  - An *operator* (e.g., *****, **++** or **>=**)
  - An *explicit separator* (e.g., **(**, **}** or **;**)
- White space between the tokens is ignored
  (except, of course, that it can separate tokens)

Identifiers

- *Identifiers* are variable, function or other names
- We get to choose names for these, but there are rules:
  - An identifier consists of letters, digits or underscore
  - But, a variable can't start with a digit
    - So `x2` is an identifier
    - but `2x` is a number followed by an identifier
- Identifiers are case sensitive in C
  - So `myValue` is not the same as `myvalue`

Reserved Keywords

- Of course, some words in C already have special meaning
- In C89, you can't use these as identifiers

```
auto, break, case, char, const,
continue, default, do, double, else,
enum, extern, float, for, goto, if,
int, long, register, return, short,
signed, sizeof, static, struct,
switch, typedef, union, unsigned,
void, volatile, while
```

- C99 adds a few more:

```
_Bool, _Complex, _Imaginary, inline,
restrict
```

Fun with Lexical Analysis

- What are the tokens in:
  `a + ++b >= c-- -d`
- How about now:
  `a+++b>=c---d`
- There are lots of things this could mean:

```
a ++ + b >= c - -- d   ?

a ++ +b > = c- - -d   ?

a + + + b > = c - - - d   ?
```

- This isn't really about precedence.
  - We can't even think about precedence until we know what the operators are.

The Scanner is Greedy

- The scanner works from left to right, grabbing the longest token it can
  - This is called *maximal munch*
- So, for our example:

```
a +++b>=c---d        (because a+ isn't a token)
a ++ +b>=c---d       (because +++ isn't a token)
a ++ + b>=c---d      (because +b isn't a token)
a ++ + b >=c---d     (because b> isn't a token)
a ++ + b >= c---d    (because >=c isn't a token)
a ++ + b >= c ---d   (because c- isn't a token)
a ++ + b >= c -- -d  (because --- isn't a token)
a ++ + b >= c -- - d (because -d isn't a token)
```

Be the Scanner

- In the following expression, how many tokens are there?

```
----xy+-=x++y*/z;
```

- What are they?
- This expression wouldn't parse, but we can still talk about what the scanner would do with it.

Coding Style Conventions

- There are lots of ways you **can** write a working program
- But there's a difference between what you can do and what you should do
- Consider this submission from the first International Obfuscated C Code Contest:

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

  - I'm told it's a "Hello World" program

Coding Style Conventions

- Or this one from the 1993 contest:

```
O5(O2,O7,O3)char**O7;{return!(O2+=~O1+O1)?OO:!(O2-=O2>O1)?printf("\045\157\012"
,O5(O12,O7+O1,OO)):!(O2-=O2>>O1)?(**O7<=067&&**O7>057?O5(O3,O7,*(*O7)++-060+O10
   *O3):O3                    ):!(O2      -=-O3-      ~O3)?       (O72>**
   O7&&O60                    <=**O7      ?O5(O4      ,O7,O12     *O3-O60
    +*(*O7                    )++):O3     ):!(O2      -=!O3+      !!O3)?(
    **O7>057                  &&**O7      <=071?      O5(O5,      O7,*(*
     O7)+++                   O3*O20      -O60):      **O7<=      O106&&
     OO101<=                  **O7?O5     (O5,O7      ,O20*O3     +*(*O7)
      ++-O67)                 :O140<**    O7&&**      O7<O147     ?O5(O5,
     O7,-O127                 +*(*O7      )+++O20     *O3):O3     ):!(
      O2-=O2-                 O1)?(**     O7==O50     ?O50**      ++*O7,
      O5(O13,                 O7,O5(      O12,O7      ,OO)):*     *O7<O56
       &&O54<*                *O7?O55     **++*       O7,-O5(     O6,O7,
       OO):O54                >**O7&&     O52<**      O7?O50*     *(*O7)
        ++,O5(O6              ,O7,OO      ):!(**      O7^O170     )||!(
        O130^**               O7)?*++     *O7,O5      (O5,O7      ,OO):*
         *O7==O144            ||**O7      ==O104      ?++*O7      ,O5(O4,
          O7,OO):             O5(O3       ,O7,OO      )):!--      O2?(*
          *O7==O52           ?O5(O7       ,O7,O3*     (*++*O7     ,O5(O6
          ,O7,OO)            ):!(         O45-**      O7)?O5(     O7,O7,
           O3%(O3+( *O7)++,                O5(O6,      O7,OO)      )):!(**
            O7^O57)?O5(O7,                 O7,O3/(     O3-*++      *O7,O5(
            O6,O7,OO))):O3                 ):!(O2      +=O1-O2     )?O5(O7
             ,O7,O5(O6,O7,                 OO)):!(     O2+=-O2/    O2)?(!(*
*O7-O53)?O5(O11,O7,O3+(++*O7,O5(O10,O7,OO))):!(O55^**O7)?O5(O11,O7,O3-(O3+*(*O7
)++,O5(OO10,O7,OO))):O3):!(O2-=0563&0215)?O5(O11,O7,O5(O10,O7,OO)):(++*O7,O3);}
```

Coding Style Conventions

- These examples are fun (?) but deliberately hard to read and understand
- Normally, this is the opposite of what we want
- In CSC 230, we adopt some coding style conventions, rules for:
  - Naming conventions
  - Spacing and indentation
  - Where important comments go and what they contain
- Fortunately, editors can often help us with this.

CSC 230 Style Guidelines

- A Javadoc-style block comment at the top of each source file
  - With a @file tag giving the filename
  - And an @author tag with name and unity ID.
  - And a brief description of what the program does

```
/** I'm a program that reads in a list of words.
    @file wordList.c
    @author Bill Smith bsmith97
  */

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

…
```

CSC 230 Style Guidelines

- A Javadoc-style block comment at the top of each function:
  - A sentence or two describing the function's purpose
  - @param tags for each parameter
  - @return tag for the return value

Omit if not needed.

```
/** Print a given number of copies of a given character.
  * @param ch The character to print
  * @param n The number of times to print ch.
  */
void printSymbol( char ch, int n )
{
  . . .
```

Extra stars are OK, if you like them.

CSC 230 Style Guidelines

- A good, Javadoc-style comment on each constant, global variable and type definition.
- Magic numbers, avoid bare constants for:
  - **Any value that could have an explanation**

```
area = radius * radius * 3.1415926;
```

  - **Any potentially tunable parameter**

```
score += 350;
```

  - **Any value that needs to occur multiple times**

```
for ( int i = 0; i <= 99; i++ )
  …;
```

```
for ( int j = 99; j >= 0; j++ )
  …;
```

CSC 230 Style Guidelines

- Curly bracket placement
  - For function definitions, it goes on the next line (to make functions stand out)
  - For everything else, it goes on the same line
- Indentation
  - No hard tabs, just indent with spaces (why?)
  - Indent using any number of spaces you want, 2 spaces, 3 spaces … maybe 4 spaces.
  - But, be consistent.
- Just one statement per line

CSC 230 Style Guidelines

- Global Variables
  - Avoid these. Only use them where we tell you to.
  - So, functions communicate with the rest of the program via parameters and return values.

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

/** Any code can access me. */
float cost = 25.88;

void someFunction()
{
  . . .
```

Don't do this, unless the design says to.