

# CS 241: Systems Programming

## Lecture 8. Introduction to C

Fall 2019

Prof. Stephen Checkoway

# Standardization

1972 - Traditional C (Dennis Ritchie)

1978 - K&R C

1983 - ANSI C committee formed

1989 - committee adopted standard C89 (ANSI C)

- Adds in the standard library, etc.

1990 - ISO adopts C89 as international standard

1995 - Adopted two “Technical Corrigenda” and an “Amendment” to form C95

- Mostly new library functions for multibyte and wchar

1999 - More extensive revisions create C99

- Inline funcs, new data types, C++ style comments, variable len arrays

2011 - C11 approved adding features like multithreading, atomic operations, type-generic macros, Unicode support, etc.

# Hello, World!

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

```
int main(void) {  
    printf("Hello world!\n");  
    return 0;  
}
```

# Jobs of a Compiler

## Inputs

- C program file and options
- Libraries

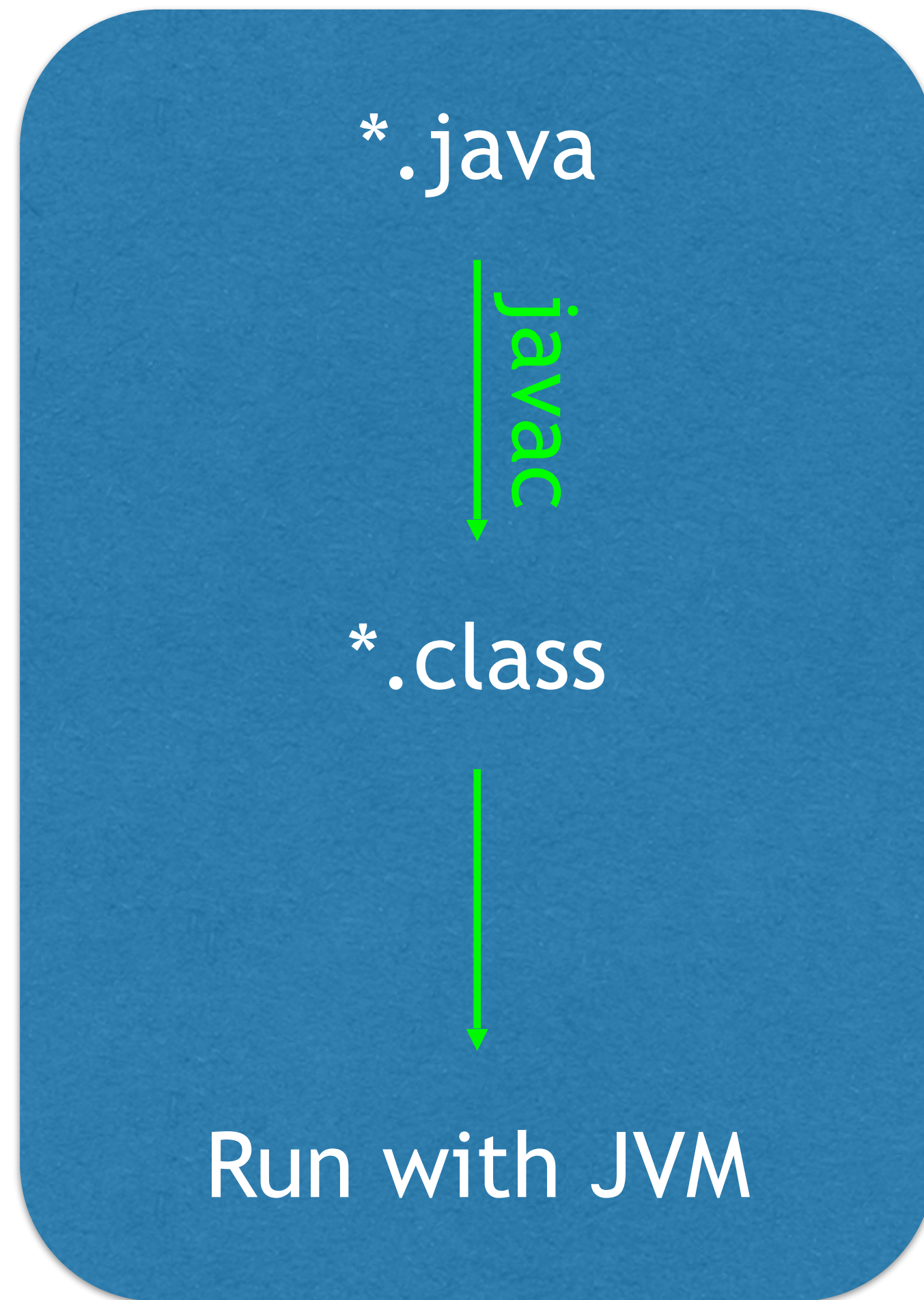
## Compilation phases

- Preprocessing
- Compilation
- Linking

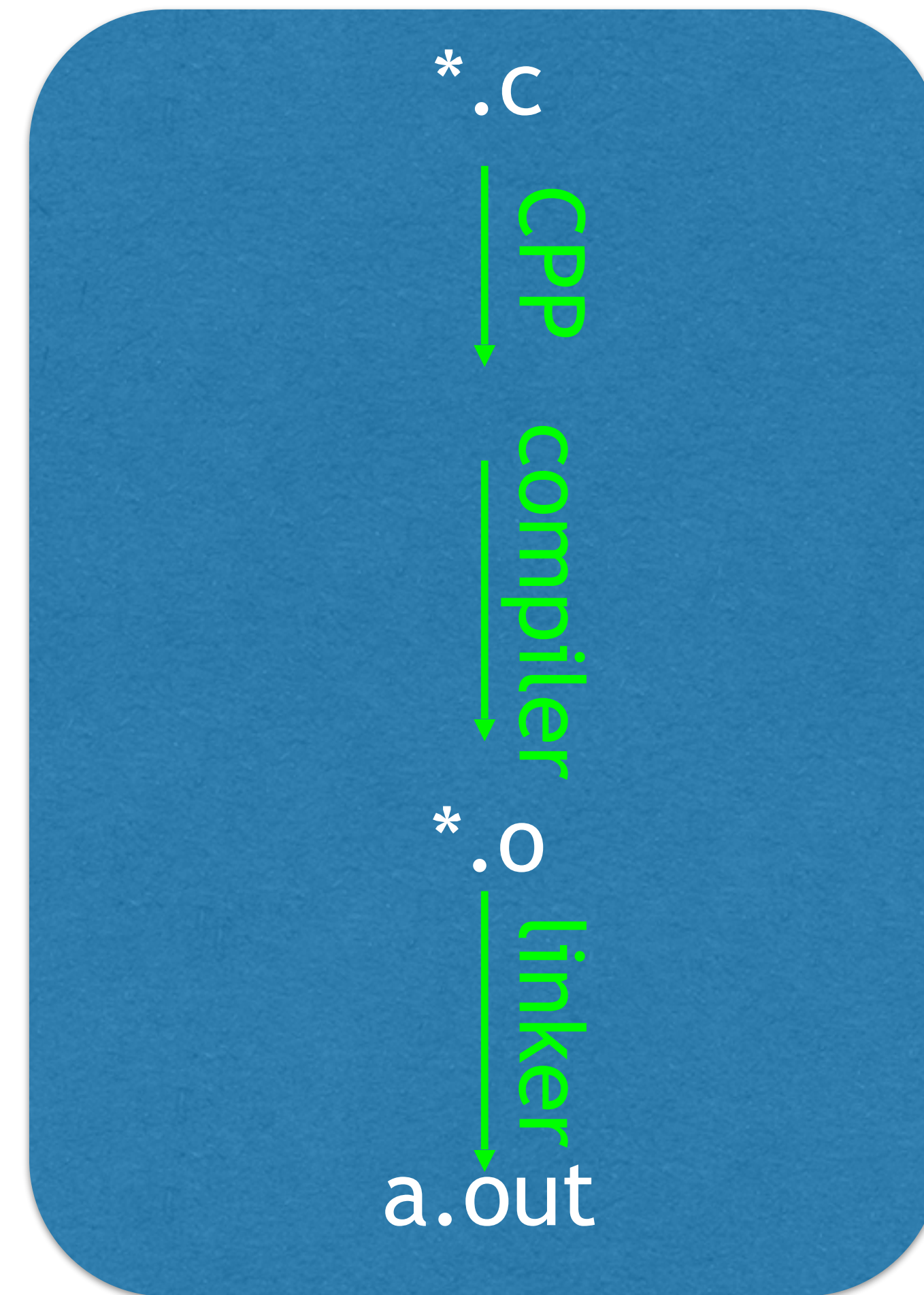
## Outputs

- Executable
- Warnings and errors

# Compilation



Java Model



C Model

# C Preprocessor Directives

`#include` — literal inclusion of a file

- ▶ `#include <foo.h>`
- ▶ `#include "foo.h"`

`#define foo bar` — literal replacement of “foo” with “bar”

- ▶ Useful for symbolic constants (and other things)
- ▶ Use UPPERCASE for constants
  - Usually these are at the top of the file



# Difference between "" and <>

2 A preprocessing directive of the form

**# include** *<h-char-sequence> new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. **How the places are specified or the header identified is implementation-defined.**

# Difference between "" and <>

3 A preprocessing directive of the form

```
# include "q-char-sequence" new-line
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched

for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
# include <h-char-sequence> new-line
```

with the identical contained sequence (including > characters, if any) from the original directive.



Consider the two files header\_file.h and source\_file.c shown to the right.

What is the value of x after the first line of main?

A. 10

B. 12

C. 20

```
// In header_file.h
#define BAR 10
#define FOO BAR+1
```

```
// In source_file.c
#include "header_file.h"
int main(void) {
    int x = FOO * 2;
    /* ... */
}
```

D. 22

E. It's an error

# Functions

```
/* Function declaration.  
 * - No return value.  
 * - Has three parameters, parameter names are optional.  
 * - Ends with a semicolon.  
 */
```

```
void foo(int x, float y, char z);
```

```
/* Function definition.  
 * - Must match declaration.  
 * - Parameter names are not optional.  
 * - Body of function wrapped in { }.  
 */
```

```
void foo(int x, float y, char z) {  
    /* ... */  
}
```

# Main function

```
// The main function is where execution begins.  
// - Returns an int, 0 is success, 1-127 are failure.  
// - Takes 0, 2, or implementation-defined number of parameters.  
// - argc is the number of command line parameters.  
// - argv points to an array of command line parameters.  
int main(void) { /* ... */ }  
int main(int argc, char **argv) { /* ... */ } // Use this one.  
int main(int argc, char **argv, char **envp) { /* ... */ }
```

# Command line parameters

```
1 // stdio.h contains printf's declaration.
2 #include <stdio.h>
3
4 // argc is like Bash's $# (but off-by-one)
5 // argv[0] is like $0
6 // argv[1], ..., argv[argc-1] is like $1, $2 ...
7 int main(int argc, char **argv) {
8     for (int idx = 0; idx < argc; ++idx) {
9         // %d means print an integer,
10        // %s means print a string
11        printf("%d: %s\n", idx, argv[idx]);
12    }
13    return 0;
14 }
```

# Command line parameters

```
$ ./arguments 'First argument' second third etc.  
0: ./arguments  
1: First argument  
2: second  
3: third  
4: etc.
```



# Basic types

class		systematic name	other name
integers	unsigned	<b>_Bool</b>	<b>bool</b>
		<b>unsigned char</b>	
		<b>unsigned short</b>	
		<b>unsigned int</b>	<b>unsigned</b>
		<b>unsigned long</b>	
		<b>unsigned long long</b>	
	<u>[un]signed</u>	<b>char</b>	
		<b>signed char</b>	
	signed	<b>signed short</b>	<b>short</b>
		<b>signed int</b>	<b>signed or int</b>
		<b>signed long</b>	<b>long</b>
		<b>signed long long</b>	<b>long long</b>
floating point	real	<b>float</b>	
		<b>double</b>	
		<b>long double</b>	
	complex	<b>float _Complex</b>	<b>float complex</b>
		<b>double _Complex</b>	<b>double complex</b>
		<b>long double _Complex</b>	<b>long double complex</b>

# Integer type sizes

`sizeof (type)` is the number of bytes a variable of `type` has

$$1 = \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \\ \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$$

`sizeof (type) = sizeof (signed type) = sizeof (unsigned type)`

`sizeof (bool)` is implementation defined

A byte isn't always 8 bits! (But it is on most systems.)

On a system where `sizeof(unsigned short)` is 2 (and bytes are 8 bits), the maximum value of an unsigned short is 0xFFFF (or 65535). Consider

```
unsigned short x = 0xFFFF;  
unsigned short y = 1;  
unsigned short z = x + y;
```

What value does `z` have after this code runs?

- A. 0
- B. 0xFFFF (65535)
- C. 0x10000 (65536)
- D. It depends on the compiler
- E. It's a runtime error.

On a system where `sizeof(unsigned short)` is 2 (and bytes are 8 bits), the maximum value of an unsigned short is 0xFFFF (or 65535). Consider

```
unsigned short x = 0xFFFF;  
unsigned short y = 1;  
unsigned int z = x + y;
```

What value does `z` have after this code runs?

- A. 0
- B. 0xFFFF (65535)
- C. 0x10000 (65536)
- D. It depends on the compiler
- E. It's a runtime error.

# Operators

The same as Java

- ▶ Arithmetic: +, −, \*, /, %
- ▶ Logical: &&, ||, !
- ▶ Bitwise: &, |, ^, ~, <<, >>
- ▶ Pre/post increment, decrement: ++, --
- ▶ Relational: ==, !=, <, <=, >, >=
- ▶ Assignment: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=

There are some others we'll talk about later

- ▶ `sizeof`
- ▶ `.`
- ▶ `->`



# Control flow

if statements; for, while, do-while loops almost identical to Java

zero is **false**, nonzero is **true**

# Compiler options (gcc/clang)

-E	preprocessor only
-S	compile only (no assembly or linking)
-c	compile/assemble (produce .o file)
-o <b>foo</b>	specify output file as <b>foo</b>
-l <b>xxx</b>	use library named <b>libxxx.so</b> or <b>libxxx.a</b>
-g	emit debugging symbols (enables debugging)
-std=c11	use C11 standard
-pedantic	be pedantic
-W <b>all</b>	turn on "all" warnings
-W <b>extra</b>	turn on extra warnings
-W <b>error</b>	make warnings into errors

# Compiling code (short version)

```
$ <compiler> <options> <.c files> <libraries>
```

```
$ clang -Wall -o program -std=c11 *.c -lm
```

If you omit `-o output`, the default is `a.out`

If you omit `-std=c11`, `clang` and `gcc` have different defaults!

# Formatting your code

It's important to be consistent more than anything else

Use tools!

```
$ clang-format foo.c      # Writes formatted code to stdout
```

```
$ clang-format -i foo.c   # Writes formatted code back to foo.c
```

# In-class exercise

<https://checkoway.net/teaching/cs241/2019-fall/exercises/Lecture-08.html>

Grab a laptop and a partner and try to get as much of that done as you can!