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!

```c
#include <stdio.h>

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

Jobs of a Compiler

Inputs

- C program file and options
- Libraries

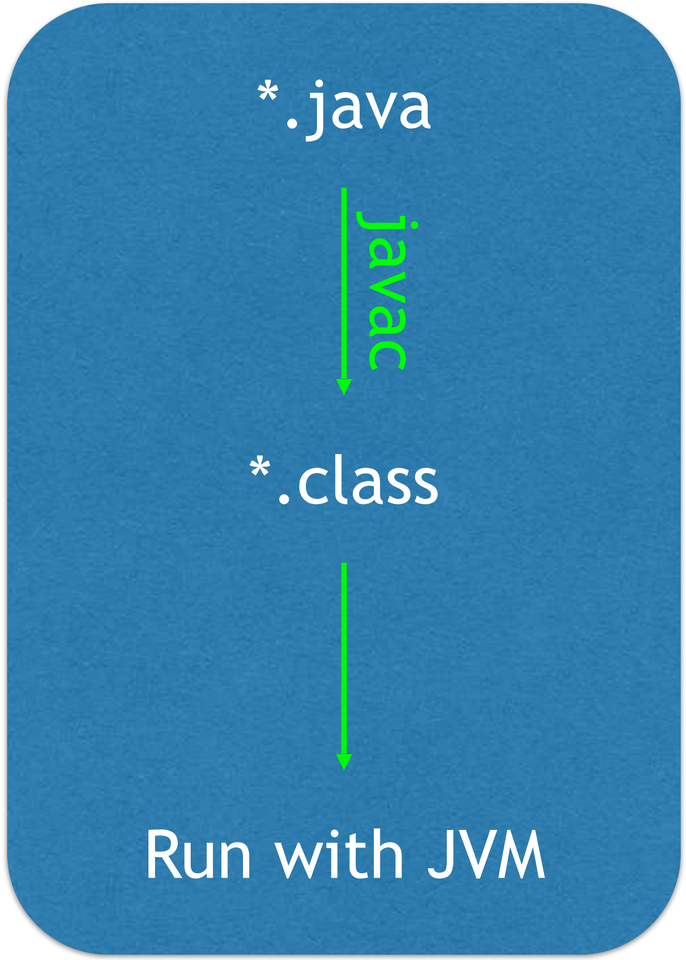
Compilation phases

- Preprocessing
- Compilation
- Linking

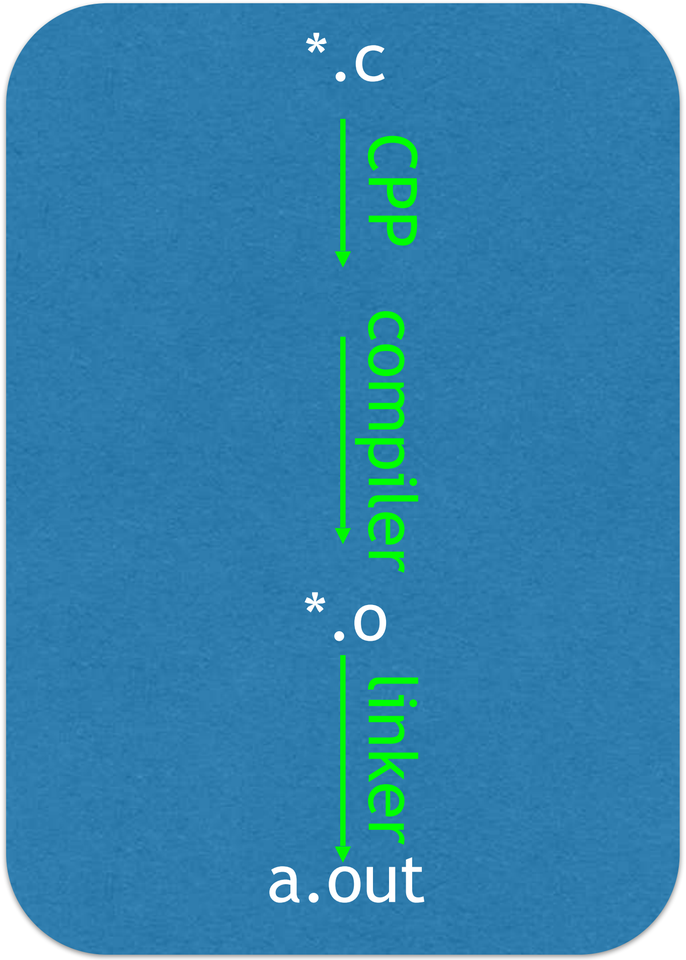
Outputs

- 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?

```c
// 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;
  /* ... */
}
```

A. 10

B. 12

C. 20

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	`_Bool`	`bool`
		`unsigned char`	
		`unsigned short`	
		`unsigned int`	`unsigned`
		`unsigned long`	
		`unsigned long long`	
	[un]signed	`char`	
	signed	`signed char`	
		`signed short`	`short`
		`signed int`	`signed` or `int`
		`signed long`	`long`
		`signed long long`	`long long`
floating point	real	`float`	
		`double`	
		`long double`	
	complex	`float _Complex`	`float complex`
		`double _Complex`	`double complex`
		`long double _Complex`	`long double complex`

Integer type sizes

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

```
1 = sizeof(char) ≤ sizeof(short) ≤ sizeof(int)
  ≤ sizeof(long) ≤ sizeof(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.