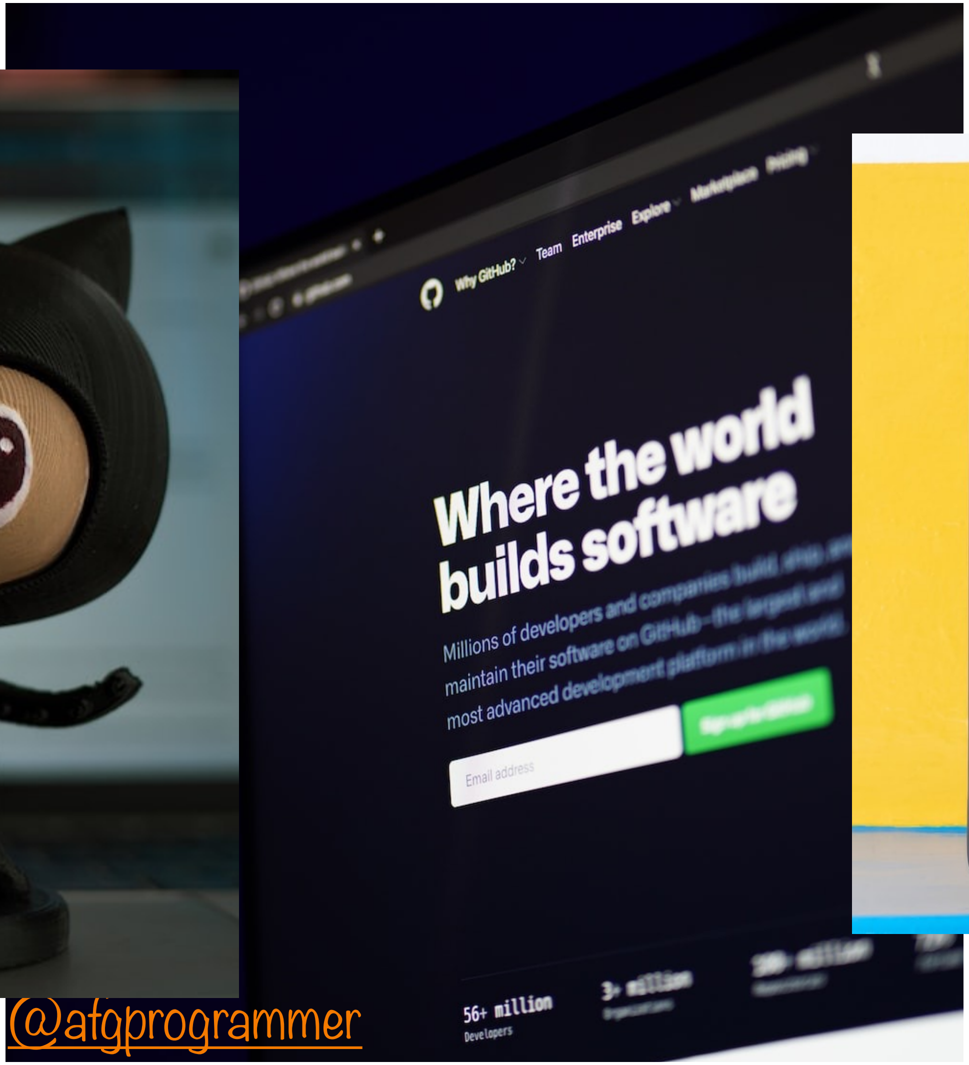
Git and GitHub ... then C

@synkevych



@afgprogrammer

@pawel_czerwinski

Agenda

Our computing environment

- Lecture 1 and Precepts 1 and 2: Linux and Bash
- Lecture 2: git and GitHub

A taste of C

- History of C
- Building and running C programs
- Characteristics of C
- Example program: `charcount`

Revision Control Systems

Problems often faced by programmers:

- Help! I've deleted my code! How do I get it back?
- How can I try out one way of writing this function, and go back if it doesn't work?
- Help! I've introduced a subtle bug that I can't find. How can I see what I've changed since the last working version?
- How do I work with source code on multiple computers?

- How do I work with others (e.g., a COS 217 partner) on the same program?
- What changes did my partner just make?
- If my partner and I make changes to different parts of a program, how do we merge those changes?

All of these problems are solved by revision control tools, e.g.:

`git`

Repository vs. Working Copy

WORKING COPY

- Represents single version of the code
- Plain files (e.g, .c)
- Make a coherent set of modifications, then *commit* this version of code to the repository
- Best practice: write a meaningful *commit message*

`git commit`

`git checkout`‡

REPOSITORY (or “repo”)

- Contains all checked-in versions of the code
- Specialized format, located in .git directory
- Can view commit history
- Can diff any versions
- Can *check out* any version, by default the most recent (known as HEAD)

‡ We'll rarely use checkout except to throw away local changes (see slide 6)

Relevant *xkcd*

https://xkcd.com/1296/

Local vs. Remote Repositories

LOCAL REPOSITORY

- Located in .git directory
- Only accessible from the current computer
- Commit early, commit often – you can only go back to versions you've committed
- Can *push* current state (i.e., complete checked-in history) to a remote repository

`git push`

`git clone`
`git pull`

REMOTE REPOSITORY

- Located in the cloud
  E.g., github.com
- Can *clone* working copies on multiple machines
- Any clone can *pull* the current state

COS 217 🧡 GitHub

We distribute assignment code through a github.com repo

- But you can’t push to our repo!

Need to create your own (private!) repo for each assignment

- Two methods in git primer handout
- One clone on armlab, to test and submit
- If developing on your own machine, another clone there:
  be sure to commit and push "up" to github,
  then pull "down" onto armlab

Agenda

Our computing environment

- Lecture 1 and Precepts 1 and 2: Linux and Bash
- Lecture 2: git

A taste of C

- History of C
- Building and running C programs
- Characteristics of C
- Example program: `charcount`

The C Programming Language

Who? Dennis Ritchie

When? ~1972

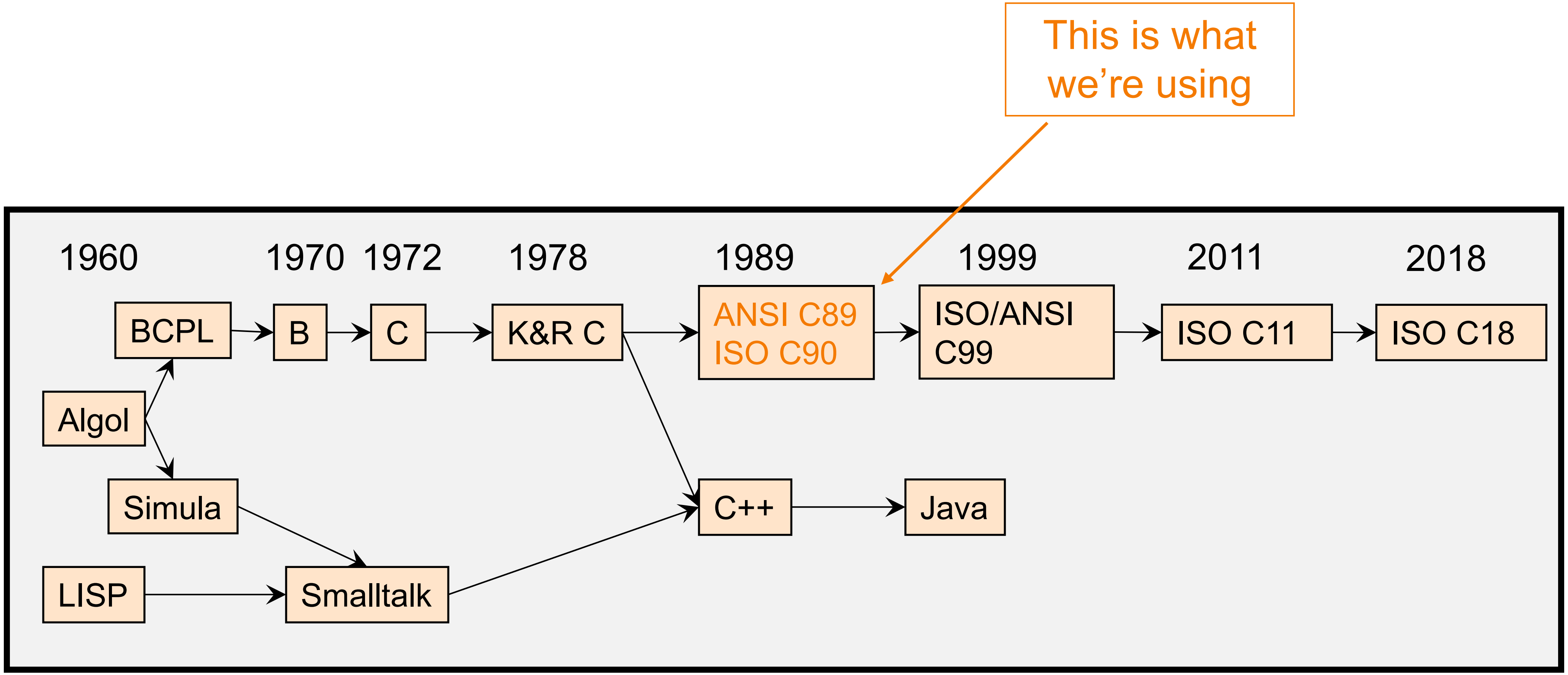
Where? Bell Labs

Why? Build the Unix OS

Read more history:

https://www.bell-labs.com/usr/dmr/www/chist.html

Java vs. C: History



C vs. Java: Design Goals

C Design Goals (1972)	Java Design Goals (1995)
Build the Unix OS	Language of the Internet
Low-level; close to HW and OS	High-level; insulated from hardware and OS
Good for system-level programming	Good for application-level programming
Support structured programming	Support object-oriented programming
Unsafe: don't get in the programmer's way	Safe: can't step "outside the sandbox"
	Look like C!

Agenda

Our computing environment

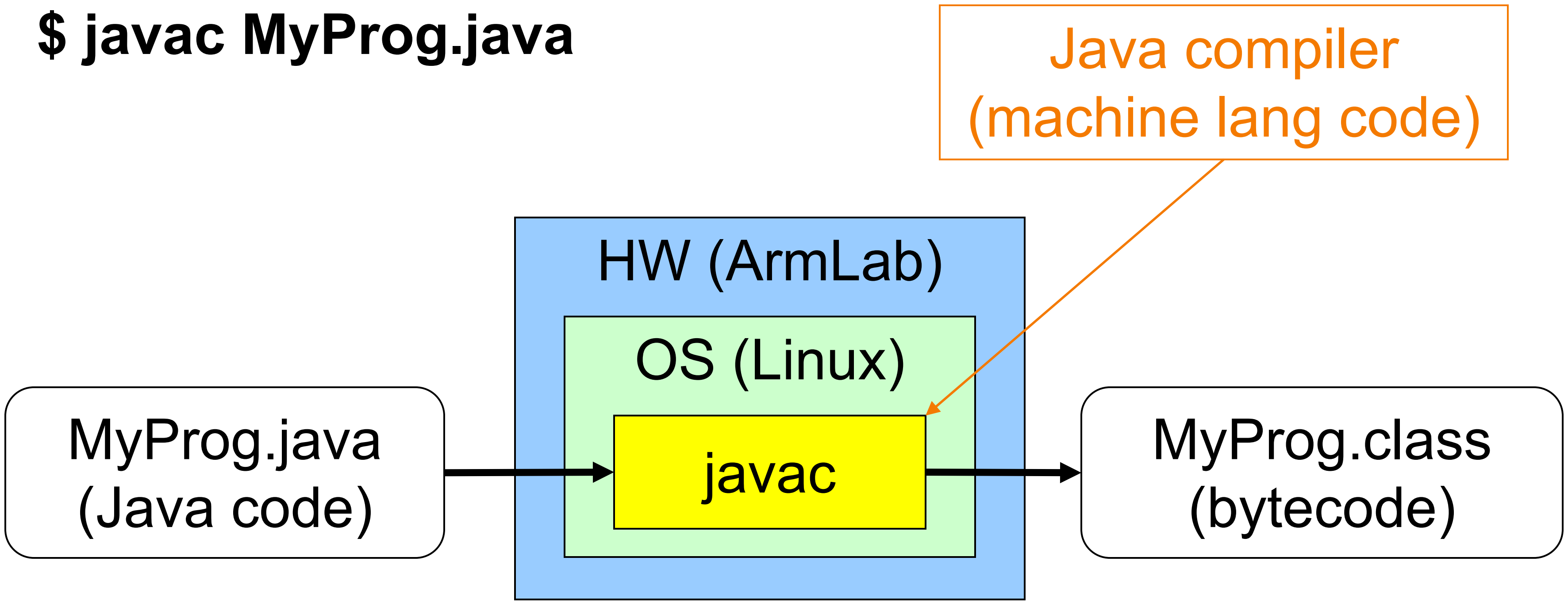
- Lecture 1 and Precepts 1 and 2: Linux and Bash
- Lecture 2: git

A taste of C

- History of C
- Building and running C programs
- Characteristics of C
- Example program: `charcount`

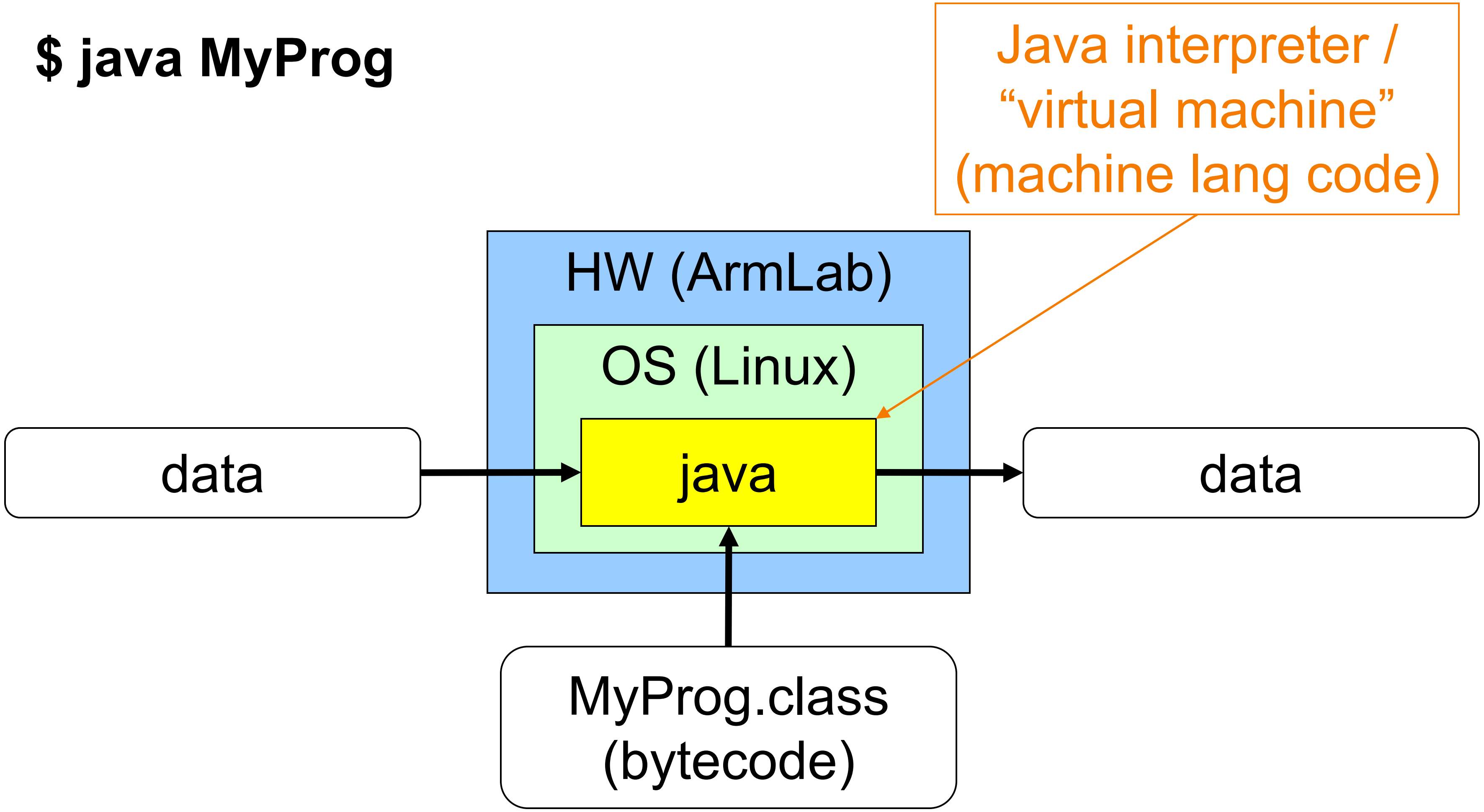
Building Java Programs

$ javac MyProg.java



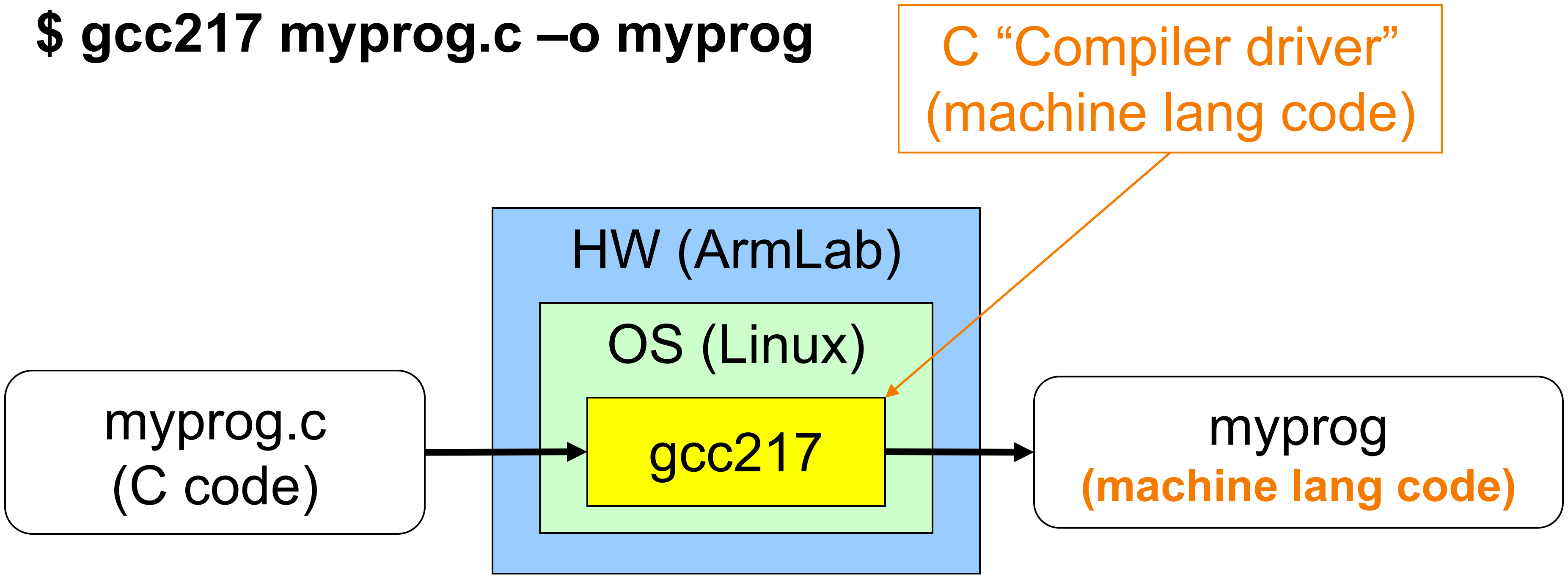
Running Java Programs

$ java MyProg



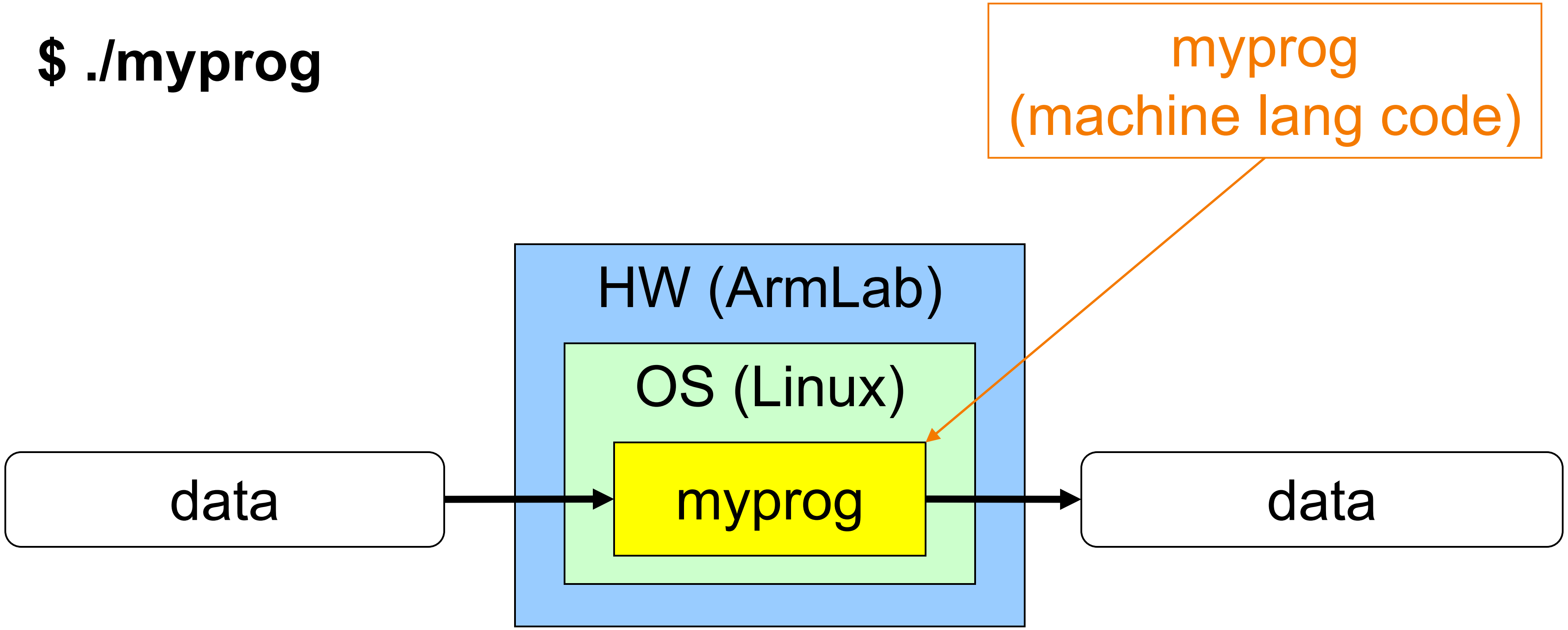
Building C Programs

$ gcc217 myprog.c –o myprog



Running C Programs

$ ./myprog



Agenda

Our computing environment

- Lecture 1 and Precepts 1 and 2: Linux and Bash
- Lecture 2: git

A taste of C

- History of C
- Building and running C programs
- Characteristics of C
- Example program: `charcount`

Java vs. C: Portability

Program	Code Type	Portable?
MyProg.java	Java source code	Yes
myprog.c	C source code	Mostly
MyProg.class	Bytecode	Yes
myprog	Machine lang code	No

Conclusion: Java programs are more portable

(For example, COS 217 has used many architectures over the years, and every time we've switched, all our programs have had to be recompiled!)

Java vs. C: Safety & Efficiency

Java

- null reference checking
- Automatic array-bounds checking
- Automatic memory management (garbage collection)
- Other safety features

C

- NULL pointer checking,
- Manual bounds checking
- Manual memory management

Conclusion 1: Java is often safer than C

Conclusion 2: Java is often slower than C

iClicker

Occasional questions in class, graded on participation not correctness.

- Using an app on your phone or the web client
- Setup is "iClicker Cloud", integrated with our course's Canvas.
- Register, select Princeton University, and find course "COS 217 – Fall 2023"

iClicker Question

Q: Can you answer this iClicker question today?

- A. Yes
- B. No, but I’ve been practicing my mental electrotelekinesis and the response is being registered anyway
- C. I’m not here, but someone is iClicking for me (don’t do this – it’s a violation of our course policies!)

C is for … car?

Q: Which corresponds to the C programming language?

A.

B.

C.

Java vs. C: Details

Next 7 slides show C language details by way of Java comparisons.

For now, use as a comparative language overview reference to start the simple "syntax mapping" stage of learning C, so that you're well prepared to dive into the less rote aspects in the coming weeks.

Java vs. C: Details

	Java	C

Overall Program Structure	Hello.java:
	
	public class Hello
	{ public static void main
	          (String[] args)
	    { System.out.println(
	            "hello, world");
	    }
	}
		hello.c:
		
		#include <stdio.h>
		
		int main(void)
		{ printf("hello, world\n");
		    return 0;
		}

Building	$ javac Hello.java	$ gcc217 hello.c -o hello
Running	$ java Hello hello, world $	$ ./hello hello, world $

Java vs. C: Details

	Java	C
Character type	`char     // 16-bit Unicode`	`char /* 8 bits */`

Integral types	`byte     // 8 bits`
	`short    // 16 bits`
	`int      // 32 bits`
	`long     // 64 bits`
		`(unsigned, signed) char`
		`(unsigned, signed) short`
		`(unsigned, signed) int`
		`(unsigned, signed) long`

Floating point types	`float    // 32 bits`
	`double   // 64 bits`
		`float`
		`double`
		`long double`

Logical type	`boolean`	`/* no equivalent */` `/* use 0 and non-0 */`
Generic pointer type	`Object`	`void*`

Constants	`final int MAX = 1000;`	`#define MAX 1000`
		`const int MAX = 1000;`
		`enum {MAX = 1000};`

Java vs. C: Details

	Java	C

Arrays	`int [] a = new int [10];`
	`float [][] b =`
	`   new float [5][20];`
		`int a[10];`
		`float b[5][20];`

Array bound checking	`// run-time check`	`/* no run-time check */`
Pointer type	`// Object reference is an` `// implicit pointer`	`int *p;`

Record type	`class Mine`
	`{   int x;`
	`    float y;`
	`}`
		`struct Mine`
		`{   int x;`
		`    float y;`
		`};`

Java vs. C: Details

<table>
<tr><th></th><th>Java</th><th>C</th></tr>
<tr><td>Strings</td><td><code>String s1 = "Hello";
String s2 = new
   String("hello");</code></td><td><code>char *s1 = "Hello";
char s2[6];
strcpy(s2, "hello");</code></td></tr>
<tr><td>String concatenation</td><td><code>s1 + s2
s1 += s2</code></td><td><code>#include <string.h>
strcat(s1, s2);</code></td></tr>
<tr><td>Logical ops *</td><td><code>&&, ||, !</code></td><td><code>&&, ||, !</code></td></tr>
<tr><td>Relational ops *</td><td><code>==, !=, <, >, <=, >=</code></td><td><code>==, !=, <, >, <=, >=</code></td></tr>
<tr><td>Arithmetic ops *</td><td><code>+, -, *, /, %, unary -</code></td><td><code>+, -, *, /, %, unary -</code></td></tr>
<tr><td>Bitwise ops</td><td><code><<, >>, >>>, &, ^, |, ~</code></td><td><code><<, >>, &, ^, |, ~</code></td></tr>
<tr><td>Assignment ops</td><td><code>=, +=, -=, *=, /=, %=,
<<=, >>=, >>>=, &=, ^=, |=</code></td><td><code>=, +=, -=, *=, /=, %=,
<<=, >>=, &=, ^=, |=</code></td></tr>
</table>

* Essentially the same in the two languages

Java vs. C: Details

	Java	C

if stmt *	`if (i < 0)`
	`    statement1;`
	`else`
	`    statement2;`
		`if (i < 0)`
		`    statement1;`
		`else`
		`    statement2;`

switch stmt *	`switch (i)`
	`{   case 1:`
	`        ...`
	`        break;`
	`    case 2:`
	`        ...`
	`        break;`
	`    default:`
	`        ...`
	`}`
		`switch (i)`
		`{   case 1:`
		`        ...`
		`        break;`
		`    case 2:`
		`        ...`
		`        break;`
		`    default:`
		`        ...`
		`}`

goto stmt	`// no equivalent`	`goto someLabel;`

* Essentially the same in the two languages

Java vs. C: Details

	Java	C

for stmt	`for (int i=0; i<10; i++)`
	`    statement;`
		`int i;`
		`for (i=0; i<10; i++)`
		`    statement;`

while stmt *	`while (i < 0)` `    statement;`	`while (i < 0)` `    statement;`

do-while stmt *	`do`
	`    statement;`
	`while (i < 0)`
		`do`
		`    statement;`
		`while (i < 0);`

continue stmt *	`continue;`	`continue;`
labeled continue stmt	`continue someLabel;`	`/* no equivalent */`
break stmt *	`break;`	`break;`
labeled break stmt	`break someLabel;`	`/* no equivalent */`

* Essentially the same in the two languages

Java vs. C: Details

	Java	C
return stmt *	`return 5;` `return;`	`return 5;` `return;`

Compound stmt (alias block) *	`{`
	    `statement1``;`
	    `statement2``;`
	`}`
		`{`
		    `statement1``;`
		    `statement2``;`
		`}`

Exceptions	`throw, try-catch-finally`	`/* no equivalent */`
Comments	`/* comment */` `// another kind`	`/* comment */`
Method / function call	`f(x, y, z);` *`someObject`*`.f(x, y, z);` *`SomeClass`*`.f(x, y, z);`	`f(x, y, z);`

* Essentially the same in the two languages

Agenda

Our computing environment

- Lecture 1 and Precepts 1 and 2: Linux and Bash
- Lecture 2: git

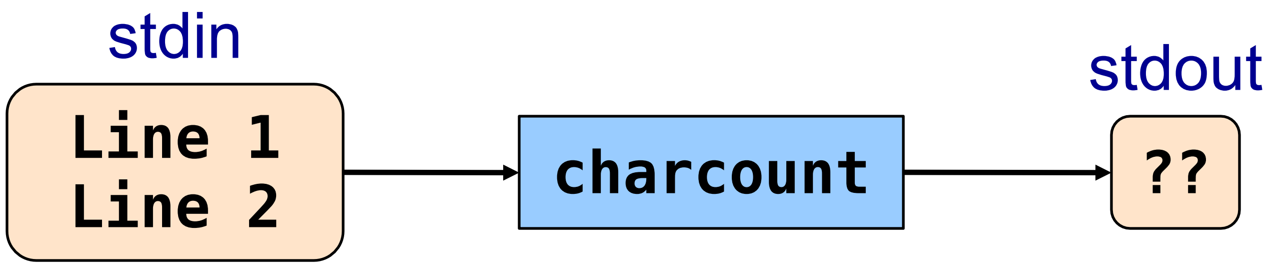
A taste of C

- History of C
- Building and running C programs
- Characteristics of C
- Example program: `charcount`

The `charcount` Program

Functionality:

- Read all characters from standard input stream
- Write to standard output stream the number of characters read



The `charcount` Program

The program: `charcount.c`

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void) {
   int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF) {
      charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

charcount Building and Running

```
$ gcc217 charcount.c
$ ls
.     ..    a.out
$ gcc217 charcount.c –o charcount
$ ls
.     ..    a.out
      charcount
$
```

charcount Building and Running

```
$ gcc217 charcount.c –o charcount
$ ./charcount
Line 1
Line 2
^D
```

What is this?
What is the effect?
What is printed?

charcount Building and Running

```
$ gcc217 charcount.c –o charcount
$ ./charcount
Line 1
Line 2
^D
14
$
```

Includes visible characters plus two newlines

`charcount` Building and Running

```
$ cat somefile
Line 1
Line 2
$ ./charcount < somefile
14
$
```

What is this?
What is the effect?

`charcount` Building and Running

```
$ ./charcount > someotherfile
Line 1
Line 2
^D
$ cat someotherfile
14
$
```

What is this?
What is the effect?

Running charcount

Run-time trace, referencing the original C code...

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

Execution begins at **`main()`** function

- No classes in the C language.

Running charcount

Run-time trace, referencing the original C code…

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

We allocate space for c and charCount in the stack section of memory

Why **int** instead of **char**?

Running charcount

Run-time trace, referencing the original C code…

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

getchar() tries to read char from stdin

- Success ⇒ returns that char value (within an int)
- Failure ⇒ returns **EOF**

EOF is a special value, distinct from all possible chars

Running charcount

Run-time trace, referencing the original C code…

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

Assuming $c \neq$ EOF, we increment charCount

Running charcount

Run-time trace, referencing the original C code…

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

We call getchar() again and recheck loop condition

Running charcount

Run-time trace, referencing the original C code…

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

- Eventually getchar() returns EOF
- Loop condition fails
- We call printf() to write final charCount

Running charcount

Run-time trace, referencing the original C code…

charcount.c

```
#include <stdio.h>
/* Write to stdout the number of
   chars in stdin. Return 0. */
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  charCount++;
      c = getchar();
   }
   printf("%d\n", charCount);
   return 0;
}
```

- return statement returns to calling function
- return from main() returns to _start, terminates program

Normal execution ⇒ 0 or **EXIT_SUCCESS**
Abnormal execution ⇒ **EXIT_FAILURE**

← #include <stdlib.h> to use these constants

Coming up next ...

More character processing, structured exactly how we'll want you to design your Assignment 1 solution!

Read the A1 specs soon: you'll be ready to start after Lecture 3!