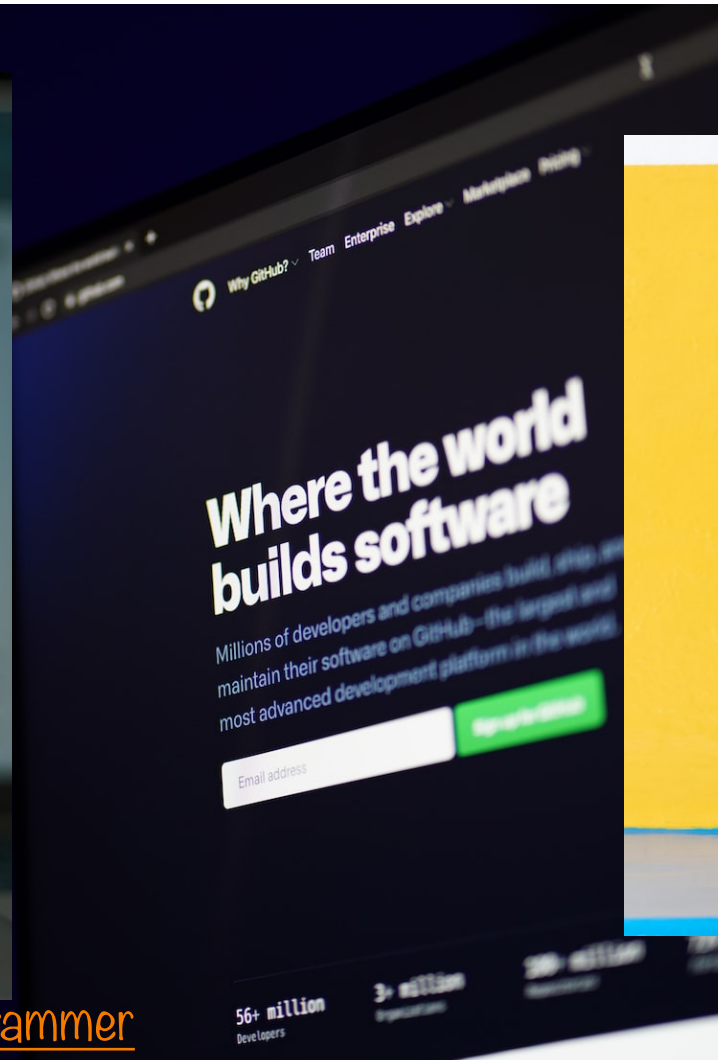


Git and GitHub ... then C



[@pawel_czerwinski](#)

[@atgprogrammer](#)

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.:

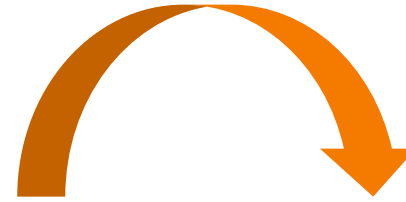
git



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

REPOSITORY

- 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)

git checkout[#]



[#] We'll rarely use checkout (see slide 6)



Relevant *xkcd*

| | COMMENT | DATE |
|---|------------------------------------|--------------|
| ○ | CREATED MAIN LOOP & TIMING CONTROL | 14 HOURS AGO |
| ○ | ENABLED CONFIG FILE PARSING | 9 HOURS AGO |
| ○ | MISC BUGFIXES | 5 HOURS AGO |
| ○ | CODE ADDITIONS/EDITS | 4 HOURS AGO |
| ○ | MORE CODE | 4 HOURS AGO |
| ○ | HERE HAVE CODE | 4 HOURS AGO |
| ○ | AAAAAAAAA | 3 HOURS AGO |
| ○ | ADKFJSLKDFJSDKLFJ | 3 HOURS AGO |
| ○ | MY HANDS ARE TYPING WORDS | 2 HOURS AGO |
| ○ | HAAAAAAAAAANDS | 2 HOURS AGO |

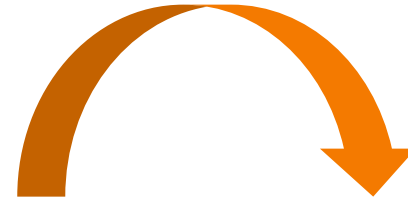
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.



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`

REMOTE REPOSITORY

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

`git clone`
`git pull`



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



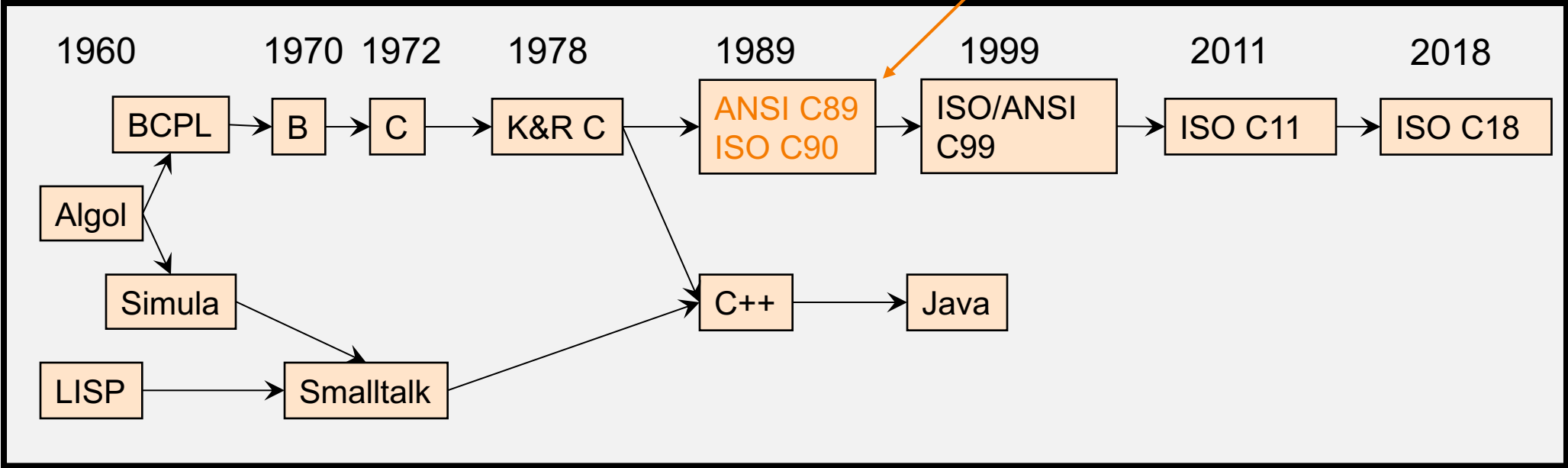
Read more history:

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

Java vs. C: History



This is what we're using





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

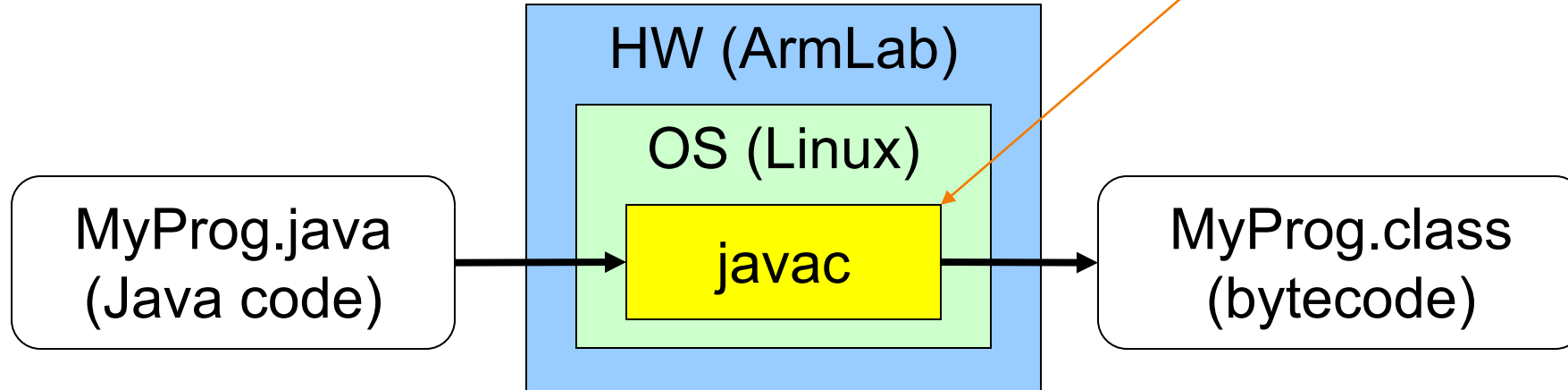
- History of C
- Building and running C programs
- Characteristics of C
- Example program: charcount



Building Java Programs

`$ javac MyProg.java`

Java compiler
(machine lang code)

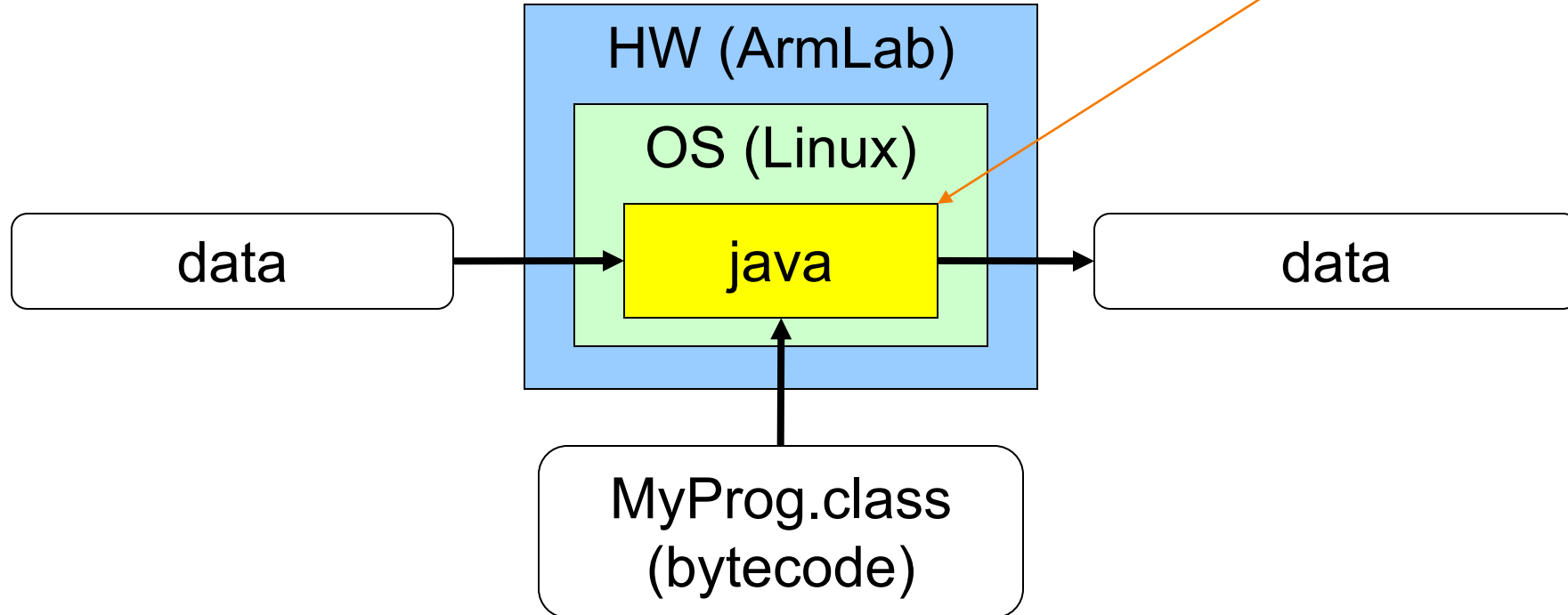




Running Java Programs

`$ java MyProg`

Java interpreter /
“virtual machine”
(machine lang code)

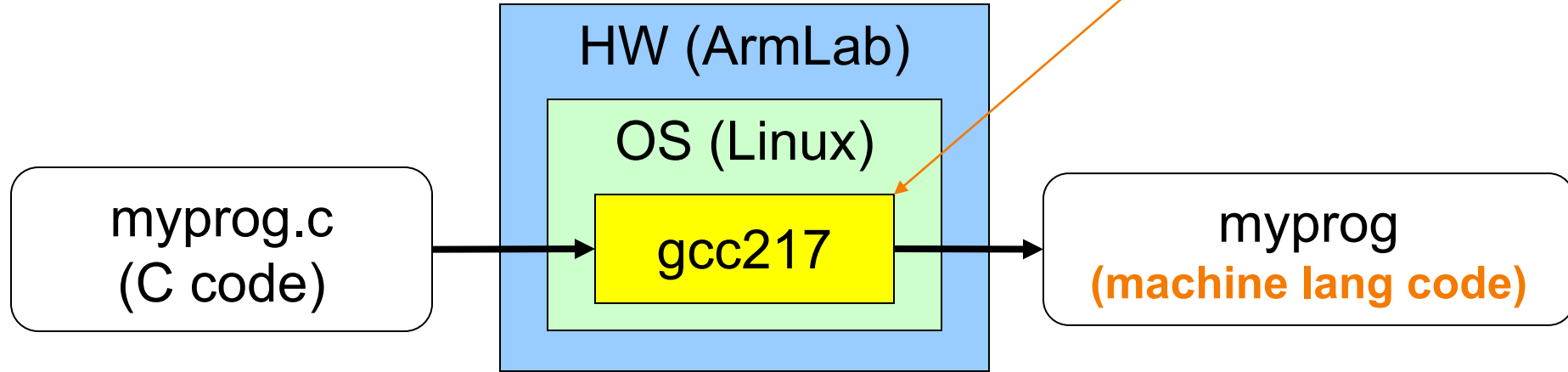




Building C Programs

```
$ gcc217 myprog.c -o myprog
```

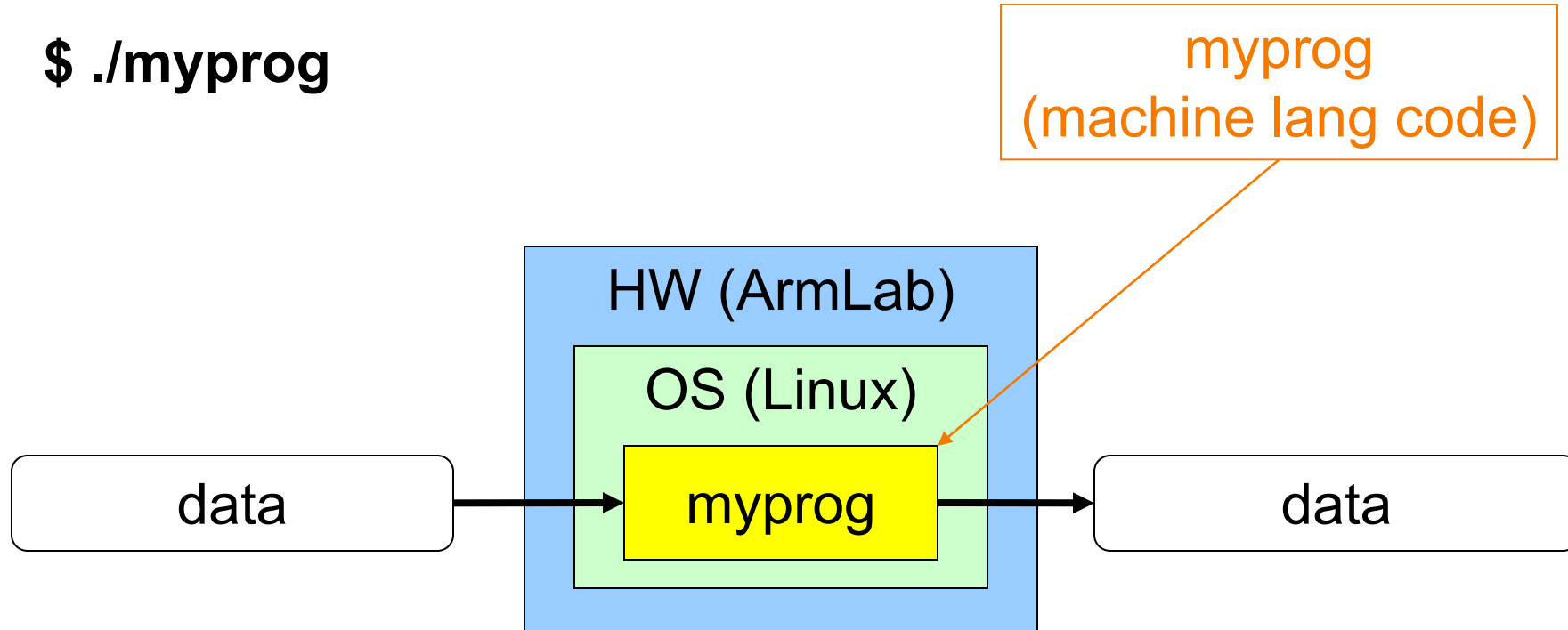
C “Compiler driver”
(machine lang code)



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

▶ 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 | <pre>Hello.java: public class Hello { public static void main (String[] args) { System.out.println("hello, world"); } }</pre> | <pre>hello.c: #include <stdio.h> int main(void) { printf("hello, world\n"); return 0; }</pre> |
| Building | <pre>\$ javac Hello.java</pre> | <pre>\$ gcc217 hello.c -o hello</pre> |
| Running | <pre>\$ java Hello hello, world \$</pre> | <pre>\$./hello hello, world \$</pre> |

Java vs. C: Details



| | Java | C |
|----------------------|---|--|
| Character type | <code>char // 16-bit Unicode</code> | <code>char /* 8 bits */</code> |
| Integral types | <code>byte // 8 bits</code> <code>short // 16 bits</code> <code>int // 32 bits</code> <code>long // 64 bits</code> | <code>(unsigned, signed) char</code> <code>(unsigned, signed) short</code> <code>(unsigned, signed) int</code> <code>(unsigned, signed) long</code> |
| Floating point types | <code>float // 32 bits</code> <code>double // 64 bits</code> | <code>float</code> <code>double</code> <code>long double</code> |
| Logical type | <code>boolean</code> | <code>/* no equivalent */</code> <code>/* use 0 and non-0 */</code> |
| Generic pointer type | <code>Object</code> | <code>void*</code> |
| Constants | <code>final int MAX = 1000;</code> | <code>#define MAX 1000</code> <code>const int MAX = 1000;</code> <code>enum {MAX = 1000};</code> |

Java vs. C: Details



| | Java | C |
|----------------------|---|---|
| Arrays | <pre>int [] a = new int [10]; float [][] b = new float [5][20];</pre> | <pre>int a[10]; float b[5][20];</pre> |
| Array bound checking | <pre>// run-time check</pre> | <pre>/* no run-time check */</pre> |
| Pointer type | <pre>// Object reference is an // implicit pointer</pre> | <pre>int *p;</pre> |
| Record type | <pre>class Mine { int x; float y; }</pre> | <pre>struct Mine { int x; float y; };</pre> |



Java vs. C: Details

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

* Essentially the same in the two languages

Java vs. C: Details



| | Java | C |
|---------------|--|--|
| if stmt * | <pre>if (i < 0) statement1; else statement2;</pre> | <pre>if (i < 0) statement1; else statement2;</pre> |
| switch stmt * | <pre>switch (i) { case 1: ... break; case 2: ... break; default: ... }</pre> | <pre>switch (i) { case 1: ... break; case 2: ... break; default: ... }</pre> |
| goto stmt | // no equivalent | <pre>goto someLabel;</pre> |

* Essentially the same in the two languages

Java vs. C: Details



| | Java | C |
|-----------------------|---|--|
| for stmt | <pre>for (int i=0; i<10; i++) statement;</pre> | <pre>int i; for (i=0; i<10; i++) statement;</pre> |
| while stmt * | <pre>while (i < 0) statement;</pre> | <pre>while (i < 0) statement;</pre> |
| do-while stmt * | <pre>do statement; while (i < 0)</pre> | <pre>do statement; while (i < 0);</pre> |
| continue stmt * | <pre>continue;</pre> | <pre>continue;</pre> |
| labeled continue stmt | <pre>continue someLabel;</pre> | <pre>/* no equivalent */</pre> |
| break stmt * | <pre>break;</pre> | <pre>break;</pre> |
| labeled break stmt | <pre>break someLabel;</pre> | <pre>/* no equivalent */</pre> |

* Essentially the same in the two languages

Java vs. C: Details



| | Java | C |
|----------------------------------|---|--|
| return stmt * | <code>return 5;</code> <code>return;</code> | <code>return 5;</code> <code>return;</code> |
| Compound stmt (alias block) * | <code>{</code> <i>statement1;</i> <i>statement2;</i> <code>}</code> | <code>{</code> <i>statement1;</i> <i>statement2;</i> <code>}</code> |
| Exceptions | <code>throw, try-catch-finally</code> | <code>/* no equivalent */</code> |
| Comments | <code>/* comment */</code> <code>// another kind</code> | <code>/* comment */</code> |
| Method / function call | <code>f(x, y, z);</code> <code>someObject.f(x, y, z);</code> <code>SomeClass.f(x, y, z);</code> | <code>f(x, y, z);</code> |

* 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

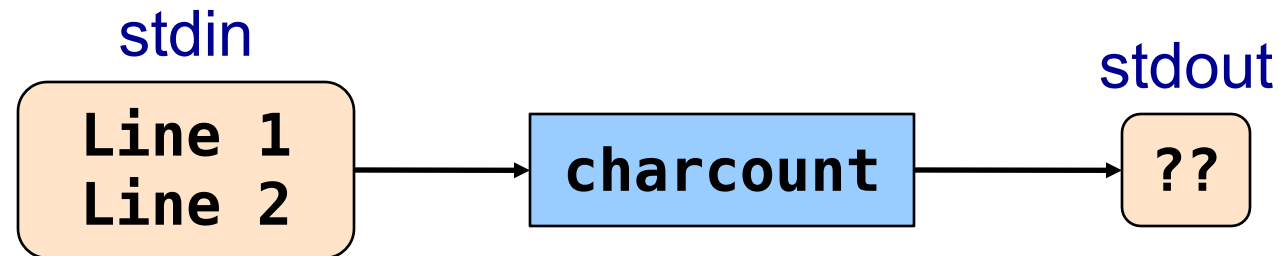
- History of C
- Building and running C programs
- Characteristics of C
- Example program: charcount



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 \Rightarrow returns that char value (within an int)
- Failure \Rightarrow 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 $c \neq \text{EOF}$,
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 \Rightarrow 0 or **EXIT_SUCCESS**
Abnormal execution \Rightarrow **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!