COS 217: Introduction to Programming Systems

Simple C Programs

PRINCETON UNIVERSITY

Agenda

Precept: Our computing environment – Linux, bash, git

- Some slides on git here: please read on your own

A taste of C (and comparison with Java)

- History of C
- Building and running C programs
- Some characteristics of the C language

Writing a simple character processing program

Next: Writing a more complex character processing program

Revision Control Systems

Problems often faced by programmers:

- I've deleted my code! How do I get it back?
- I tried a way of writing functions, but I want to go back to the old way
  - Save versions. But how many? Gets complicated
- What have I changed since the last working version?
- How do I work with source code on multiple computers (laptop and armlab)?
- How do I work with others (e.g., a COS 217 partner) on the same program?
- What changes did my partner just make?
- My partner and I make changes to different parts of a program; how do we merge those changes?

All solved by revision control tools, e.g. `git`

Repository vs. Working Copy

WORKING COPY

- Represents one 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)

Information Content of Git Comments

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

https://xkcd.com/1296/

Local vs. Remote Repositories

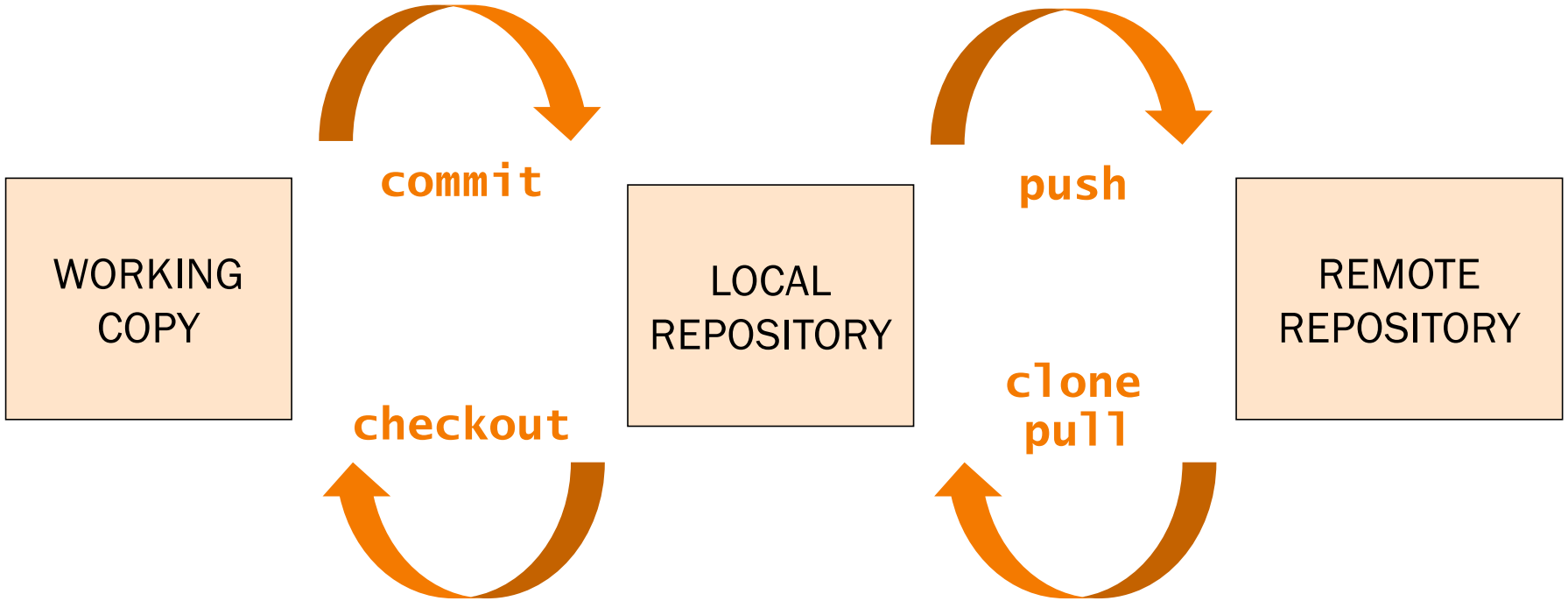
LOCAL REPOSITORY

- Located in .git directory
- Only accessible from the computer where it lives
- Commit early, commit often: you can only go back to versions you've committed
- Can *push* current state (i.e., complete committed history) of a local repo to remote repo

REMOTE REPOSITORY

- Located in the cloud
  E.g., github.com
- Can *clone* remote repo into local repo + working copy on multiple machines
- Any clone can *pull* the current state from remote repo

Another Level of Indirection



COS 217 🧡 GitHub

We distribute assignment code through a github.com repo

But we're not very nice

- You can't push code to our repo
- You create your own remote (private) repo for each assignment
  - Two methods for this in git primer handout
- Then create clones of that repo for yourself
  - Create one clone on armlab, to test and submit
  - If developing on your own machine, create another clone there
  - Commit from working copy to local dev clone, push from it "up" to github, then pull "down" onto armlab, before testing and submitting

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

Dennis Ritchie on C

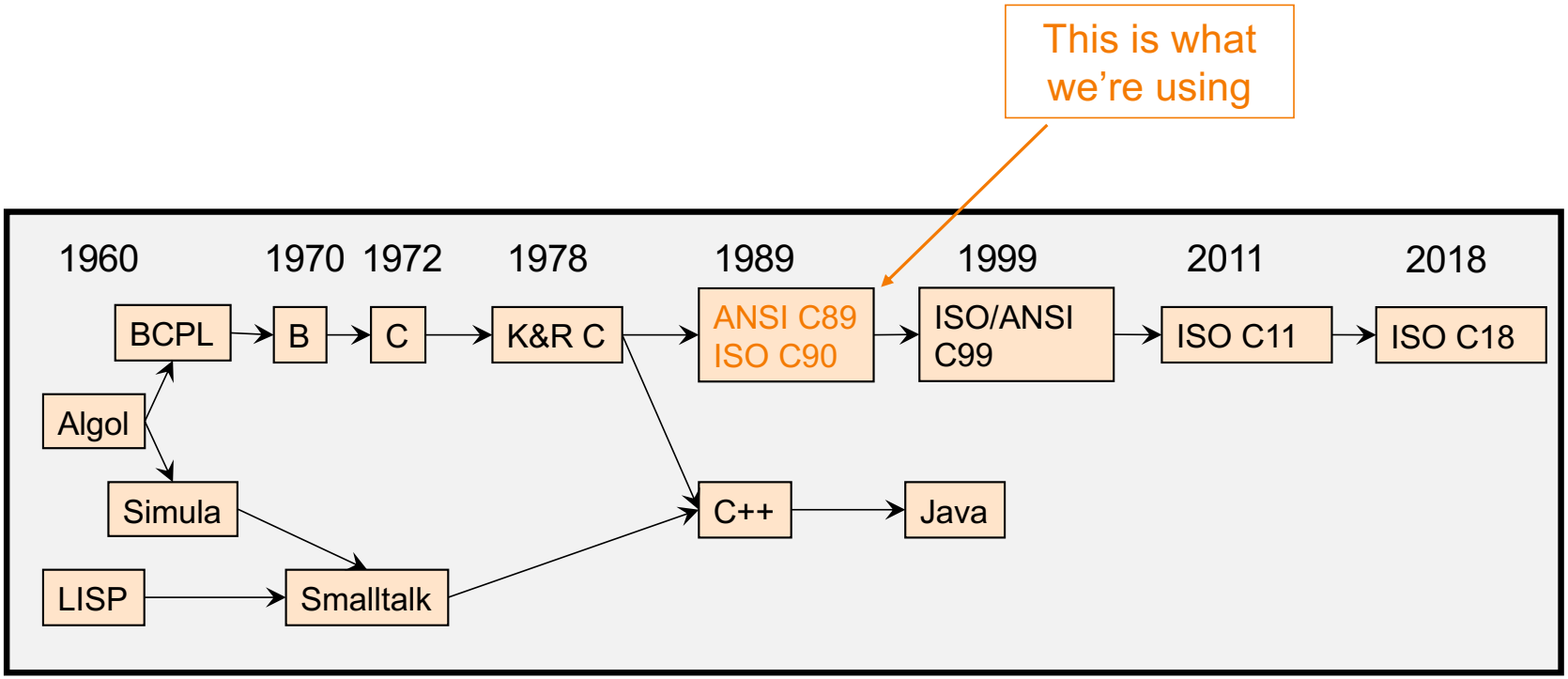
"C … **never attempts to tie a programmer down**."

"C has always appealed to systems programmers who like the **terse, concise** manner in which powerful expressions can be coded."

"C allowed programmers to (while sacrificing portability) have **direct access to many machine-level features** that would otherwise require the use of assembly language."

"C is **quirky, flawed, and an enormous success**. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

History of C and Java



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 restrict the programmer much	*Safe*: can't step "outside the sandbox"
	Look like C

Agenda

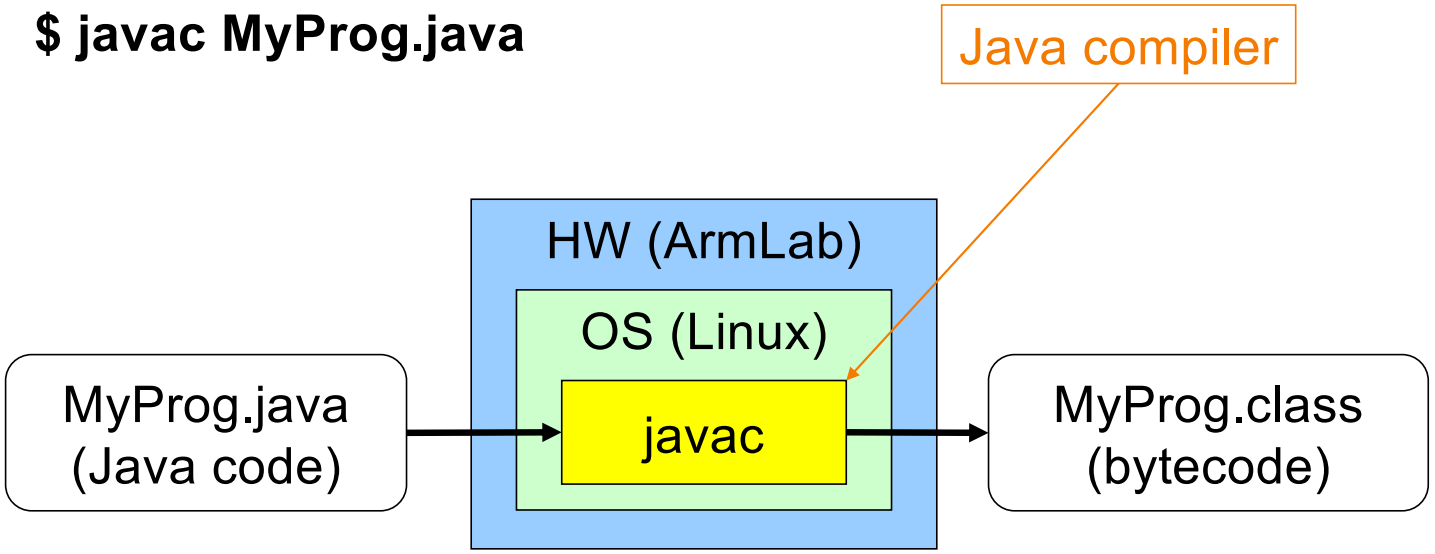
A taste of C (compared with Java)

- History of C
- Building and running programs
- Some characteristics of C

Writing a simple character processing program

Building Java Programs

$ javac MyProg.java

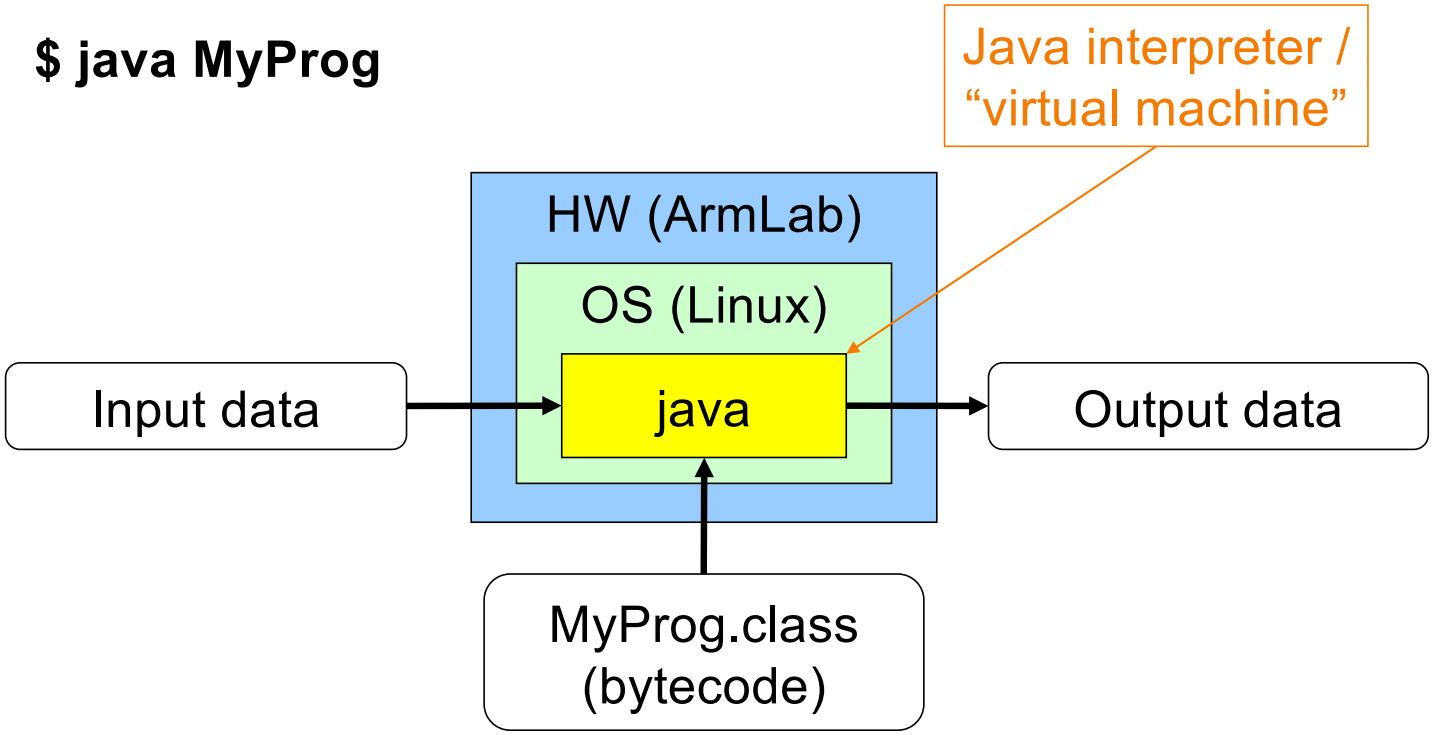


Myprog.class is bytecode, which is machine independent and runs on an interpreter, which is a software layer that runs on the hardware

- The bytecode is an input to the interpreter software

Running Java Programs

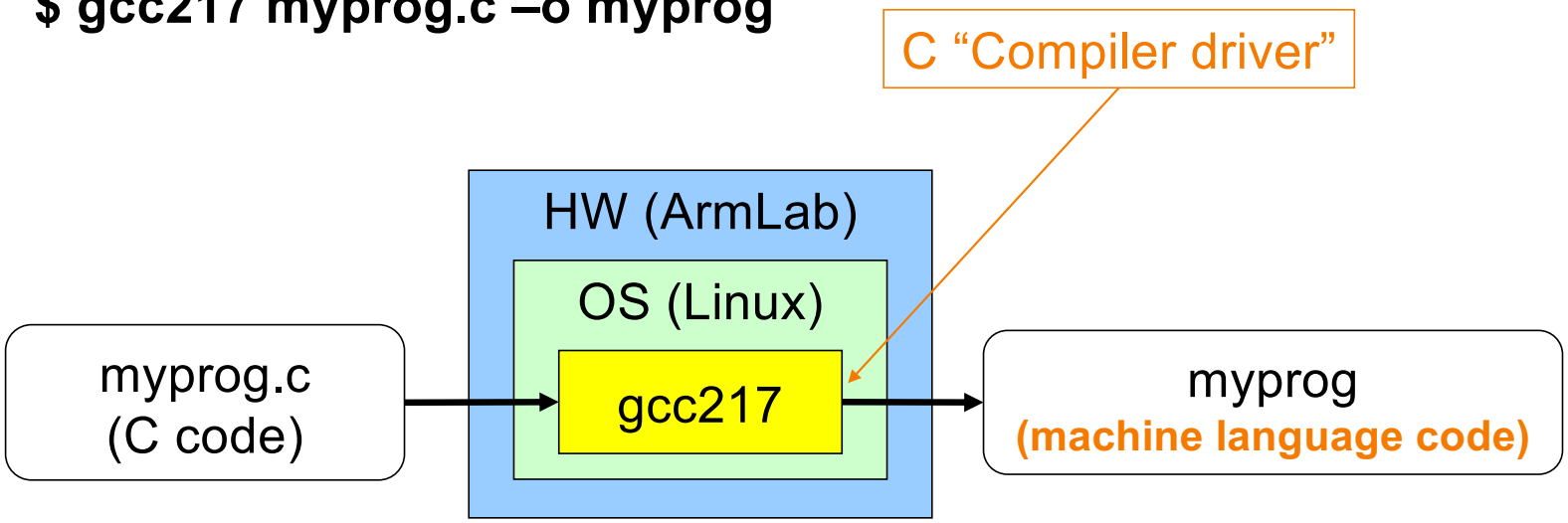
$ java MyProg



The java interpreter is a good example of abstraction (provides an abstracted view of the hardware) and separation of interface from implementation

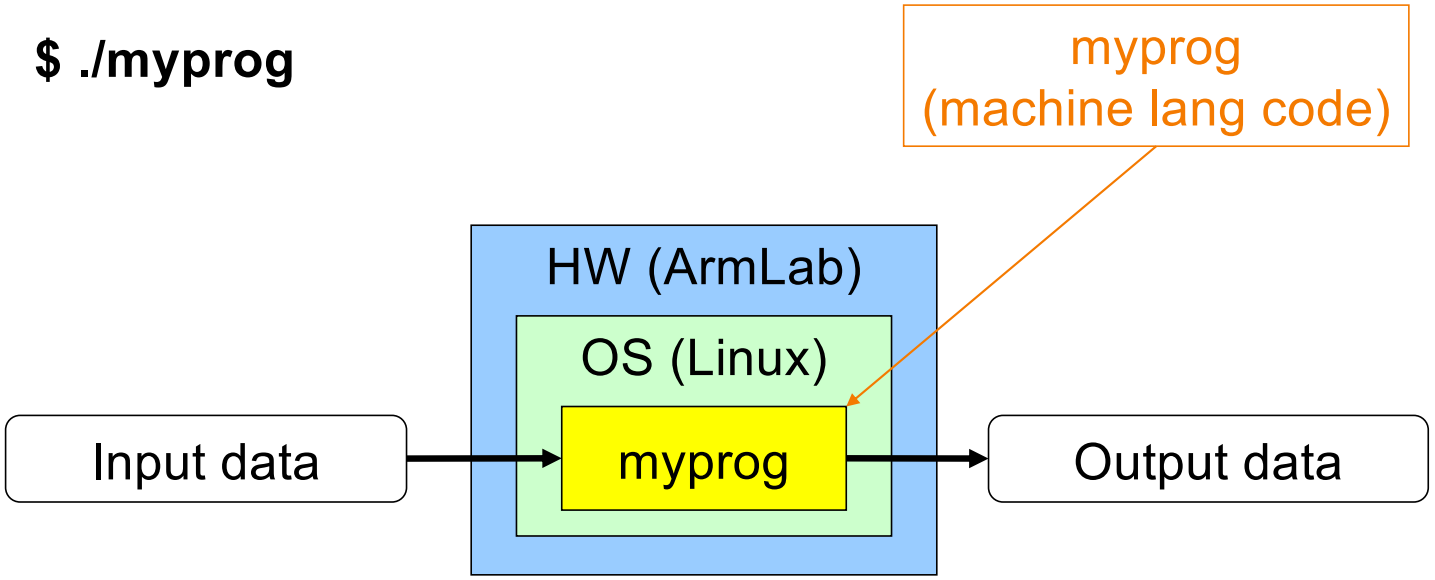
Building C Programs

$ gcc217 myprog.c –o myprog



Myprog is machine language, which is machine dependent and runs on the raw hardware (unlike Java’s bytecode)

Running C Programs



Agenda

A taste of C (compared with Java)

- History of C
- Building and running programs
- Some characteristics of the C language

Writing a simple character processing program

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

* C leaves a lot to implementation: sizes for int, char; byte order, overflow handling. COS 217 has switched *many* architectures over the years, and every time all programs had to be recompiled

** Java interpreter provides a consistent interface across OSes and hardware, though its own implementation is different across them

Is gcc217 portable? Is javac? Is java?

Java vs. C: Safety & Efficiency

Java does a lot more checking

- `null` reference checking (crash)
- Automatic array-bounds checking (crash)
- Automatic memory management (garbage collection – McCarthy 1950s for Lisp)
- Other safety features

C

- NULL reference checking (crash)
- Manual bounds checking (keep going …)
- Manual memory management

Conclusion 1: Java is often safer than C

Conclusion 2: Java is often slower than C (can slow be unsafe?)

Java vs. C: Details

Next 7 slides show C details through comparisons with the Java you know.

We're going to skip them in lecture. You can read them later

- This is largely syntax mapping. But it prepares us for the more significant differences later

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

	Java	C

Strings	`String s1 = "Hello";`
	`String s2 =`
	`  new String("hello");`
		`char *s1 = "Hello";`
		`char s2[6];`
		`strcpy(s2, "hello");`

String concatenation	`s1 + s2` `s1 += s2`	`#include <string.h>` `strcat(s1, s2);`
Logical ops *	`&&, ||, !`	`&&, ||, !`
Relational ops *	`==, !=, <, >, <=, >=`	`==, !=, <, >, <=, >=`
Arithmetic ops *	`+, -, *, /, %, unary -`	`+, -, *, /, %, unary -`
Bitwise ops	`<<, >>, >>>, &, ^, |, ~`	`<<, >>, &, ^, |, ~`
Assignment ops	`=, +=, -=, *=, /=, %=,` `<<=, >>=, >>>=, &=, ^=, |=`	`=, +=, -=, *=, /=, %=,` `<<=, >>=, &=, ^=, |=`

* 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

A taste of C (compared with Java)

- History of C
- Building and running programs
- Some characteristics of C

Writing a simple character processing program

Writing a Simple Character Processor Program

From product specification to design to code

Structured 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

Plan for Next Couple of Lectures

A simple character processing program: Upper

- Learn to structure a simple C program
- Trace its execution beginning to end
- Learn to use C libraries (stdio, ctype)

Next: A more complex character processing program: Upper1. Why?

- Assignment 1
- Design step more involved: designing the simple DFA model
- Coding Step: develop a C program to implement the DFA

Building an executable C program

Design decisions in upper, upper1

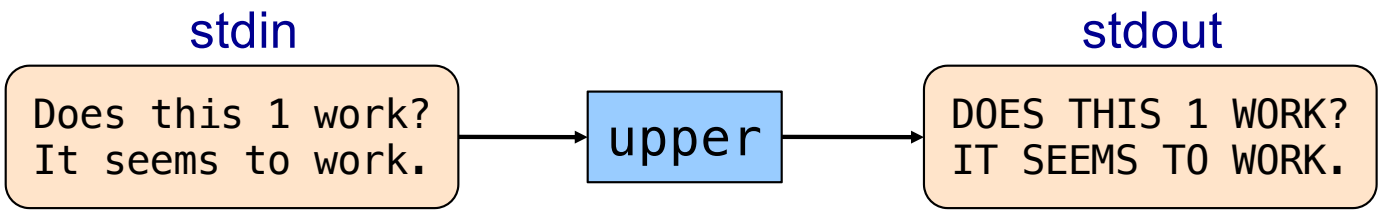
`upper` Product Specification

Read all chars from stdin

Convert every lower-case alphabetic character to upper case

- Leave other kinds of characters alone

Print results to stdout



`upper` Program Design

Read a char from stdin

If it is lowercase alphabetical, turn it into uppercase and write it to stdout

If it is not, keep it as is and write it to stdout

Can we optimize this, from a code size perspective?

- Read a char from stdin
- If it is lowercase alphabetical, turn it into uppercase
- If it is not, keep it as is
- Write the resulting character to stdout

upper C Program

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

Now let's walk through this element by element and see how the program is executed

Tracing through **upper**: Starting up

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

Block /* */ comments are the **only** legal ones in C90: no //

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

- No classes in C

Tracing through `upper`: Defining Variables

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

Variables must be declared at the top of a block

We allocate space for c in the stack section of memory

Why **`int`** not **`char`**?

Tracing through `upper`: Reading and Processing Input

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

getchar() tries to read a char from stdin

- Success ⇒ returns that char value (as int)
- Failure ⇒ returns a special value: **EOF**
- EOF is just a number, defined in stdio.h (-1 on our systems)

Tracing through upper: The Loop

We read a character at a time in a `while` loop until we hit EOF, for every character we read (not EOF), we process and print it

Simple version of loop:

```
c = getchar();
while (c != EOF)
{   if (islower(c))
      c = toupper(c);
    putchar(c);
    c = getchar(c);
}
```

More typical version of loop:

```
while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
```

Tracing through `upper`: Using Library Functions

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

ctype.h Functions

```
$ man islower
NAME
       isalnum, isalpha, isascii, isblank, iscntrl, isdigit, isgraph,
       islower, isprint, ispunct, isspace, isupper, isxdigit –
       character classification routines

SYNOPSIS
       #include <ctype.h>
       int isalnum(int c);
       int isalpha(int c);
       int isascii(int c);
       int isblank(int c);
       int iscntrl(int c);
       int isdigit(int c);
       int isgraph(int c);
       int islower(int c);
       int isprint(int c);
       int ispunct(int c);
       int isspace(int c);
       int isupper(int c);
       int isxdigit(int c);
```

These functions check whether c, which must have the value of an unsigned char or EOF, falls into a certain character class.

...

islower() checks for a lowercase character.

Tracing through upper: The End Game

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- Eventually getchar() returns EOF
- Loop condition fails
- We exit the loop, having output what we needed to output

Tracing through upper: The Exit

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- return statement returns control 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