A Taste of C

Agenda

Getting started with C

- **History of C**
- Building and running C programs
- Characteristics of C

Three Simple C Programs

- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Java versus C Details

- For initial cram and/or later reference

The C Programming Language

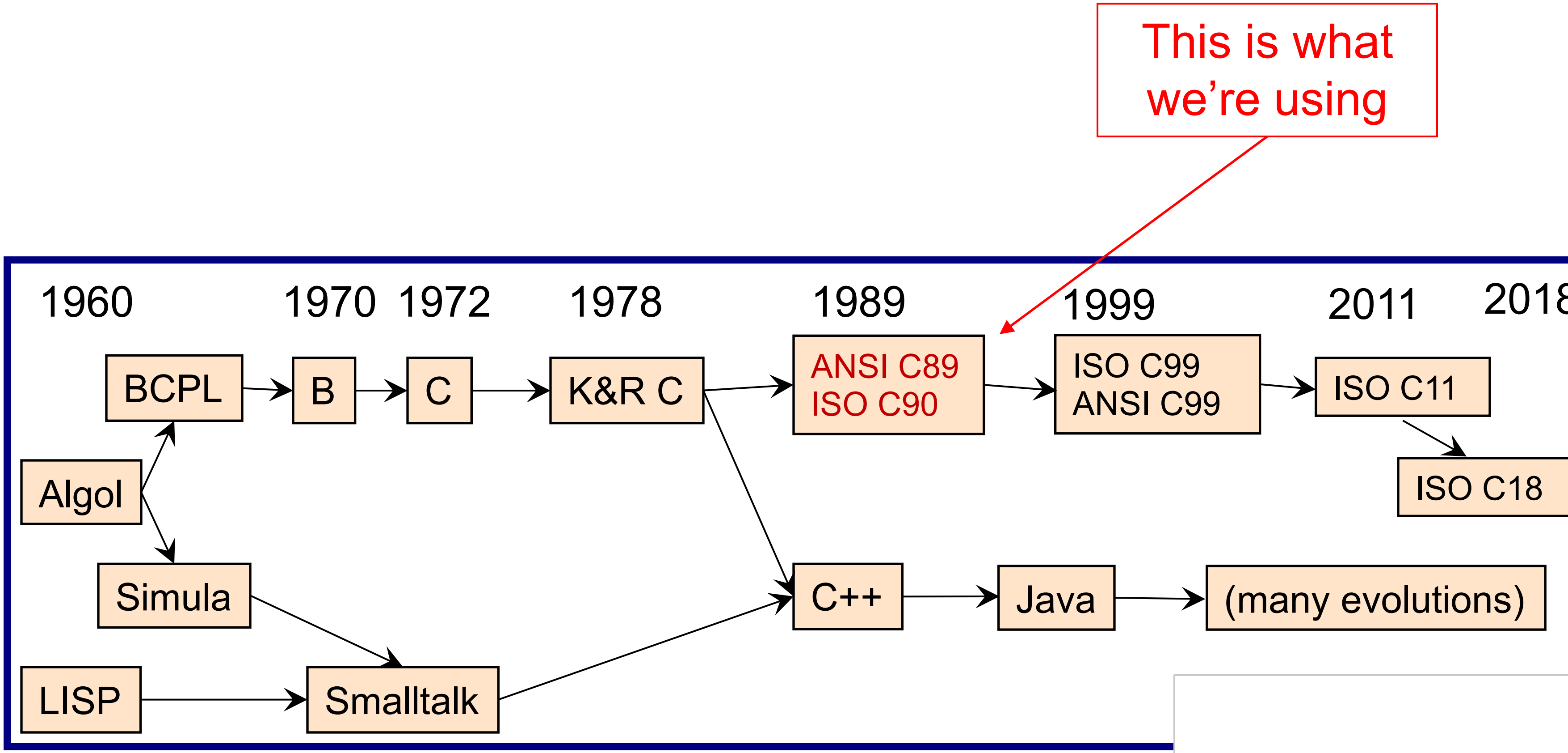
Who? Dennis Ritchie

When? ~1972

Where? Bell Labs

Why? Build the Unix OS

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

Getting started with C

- History of C
- **Building and running C programs**
- Characteristics of C

Three Simple C Programs

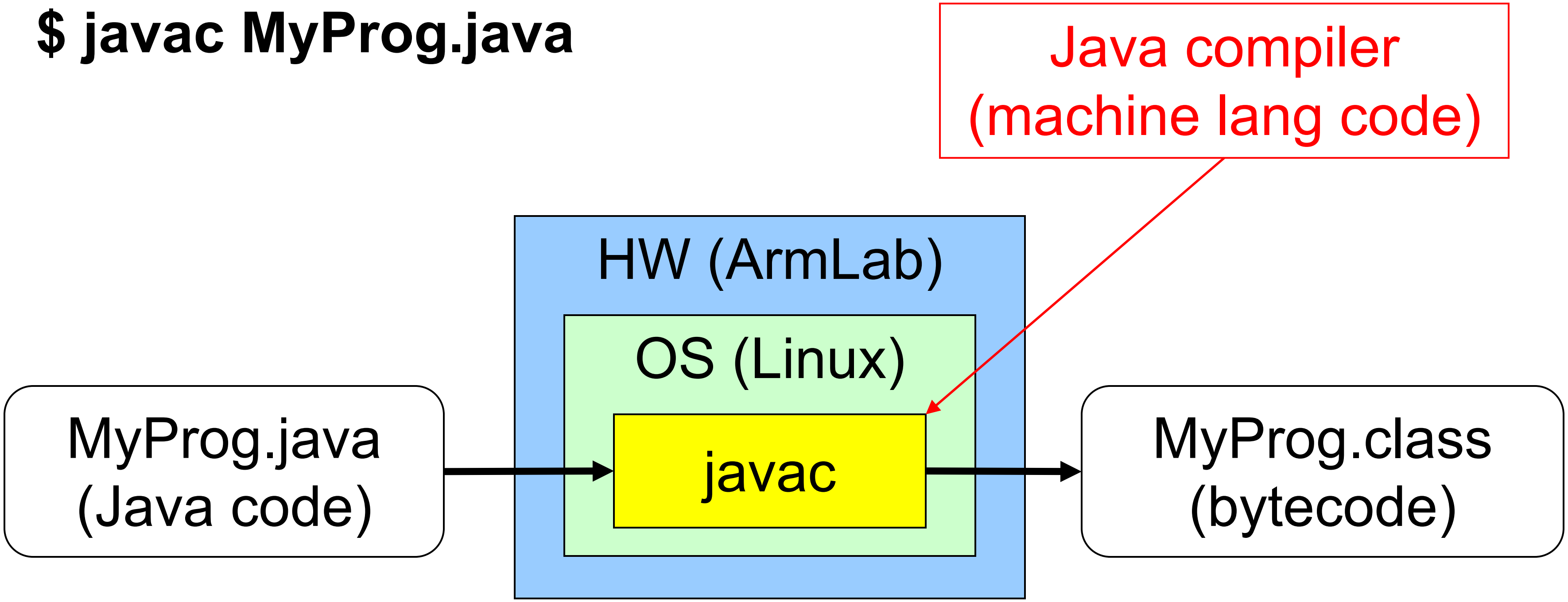
- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Java versus C Details

- For initial cram and/or later reference

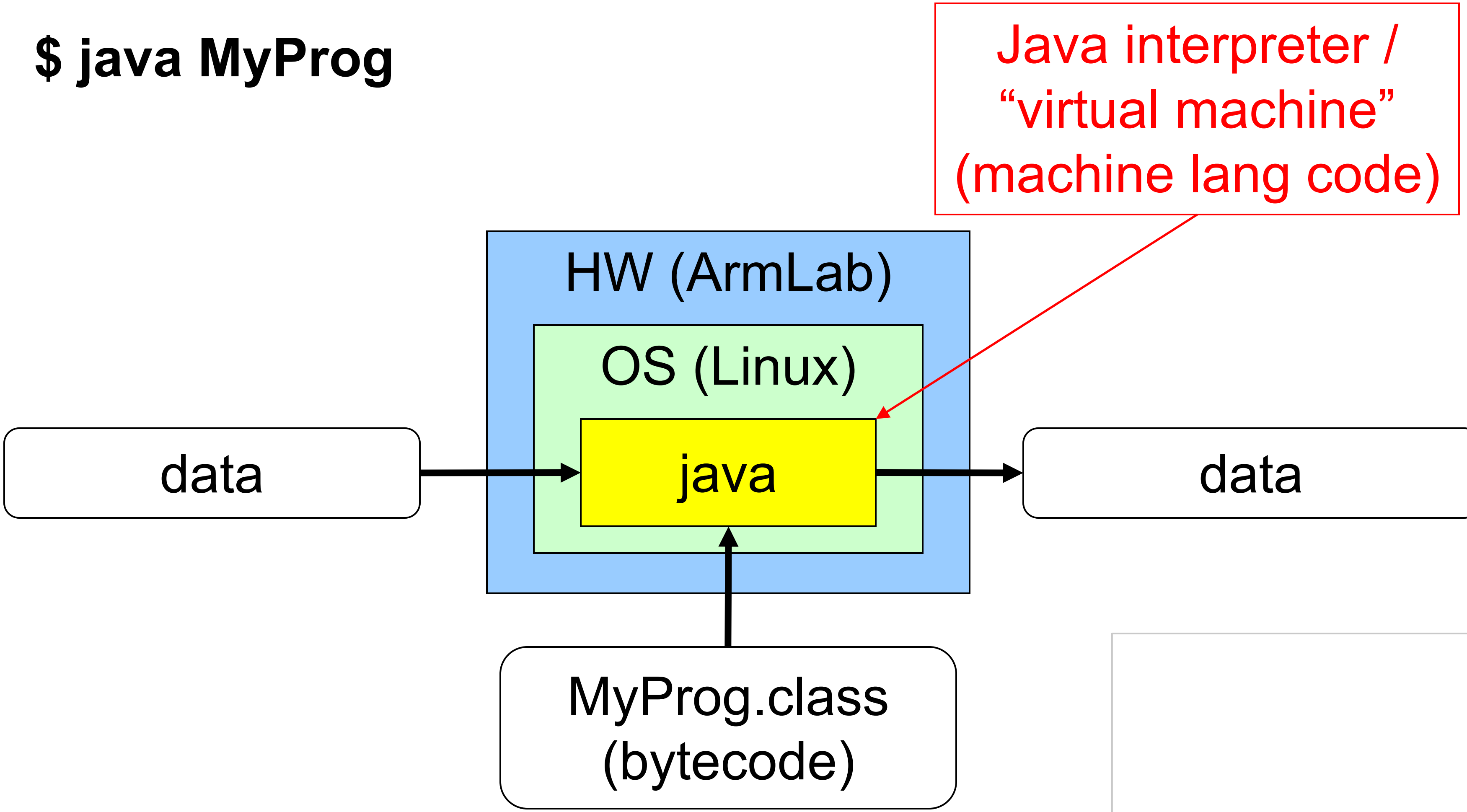
Building Java Programs

$ javac MyProg.java



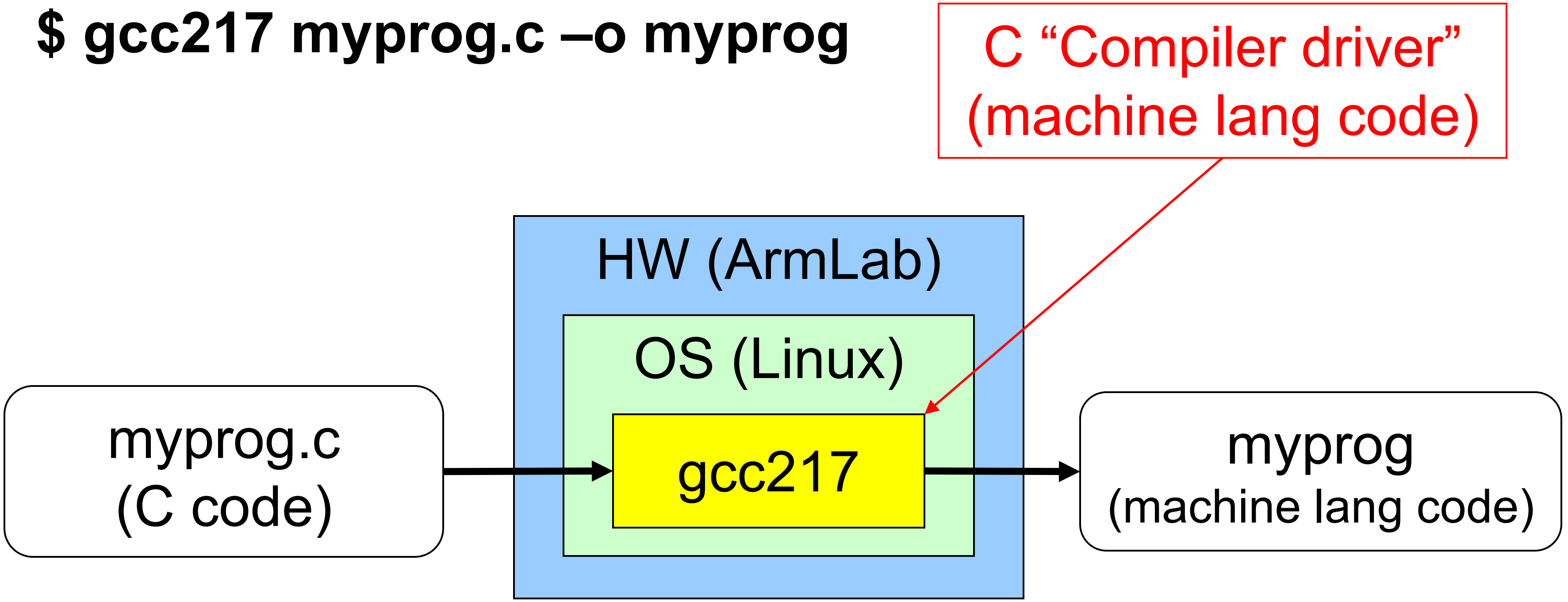
Running Java Programs

$ java MyProg



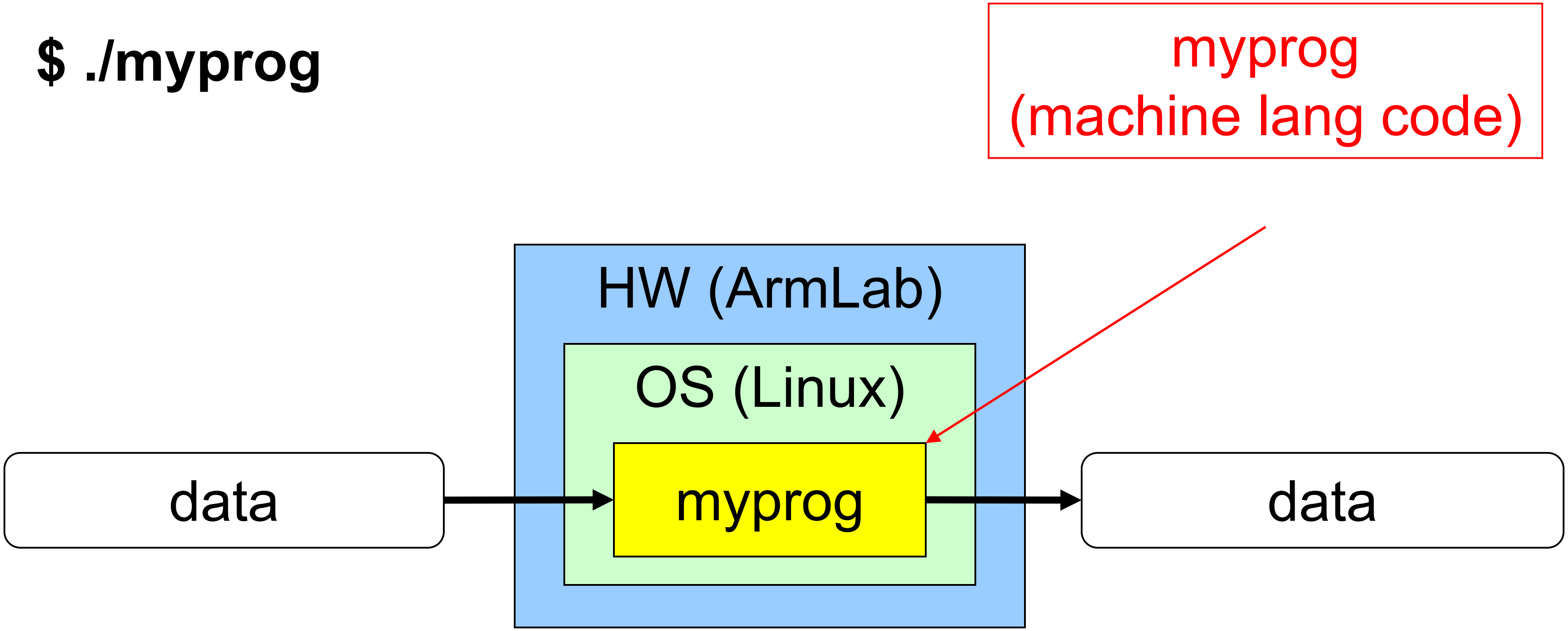
Building C Programs

$ gcc217 myprog.c –o myprog



Running C Programs

$ ./myprog



Agenda

Getting started with C

- History of C
- Building and running C programs
- **Characteristics of C**

Three Simple C Programs

- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Java versus C Details

- For initial cram and/or later reference

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

Example: since I've been here, we've used three architectures (x86, x86_64, and AArch64) and all our programs … class samples, assignment reference implementations, grading infrastructure, etc. had to be recompiled with each change!

Java vs. C: Safety & Efficiency

Java

- Automatic array-bounds checking,
- NULL pointer checking,
- Automatic memory management (garbage collection)
- Other safety features

C

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

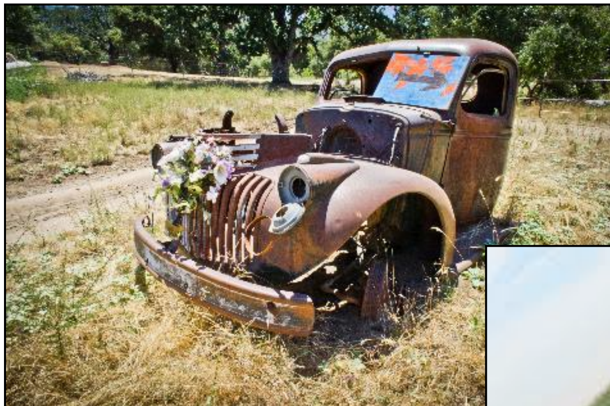
Conclusion 1: Java is often safer than C

Conclusion 2: Java is often slower than C

iClicker Question

Q: Which corresponds to the C programming language?

A.



B.

C.

Goals of the rest of this Lecture

Help you learn about:

- The basics of C
- Deterministic finite-state automata (DFA)
- Expectations for programming assignments

Why?

- Help you get started with Assignment 1
  - Required readings…
  - + coverage of programming environment in precepts…
  - + minimal coverage of C in this lecture…
  - = enough info to start Assignment 1
- DFAs are useful in many contexts
  - Theoretical problem characteristics + modeling
  - Practical system/program design (e.g. A1)

Agenda

Getting started with C

- History of C
- Building and running C programs
- Characteristics of C

Three Simple C Programs

- **charcount (loops, standard input)**
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

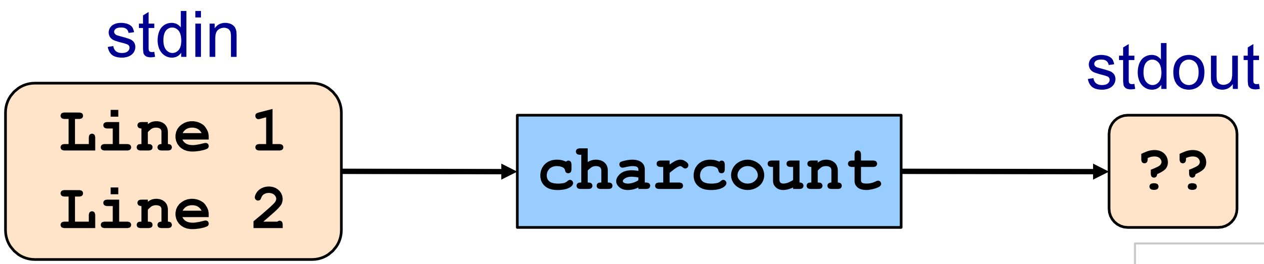
Java versus C Details

- For initial cram and/or later reference

The “charcount” Program

Functionality:

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



iClicker Question

Q: What is the output of **`charcount`** on this input?

```
[armlab01:lecture2$./charcount
[Line 1
[Line 2
```

stdout

??

A. 10

B. 12

C. 13

D. 14

E. 15

```
[armlab01:lecture2$wc -c
[Line 1
[Line 2
14
```

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;
}
```

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 ≠ 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() terminates program

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

"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
14
$
```

What is this?
What is the effect?

"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?

"charcount" Build Process in Detail

Question:

- Exactly what happens when you issue the command
  `gcc217 charcount.c –o charcount`

Answer: Four steps

- Preprocess
- Compile
- Assemble
- Link

“charcount” Build Process in Detail

The starting point

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;
}
```

- C language
- Missing declarations of getchar() and printf()
- Missing definitions of getchar() and printf()

Preprocessing “charcount”

Command to preprocess:

- `gcc217 -E charcount.c > charcount.i`

Preprocessor functionality

- Removes comments
- Handles preprocessor directives

Preprocessing “charcount”

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;
}
```

Preprocessor removes comment (this is A1!)

Preprocessing “charcount”

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;
}
```

Preprocessor replaces
#include <stdio.h>
with contents of
/usr/include/stdio.h

Preprocessor replaces
EOF with -1

Preprocessing “charcount”

The result

charcount.i

```
...
int getchar();
int printf(char *fmt, ...);
...


int main(void)
{   int c;
    int charCount = 0;
    c = getchar();
    while (c != -1)
    {   charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

- C language
- Missing comments
- Missing preprocessor directives
- Contains code from stdio.h: **declarations** of getchar() and printf()
- Missing **definitions** of getchar() and printf()
- Contains value for EOF

Compiling "charcount"

Command to compile:

- `gcc217 -S charcount.i`

Compiler functionality

- Translate from C to assembly language
- Use function declarations to check calls of getchar() and printf()

Compiling "charcount"

charcount.i

```
...
int getchar();
int printf(char *fmt, ...);
...
int main(void)
{   int c;
    int charCount = 0;
    c = getchar();
    while (c != -1)
    {   charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

- Compiler sees function declarations
- So compiler has enough information to check subsequent calls of getchar() and printf()

Compiling "charcount"

charcount.i

```
...
int getchar();
int printf(char *fmt, ...);
...
int main(void)
{   int c;
    int charCount = 0;
    c = getchar();
    while (c != -1)
    {   charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

- Definition of main() function
- Compiler checks calls of getchar() and printf() when encountered
- Compiler translates to assembly language

Compiling "charcount"

The result: charcount.s

```
        .section        .rodata
.LC0:
        .string "%d\n"

        .section        .text
        .global main
main:
        stp     x29, x30, [sp, -32]!
        add     x29, sp, 0
        str     wzr, [x29,24]
        bl      getchar
        str     w0, [x29,28]
        b       .L2
.L3:
        ldr     w0, [x29,24]
        add     w0, w0, 1
        str     w0, [x29,24]
        bl      getchar
        str     w0, [x29,28]
.L2:
        ldr     w0, [x29,28]
        cmn     w0, #1
        bne     .L3
        adrp    x0, .LC0
        add     x0, x0, :lo12:.LC0
        ldr     w1, [x29,24]
        bl      printf
        mov     w0, 0
        ldp     x29, x30, [sp], 32
        ret
```

- Assembly language
- Missing definitions of getchar() and printf()

Assembling “charcount”

Command to assemble:

- `gcc217 –c charcount.s`

Assembler functionality

- Translate from assembly language to machine language

Assembling "charcount"

The result:

charcount.o

```
Machine language
version of the
program

No longer human
readable
```

- Machine language
- Missing definitions of getchar() and printf()

Linking "charcount"

Command to link:

- `gcc217 charcount.o –o charcount`

Linker functionality

- Resolve references within the code
- Fetch machine language code from the standard C library (/usr/lib/libc.a) to make the program complete

Linking "charcount"

The result:

charcount

```
Machine language
version of the
program

No longer human
readable
```

- Machine language
- Contains definitions of getchar() and printf()

Complete! Executable!

iClicker Question

Q: There are other ways to **`charcount`** – which is best?

A.
```
for (c=getchar(); c!=EOF; c=getchar())
    charCount++;
```

B.
```
while ((c=getchar()) != EOF)
    charCount++;
```

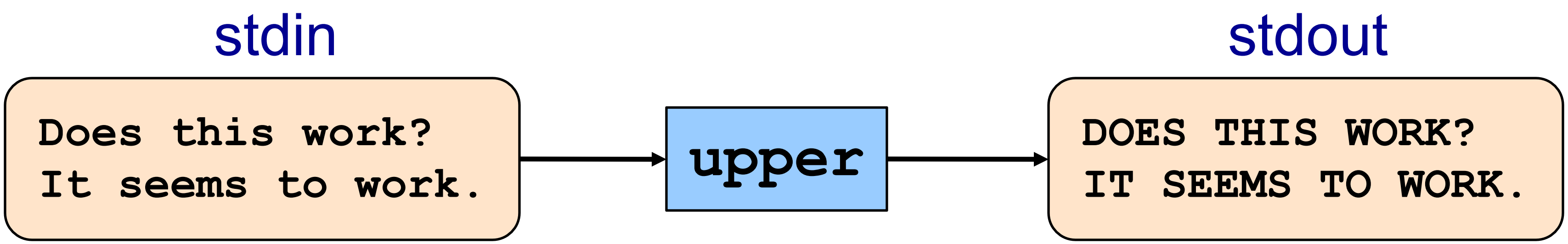
C.
```
for (;;)
{   c = getchar();
    if (c == EOF)
       break;
    charCount++;
}
```

D.
```
c = getchar();
while (c!=EOF)
{   charCount++;
    c =
    getchar();
}
```

Example 2: "upper"

Functionality

- Read all chars from stdin
- Convert each lower-case alphabetic char to upper case
    - Leave other kinds of chars alone
- Write result to stdout



ASCII

American Standard Code for Information Interchange

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL									HT	LF					
16																
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Partial map

Note: Lower-case and upper-case letters are 32 apart

"upper" Version 1

```
#include <stdio.h>
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if ((c >= 97) && (c <= 122))
         c -= 32;
      putchar(c);
   }
   return 0;
}
```

What's wrong?

Character Literals

Examples

```
'a'         the a character
                   97 on ASCII systems

'\n'        newline
                   10 on ASCII systems

'\t'        horizontal tab
                   9 on ASCII systems

'\\'        backslash
                   92 on ASCII systems

'\''        single quote
                   39 on ASCII systems

'\0'        the null character (alias NUL)
                   0 on all systems
```

"upper" Version 2

```
#include <stdio.h>
int main(void)
{   int c;
    while ((c = getchar()) != EOF)
    {   if ((c >= 'a') && (c <= 'z'))
            c += 'A' - 'a';
        putchar(c);
    }
    return 0;
}
```

Arithmetic on chars?

What's wrong now?

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 falls into various character classes

ctype.h Functions

```
$ man toupper
NAME
       toupper, tolower - convert letter to upper or lower case

SYNOPSIS
       #include <ctype.h>
       int toupper(int c);
       int tolower(int c);

DESCRIPTION
       toupper() converts the letter c to upper case, if possible.
       tolower() converts the letter c to lower case, if possible.

       If c is not an unsigned char value, or EOF, the behavior of
       these functions is undefined.

RETURN VALUE
       The value returned is that of the converted letter
       or c if the conversion was not possible.
```

“upper” Version 3

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{   int c;
    while ((c = getchar()) != EOF)
    {   if (islower(c))
            c = toupper(c);
        putchar(c);
    }
    return 0;
}
```

iClicker Question

Q: Is the **if** statement really necessary?

A. Gee, I don’t know. Let me check the man page (again)!

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
            c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

ctype.h Functions

```
$ man toupper
NAME
      toupper, tolower - convert letter to upper or lower case

SYNOPSIS
      #include <ctype.h>
      int toupper(int c);
      int tolower(int c);

DESCRIPTION
      toupper() converts the letter c to upper case, if possible.
      tolower() converts the letter c to lower case, if possible.

      If c is not an unsigned char value, or EOF, the behavior of
      these functions is undefined.

RETURN VALUE
      The value returned is that of the converted letter
      or c if the conversion was not possible.
```

iClicker Question

Q: Is the **if** statement really necessary?

A. Yes, necessary for correctness.

B. Not necessary, but I'd leave it in.

C. Not necessary, and I'd get rid of it.

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{   int c;
    while ((c = getchar()) != EOF)
    {  if (islower(c))
             c = toupper(c);
       putchar(c);
    }
    return 0;
}
```

Agenda

Getting started with C

- History of C
- Building and running C programs
- Characteristics of C

Three Simple C Programs

- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- **upper1 (switch statements, enums, functions)**
  - DFA program design

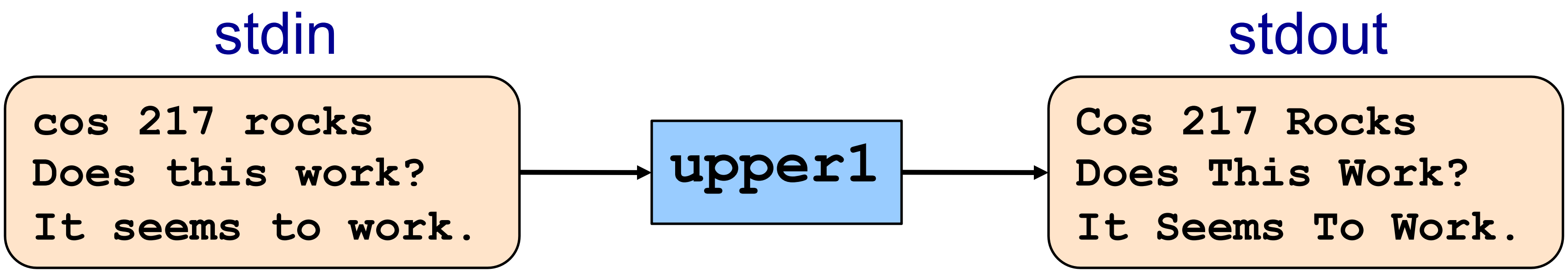
Java versus C Details

- For initial cram and/or later reference

Example 3: "upper1"

Functionality

- Read all chars from stdin
- Capitalize the first letter of each word
  - "cos 217 rocks" ⇒ "Cos 217 Rocks"
- Write result to stdout



“upper1” Challenge

Problem

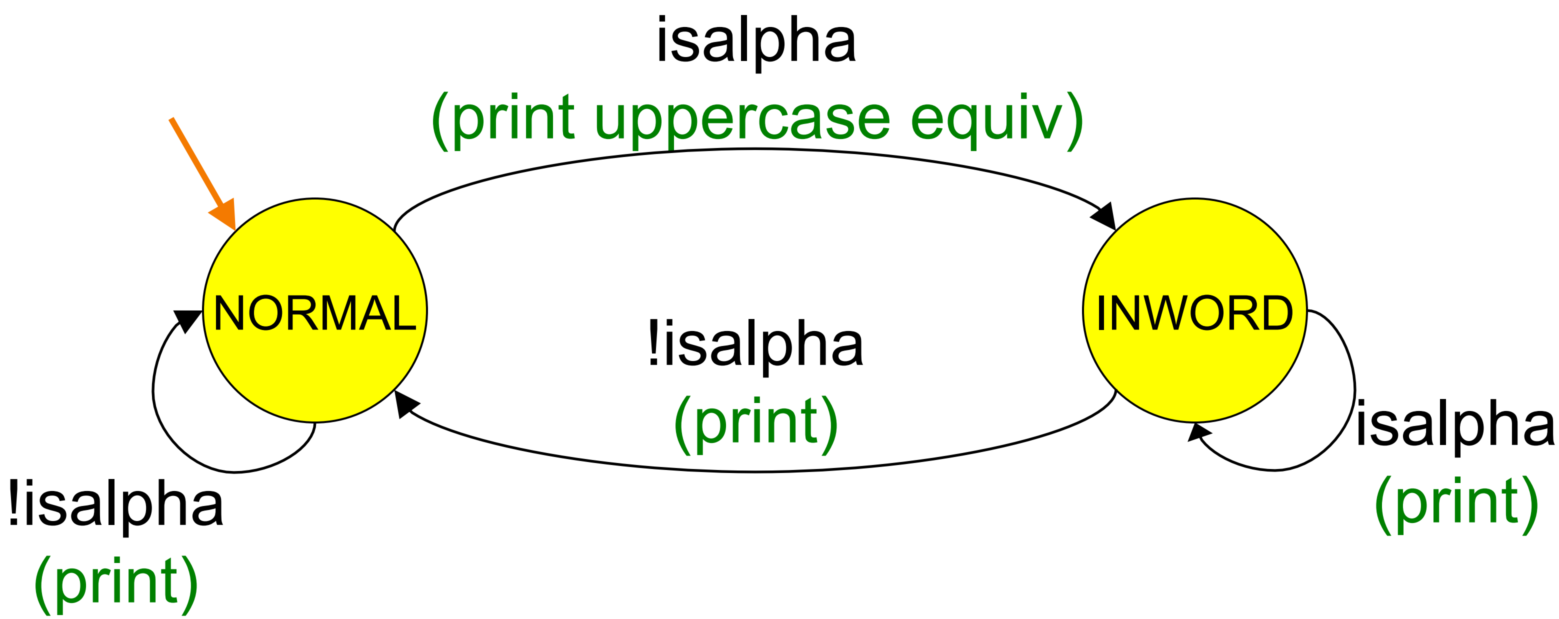
- Must remember where you are
- Capitalize “c” in “cos”, but not “o” in “cos” or “c” in “rocks”

Solution

- Maintain some extra information
- “In a word” vs “not in a word”

Deterministic Finite Automaton

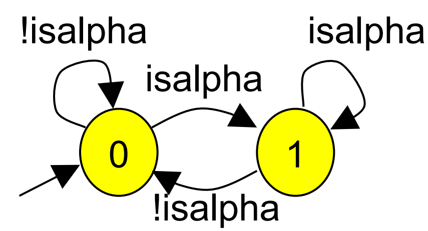
Deterministic Finite State Automaton (DFA)



- **States**, one of which denotes the **start**
- **Transitions** labeled by chars or categories
- Optionally, actions on transitions

“upper1” Version 1

```c
#include <stdio.h>
#include <ctype.h>
int main(void)
{   int c;
    int state = 0;
    while ((c = getchar()) != EOF)
    {   switch (state)
        {   case 0:
                if (isalpha(c))
                {   putchar(toupper(c)); state = 1; }
                else
                {   putchar(c); state = 0; }
                break;
            case 1:
                if (isalpha(c))
                {   putchar(c); state = 1; }
                else
                {   putchar(c); state = 0; }
                break;
        }
    }
    return 0;
}
```



That’s a B.
What’s wrong?

“upper1” Toward Version 2

Problem:

- The program works, but…
- States should have names

Solution:

- Define your own named constants
- **`enum Statetype {NORMAL, INWORD};`**
  - Define an enumeration type
- **`enum Statetype state;`**
  - Define a variable of that type

“upper1” Version 2

```
#include <stdio.h>
#include <ctype.h>
enum Statetype {NORMAL, INWORD};
int main(void)
{  int c;
   enum Statetype state = NORMAL;
   while ((c = getchar()) != EOF)
   {  switch (state)
      {  case NORMAL:
            if (isalpha(c))
            {  putchar(toupper(c)); state = INWORD; }
            else
            {  putchar(c); state = NORMAL; }
            break;
         case INWORD:
            if (isalpha(c))
            {  putchar(c); state = INWORD; }
            else
            {  putchar(c); state = NORMAL; }
            break;
      }
   }
   return 0;
}
```

That’s a B+.
What’s wrong?

"upper1" Toward Version 3

Problem:

- The program works, but…
- Deeply nested statements
- No modularity

Solution:

- Handle each state in a separate function

"upper1" Version 3

```
#include <stdio.h>
#include <ctype.h>
enum Statetype {NORMAL, INWORD};

enum Statetype handleNormalState(int c)
{  enum Statetype state;
   if (isalpha(c))
   {  putchar(toupper(c));
      state = INWORD;
   }
   else
   {  putchar(c);
      state = NORMAL;
   }
   return state;
}

enum Statetype handleInwordState(int c)
{  enum Statetype state;
   if (!isalpha(c))
   {  putchar(c);
      state = NORMAL;
   }
   else
   {  putchar(c);
      state = INWORD;
   }
   return state;
}
```

```
int main(void)
{  int c;
   enum Statetype state = NORMAL;
   while ((c = getchar()) != EOF)
   {  switch (state)
      {  case NORMAL:
            state = handleNormalState(c);
            break;
         case INWORD:
            state = handleInwordState(c);
            break;
      }
   }
   return 0;
}
```

That's an A-.
What's wrong?

“upper1” Toward Final Version

Problem:

- The program works, but…
- No comments

Solution:

- Add (at least) function-level comments

Function Comments

Function comment should describe
what the function does (from the caller's viewpoint)

- Input to the function
  - Parameters, input streams
- Output from the function
  - Return value, output streams, (call-by-reference parameters)

Function comment should **not** describe
how the function works

Function Comment Examples

Bad main() function comment

```
Read a character from stdin. Depending upon
the current DFA state, pass the character to
an appropriate state-handling function. The
value returned by the state-handling function
is the next DFA state. Repeat until end-of-file.
```

Describes **how the function works**

Good main() function comment

```
Read text from stdin. Convert the first character
of each "word" to uppercase, where a word is a
sequence of characters. Write the result
to stdout. Return 0.
```

Describes **what the function does**
(from caller's viewpoint)

"upper1" Final Version

```
/*------------------------------------------------------------*/
/* upper1.c                                                   */
/* Author: Bob Dondero                                        */
/*------------------------------------------------------------*/

#include <stdio.h>
#include <ctype.h>

enum Statetype {NORMAL, INWORD};
```

Continued on next page

“upper1” Final Version

```
/*------------------------------------------------------------*/

/* Implement the NORMAL state of the DFA. c is the current
   DFA character. Write c or its uppercase equivalent to
   stdout, as specified by the DFA. Return the next state. */

enum Statetype handleNormalState(int c)
{  enum Statetype state;
   if (isalpha(c))
   {  putchar(toupper(c));
      state = INWORD;
   }
   else
   {  putchar(c);
      state = NORMAL;
   }
   return state;
}
```

Continued on next page

"upper1" Final Version

```
/*------------------------------------------------------------*/

/* Implement the INWORD state of the DFA. c is the current
   DFA character. Write c to stdout, as specified by the DFA.
   Return the next state. */

enum Statetype handleInwordState(int c)
{  enum Statetype state;
   if (!isalpha(c))
   {  putchar(c);
      state = NORMAL;
   }
   else
   {  putchar(c);
      state = INWORD;
   }
   return state;
}
```

Continued on next page

“upper1” Final Version

```
/*----------------------------------------------------------*/

/* Read text from stdin. Convert the first character of each
   "word" to uppercase, where a word is a sequence of
   letters. Write the result to stdout. Return 0. */

int main(void)
{  int c;
   /* Use a DFA approach.  state indicates the DFA state. */
   enum Statetype state = NORMAL;
   while ((c = getchar()) != EOF)
   {  switch (state)
      {  case NORMAL:
            state = handleNormalState(c);
            break;
         case INWORD:
            state = handleInwordState(c);
            break;
      }
   }
   return 0;
}
```

Review of Example 3

Deterministic finite-state automaton

- Two or more states
- Transitions between states
  - Next state is a function of current state and current character
- Actions can occur during transitions

Expectations for COS 217 assignments

- Readable
  - Meaningful names for variables, constants, and literals
  - Reasonable max nesting depth
- Modular
  - Multiple functions, each with 1 well-defined job
- Function-level comments
  - Should describe what function does
- See K&P book for style guidelines specification

Agenda

Getting started with C

- History of C
- Building and running C programs
- Characteristics of C

Three Simple C Programs

- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Java versus C Details

- **For initial cram and/or later reference**

82

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 $

83

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};`

84

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;`
		`};`

85

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

86

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

87

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

88

Java vs. C: Details

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

Compound stmt (aka: 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

89

Example C Program

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

int main(void)
{  const double KMETERS_PER_MILE = 1.609;
   int miles;
   double kMeters;

   printf("miles: ");
   if (scanf("%d", &miles) != 1)
   {  fprintf(stderr, "Error: Expected a number.\n");
      exit(EXIT_FAILURE);
   }

   kMeters = (double)miles * KMETERS_PER_MILE;
   printf("%d miles is %f kilometers.\n",
      miles, kMeters);
   return 0;
}
```


90