COS 217: Introduction to Programming Systems

Character Manipulation and DFAs

PRINCETON UNIVERSITY

Agenda

Simple C Programs

- upper (character data and I/O, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Two big differences from Java

- Variable declarations
- Logical operators

Agenda

Simple C Programs

- upper (character data and I/O, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

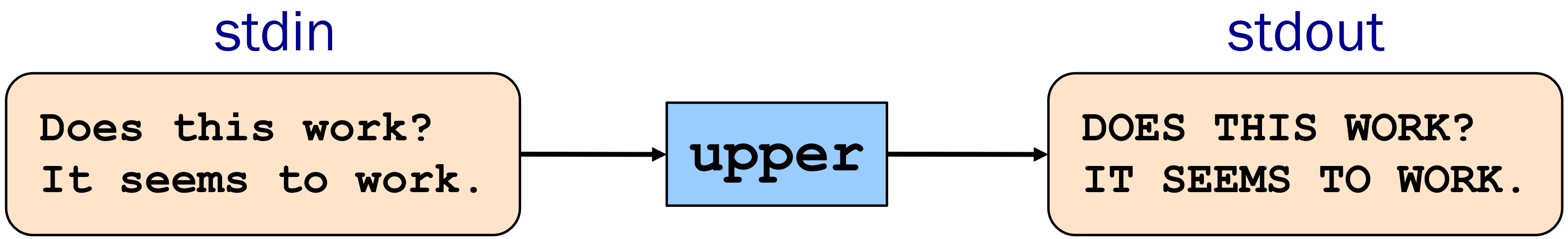
Two big differences from Java

- Variable declarations
- Logical operators

Simple C program: “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



What we need: character representation, I/O

The C char Data Type

char is 1 byte – designed to hold a single character

- Might be signed (-128..127) or unsigned (0..255)
- If using chars for arbitrary one-byte data, good to specify as “signed char” or “unsigned char”

Mapping from char values to characters on pretty much all machines:

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	{			}	~	

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

Character Literals

Single quote syntax: 'a' is a value of type char with the value 97

Use backslash to write special characters

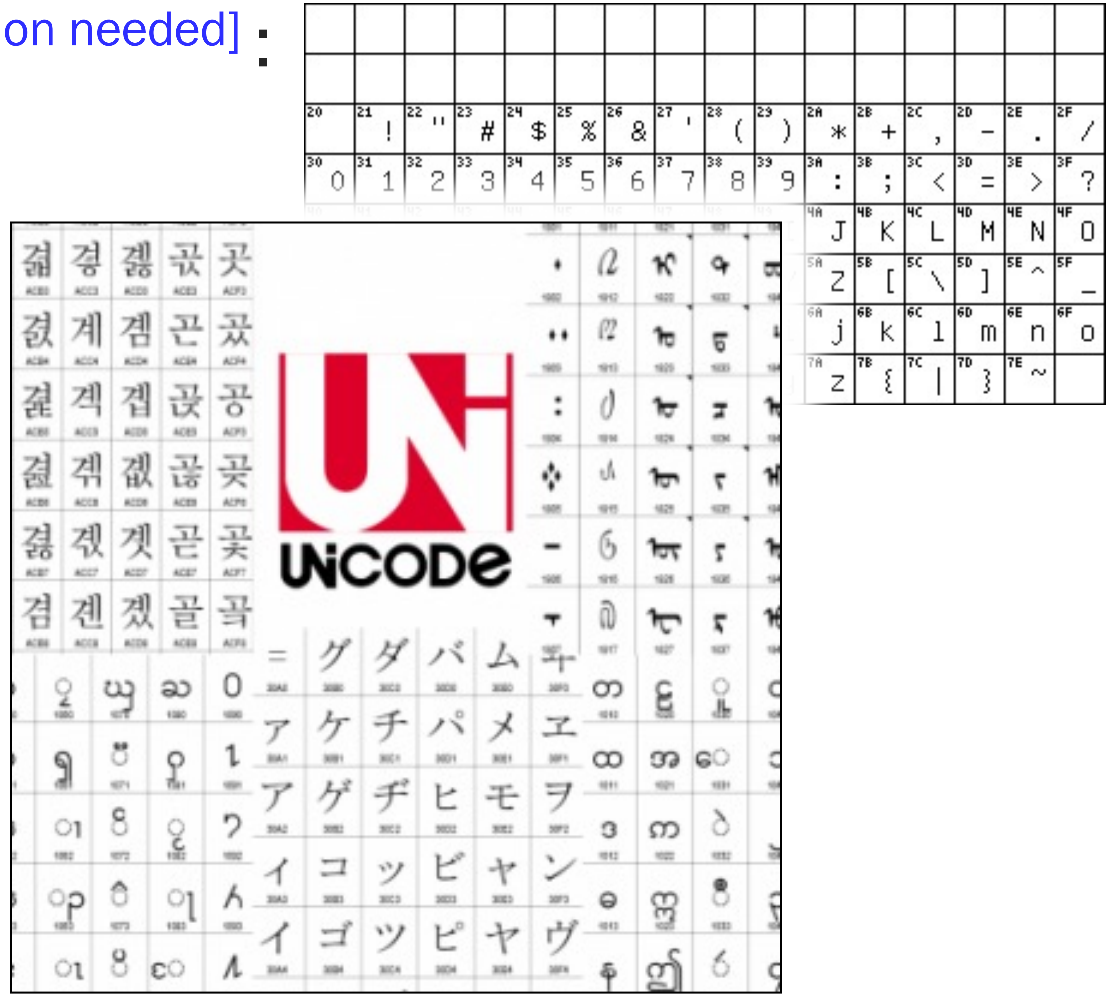
- Examples (with numeric equivalents in ASCII):

```
'a'    the a character (97)
'A'    the A character (65)
'0'    the zero character (48)
'\0'   the null character (0)
'\n'   the newline character (10)
'\t'   the horizontal tab character (9)
'\\'   the backslash character (92)
'\''   the single quote character (39)
'"'    the double quote character (34)
```

Aside: Unicode

Back in 1970s, English was the only language in the world[citation needed]

so we all used this alphabet [citation needed] :



ASCII:

American Standard Code
for Information Interchange

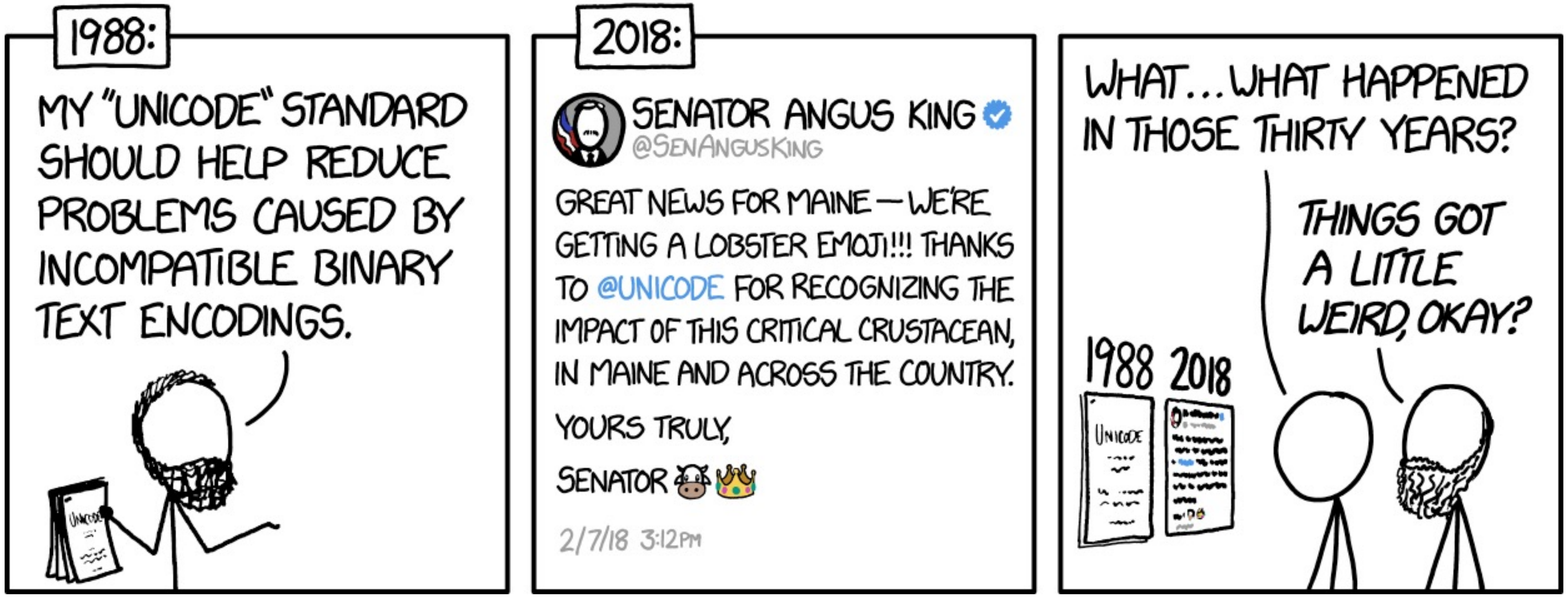
In the 21st century, it turns out
there are other languages!

Modern Unicode

When C was designed, characters fit into 8 (really 7) bits, so C's chars are 8 bits long.

When Java was designed, Unicode fit into 16 bits, so Java's chars are 16 bits long.

Then this happened:



https://xkcd.com/1953/

Result: modern systems use *variable length* (UTF-8) encoding for Unicode.

Character Input/Output (I/O) in C

Design of C:

- Does not provide I/O facilities in the language
- Instead provides I/O facilities in standard library, declared in stdio.h
  - Constant: EOF
  - Data type: FILE (described later in course)
  - Variables: stdin, stdout, and stderr
  - Functions: ...

Reading characters

- getchar() function with return type wider than char (specifically, int)
- Returns EOF (a special non-character int) to indicate failure
- Reminder: there is no such thing as "the EOF character”

Writing characters

- putchar() function accepting one parameter
- For symmetry with getchar(), parameter is an int

“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?

“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

Simple C Programs

- upper (character data and I/O, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

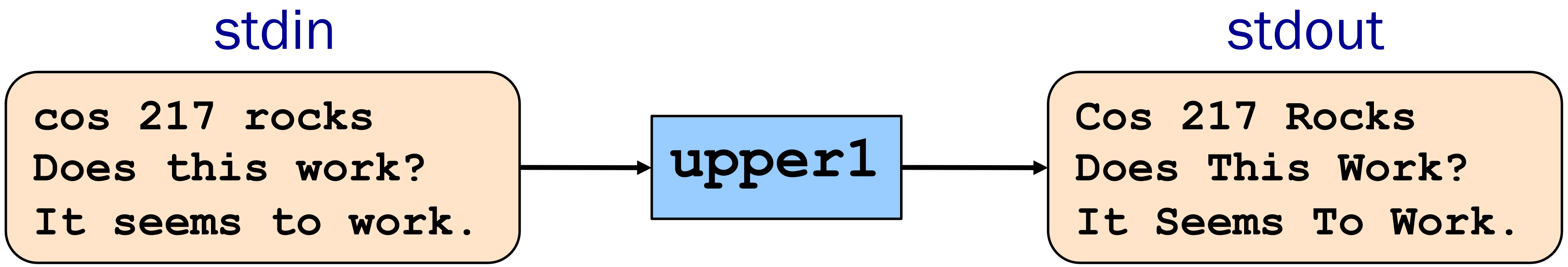
Two big differences from Java

- Variable declarations
- Logical operators

The "upper1" program

Functionality

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

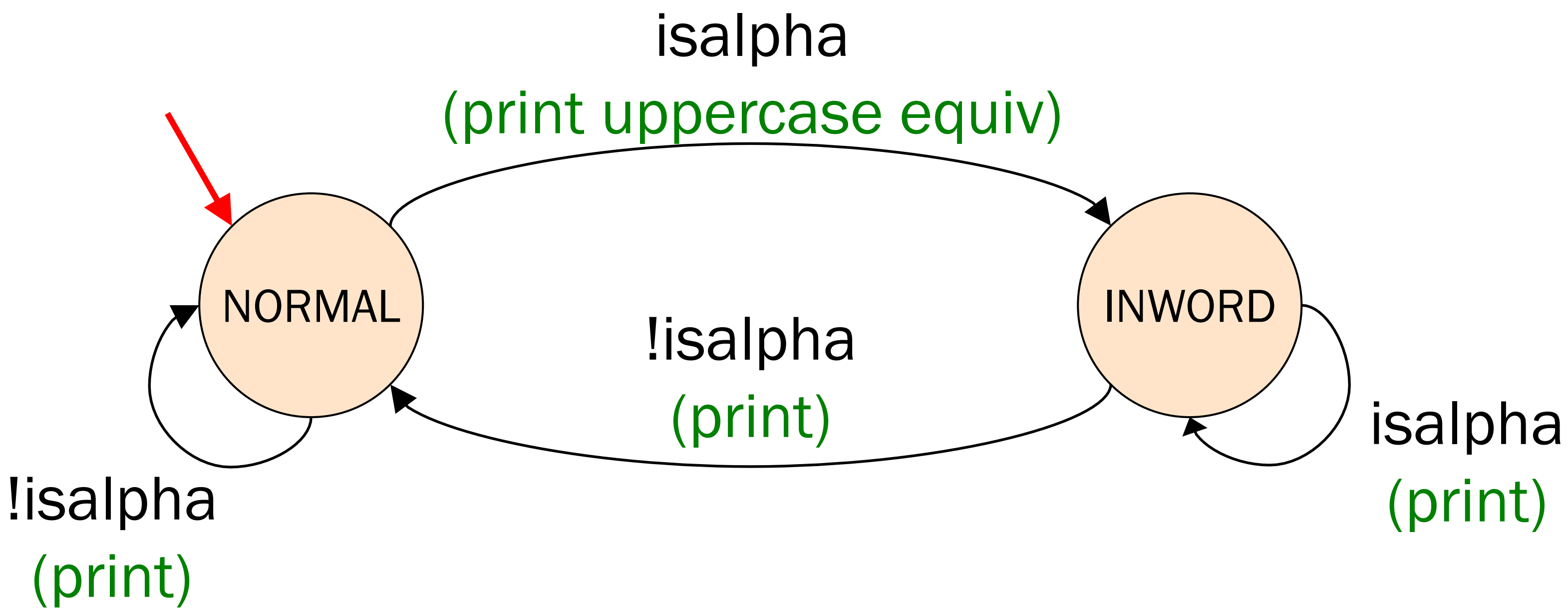


What we need: maintain extra information, namely "in a word" vs "*not* in a word"

- Need systematic way of reasoning about what to do with that information

Deterministic Finite Automaton

Deterministic Finite State Automaton (DFA)

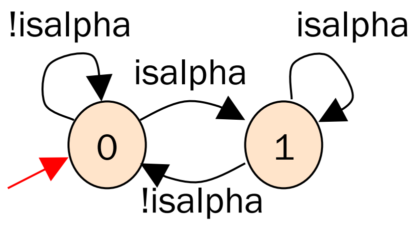


- States, one of which denotes the start
- Transitions labeled by chars or categories
- Optionally, actions on transitions

“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 slide

“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 slide

“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 slide

“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;
}
```

Agenda

Simple C Programs

- upper (character data and I/O, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Two big differences from Java

- Variable declarations
- Logical operators

Declaring Variables

C requires variable declarations.

Motivation:

- Declaring variables allows compiler to check “spelling”
- Declaring variables allows compiler to allocate memory more efficiently
- Declaring variables’ types produces fewer surprises at runtime
- Declaring variables requires more from the programmer
  - Extra verbiage
  - Type foresight
  - “Do what I mean, not what I say”

Declaring Variables

C requires variable declarations.

- Declaration statement specifies type of variable (and other attributes too)

Examples:

```
int i;
int i, j;
int i = 5;
const int i = 5; /* value of i cannot change */
static int i; /* covered later in course */
extern int i; /* covered later in course */
```

Declaring Variables

C requires variable declarations.

- Declaration statement specifies type of variable (and other attributes too)
- Unlike Java, declaration statements in C90 must appear before any other kind of statement in compound statement

```
{
   int i;
   /* Non-declaration
      stmts that use i. */
   …
   int j;
   /* Non-declaration
      stmts that use j. */
   …
}
```

Illegal in C

```
{
   int i;
   int j;
   /* Non-declaration
      stmts that use i. */
   …
   /* Non-declaration
      stmts that use j. */
   …
}
```

Legal in C

Agenda

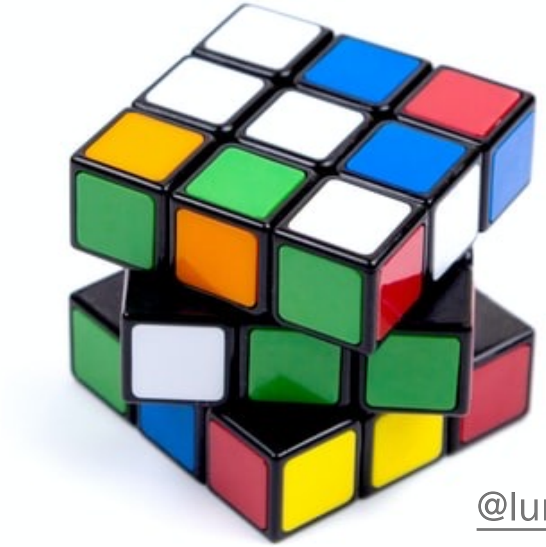
Simple C Programs

- upper (character data and I/O, ctype library)
  - portability concerns
- upper1 (switch statements, enums, functions)
  - DFA program design

Two big differences from Java

- Variable declarations
- Logical operators

Logical Data Types



@lunarts

- No separate logical or Boolean data type
- Represent logical data using type char or int
  - Or any primitive type! :/
- Conventions:
  - Statements (if, while, etc.) use 0 ⇒ FALSE, ≠0 ⇒ TRUE
  - Relational operators (<, >, etc.) and logical operators (!, &&, ||) produce the result 0 or 1

Logical Data Type Shortcuts

Using integers to represent logical data permits shortcuts

```
…
int i;
…
if (i)   /* same as (i != 0) */
    statement1;
else
    statement2;
…
```

It also permits some really bad code…

```
i = (1 != 2) + (3 > 4);
```

iClicker Brainteaser

Q: What is **`i`** set to in the following code?

```
i = (1 != 2) + (3 > 4);
```

A. 0

B. 1

C. 2

D. 3

E. 4

Logical Data Type Dangers

Beware: the following code will cause loss of sleep

```
…
int i;
…
i = 0;
…
if (i = 5)
   statement1;
…
```

What happens in Java?

What happens in C?

Appendix:

Additional DFA Examples

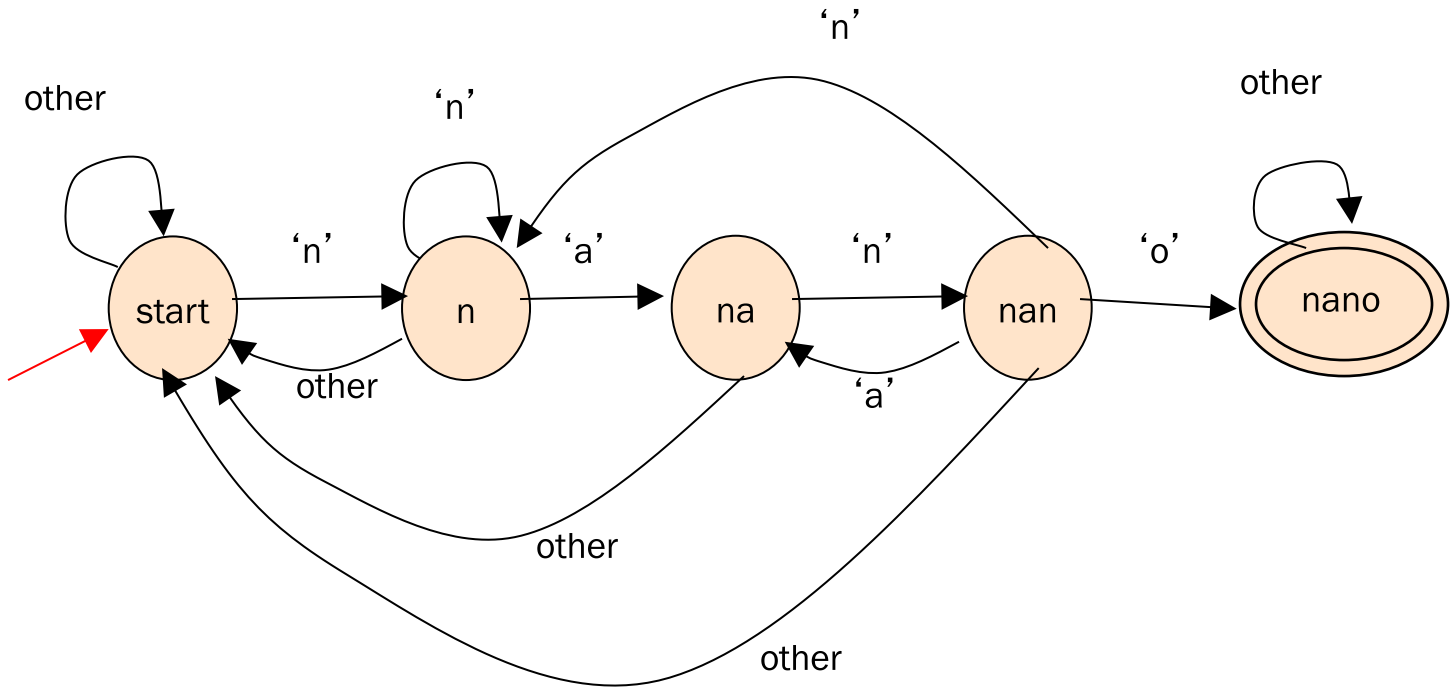
Another DFA Example

Does the string have "nano" in it?

- "banano" ⇒ yes
- "nnnnnnnanofff" ⇒ yes
- "banananonano" ⇒ yes
- "bananananashanana" ⇒ no

Double circle is accepting state

Single circle is rejecting state



Yet Another DFA Example

Old Exam Question

Compose a DFA to identify whether or not a string is a floating-point literal

Valid literals

- "-34"
- "78.1"
- "+298.3"
- "-34.7e-1"
- "34.7E-1"
- "7."
- ".7"
- "999.99e99"

Invalid literals

- "abc"
- "-e9"
- "1e"
- "+"
- "17.9A"
- "0.38+"
- "."
- "38.38f9"