# COS 217: Introduction to Programming Systems

## Building C Programs
## &
## Implementing DFAs in C

PRINCETON UNIVERSITY

# Agenda

Building simple C programs

- examine 4-stage build process for charcount

"DFA model" character processing programs

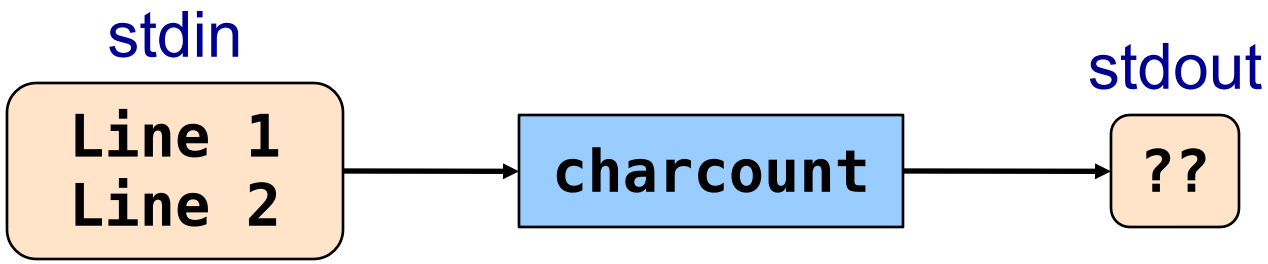
- upper: demonstrate ctype library for character data
- upper1: DFA model
- upper1: develop a C program to implement the DFA

Next time: design decisions in charcount, upper, upper1

# Last time: The `charcount` Program

Functionality:

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



# Last time: 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;
}
```

# Last time: `charcount` Building and Running

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

# `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()`

# `charcount` Build Process: Preprocessor

Command to preprocess:

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

Preprocessor functionality

- Removes comments
- Handles preprocessor directives

# charcount Build Process: Preprocessor

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!)

# charcount Build Process: Preprocessor

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

# `charcount` Build Process: Preprocessor

The result

`charcount.c`

```
...
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`

# `charcount` Build Process: Compiler

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()`

# charcount Build Process: Compiler

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
- These give compiler enough information to check subsequent calls of `getchar()` and `printf()`

# `charcount` Build Process: Compiler

`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()`
- Compiler translates code to assembly language instructions

# `charcount` Build Process: Compiler

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()`

# `charcount` Build Process: Assembler

Command to assemble:

- `gcc217 –c charcount.s`

Assembler functionality

- Translate from assembly language to machine language

# `charcount` Build Process: Assembler

The result:

`charcount.o`

```
Machine language
version of the
program

No longer human
readable
```

- Machine language
- (Still!) Missing definitions of `getchar()` and `printf()`

# `charcount` Build Process: Linker

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
- Produce final executable

# `charcount` Build Process: Linker

The result:

`charcount`

```
Machine language
version of the
program

No longer human
readable
```

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

Complete! Executable!

# Agenda

Building simple C programs

- examine 4-stage build process for charcount

"DFA model" character processing programs

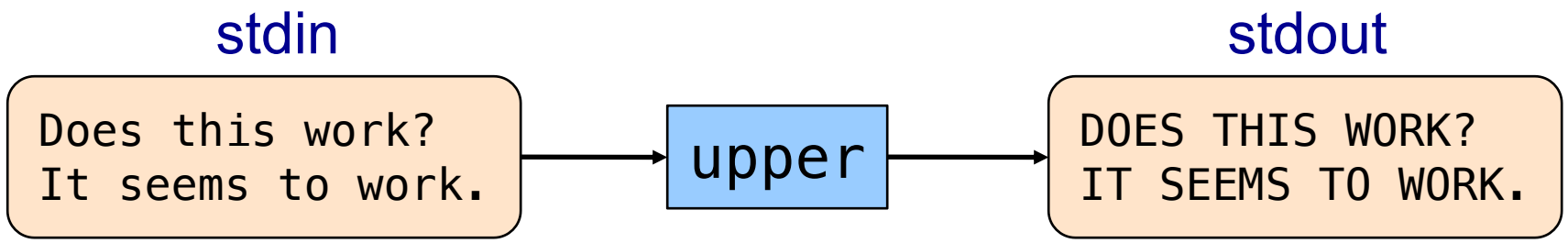
- upper: demonstrate ctype library for character data
- upper1: DFA model
- upper1: develop a C program to implement the DFA

Next time: design decisions in charcount, upper, upper1

# Getting closer: `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



# upper (starting at Version 3 ... 1 and 2 next time!)

```
#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 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.

# 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.
```

# Agenda

Building simple C programs

- examine 4-stage build process for charcount

"DFA model" character processing programs

- upper: demonstrate ctype library for character data
- upper1: DFA model
- upper1: develop a C program to implement the DFA

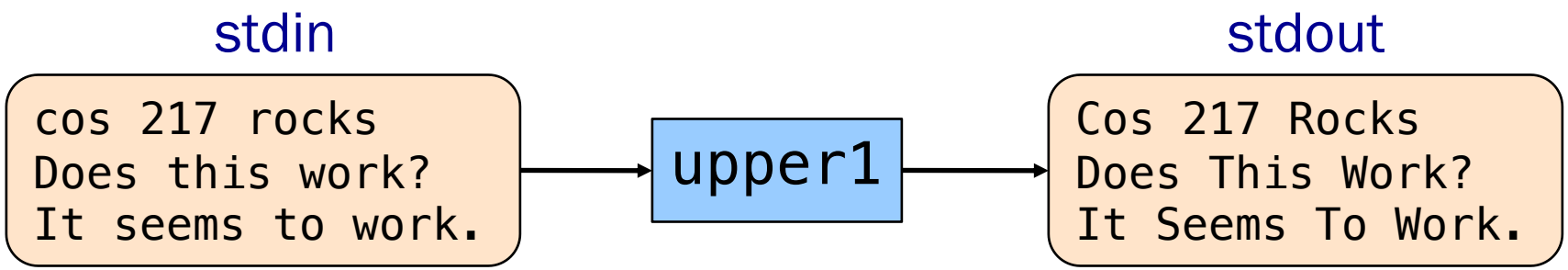
Next time: design decisions in charcount, upper, upper1

# 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



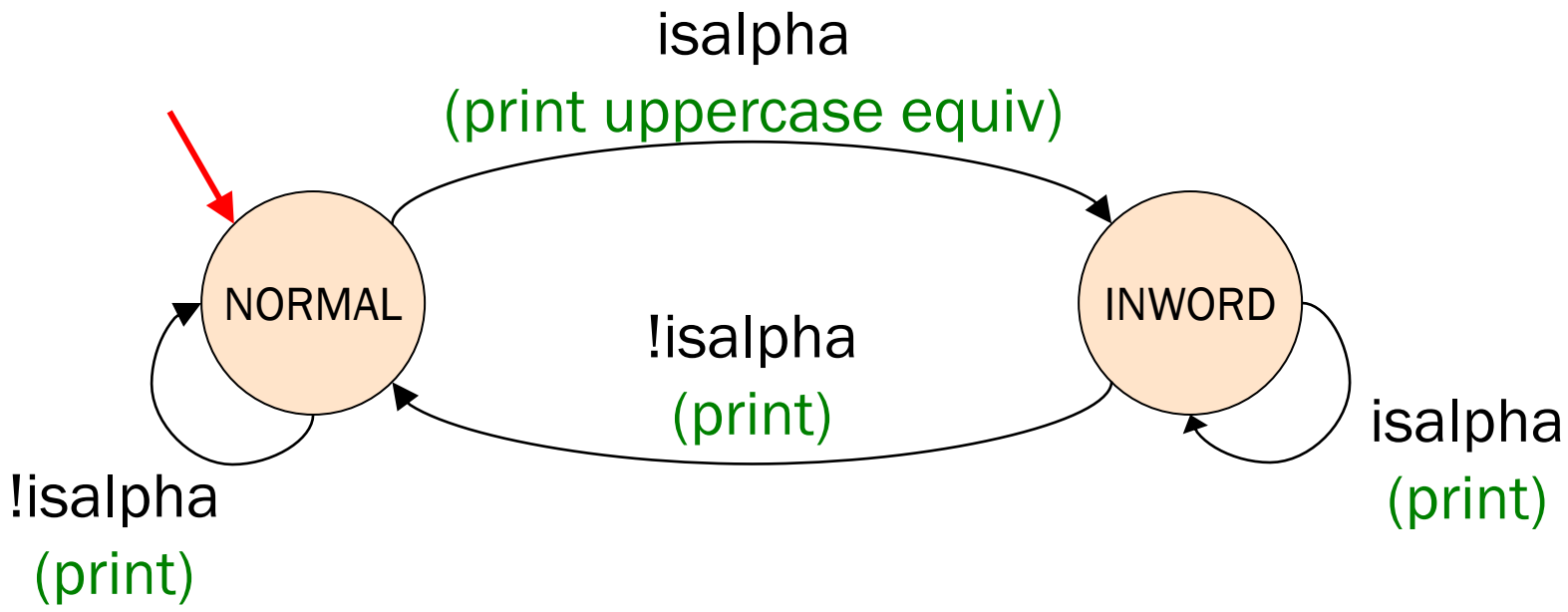

What we need:

1. to recognize when we're *"in a word"* vs *"**not** in a word"*
2. to reason about what to do with that information in a systematic way

# Deterministic Finite Automaton



Deterministic Finite State Automaton (DFA)




- States, one of which is designated as the start
- Transitions labeled by individual or categories of chars
- Optionally, actions on transitions

# Agenda

Building simple C programs

- examine 4-stage build process for charcount

"DFA model" character processing programs

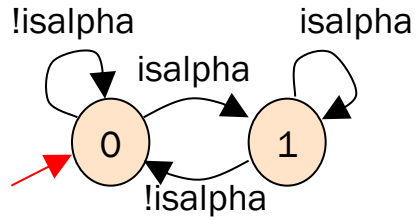
- upper: demonstrate ctype library for character data
- upper1: DFA model
- upper1: develop a C program to implement the DFA

Next time: design decisions in charcount, upper, upper1

# upper1 Version 1

```
#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

```
…
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?

# Agenda

Building simple C programs

- examine 4-stage build process for charcount

"DFA model" character processing programs

- upper: demonstrate ctype library for character data
- upper1: DFA model
- upper1: develop a C program to implement the DFA

Next time: design decisions in charcount, upper, upper1

# Appendix:

# Additional DFA Examples

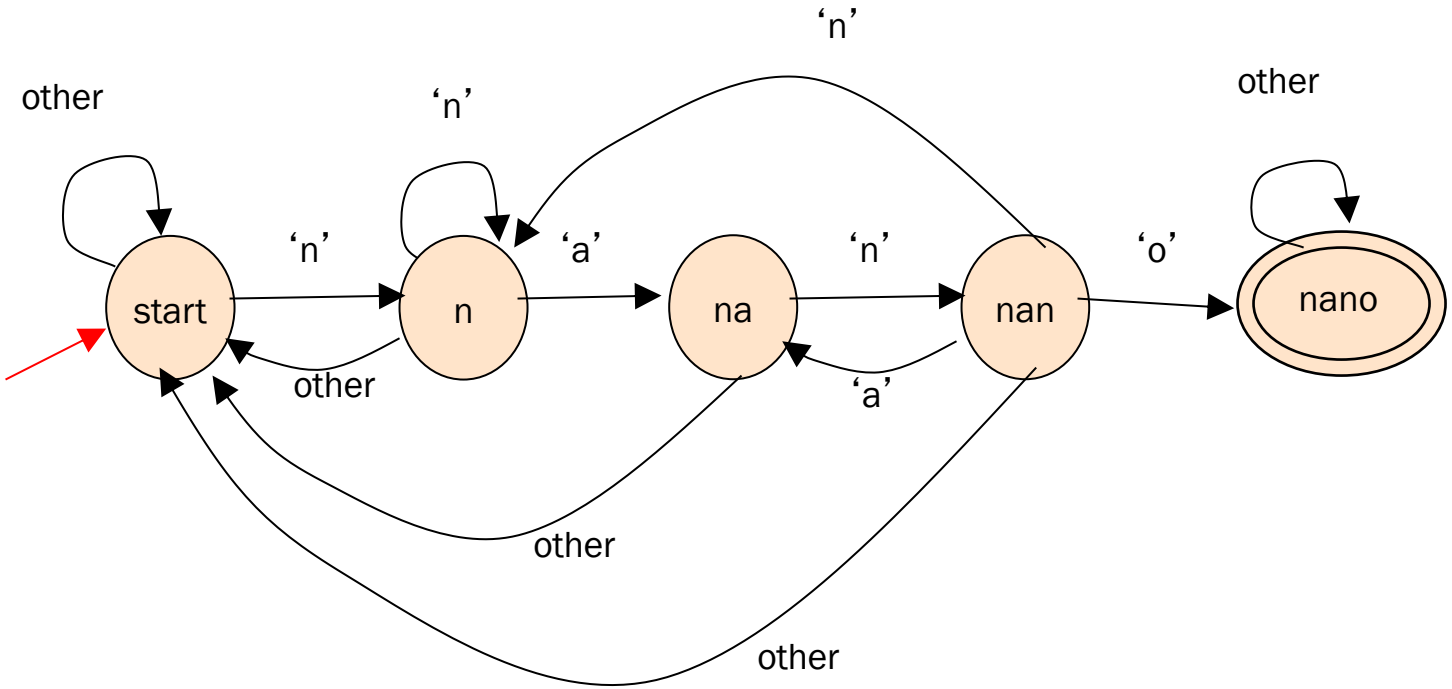
# Another DFA Example

Does the string have “nano” in it?

- “banano” ⇒ yes
- “nnnnnnnanofff” ⇒ yes
- “bananananonano” ⇒ 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”