Character Manipulation and DFAs
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 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)

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

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):

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>ASCII Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>the a character (97)</td>
<td></td>
</tr>
<tr>
<td>'A'</td>
<td>the A character (65)</td>
<td></td>
</tr>
<tr>
<td>'0'</td>
<td>the zero character (48)</td>
<td></td>
</tr>
<tr>
<td>'\0'</td>
<td>the null character (0)</td>
<td></td>
</tr>
<tr>
<td>'\n'</td>
<td>the newline character (10)</td>
<td></td>
</tr>
<tr>
<td>'\t'</td>
<td>the horizontal tab character (9)</td>
<td></td>
</tr>
<tr>
<td>'\'</td>
<td>the backslash character (92)</td>
<td></td>
</tr>
<tr>
<td>'&quot;'</td>
<td>the single quote character (39)</td>
<td></td>
</tr>
<tr>
<td>'''</td>
<td>the double quote character (34)</td>
<td></td>
</tr>
</tbody>
</table>
Aside: Unicode

Back in 1970s, English was the only language in the world \[\text{[citation needed]}\] so we all used this alphabet \[\text{[citation needed]}\]:

**ASCII:**

American Standard Code for Information Interchange

In the 21\textsuperscript{st} century, it turns out there are other languages!
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:

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

What's wrong now?

Arithmetic on chars?
$ 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
$ 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.
```
#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;
}
```
Q: Is the if statement really necessary?

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

```c
#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;
}
```
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.
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.

code:
```c
#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

![Diagram](https://via.placeholder.com/150)
```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?
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
```
```
#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;
}
```
Problem:
• The program works, but...
• Deeply nested statements
• No modularity

Solution:
• Handle each state in a separate function
#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.c                                 */
/* Author: Bob Dondero                      */
/*----------------------------------------*/

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

enum Statetype {NORMAL, INWORD};

Continued on
next slide
```c
/* 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;
}
```
/*-----------------------------------------------*/

/* 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;
}
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”
C requires variable declarations.

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

Examples:

```c
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
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

• 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);
```
Q: What is \( i \) set to in the following code?

\[ i = (1 \neq 2) + (3 > 4); \]

A. 0
B. 1
C. 2
D. 3
E. 4
Logical Data Type Dangers

Beware: the following code will cause loss of sleep

```java
...  
int i;  
...  
i = 0;  
...  
if (i = 5)  
    statement1;  
...  
```

What happens in Java?

What happens in C?
Appendix:

Additional DFA Examples
Does the string have “nano” in it?

- “banano” ⇒ yes
- “nnnnnnnанофф” ⇒ yes
- “banananonano” ⇒ yes
- “bananananashanana” ⇒ no

**Another DFA Example**

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”