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
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)

<table>
<thead>
<tr>
<th>char value</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>1</td>
<td>HT</td>
</tr>
<tr>
<td>2</td>
<td>LF</td>
</tr>
<tr>
<td>32</td>
<td>SP</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>;</td>
</tr>
<tr>
<td>59</td>
<td>&lt;</td>
</tr>
<tr>
<td>60</td>
<td>=</td>
</tr>
<tr>
<td>61</td>
<td>&gt;</td>
</tr>
<tr>
<td>62</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>126</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):

```
'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 21\textsuperscript{st} 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:

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

https://xkcd.com/1953/
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;
}
```

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

```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**
- Transitions labeled by chars or categories
- Optionally, actions on transitions

### DFA Example

- **States**
  - NORMAL
  - INWORD
- **Transitions**
  - isalpha (print uppercase equiv)
  - !isalpha (print)

21
```
#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;
}
```
Problem:
- The program works, but...
- States should have names

Solution:
- Define your own named constants

```c
enum Statetype {NORMAL, INWORD};
- Define an enumeration type
```
- Define a variable of that type

```c
enum Statetype state;
```
```c
#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?
```
Problem:
• The program works, but...
• Deeply nested statements
• No modularity

Solution:
• Handle each state in a separate function
```c
#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?
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)
```c
#include <stdio.h>
#include <ctype.h>

enum Statetype {NORMAL, INWORD};
```
/* 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”
Declaring Variables

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 */
```
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

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

**Illegal in C**

```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

• 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
Using integers to represent logical data permits shortcuts

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

It also permits some really bad code...

```c
i = (1 != 2) + (3 > 4);
```
Q: What is $i$ set to in the following code?

```c
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

```java
...  
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”