A Taste of C

Goals 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
  - E.g. Assignment 1, Assignment 7

Agenda

The charcount program
- The upper program
- The upper1 program

The “charcount” Program

Functionality:
- Read all chars from stdin (standard input stream)
- Write to stdout (standard output stream) the number of chars read

```
#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;
}
```

“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 > someotherfile
14
$

What is this? What is the effect?

“charcount” Building and Running

Question:
- Exactly what happens when you issue the command
gcc charcount.c -o charcount

Answer: Four steps
- Preprocess
- Compile
- Assemble
- Link

The starting point

charcount.c

```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 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

```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”

```
#include <stdio.h>

int main(void)
{
    printf("Welcome to calculate the number of chars in stdin. Return 0. ", 0);
    int charCount = 0;
    int c = getchar();
    while (c != -1)
        {
            charCount++;
            c = getchar();
        }
    printf("%d\n", charCount);
    return 0;
}
```

The result

```
... 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;
}
```

Why int instead of char?

- C language
- Missing comments
- Missing preprocessor directives
- Contains code from stdio.h
  - Declarations of getchar() and printf()
  - Missing definitions of getchar() and printf()

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()

```
... 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

```
charcount.s
```

- 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
- 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!

Running “charcount”

Command to run:
- ./charcount < somefile

Running “charcount”

Run-time trace, referencing the original C code...

ccharcount.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("\td\n", charCount);
  return 0;
}
```

Computer allocates 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...

```
#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;
}
```

• Computer calls getchar()
• getchar() tries to read char from stdin
  • Success ⇒ returns char (within an int)
  • Failure ⇒ returns EOF

EOF is a special non-char value that getchar() returns to indicate failure

Running “charcount”

Run-time trace, referencing the original C code...

```
#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, computer increments charCount

Running “charcount”

Run-time trace, referencing the original C code...

```
#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
• Computer breaks out of loop
• Computer calls printf() to write charCount

Running “charcount”

Run-time trace, referencing the original C code...

```
#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;
}
```

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

Other Ways to “charcount”

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

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

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

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

Which way is best?
Review of Example 1

Input/Output
- Including stdio.h
- Functions getchar() and printf()
- Representation of a character as an integer
- Predefined constant EOF

Program control flow
- The for and while statements
- The break statement
- The return statement

Operators
- Assignment: =
- Increment: ++
- Relational: == !=

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

stdin
Does this work?
It seems to work.

upper

stdout
DOES THIS WORK?
IT SEEMS TO WORK.

“upper” Building and Running

$ gcc217 upper.c -o upper
$ cat somefile
Does this work?
It seems to work.
$ ./upper < somefile
DOES THIS WORK?
IT SEEMS TO WORK.
$ 

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>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
<td>08</td>
<td>09</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
<tr>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
</tr>
</tbody>
</table>

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)
    {
        // ASCII lower case and upper case
        if (c >= 97) & (c <= 122)
        c -= 32;
        putchar(c);
    }
    return 0;
}
```
### EBCDIC

**Extended Binary Coded Decimal Interchange Code**

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 NUL</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>64 SP</td>
</tr>
<tr>
<td>80 A</td>
</tr>
<tr>
<td>96 - /</td>
</tr>
<tr>
<td>112</td>
</tr>
<tr>
<td>128 a b c d e f g h i</td>
</tr>
<tr>
<td>144 j k l m n o p q r</td>
</tr>
<tr>
<td>160 s t u v w x y z</td>
</tr>
<tr>
<td>176</td>
</tr>
<tr>
<td>192 A B C D E F G H I</td>
</tr>
<tr>
<td>208 J K L M N O P Q R</td>
</tr>
<tr>
<td>224 \ S T U V W X Y Z</td>
</tr>
<tr>
<td>240 0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

**Note:** Lower case not contiguous; same for upper case

---

### Character Literals

#### Examples

- `'a'` the a character
  - 97 on ASCII systems
  - 129 on EBCDIC systems
- `'\n'` newline
  - 10 on ASCII systems
  - 37 on EBCDIC systems
- `'\t'` horizontal tab
  - 9 on ASCII systems
  - 5 on EBCDIC systems
- `'\'` backslash
  - 92 on ASCII systems
  - 92 on EBCDIC systems
- `'\''` single quote
  - 39 on ASCII systems
  - 39 on EBCDIC systems
- `'\0'` the null character (alias NUL)
  - 0 on all systems

---

### “upper” Version 2

```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?

---

### ctype.h Functions

####Synopsis

```c
#include <ctype.h>
int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isgraph(int c);
int isdigit(int c);

These functions check whether c... falls into a certain character class...
```

---

### ctype.h Functions

####Synopsis

```c
#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” Final Version

```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;
}
```

Is the if statement really necessary?
Review of Example 2

Representing characters
- ASCII and EBCDIC character sets
- Character literals (e.g., 'A' or 'a')

Manipulating characters
- Arithmetic on characters
- Functions such as islower() and toupper()

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

```
stdin
  cos 217 rocks
  Does this work?
  It seems to work.
```

```
stdout
  upper1
  Does This Work?
  It Seems To Work.
```

“upper1” Building and Running

```
$ gcc217 upper1.c -o upper1
$ cat somefile
  cos 217 rocks
  Does this work?
  It seems to work.
$ ./upper1 < somefile
  Cos 217 Rocks
  Does This Work?
  It Seems To Work.
```

“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)

```
  NORMAL
  isalpha
    (print)
    (print uppercase equiv)
  isalpha
    (print)
  INWORD
  isalpha
    (print)
```

- **States**, one of which is denoted the **start** state
- **Transitions** labeled by chars or char 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

```
#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” 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;
}
```

“upper1” Toward Version 3

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);

int main(void)
{
    int c;
    enum Statetype state = NORMAL;
    while (c = getchar()) != EOF
    { switch (state)
        { case NORMAL:
            if (isalpha(c))
            { putstr(toupper(c)); state = INWORD; }
            else
            { putstr(c); state = NORMAL; }
            break;
        case INWORD:
            if (isalpha(c))
            { putstr(c); state = INWORD; }
            else
            { putstr(c); state = NORMAL; }
            break;
        }
    return 0;
}
```

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

“upper1” Version 3

```
#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))
            { putstr(toupper(c)); state = INWORD; }
            else
            { putstr(c); state = NORMAL; }
            return state;
        }
    return state;
}
```

“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 letters. Write the result to stdout. Return 0.

- Describes what the function does from caller’s viewpoint

```
/*---------------------------------------------*/
/* Author: Bob Benderso */
/*---------------------------------------------*/
#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 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;
}
```

```
/* 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;
  }
```

“upper1” Final Version

Continued on next page
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 and literals
- Reasonable max nesting depth
- Modular
- Multiple functions, each of which does one well-defined job
- Function-level comments
  - Should describe what function does
- See K&P book for style guidelines specification

Summary

The C programming language
- Overall program structure
- Control statements (if, while, for, and switch)
- Character I/O functions (getchar() and putchar())

Deterministic finite state automata (DFA)

Expectations for programming assignments
- Especially Assignment 1

Start Assignment 1 soon!

Appendix:

Additional DFA Examples

Another DFA Example

Does the string have “nano” in it?
- “banano” ⇒ yes
- “nnnnnnnanoff” ⇒ 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”