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

stdin

Line 1
Line 2

charcount

??

stdout
Q: What is the output of this program, on this input?

• A. 10
• B. 12
• C. 13
• D. 14
• E. 15
The "charcount" Program

The program:

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;
}
```
$ gcc217 charcount.c -o charcount
$ ./charcount
Line 1
Line 2
^D
14
$
$ cat somefile
Line 1
Line 2
$ ./charcount < somefile
14
$

What is this? What is the effect?
Building and Running

$ ./charcount > someotherfile
Line 1
Line 2
^D
$ cat someotherfile
14

What is this?
What is the effect?
“charcount” Building and Running in Detail

**Question:**
- Exactly what happens when you issue the command
  \[
  \text{gcc217 charcount.c -o charcount}
  \]

**Answer:** Four steps
- Preprocess
- Compile
- Assemble
- Link
Building and Running in Detail

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**
```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 != -1)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```
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`
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;
}
```

- 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()
Compiling “charcount”

charcount.i

```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;
}
```

- 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
Compiling “charcount”

The result:  charcount.s

```assembly
.section ".rodata"
format:
    .string "%d\n"
.section ".text"
.globl main
.type main,@function
main:
    pushq %rbp
    movq %rsp, %rbp
    subq $4, %rsp
    call getchar
loop:
    cmpl $-1, %eax
    je endloop
    incl -4(%rbp)
    call getchar
    jmp loop
endloop:
    movq $format, %rdi
    movl -4(%rbp), %esi
    movl $0, %eax
    call printf
    movl $0, %eax
    movq %rbp, %rsp
    popq %rbp
    ret
```

- 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
- 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
#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;
}
**Running “charcount”**

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

### 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;
}
```

- 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…

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;
}
```

Assuming `c ≠ EOF`, computer increments `charCount`
Running “charcount”

Run-time trace, referencing the original C code…

```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;
}
```

Computer calls getchar() again, and repeats
Running “charcount”

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

```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;
}
```

- 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...

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;
}
```

- Computer executes return statement
- Return from main() terminates program

Normal execution ⇒ return 0 or EXIT_SUCCESS
Abnormal execution ⇒ return EXIT_FAILURE
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: `==` `!=`
iClicker Question

Q: There are other ways to `charcount` – which is best?

A
```c
for (c=getchar(); c!=EOF; c=getchar())
    charCount++;
```

B
```c
while ((c=getchar())!= EOF)
    charCount++;
```

C
```c
for (;;)
{ c = getchar();
    if (c == EOF)
        break;
    charCount++;
}
```

D
```c
c = getchar();
while (c!=EOF)
{ charCount++;
    c = getchar();
}
```
Agenda

The charcount program
The upper program
The upper1 program
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.
$
```
American Standard Code for Information Interchange

Partial map

Note: Lower case and upper case letters are 32 apart
#include <stdio.h>
int main(void)
{
    int c;
    while ((c = getchar()) != EOF)
    {
        if ((c >= 97) && (c <= 122))
        {
            c -= 32;
            putchar(c);
        }
        return 0;
    }
}
Character Literals

Examples

'a'  the a character
      97 on ASCII systems

'\n'  newline
      10 on ASCII systems

'\t'  horizontal tab
      9 on ASCII systems

'\\'  backslash
      92 on ASCII systems

'\''  single quote
      39 on ASCII systems

'\0'  the null character (alias NUL)
      0 on all systems
```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;
}
```
## Extended Binary Coded Decimal Interchange Code

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
</tr>
<tr>
<td>16</td>
<td>HT</td>
</tr>
<tr>
<td>32</td>
<td>LF</td>
</tr>
<tr>
<td>64</td>
<td>SP</td>
</tr>
<tr>
<td>80</td>
<td>&amp;</td>
</tr>
<tr>
<td>96</td>
<td>/</td>
</tr>
<tr>
<td>112</td>
<td>\</td>
</tr>
<tr>
<td>128</td>
<td>a</td>
</tr>
<tr>
<td>144</td>
<td>j</td>
</tr>
<tr>
<td>160</td>
<td>~</td>
</tr>
<tr>
<td>192</td>
<td>A</td>
</tr>
<tr>
<td>208</td>
<td>J</td>
</tr>
<tr>
<td>224</td>
<td>\</td>
</tr>
<tr>
<td>240</td>
<td>0</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
   224 on EBCDIC systems

'\'' single quote
   39 on ASCII systems
   125 on EBCDIC systems

'\0' the null character (alias NUL)
   0 on all systems
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 a certain character class...
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.
```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;
}
```
Q: Is the `if` statement really necessary?

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

```c
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    int c;
    while (((c = getchar()) != EOF)
    {
        if (islower(c))
        {
            c = toupper(c);
            putchar(c);
        }
        putchar(c);
    }
    return 0;
}
```
$ 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.

```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;
}
```
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()
Agenda

The charcount program
The upper program
The upper1 program
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
\[cos\ 217\ \text{rocks}\]
\[\text{Does this work?}\]
\[\text{It seems to work.}\]

\text{upper1}

\text{stdout}
\[\text{Cos}\ 217\ \text{Rocks}\]
\[\text{Does This Work?}\]
\[\text{It Seems To Work.}\]
```
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)

- **States**, one of which is denoted the **start** state
- **Transitions** labeled by chars or char categories
- Optionally, **actions** on transitions
```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
```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 letters. 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 page
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;
}

Continued on next page
/*----------------------------------------------------------*/

/* 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;
}
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
Q: Convert binary 101010 into decimal and hex

A. 42 decimal, 2A hex
B. 48 decimal, 32 hex
C. 55 decimal, 3G hex
D. I know what this means, but I need a calculator…
E. Huh? Hex? Is this COS or witchcraft?
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
- “nnnnnnnanonoff” ⇒ yes
- “banananonano” ⇒ yes
- “banananananashanana” ⇒ 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”