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

The “charcount” Program

The program:

```
#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
", charCount);
  return 0;
}
```

“charcount” Building and Running

```
gcc217 charcount.c -o charcount
./charcount
```

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

1. cat somefile
   Line 1
   Line 2
   $ ./charcount < somefile
   14

What is this? What is the effect?

2. ./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
gcc217 charcount.c -o charcount

Answer: Four steps
1. Preprocess
2. Compile
3. Assemble
4. Link

Question:
• What is the effect?

(1) Preprocessing “charcount”

Command to preprocess:
• gcc217 -E charcount.c > charcount.i

Preprocessor functionality
• Removes comments
• Handles preprocessor directives

Preprocessor replaces
#include <stdio.h>
with contents of /usr/include/stdio.h
Preprocessor replaces
EOF with -1

(charcount)

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

Preprocessor removes comment

The result

```
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;
}
```

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()

(2) 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()

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
• So compiler has enough information to check subsequent calls of getchar() and printf()

The result: charcount.s

```
.section .rodata
.format .string "%d\n"
.section .text
.globl main
.type main,@function
main:
pushq %rbp
movq %rsp, %rbp
subq $4, %rsp
lea %r14, [%rbp-4]
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()

• Definition of main() function
• Compiler checks calls of getchar() and printf() when encountered
• Compiler translates to assembly language
(3) Assembling “charcount”

Command to assemble:
• gcc217 -c charcount.s

Assembler functionality
• Translate from assembly language to machine language

The result:
charcount.o
Machine language version of the program
No longer human readable

• Machine language
• Missing definitions of getchar() and printf()

(4) 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

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
", charCount);
    return 0;
}

charcount.c

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...

```c
#include <stdio.h>
/* Write to stdout the number of chars in stdin. Return 0. */
int main(void)
{
  int c;
  int charCount = 0;
  while (c != EOF)
  {
    charCount++;
    c = getchar();
  }
  printf("%d
", 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...

```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
", 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
", charCount);
  return 0;
}
```

• Eventually `getchar()` returns `EOF`
• Computer breaks out of loop
• Computer calls `printf()` to write `charCount`

Other Ways to “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
", charCount);
  return 0;
}
```

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

Which way is best?

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(); }`
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: == !=

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

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

“upper” Version 1

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

ASCII

American Standard Code for Information Interchange

Partial map

Note: Lower case and upper case letters are 32 apart

“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.
```

“upper” Version 1

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

What's wrong?
**EBCDIC**

Extended Binary Coded Decimal Interchange Code

Note: Lower case not contiguous; same for upper case

**Character Literals**

Examples

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>a character</td>
</tr>
<tr>
<td>'\n'</td>
<td>newline</td>
</tr>
<tr>
<td>'\t'</td>
<td>horizontal tab</td>
</tr>
<tr>
<td>'\r'</td>
<td>backslash</td>
</tr>
<tr>
<td>'\0'</td>
<td>null character (alias NUL)</td>
</tr>
</tbody>
</table>

**ctype.h Functions**

- isalnum(int c)
- isalpha(int c)
- isascii(int c)
- isblank(int c)
- iscntrl(int c)
- isdigit(int c)
- isgraph(int c)
- islower(int c)
- isprint(int c)
- ispunct(int c)
- isspace(int c)
- isupper(int c)
- isxdigit(int c)

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

**“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?

**“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

```
clang upper1.c -o upper1
cat somefile < upper1
```

“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
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

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

That’s a B+. 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” Final Version

```
#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:
                state = handleNormalState(c);
                break;
            case INWORD:
                state = handleInwordState(c);
                break;
        }
    }
    return 0;
}
```

Continued on next page

```
/* upper1.c                                                   */
/* Author: Bob Dondero                                        */
/*------------------------------------------------------------*/
#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:
                state = handleNormalState(c);
                break;
            case INWORD:
                state = handleInwordState(c);
                break;
        }
    }
    return 0;
}
```

Continued on next page

```
/* 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;
}
```

Continued on next page

```
/* 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
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
- "nnnnnnnanofff" ⇒ yes
- "bananananano" ⇒ 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"
- "+g9"
- "1e"
- "+"
- "17.9A"
- "0.38+"
- "-"