A Taste of C

COS 217 Spring 2015
Lecture 2
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 env 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

14

stdout
The “charcount” Program

The program:

```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;
}
```

What is this? What is the effect?

```
$ gcc217 charcount.c -o charcount
$ charcount
Line 1
Line 2
^D
14
$  
```
What is this? What is the effect?

```
$ cat somefile
Line 1
Line 2
$ charcount < somefile
14
$
```
```
$ charcount > someotherfile
Line 1
Line 2
^D
$ cat someotherfile
14
```

What is this? What is the effect?
The “charcount” Program

The program:

ccharcount.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 Directive

Comment

Functions: getchar, printf
“charcount” Building Steps

Question:
- Exactly what happens when you issue the command
  \texttt{gcc217 charcount.c -o charcount}

Answer: Four steps
1. Preprocess: removes comments, handles preprocessor directives
2. Compile: translates to assembly language, matches function calls with declarations
3. Assemble: translates to machine language
4. Link: resolves references, fetches functions from libraries

charcount.c charcount.i charcount.s charcount.o charcount

More details in Appendix
Running “charcount”

Command to run:

- charcount < somefile
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 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…

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
#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() again, and repeats
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...

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 stmt
- Return from main() terminates program

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: *== !==*
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 → upper → stdout

Does this work? It seems to work.

DOES THIS WORK? IT SEEMS TO WORK.
Does this work?
It seems to work.

$ upper < somefile
DOES THIS WORK?
IT SEEMS TO WORK.
$
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>NUL</td>
<td>HT</td>
<td>LF</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>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>
<td>/</td>
</tr>
<tr>
<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>
<td>?</td>
</tr>
<tr>
<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>
<td>O</td>
</tr>
<tr>
<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>
<td>_</td>
</tr>
<tr>
<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>
<td>o</td>
</tr>
<tr>
<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>
<td>~</td>
</tr>
</tbody>
</table>

Partial map

Note: Lower case and upper case letters are 32 apart
**Extended Binary Coded Decimal Interchange Code**

<table>
<thead>
<tr>
<th></th>
<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></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>
<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>
<td></td>
</tr>
<tr>
<td>32</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>
<td></td>
</tr>
<tr>
<td>48</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>
<td></td>
</tr>
<tr>
<td>64</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.</td>
<td>&lt;</td>
<td>(</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>&amp;</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>96</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>&gt;</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</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>
<td></td>
</tr>
<tr>
<td>128</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>~</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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>\</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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Lower case not contiguous; same for upper case
```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?
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
#include <stdio.h>
int main(void)
{
    int c;
    while ((c = getchar()) != EOF)
    {
        if ((c >= 'a') && (c <= 'z'))
        {
            c += 'A' - 'a';
            putchar(c);
        }
        putchar(c);
    }
    return 0;
}
$ man islower

NAME

isanum, 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.
#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()
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 rocks
Does this work?
It seems to work.
```

upper1

```
Cos 217 Rocks
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 Autonmaton

Deterministic Finite State Automaton (DFA)

isalpha
(print uppercase equiv)

State 0

State 1

! isalpha
(print)

! isalpha
(print)

• **States**, one of which is denoted the **start** state
• **Transitions** labeled by chars or char categories
• Optionally, **actions** on transitions (not part of DFA)
Implementation Skeleton

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

int main (void) {
    int c;
    while ((c = getchar()) != EOF) {
        // <process one character>
    }
    return 0;
}
```
Implementation

```c
<process one character> =
switch (state) {
    case 0:
        <state 0 action>
        break;
    case 1:
        <state 1 action>
        break;
    default:
        <this should never happen>
}
```

```
Implementation

if (isalpha(c)) {
    putchar(toupper(c));
    state = 1;
} else {
    putchar(c); state = 0;
}
```
```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 …
- Mysterious integer constants (“magic numbers”)
- Instead, 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?
“upper1” Toward Version 3

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};

denum Statetype handleNormalState(int c) {
    enum Statetype state;
    if (isalpha(c)) {
        putchar(toupper(c));
        state = INWORD;
    } else {
        putchar(c);
        state = NORMAL;
    }
    return state;
}

denum 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};
/* 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. */

def handleNormalState(c):
  state = NORMAL
  if isalpha(c):
    put(toupper(c))
    state = INWORD
  else:
    put(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;
}
/*----------------------------------------------------------*/
/* 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
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
- “nnnnnnnanonoff” => 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”
Appendix:
Building “charcount” 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()
Step 1: Preprocessing “charcount”

Command to preprocess:

- `gcc217 -E charcount.c > charcount.i`

Preprocessor functionality

- Removes comments
- Handles **preprocessor directives**
Step 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 != EOF)
    {
        charCount++;
        c = getchar();
    }
    printf("%d\n", charCount);
    return 0;
}
```

Preprocessor replaces `#include <stdio.h>` with contents of /
/usr/include/stdio.h
Step 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 != EOF)
  {
    charCount++;
    c = getchar();
  }
  printf("%d\n", charCount);
  return 0;
}
```

Preprocessor removes comment
Step 1: Preprocessing “charcount”

The result

```
... int getchar();
int printf(char *fmt, ...);
...
int main(void)
{
  int c;
  int charCount = 0;
  c = getchar();
  while (c != EOF)
  {
    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()
Step 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()
Step 2: Compiling “charcount”

charcount.i

...  
int getchar();
int printf(char *fmt, ...);
...  
int main(void)
{  int c;
   int charCount = 0;
   c = getchar();
   while (c != EOF)
   {  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()
Step 2: Compiling “charcount”

charcount.i

... int getchar(); int printf(char *fmt, ...); ...
int main(void)
{ int c;
  int charCount = 0;
  c = getchar();
  while (c != EOF)
  { charCount++;
    c = getchar();
  }
  printf("%d\n", charCount);
  return 0;
}
Step 2: Compiling “charcount”

The result: `charcount.s`

```
.section ".rodata"
format:
 .string "\d\n"
.section ".text"
.globl main
.type main,@function
main:
 pushl %ebp
 movl %esp, %ebp
 subl $4, %esp
 call getchar
loop:
 cmpl $-1, %eax
 je endloop
 incl -4(%ebp)
 call getchar
 jmp loop
endloop:
 pushl -4(%ebp)
pushl $format
 call printf
 addl $8, %esp
 movl $0, %eax
 movl %ebp, %esp
 popl %ebp
ret
```

- Assembly language
- Missing definitions of `getchar()` and `printf()`
Step 3: Assembling “charcount”

Command to assemble:
  • gcc217 –c charcount.s

Assembler functionality
  • Translate from assembly language to machine language
Assembling “charcount” (Step 3)

The result:

- charcount.o

- Machine language version of the program
- No longer human readable

- Machine language
- Missing definitions of getchar() and printf()
Step 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
Step 4: Linking “charcount”

The result:

charcount

- Machine language
- Contains definitions of getchar() and printf()

Complete! Executable!