COS 217: Introduction to Programming Systems

A Taste of C
Agenda

Simple C Programs
  • charcount (loops, standard input)
    • 4-stage build process
  • upper (character data, ctype library)
    • portability concerns

Source code control with git
Agenda

Simple C Programs

- **charcount** (loops, standard input)
  - 4-stage build process
- **upper** (character data, ctype library)
  - portability concerns

Source code control with **git**
The “charcount” Program

Functionality:
• Read all characters from standard input stream
• Write to standard output stream the number of characters read
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;
}
```

charcount.c
“charcount” Building and Running

```
$ gcc217 charcount.c
$ ls
.  ..  a.out
$ gcc217 charcount.c -o charcount
$ ls
.  ..  a.out  charcount
$ 
```
“charcount” Building and Running

```
$ gcc217 charcount.c -o charcount
$ ./charcount
Line 1
Line 2
^D
```

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

$ gcc217 charcount.c -o charcount
$ ./charcount

Line 1
Line 2
^D
14
$

Includes visible characters plus two newlines
**“charcount” Building and Running**

```bash
$ 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?
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;
}
```

Execution begins at

- **main()** function

- No classes in the C language.
"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;
}
```

We allocate 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;
}
```

cvarchar() tries to read char from stdin
- Success ⇒ returns that char value (within an int)
- Failure ⇒ returns EOF

**EOF** is a special value, distinct from all possible chars
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 \( c \neq EOF \), we increment 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;
}
```

We call getchar() again and recheck loop condition
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;
}
```

- Eventually `getchar()` returns EOF
- Loop condition fails
- We call `printf()` to write final `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;
}
```

- return statement returns to calling function
- return from main() terminates program

Normal execution ⇒ 0 or EXIT_SUCCESS
Abnormal execution ⇒ EXIT_FAILURE
“charcount” Build Process in Detail

Question:
• Exactly what happens when you issue the command
  gcc217 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
chrs 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 declarations of `getchar()` and `printf()`
• 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”

Preprocessor removes comment (this is A1!)

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
#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”

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;
}

- C language
- Missing comments
- Missing preprocessor directives
- Contains code from stdio.h: declarations of getchar() and printf()
- Missing definitions of getchar() and printf()
- Contains value for EOF
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

```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;
}
```

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

The result: `charcount.s`

```
.section .rodata
.LC0:
.string "%d\n"

.section .text
.global main
main:
  stp  x29, x30, [sp, -32]!
  add  x29, sp, 0
  str  wzr, [x29,24]
  bl   getchar
  str  w0, [x29,28]
  b    .L2
.L3:  
  ldr  w0, [x29,24]
  add  w0, w0, 1
  str  w0, [x29,24]
  bl   getchar
  str  w0, [x29,28]
.L2:  
  ldr  w0, [x29,28]
  cmn  w0, #1
  bne  .L3
  adrp x0, .LC0
  add  x0, x0, :lo12:.LC0
  ldr  w1, [x29,24]
  bl   printf
  mov  w0, 0
  ldp  x29, x30, [sp], 32
  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 within the code
  • Fetch machine language code from the standard C library (/usr/lib/libc.a) to make the program complete
Linking “charcount”

The result:

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

Complete! Executable!
Q: There are other ways to `charCount` - which is best?

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

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

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

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

Simple C Programs
• charcount (loops, standard input)
  • 4-stage build process
• upper (character data, ctype library)
  • portability concerns

Source code control with git
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

```
Does this work?
It seems to work.
```

```
upper
```

```
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>0</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>
</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>
</tr>
<tr>
<td>32</td>
<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>
</tr>
<tr>
<td>48</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>9</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
</tr>
<tr>
<td>64</td>
<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>
</tr>
<tr>
<td>80</td>
<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>
</tr>
<tr>
<td>96</td>
<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>
</tr>
<tr>
<td>112</td>
<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>
</tr>
</tbody>
</table>

**Partial map**

Note: Lower-case and upper-case letters are 32 apart.
```c
#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

37
```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 now? Arithmetic on chars?
$ 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.
```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 (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);
        }
        putchar(c);
    }
    return 0;
}
```
Agenda

Simple C Programs

- charcount (loops, standard input)
  - 4-stage build process
- upper (character data, ctype library)
  - portability concerns

Source code control with **git**
Problems often faced by programmers:

• How do I work with source code on multiple computers?
• How do I work with others (e.g. a COS 217 partner) on the same program?
• What changes did my partner just make?
• If my partner and I make changes to different parts of a program, how do we merge those changes?
• How can I try out one way of writing this function, and go back if it doesn’t work?
• Help! I’ve deleted my code! How do I get it back?
• Help! I’ve introduced a subtle bug that I can’t find. How can I see what I’ve changed since the last working version?

All of these problems solved by specialized tools, such as git
**WORKING COPY**

- Represents single version of the code
- Plain files (e.g., .c)
- Make a coherent set of modifications, then commit this version of code to the repository
- Best practice: write a meaningful commit message

**REPOSITORY**

- Contains all checked-in versions of the code
- Specialized format, located in .git directory
- Can view commit history
- Can diff any versions
- Can check out any version, by default the most recent (known as HEAD)
Local vs. Remote Repositories

LOCAL REPO

- Located in .git directory
- Only accessible from the current computer
- Commit early, commit often – you can only go back to versions you’ve committed
- Can push current state (i.e., complete checked-in history) to a remote repository

REMOTE REPO

- Located in the cloud, e.g. github.com
- Can clone to multiple machines
- Any clone can pull the current state
We distribute assignment code through a github.com repo

- But you can’t push to our repo!

Need to create your own (private!) repo for each assignment

- Two methods in git primer handout
- One clone on armlab, to test and submit
- If developing on your own machine, another clone there: be sure to commit and push to github, then pull on armlab