Overview of Today’s Lecture

• Goals of the lecture
  ◦ Important C constructs
    – Program flow (if/else, loops, and switch)
    – Character input/output (getchar and putchar)
  ◦ Deterministic finite automata (i.e., state machine)
  ◦ Expectations for programming assignments

• C programming examples
  ◦ Echo the input directly to the output
  ◦ Put all lower-case letters in upper case
  ◦ Put the first letter of each word in upper case

• Glossing over some details related to “pointers”
  ◦ … which will be covered in the next lecture
Echo Input Directly to Output

- **Including the Standard Input/Output (stdio) library**
  - Makes names of functions, variables, and macros available
  - `#include <stdio.h>`

- **Defining procedure main()**
  - Starting point of the program, a standard boilerplate
  - `int main(void)`
  - `int main(int argc, char **argv)`
  - Hand-waving: `argc` and `argv` are for input arguments

- **Read a single character**
  - Returns a single character from the text stream “standard in” (stdin)
  - `c = getchar();`

- **Write a single character**
  - Writes a single character to “standard out” (stdout)
  - `putchar(c);`
#include <stdio.h>

int main(void) { 
    int c;
    c = getchar();
    putchar(c);
    return 0;
}
Why is the Character an “int”

• Meaning of a data type
  ○ Determines the size of a variable
  ○ … and how it is interpreted and manipulated

• Difference between char and int
  ○ char: character, a single byte
  ○ int: integer, machine-dependent (e.g., -32,768 to 32,767)

• One byte is just not big enough
  ○ Need to be able to store any character
  ○ … plus, special value like End-Of-File (typically “-1”)
  ○ We’ll see an example with EOF in a few slides
Read and Write Ten Characters

- Loop to repeat a set of lines (e.g., \textbf{for} loop)
  - Three arguments: initialization, condition, and re-initialization
  - E.g., start at 0, test for less than 10, and increment per iteration

\begin{verbatim}
#include <stdio.h>

int main(void) {
    int c, i;

    for (i=0; i<10; i++) {
        c = getchar();
        putchar(c);
    }

    return 0;
}
\end{verbatim}
Read and Write Forever

- Infinite `for` loop
  - Simply leave the arguments blank
  - E.g., `for ( ; ; )`

```c
#include <stdio.h>
int main(void) {
    int c;

    for ( ; ; ) {
        c = getchar();
        putchar(c);
    }

    return 0;
}
```
Read and Write Till End-Of-File

- Test for end-of-file (EOF)
  - EOF is a special global constant, defined in stdio
  - The break statement jumps out of the current scope

```c
#include <stdio.h>
int main(void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF)
            break;
        putchar(c);
    }
    return 0;
}
```
Many Ways to Say the Same Thing

```c
for (c=getchar(); c!=EOF; c=getchar())
    putchar(c);
```

```c
while ((c=getchar())!=EOF)
    putchar(c);
```

```c
for (;;) {
    c = getchar();
    if (c == EOF)
        break;
    putchar(c);
}
```

```c
c = getchar();
while (c!=EOF) {
    putchar(c);
    c = getchar();
}
```

Very typical idiom in C, but messy side-effects in loop test
Review of Example #1

• Character I/O
  ◦ Including stdio.h
  ◦ Functions getchar() and putchar()
  ◦ Representation of a character as an integer
  ◦ Predefined constant EOF

• Program control flow
  ◦ The for loop and while loop
  ◦ The break statement
  ◦ The return statement

• Assignment and comparison
  ◦ Assignment: “=”
  ◦ Increment: “i++”
  ◦ Comparing for equality “==”
  ◦ Comparing for inequality “!=”
Example #2: Convert Upper Case

• Problem: write a program to convert a file to all upper-case (leave nonalphabetic characters alone)

• Program design:

  repeat
  
  read a character
  
  if it’s lower-case, convert to upper-case
  
  write the character

  until end-of-file
# ASCII

### American Standard Code for Information Interchange

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<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>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
</tr>
<tr>
<td>16</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</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>
</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>

**Lower case: 97-122 and upper case: 65-90**

E.g., ‘a’ is 97 and ‘A’ is 65 (i.e., 32 apart)
#include <stdio.h>

int main(void) {
    int c;
    for (; ; ) {
        c = getchar();
        if (c == EOF) break;
        if ((c >= 97) && (c < 123))
            c -= 32;
        putchar(c);
    }
    return 0;
}
That’s a B-minus

• Programming well means programs that are
  ◦ Clean
  ◦ Readable
  ◦ Maintainable

• It’s not enough that your program works!
  ◦ We take this seriously in COS 217.
Correct, but ugly to have all these hard-wired constants in the program.
#include <stdio.h>

int main(void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        if ((c >= 'a') && (c <= 'z'))
            c += 'A' - 'a';
        putchar(c);
    }
    return 0;
}
Standard C Library Functions

NAME
cctype, isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, isascii - character handling

SYNOPSIS
#include <ctype.h>
int isalpha(int c);
int isupper(int c);
int islower(int c);
int isdigit(int c);
int isalnum(int c);
int isspace(int c);
int ispunct(int c);
int isprint(int c);
int isgraph(int c);
int iscntrl(int c);
int toupper(int c);
int tolower(int c);

DESCRIPTION

These macros classify character-coded integer values. Each is a predicate returning non-zero for true, 0 for false...

The toupper() function has as a domain a type int, the value of which is representable as an unsigned char or the value of EOF.... If the argument of toupper() represents a lower-case letter ... the result is the corresponding upper-case letter. All other arguments in the domain are returned unchanged.
#include <stdio.h>
#include <ctype.h>

int main(void) {
    int c;
    for (; ; ) {
        c = getchar();
        if (c == EOF) break;
        if (islower(c))
            c = toupper(c);
        putchar(c);
    }
    return 0;
}
Compiling and Running

% ls
get-upper.c
% gcc get-upper.c
% ls
a.out get-upper.c
% a.out
We’ll be on time today!
WE’LL BE ON TIME TODAY!
^D
%
% a.out < get-upper.c

#include <stdio.h>
#include <ctype.h>
int main(void) {
    int c;
    for (;;) {
        c = getchar();
        if (c == EOF) break;
        if (islower(c))
            c = toupper(c);
        putchar(c);
    }
    return 0;
}
Output Redirection

```c
% a.out < get-upper.c > test.c
% gcc test.c
```
Review of Example #2

- **Representing characters**
  - ASCII character set
  - Character constants (e.g., ‘A’ or ‘a’)

- **Manipulating characters**
  - Arithmetic on characters
  - Functions like `islower()` and `toupper()`

- **Compiling and running C code**
  - Compile to generate a.out
  - Invoke a.out to run program
  - Can redirect stdin and/or stdout
Example #3: Capitalize First Letter

Deterministic Finite Automaton (DFA)

State #1: before the 1\textsuperscript{st} letter of a word
State #2: after the 1\textsuperscript{st} letter of a word
Capitalize on transition from state 1 to 2

“cos 217 rocks” $\rightarrow$ “Cos 217 Rocks”
Implementation Skeleton

```c
#include <stdio.h>
#include <ctype.h>
int main (void) {
    int c;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        <process one character>
    }
    return 0;
}
```
Implementation

<process one character> =

switch (state) {
    case 1:
        <state 1 action>
        break;
    case 2:
        <state 2 action>
        break;
    default:
        <this should never happen>
}

if (isalpha(c)) {
    putchar(toupper(c));
    state = 2;
} else putchar(c);

if (!isalpha(c))
    state = 1;
putchar(c);
```c
#include <stdio.h>
#include <ctype.h>

int main(void) {
    int c; int state=1;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        switch (state) {
        case 1:
            if (isalpha(c)) {
                putchar(toupper(c));
                state = 2;
            } else putchar(c);
            break;
        case 2:
            if (!isalpha(c)) state = 1;
            putchar(c);
            break;
        }
    }
    return 0;
}
```
% gcc upper1.c
% a.out < upper1.c
#include <stdio.h>
#include <ctype.h>
int main(void) {
    int c; int state = 1;
    for (; ; ) {
        c = getchar();
        if (c == EOF) break;
        switch (state) {
        case 1:
            if (isalpha(c)) {
                putchar toupper(c);
                state = 2;
            } else putchar(c);
            break;
        case 2:
            if (!isalpha(c)) state = 1;
            putchar(c);
            break;
        }
    }
    return 0;
}
OK, That’s a B+

• Works correctly, but
  ◦ Mysterious integer constants (“magic numbers”)
  ◦ No modularization
  ◦ No checking for states besides 1 and 2

• What now?
  ◦ States should have names, not just 1,2
  ◦ Should handle each state in a separate function
  ◦ Good to check for unexpected variable value
Improvement: Names for States

• Define your own named constants
  ◦ Enumeration of a list of items
  ◦ `enum Statetype {NORMAL, INWORD};`
  ◦ Declare a variable of that type
  ◦ `enum Statetype state;`
#include <stdio.h>
#include <ctype.h>

enum Statetype {NORMAL, INWORD};

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

```c
#include <stdio.h>
#include <ctype.h>
enum Statetype {NORMAL, INWORD};
enum Statetype handleNormalState(int c) {...}
enum Statetype handleInwordState(int c) {...}

int main(void) {
    int c;
    enum Statetype state = NORMAL;
    for ( ; ; ) {
        c = getchar();
        if (c == EOF) break;
        switch (state) {
            case NORMAL:
                state = handleNormalState(c);
                break;
            case INWORD:
                state = handleInwordState(c);
                break;
        }
    }
    return 0;
}
```
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;
    putchar(c);
    if (!isalpha(c))
        state = NORMAL;
    else
        state = INWORD;
    return state;
}
Improvement: Defensive Programming

• Assertion checks for diagnostics
  ○ Check that an expected assumption holds
  ○ Print message to standard error (stderr) when expression is false
  ○ E.g., `assert(expression);`
  ○ Makes program easier to read, and to debug

```c
switch (state) {
    case NORMAL:
        ...
        break;
    case INWORD:
        ...
        break;
    default:
        assert(0);
}
```

Should never, ever get here.
Putting it Together: An “A” Effort

```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;
    putchar(c);
    if (!isalpha(c))
        state = NORMAL;
    else
        state = INWORD;
    return state;
}
int main(void) {
    int c;
    enum Statetype state = NORMAL;
    for (; ; ) {
        c = getchar();
        if (c == EOF) break;
        switch (state) {
        case NORMAL:
            state = handleNormalState(c);
            break;
        case INWORD:
            state = handleInwordState(c);
            break;
        }
    }
    return 0;
}
Review of Example #3

• Deterministic Finite Automaton
  ◦ Two or more states
  ◦ Actions in each state, or during transition
  ◦ Conditions for transitioning between states

• Expectations for COS 217 assignments
  ◦ Modularity (breaking into distinct functions)
  ◦ Readability (meaningful names for variables and values)
  ◦ Diagnostics (assertion checks to catch mistakes)
  ◦ See K&P book for style guidelines specification