Character Input/Output in C

COS 217

http://www.cs.princeton.edu/courses/archive/fall05/cos217/

Precepts and Office Hours

- Four precept sections (assignments sent via e-mail)
  - MW 1:30-2:20, Friend Center 009
  - TTh 12:30-1:20 Computer Science Building 102
  - TTh 1:30-2:20 Friend Center 108
  - TTh 4:30-5:20 Computer Science Building 102
- Office hours
  - Jennifer Rexford, Computer Science Building 306
    - T 11:00-11:50, Th 9:00-9:50, or by appointment
  - Bob Dondero, Computer Science Building 206
    - TTh 2:30-3:20, TTh 3:30-4:20, or by appointment
  - Chris DeCoro
    - TW 1:30-2:20, or by appointment

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(int argc, char **argv)`**
  - Hand-waving #1: **`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);`**

Putting it All Together

```
#include <stdio.h>

int main(int argc, char **argv) {
  int c;

  c = getchar();
  putchar(c);

  return 0;
}
```

Why an “int”?

Why a return value?

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., **`for`** loop)
  - Three arguments: initialization, condition, and re-initialization
  - E.g., start at 0, test for less than 10, and increment per iteration

```
#include <stdio.h>

int main(int argc, char **argv) {
   int c, i;

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

   return 0;
}
```

Read and Write Forever

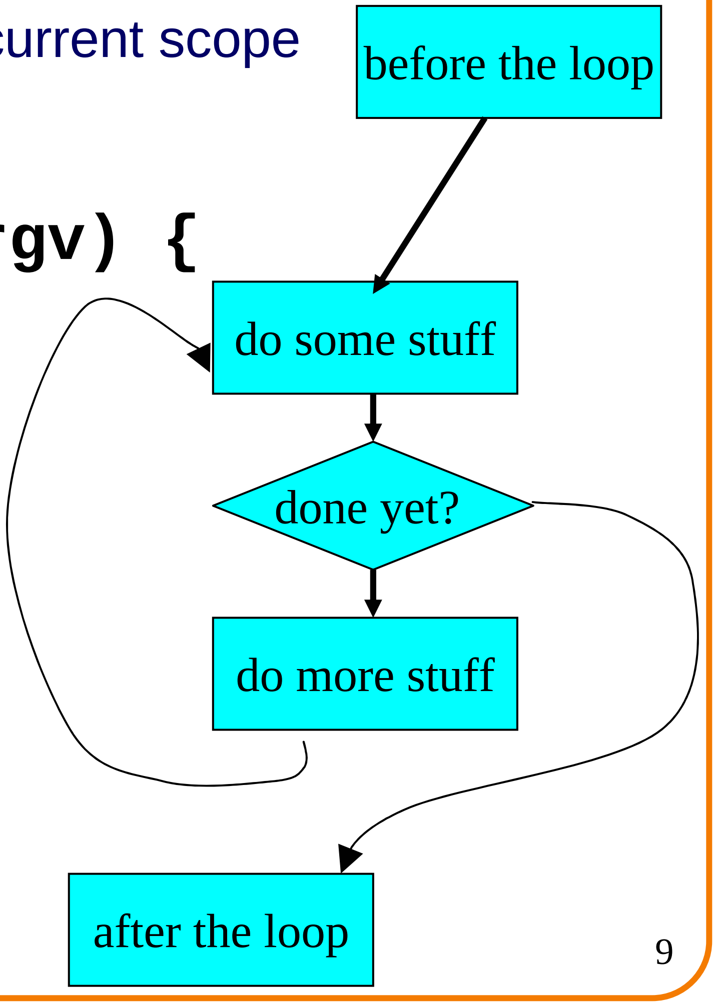
- Infinite **`for`** loop
  - Simply leave the arguments blank
  - E.g., **`for ( ; ; )`**

```
#include <stdio.h>
int main(int argc, char **argv) {
   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

```
#include <stdio.h>
int main(int argc, char **argv) {
   int c;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF)
        break;
      putchar(c);
   }
   return 0;
}
```



Many Ways to Say the Same Thing

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

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

Very typical idiom in C, but it's messy to have side effects in loop test

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

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

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

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

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

E.g., ‘a’ is 97 and ‘A’ is 65 (i.e., 32 apart)

Implementation in C

```
#include <stdio.h>
int main(int argc, char **argv) {
   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.

Avoid Mysterious Numbers

```
#include <stdio.h>
int main(int argc, char **argv) {
   int c;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF) break;
      if ((c >= 97) && (c < 123))
           c -= 32;
      putchar(c);
   }
   return 0;
}
```

Correct, but ugly to have all these hard-wired constants in the program.

Improvement: Character Literals

```
#include <stdio.h>
int main(int argc, char **argv) {
   int c;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF) break;
      if ((c >= 'a') && (c <= 'z'))
         c += 'A' - 'a';
      putchar(c);
   }
   return 0;
}
```

Improvement: Existing Libraries

Standard C Library Functions ctype(3C)

NAME

ctype, 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.

Using the ctype Library

```
#include <stdio.h>
#include <ctype.h>
int main(int argc, char **argv) {
   int c;
   for ( ; ; ) {
      c = getchar();
      if (c == EOF) break;
      if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

Returns 1 (true) if c is between ‘a’ and ‘z’

Compiling and Running

```
% ls
get-upper.c
% gcc get-upper.c
% ls
a.out get-upper.c
% a.out
We have Air Conditioning Today!
WE HAVE AIR CONDITIONING TODAY!
^D
%
```

Run the Code on Itself

```
% a.out < get-upper.c
#INCLUDE <STDIO.H>
#INCLUDE <CTYPE.H>
INT MAIN(INT ARGC, CHAR **ARGV) {
   INT C;
   FOR ( ; ; ) {
      C = GETCHAR();
      IF (C == EOF) BREAK;
      IF (ISLOWER(C))
            C = TOUPPER(C);
       PUTCHAR(C);
   }
   RETURN 0;
}
```

Output Redirection

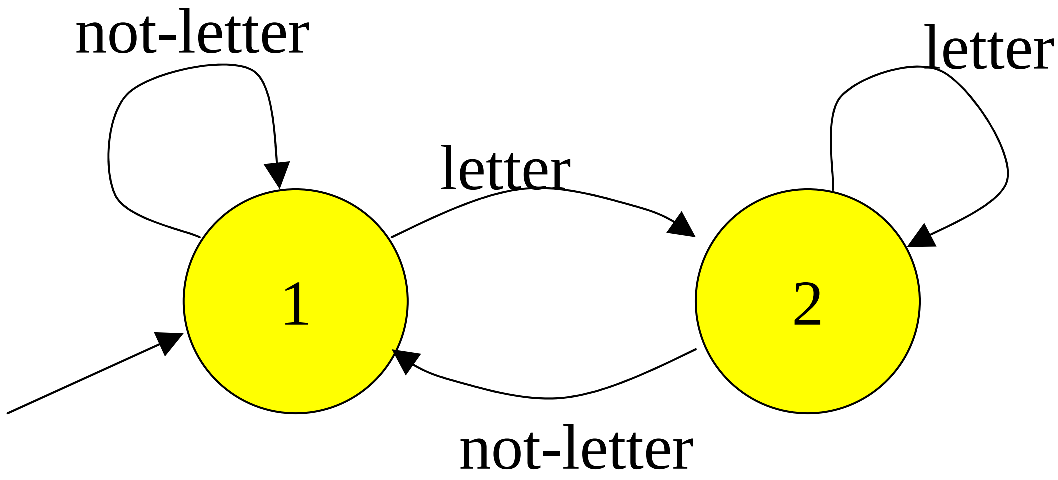
```
% a.out < get-upper.c  > test.c

% gcc test.c
test.c:1:2: invalid preprocessing directive #INCLUDE
test.c:2:2: invalid preprocessing directive #INCLUDE
test.c:3: syntax error before "MAIN"
test.c:3: syntax error before "ARGC"
etc...
```

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 1st letter of a word

State #2: after the 1st letter of a word

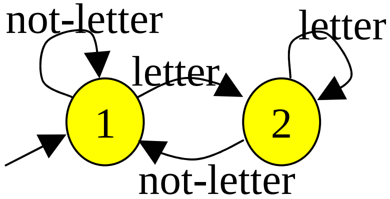
Capitalize on transition from state 1 to 2

“air conditioning rocks” → “Air Conditioning Rocks”

Implementation Skeleton

```
#include <stdio.h>
#include <ctype.h>
int main (int argc, char **argv) {
   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);
```

Complete Implementation

```
#include <stdio.h>
#include <ctype.h>
int main(int argc, char **argv) {
  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;
}
```

Running Code on Itself

```
% gcc upper1.c

% a.out < upper1.c
#Include <Stdio.H>
#Include <Ctype.H>
Int Main(Int Argc, Char **Argv) {
  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
  - No modularization
  - Mysterious integer constants
  - No checking for states besides 1 and 2
- What now?
  - <*process one character*> should be a function!
  - States should have names, not just 1,2
  - Good to check for unexpected variable value

Improvement: Modularity

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

void process_one_character(char c) {
  …
}

int main(int argc, char **argv) {
  int c;

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

Improvement: Names for States

- Define your own named constants
  - Enumeration of a list of items
  - **`enum statetype {NORMAL,INWORD};`**

```
void process_one_character(char c) {
  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;
  }
}
```

Problem: Persistent “state”

- State variable spans multiple function calls
  - Variable **`state`** should start as **`NORMAL`**
  - Value of **`state`** should persist across successive function calls
  - But, all C functions are “call by value”
  - Hand-waving #2: make **`state`** a global variable (for now)

```
enum statetype {NORMAL, INWORD};
enum statetype state = NORMAL;

void process_one_character(char c) {
  extern enum statetype state;
  switch (state) {
    case NORMAL:
      …
    case INWORD:
      …
  }
}
```

Declaration optional if the variable is defined earlier in the file.

Improvement: Defensive Programming

- Assertion checks for diagnostics
  - Check that 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

```
void process_one_character(char c) {
  switch (state) {
    case NORMAL:
      …
      break;
    case INWORD:
      …
     break;
    default:
     assert(0);
  }
}
```

Should never, ever get here.

Putting it Together: An “A” Effort

```
enum statetype {NORMAL, INWORD};

enum statetype state = NORMAL;

void process_one_character(char c) {
  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;
   default: assert(0);
  }
}
```

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

void process_one_character(char);

int main(int argc, char **argv) {
  int c;
  for ( ; ; ) {
    c = getchar();
    if (c == EOF) break;
    process_one_character(c);
  }
}
```

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 in to distinct functions)
  - Readability (meaningful names for variables and values)
  - Diagnostics (assertion checks to catch mistakes)
- Note: some vigorous hand-waving in today's lecture
  - E.g., use of global variables (okay for assignment #1)
  - Next lecture will introduce pointers

Precepts and Office Hours

- Four precept sections (assignments sent via e-mail)
  - MW 1:30-2:20, Friend Center 009
  - TTh 12:30-1:20 Computer Science Building 102
  - TTh 1:30-2:20 Friend Center 108
  - TTh 4:30-5:20 Computer Science Building 102
- Office hours
  - Jennifer Rexford, Computer Science Building 306
    - T 11:00-11:50, Th 9:00-9:50, or by appointment
  - Bob Dondero, Computer Science Building 206
    - TTh 2:30-3:20, TTh 3:30-4:20, or by appointment
  - Chris DeCoro
    - TW 1:30-2:20, or by appointment