COS 217: Introduction to Programming Systems

Simple Programs (continued)

PRINCETON UNIVERSITY

From Lecture 2: Tracing upper: The Exit

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

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

Normal execution ⇒ 0 or **EXIT_SUCCESS**
Abnormal execution ⇒ **EXIT_FAILURE**
#include <stdlib.h> to use these constants

From Lecture 2: Tracing upper: The End Game

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- Eventually getchar() returns EOF
- Loop condition fails
- We exit the loop, having output what we needed to output

Agenda

A more complex character processing program: Upper1. Why?

- Assignment 1
- Design step more involved: designing the simple DFA model
- Coding Step: develop a C program to implement the DFA

Building an executable C program

Next time: Separating Interface and Implementation, using character processing as an example

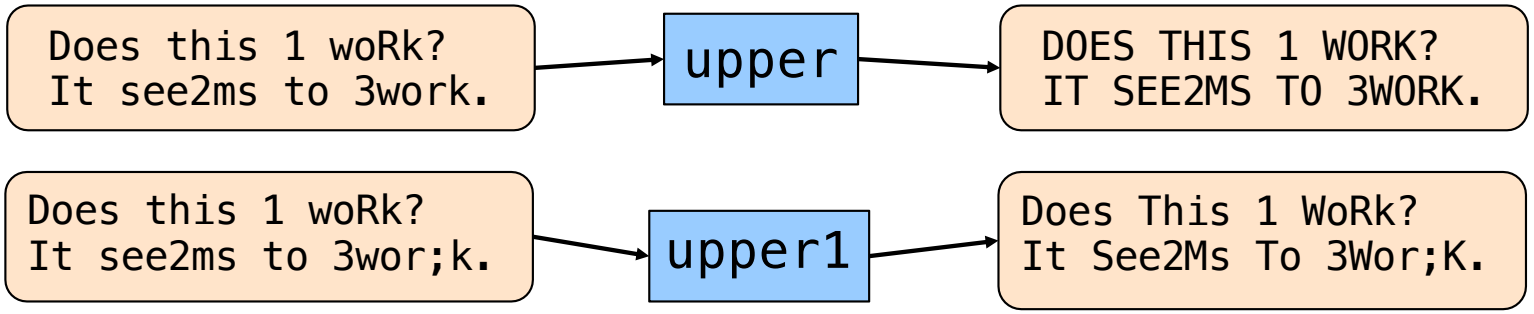
upper1 Program Specification

Read all chars from stdin

Capitalize the first letter of each 'word:'

- series of alphabetical characters terminated by non-alphabetical characters

Write result to stdout



Key new element: don't capitalize letters inside a word (see weird strings above)

What's the key new element we have to design?

upper1 Program Design

Key is to recognize when we're *"inside a word"* vs *"**not** inside a word"*

- if in a word, don't capitalize (until we have left the word)
- if not in word, capitalize first next time we see an alphabetical char

Where we are depends on where we just were (after we read the previous character)

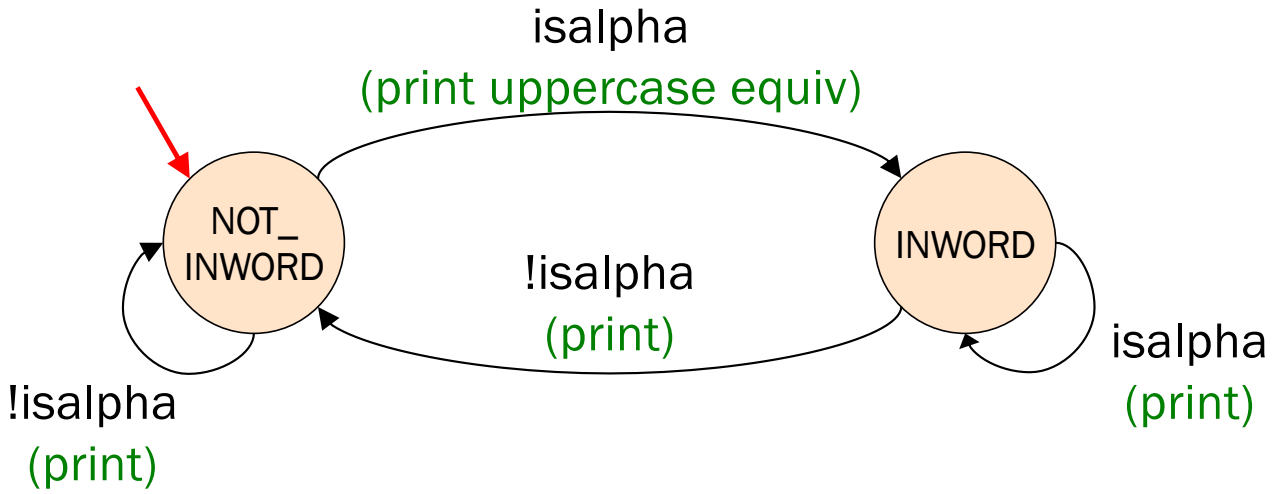
- If we were in a word:
  - If character is alphabetical: still in word. Output same as input
  - If character is not alphabetical: no longer in word. Output same as input
- If we are not in a word:
  - If character is alphabetical: we are now in a word. In this case alone, we should uppercase
  - If character is not alphabetical, we are still not in a word. Output same as input

We must keep track of the **state** (where we were, not just what we're seeing now)

- Use a state machine

upper1 Program Design

Deterministic Finite State Automaton (DFA)



- States, one of which is designated as the start
- Transitions labeled by individual or categories of chars
- Optionally, actions on transitions
- Usually (but not here) a notion of accept ✅ and reject ❌ states

Agenda

A more complex character processing program: Upper1. Why?

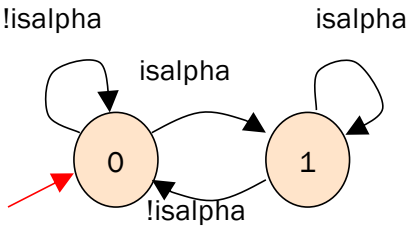
- Assignment 1
- Design step more involved: designing the simple DFA model
- Coding Step: develop a C program to implement the DFA

Building an executable C program

Next time: Separating Interface and Implementation, using character processing as an example

upper1 C Program, Version 1

```
#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 gets a ‘B’ grade.
What’s wrong?

`upper1` C Program, Toward Version 2

Problem:

- The program works, but …
- States should have names

Solution: Define your own named constants

- `enum Statetype {NOT_INWORD, INWORD};`
  - Define an enumeration type: a type with literals that are semantically meaningful names for a subset of integer values
    - (values start at 0 or a specifically assigned value)
    - (subsequent values increment by 1 over previous if not specifically assigned)
- `enum Statetype state;`
  - Define a variable of that type. state can take two values: NOT_INWORD and INWORD
  - (when not defined explicitly, C uses 0, 1, 2 etc. for enum values)

upper1 C Program, Version 2

```
…
enum Statetype {NOT_INWORD, INWORD};
int main(void) {
   int c;
   enum Statetype state = NOT_INWORD;
   while ((c = getchar()) != EOF) {
      switch (state) {
         case NOT_INWORD :
            if (isalpha(c)) {
               putchar(toupper(c)); state = INWORD;
            } else {
               putchar(c); state = NOT_INWORD;
            }
            break;
         case INWORD:
            if (isalpha(c)) {
               putchar(c); state = INWORD;
            } else {
               putchar(c); state = NOT_INWORD;
            }
            break;
      }
   }
   return 0;
}
```

That's a B+.
What's wrong?

`upper1` C Program, Toward Version 3

Problem:

- The program works, but…
- Deeply nested statements
- No modularity

Solution:

- Handle each state in a separate function

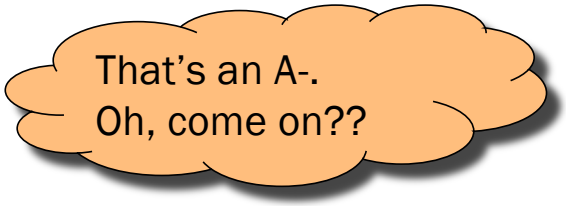
upper1 C Program, Version 3

```
#include <stdio.h>
#include <ctype.h>
enum Statetype {NOT_INWORD,
INWORD};

enum Statetype
handleNotInwordState(int c)
{
   enum Statetype state;
   if (isalpha(c)) {
      putchar(toupper(c));
      state = INWORD;
   } else {
      putchar(c);
      state = NOT_INWORD;
   }
   return state;
}
```

```
enum Statetype
handleInwordState(int c)
{
   enum Statetype state;
   if (!isalpha(c)) {
      putchar(c);
      state = NOT_INWORD;
   } else {
      putchar(c);
      state = INWORD;
   }
   return state;
}
```

```
int main(void)
{
   int c;
   enum Statetype state = NOT_INWORD;
   while ((c = getchar()) != EOF) {
      switch (state) {
         case NOT_INWORD:
            state = handleNoTInwordState(c);
            break;
         case INWORD:
            state = handleInwordState(c);
            break;
      }
   }
   return 0;
}
```



Agenda

A more complex character processing program: Upper1. Why?

- Assignment 1
- Design step more involved: designing the simple DFA model
- Coding Step: develop a C program to implement the DFA

Building an executable C program

Building the `upper` Executable

We'll go back to upper, just because the code fits better on a slide

```
$ gcc217 upper.c
$ ls
.    ..   a.out
$
$ gcc217 upper.c -o upper
$ ls
.    ..   a.out     upper
$ ./upper
Does this work?
It seems to work.
^D
```

What do we see in our terminal emulator after this?

```
DOES THIS WORK?
IT SEEMS TO WORK.
COS 217 ROCKS!
```

`upper` Build Process in Detail

The starting point:

`upper.c`

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- C language
- Missing declarations of `getchar()`, `putchar()`, islower(), toupper()
- Missing definitions of getchar(), putchar(), islower(), toupper()

`upper` Build Process in Detail

Question:

- Exactly what happens when you issue the command
  `gcc217 upper.c –o upper`

Answer: Four steps

- Preprocess
- Compile
- Assemble
- Link

`upper` Build Process: Preprocessor

Command to preprocess:

- `gcc217 –E upper.c > upper.i`

Preprocessor functionality

- Removes comments
- Handles preprocessor directives, like including code from files and expanding macros

upper Build Process: Preprocessor

upper.c

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

Preprocessor removescomment.

(This is the crux of Assignment 1)

upper Build Process: Preprocessor

upper.c

```
#include <stdio.h>
#include <ctype.h>
/* Turn letters in stdin to uppercase
and print result to stdout. Return 0. */
int main(void)
{  int c;
   while ((c = getchar()) != EOF)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

Preprocessor replaces

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

with contents of

```
/usr/include/stdio.h
```

Similarly for ctype.h

Preprocessor replaces `EOF` with –1

(it knows this because EOF is defined in stdio.h, which it has now included in the file).

upper Build Process: Preprocessor

The result `upper.i`

```
...
int getchar();
int putchar();
...
#define EOF -1;
...
int islower(int a)
int toupper(int a)
...

int main(void)
{  int c;
   while ((c = getchar()) != -1)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- C language
- No comments (removed)
- No preprocessor directives (replaced)
- Contains code from `stdio.h`: **declarations** of `getchar()`, `putchar()`, `etc.`
- Contains value for `EOF`
- Missing **definitions** of `getchar()`, `putchar()`, `etc.`

`upper` Build Process: Compiler

Command to compile:

- `gcc217 –S upper.i`

Compiler functionality

- Translate from C to assembly language
- Check syntax
- Check types. Use function declarations to check calls of `getchar()`, `putchar()`, `islower()`, `toupper()`

upper Build Process: Compiler

upper.i

```
...
int getchar();
int putchar());
...
int islower(int a);
int toupper(int a);
...
int main(void)
{  int c;
   while ((c = getchar()) != -1)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- Compiler sees function **declarations**
- These give compiler enough information to check subsequent calls of `getchar()`, `putchar()`, `islower()`, `toupper()`

upper Build Process: Compiler

upper.i

```
...
int getchar();
int putchar());
...
int islower(int a);
int toupper(int a);
...
int main(void)
{  int c;
   while ((c = getchar()) != -1)
   {  if (islower(c))
         c = toupper(c);
      putchar(c);
   }
   return 0;
}
```

- Compiler checks calls of `getchar()`, `putchar()`, `islower()`, `toupper()` in `main()`
- Compiler translates C code to assembly language directives and instructions progressively

`upper` Build Process: Compiler

`upper.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:
          ….
.L2:
        …..
```

- Assembly language
- Missing definitions of `getchar()`, `putchar()`, `islower()`, `toupper()`
- So not enough information to execute them

`upper` Build Process: Assembler

Command to assemble:

- `gcc217 –c upper.s`

Assembler functionality

- Translate from assembly language to machine language
- (Terminology: assembler does not produce assembly language)

`upper` Build Process: Assembler

The result: An "object file"

`upper.o`

```
Machine language
version of the
program

No longer human-
readable
(except some of
it by COS217
students)
```

- Machine language
- (Still) Missing definitions of `getchar`, `putchar()`, `islower()`, `toupper()`
- So cannot be executed

`upper` Build Process: Linker

Command to link:

- `gcc217 upper.o –o upper`

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. Pull the relevant .o files from the .a archive and link them together with upper.o
- Produce final executable

`upper` Build Process: Linker

The result:

`upper`

```
Machine language
version of the
program

No longer human
-readable
(except some of
it by COS217
students)
```

- Machine language
- Contains definitions of `getchar()`, `printf()`, `islower()`, `toupper()`

Complete. Executable.

Agenda

A more complex character processing program: Upper1. Why?

- Assignment 1
- Design step more involved: designing the simple DFA model
- Coding Step: develop a C program to implement the DFA

Building an executable C program

Next time: Separating interface and implementation, using character processing as an example