CS2211a Lab No. 9
Introduction to C

Tuesday November 24, 2015 (sections 3 and 2),
Wednesday November 25, 2015 (sections 6 & 8), and
Thursday November 26, 2015 (sections 9 and 5)

Location: ***MC10*** lab

The objective of this lab is to practice:

- **C Preprocessor**
- **Makefile**

If you would like to leave, and at least 30 minutes have passed, raise your hand and wait for the TA.
Show the TA what you did. If, and only if, you did a reasonable effort during the lab, he/she will give you the lab mark.

==========

In this lab, you should decide on the correct responses before running the code to test the real result.

1. Write parameterized macros that compute the following values.

(a) The cube of `x`.

(b) The remainder when `n` is divided by `4`.

(c) `1` if the product of `x` and `y` is less than `100`, `0` otherwise.

(d) The number of elements in a one-dimensional array (see the discussion of the `sizeof` operator in Section 8.1)

Write a program to test these macros. Do your macros always work? If not, describe what arguments would make them fail.

Hint: What will happen if the provided argument(s) have side effect?

2. Show what the following program will look like after preprocessing. You may **ignore** any lines added to the program as a result of including the < `stdio.h>` header. What will be the output of this program?

```
#include <stdio.h>
#define N 100

void f(void);

int main(void)
{
  f();
# ifdef N
 #  undef N
  #   endif
  return 0;
}


void f(void)
{
#if defined(N)
  printf("N is %d\n", N);
#else
  printf("N is undefined\n");
#endif
}
```

3. Let `TOUPPER` be the following macro:

```
#define TOUPPER(c) ('a' <= (c) && (c) <= 'z'? (c) - 'a' + 'A': (c))
```

Let `s` be a string and let `i` be an `int` variable.
Show the output produced by each of the following program fragments.

```
(a) strcpy(s, "abcde");
    i=0;
    putchar(TOUPPER(s[++i]));
    putchar('\n');

(b) strcpy(s, "01234");
    i=0;
    putchar(TOUPPER(s[++i]));
    putchar('\n');
```

Did you get the results that you expected? If not, explain why?

4. C compilers usually provide some method of specifying the value of a macro at the time a program is compiled. This ability makes it easy to change the value of a macro without editing any of the program's files. Most compilers (including `gcc`) support the `-D` option, which allows the value of a macro to be specified on the command line, i.e., *defines* a macro as if by using `#define`. Many compilers also support the `-U` option, which *undefines* a macro as if by using `#undef`.
Type the following program:

```
#include <stdio.h>

#ifdef DEBUG
#define PRINT_DEBUG(n) printf("Value of " #n ": %d\n", n)
#else
#define PRINT_DEBUG(n)
#endif

int main(void)
{
int i = 1, j = 2, k = 3;

#ifdef DEBUG
  printf("DEBUG is defined:\n");
#else
  printf("DEBUG is not defined:\n");
#endif

PRINT_DEBUG(i);
PRINT_DEBUG(j);
PRINT_DEBUG(k);
PRINT_DEBUG(i + j);
PRINT_DEBUG(2 * i + j - k);
return 0;
}
```

(a) Compile and the run this program without using any option during compilation
(b) Compile and the run this program using the following options during compilation:

```
     i.  -DDEBUG=1
    ii.  –DDEBUG
   iii.  -Ddebug=1
    iv.  –Ddebug
```

What are the differences between the five runs? Why?

5. The following are two C programs (**addition.c** and **add_fun.c**) and one header file (**add_fun.h**).

addition.c

```
#include <stdio.h>
#include <stdlib.h>
#include "add_fun.h"

int main(int argc, char *argv[])
{
    int operand1, operand2, result;

    // check to make sure that we have the correct number or arguments
    if(argc != 3)
    { // print an error message and exit
      printf("Usage: %s operand1 operand2\n", argv[0]);
      // return with  unsuccessful status
      return 1;
    }

    // convert the arguments from array of characters to integer
    operand1 = atoi(argv[1]);
    operand2 = atoi(argv[2]);

    // perform the addition operation
    result = add(operand1, operand2);

    // print the result
    printf("       Result = %d\n\n", result);

    // return with a successful status
    return 0;
}
```

add_fun.c

```
#include "add_fun.h"

int add(int op1, int op2)
{
   int sum;

   sum = op1 + op2;

   return sum;
}
```

add_fun.h

```
int add(int op1, int op2);
```

To automatically build and test these C programs, you need to have a **makefile**.
The following is a suggested **makefile**.

makefile

```
#format is target-name: target dependencies
#{-tab-}actions

# MACRO definitions
CC  = gcc
CFLAG = -std=c99 -Wall

# All Targets
all: addition

#Executable addition depends on the files addition.o add_fun.o
addition: addition.o add_fun.o
    $(CC) $(CFLAG) -o addition addition.o add_fun.o

# addition.o depends on the source and header files
addition.o: addition.c add_fun.h
    $(CC) $(CFLAG) -c addition.c

# add_fun.o depends on the source and header files
add_fun.o: add_fun.c add_fun.h
    $(CC) $(CFLAG) -c add_fun.c

# test cases
test: addition
    addition 1 2
    addition 6 7
    addition 11 2

#Clean the build directory
clean:
    rm -f *.o addition
```

download all these 4 files and do the following experimentations ***(you should expect he answer before executing the command)***:

(a) `make`
(b) `make clean`
(c) `make all`
(d) `make all`
(e) `make clean`
(f) `make addition`
(g) `make addition`
(h) `rm addition.o`
(i) `make`
(j) `rm add_fun.o`
(k) `make`
(l) `touch addition.h`
(m) `make`
(n) `touch add_fun.c`
(o) `make`
(p) `touch add_fun.h`
(q) `make`
(r) `make test`
(s) `make clean`
(t) `make test`