CprE 288 Spring 2016 – Homework 5
Due: Note: not collected or graded, for Exam 1 practice only
Notes:
- Homework is individual work. Adhere to the University’s policy relating to the integrity of scholarship. See http://catalog.iastate.edu/academiclife/regulations/, “Academic Dishonesty”.
- Homework must be typed and uploaded to BlackBoard as a PDF or Word Document (i.e. .doc or .docx) only.
- Late homework is accepted within three days from the due date. Late penalty is 10% per day (for each 24 hours following your class).

Note: Unless otherwise specified, all problems assume the ATMega128 is being used

Question 1: C Operator Precedence (10 pts)
Does each of the code fragment serve its stated purpose? You must explain to get the full credit. Assume the following variable declaration:

```c
unsigned int a, b, c, d, mask;
unsigned int *p = &a;
```

Hint: See final page of homework for a precedence chart.

a. [3] Check if a is equal to b, OR c is equal to d.

```c
if (a == b || c == d)
    ... // code
```
Yes. The precedence of || is lower than that of ==. The condition is equivalent to ((a == b) || (c == d)).

b. [3] Check if the upper half of a is equal to the upper half of b.

```c
mask = 0xFF00;
if (a & mask == b & mask)
    ... // code
```
No. & is lower than == in the precedence table, so the condition is equivalent to (a & (mask == b) & mask).


```c
*p++;
```
No. ++ is higher than * in the precedence table, so the statement is equivalent to *(p++) . The pointer p will be incrementd, not *p (i.e a). To increment a you would need (*p)++;
d. [3] Swap the upper half and the lower half of a.

\[ a = a << 8 + a >> 8; \]

No. Because + has higher precedence than >> and <<, this statement is equivalent to \( a << (8+a) >> 8 \)

e. [3] Swap the upper half and the lower half of b.

\[ b = b << 8 | b >> 8; \]

Yes. | has lower precedence than << and >>, so it is indeed \( (b << 8) | (b >> 8) \).

Question 2: More precedence practice (12 pts)

Place parenthesis around the following statements to explicitly show how the compiler will interpret the association of the operations. (Hint: see order of operations table at end of homework)

a. \( d + e * a + c; \) // Sol: \( d + (e*a) + c; \)

b. if( a & b == c) // Sol: if( a & (b==c) )

c. \( *x_ptr++; \) // Sol: \( *(x_ptr++); \)

d. \( *my_ptr.name; \) // Sol: \( *(my_ptr.name) \)

e. a & *b * c; // Sol: a & ( (*b) * c);

f. if(a >> b + c * var1 == var2) //Sol: if( (a >> (b + (c*var1) )) == var2);
**Question 3: Scope Practice (15 pts)**

What will be printed by `printf`?

a) **x is 5**.

**Sol:** The local instantiation of `x` within `main` is in scope. So `printf` will print: **x is 5**

```c
int x = 20;

int my_function(int my_op)
{
    int x = 0;
    x++;
    return x;
}

main()
{
    int x = 5;
    int y = 0;
    y = my_function(x);
    printf("x is %d", x);
}
```

b) **x is 1**.

**Sol:** The local instantiation of `x` within `my_function` is in scope. So `printf` will print: **x is 1**

```c
int x = 20;

int my_function(int my_op)
{
    int x = 0;
    x++;
    printf("x is %d", x);
    return x;
}

void main(void)
{
    int x = 5;
    x = my_function(x);
}
```
c) x is 1 x is 1 x is 1 x is 1 x is 1 x is 1

Sol: The local instantiation of x within my_function is in scope. So printf will print:
  x is 1 x is 1 x is 1 x is 1 x is 1

int x = 20;

int my_function(int my_op)
{
    int x = 0;

    x++;
    printf("x is %d", x);

    return x;
}

main()
{
    int x = 5;

    for(x = 10; x < 15; x++)
    {
        my_function(x);
    }

}

d) x is 1 x is 2 x is 3 x is 4 x is 5

Sol: The local instantiation of x within my_function is in scope, and x is declared to be a static variable. Thus, it will retain its value across function calls. So printf will print:
  x is 1 x is 2 x is 3 x is 4 x is 5

int x = 20;

int my_function(int my_op)
{
    static int x = 0;

    x++;
    printf("x is %d", x);

    return x;
}

main()
{
    int x = 5;
for(x = 10; x < 15; x++)
{
    my_function(x);
}
}

e) **x is 21**.

**Sol:** The global instantiation of x is in scope when printf is call. So printf will print: **x is 21**

```c
int x = 20;

int my_function(int my_op)
{
    x++;  
    printf("x is %d", x);  
    return x; 
}

main()
{
    x = my_function(x);  
}
```
Question 4: Interrupt Service Routine (8 pts)

Consider the following code which is a partial implementation of interrupt-based clock implementation of Lab 4 assignment:

```c
static int hours;
static int minutes;
static int seconds;

ISR (TIMER1_COMPA_vect)
{
    char buffer[12];

    ... // code for advancing the clock

    /* print out time in HH:MM:SS format*/
    sprintf(buffer, "%02d:%02d:%02d", hours, minutes, seconds);
    lcd_puts(buffer);
}
```

a. Discuss generally why it's a bad idea to call time-consuming functions like “lcd_puts” from the inside of an ISR function. (4pts)

It will hang up the foreground execution and prevents other ISRs from being executed for a long time. In general, ISR should complete execution as soon as possible. Time-consuming computation or data processing should be left in foreground execution.

Functions like `lcd_puts` and `printf` take a long time to execute and should not be placed inside of ISRs.

b. Discuss specifically what bad program behavior a user may experience with this implementation. (4pts)

For example, in Lab4, the push button may become unresponsive because the ISR for checking push button may not get executed.
Question 5: Interrupt Service Routine (5 pts)

Explain why a variable shared by an ISR and the foreground program execution must be declared as “volatile”. In particular, discuss what will happen if “volatile” is NOT used on the shared variable and compiler optimization is enabled. The following is an example:

```c
volatile static int clock_flag = 0;

ISR (TIMER1_COMPA_vect)
{
    clock_flag = 1;
}

int main()
{
    ... // other code

    while (1)
    {
        if (clock_flag == 1)
        {
            ... // advance the clock, print clock on LCD
        }
    }
}
```

With compilation optimization, a copy of “clock_flag” value will be loaded into a CPU register on the first checking. The register copy will be used in the following checking instead of the memory copy. For normal code, this optimization will reduce program execution time.

However, the ISR will change the value of “clock_flag” in memory, not the copy in the register. The “volatile” flag tells the compiler not to perform the above optimization, so every time “clock_flag” is checked, its value will be loaded from memory.
Appendix: Operator Precedence Chart

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Expression</td>
<td>() [] . -&gt; expr++ expr--</td>
<td>left-to-right</td>
</tr>
<tr>
<td>Operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unary Operators</td>
<td>* &amp; + - ! ~ ++expr --expr (cast) sizeof</td>
<td>right-to-left</td>
</tr>
<tr>
<td>Binary Operators</td>
<td>* / %</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td>+ -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt; &lt;&lt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; &gt; &lt;= &gt;=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>== !=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ternary Operator</td>
<td>?:</td>
<td>right-to-left</td>
</tr>
<tr>
<td>Assignment Operators</td>
<td>+= -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^=</td>
<td>=</td>
</tr>
<tr>
<td>Comma</td>
<td>,</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>