CSC 209 Review 9 Solution

September 1, 2020

1. a) 0

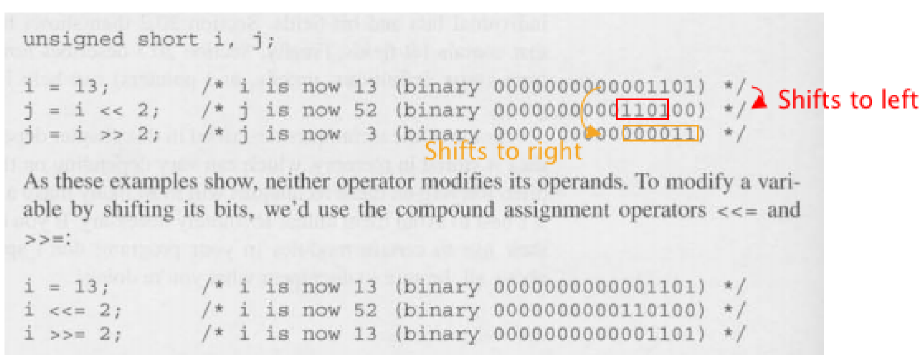
   <u>**Notes**</u>

   - `a)` is 0 because (`i >> 1 + j >> 1 = i >> 10 >> 1 = 0`)
   - **Bitwise Shift Operators**
     - has lower precedence than arithematic operators

       <u>**Example:**</u>

       `i << 2 + 1` means `i << (2+1)` and not `(i << 2) + 1`
     - << : Left Shift
     - >> : Right Shift
     - *Tip:* Always shift only on <u>unsigned</u> numbers for portability

       <u>**Example**</u>



```
unsigned short i, j;

i = 13;       /* i is now 13 (binary 0000000000001101) */
j = i << 2;   /* j is now 52 (binary 0000000000110100) */
j = i >> 2;   /* j is now  3 (binary 0000000000000011) */
```

As these examples show, neither operator modifies its operands. To modify a variable by shifting its bits, we'd use the compound assignment operators <<= and >>=:

```
i = 13;       /* i is now 13 (binary 0000000000001101) */
i <<= 2;      /* i is now 52 (binary 0000000000110100) */
i >>= 2;      /* i is now 13 (binary 0000000000001101) */
```


     - >>= / <<= : Are bitwise shift equivalent of + =

   b) 0

   <u>**Notes**</u>

- i is 1111111111111111
- i is 0000000000000000
- so i & i = 0
- : Bitwise complement (NOT)

a	∼ a
0	1
1	0

Example:

```
    0   1   1   1   //<- this is 7
    --------------
    1   0   0   0   //<- this is 8

    so, ~ 7 = 8
```

- &: Bitwise *and*

a	b	a & b
0	0	0
0	1	1
1	0	0
1	1	1

Example:

```
    0   1   1   1   //<- this is 7
    0   1   0   0   //<- this is 4
    --------------
    0   1   0   0   //<- this is 4

    so, 7 & 4 = 4
```

- : Bitwise *exclusive or*
- |: Bitwise *inclusive or*

c) 1

<u>**Notes**</u>

- i is 1111111111111110
- j is 0000000000000000
- i & j is 0000000000000000 or 1
- i & j ^ k is 1

- `^`: Bitwise XOR

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Example:

```
    0    1    1    1    //<- this is 7
    0    1    0    0    //<- this is 4
    --------------
    0    0    1    1    //<- this is 3

    so, 7 ^ 4 = 3

```

d) 0

Example

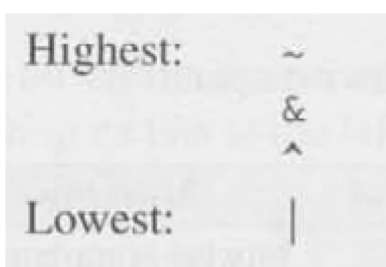
- `i` is `0000000000000111`
- `j` is `0000000000001000`
- `i ^ j` is `0000000000000000` or 0
- `k` is `0000000000001001`
- `i ^ j & k` is `0000000000000000` or 0

Correct Solution

15

Notes

- There is a precendence to the order of operations


Highest: ~
&
^
Lowest: |

e)
- toggling from 0 to 1

  `i = 0x0000;`
  `i |= 0x0001;`

  or

  `i |= 1 << 0;` where `i = 0x0000;`
- toggling from 1 to 0

  `i = 0x0001;`
  `i &= ~0x0001;`

  or

  `i &= ~(1 << 0);` where `i = 0x0001;`

Correct Solution

- toggling from 0 to 1 of 4th bit

  `i = 0x0010;`
  `i ^= 0x0000;`

  or

  `i ^= 1 << 4;` where `i = 0x0000;`
- toggling from 1 to 0 of 4th bit

  `i = 0x0010;`
  `i ^= 0x0010;`

  or

  `i ^= (1 << 4);` where `i = 0x0010;`

Notes

- Toggling can be done using bitwise XOR
- **Setting a bit**
  - Is done using | or bitwise OR

```
i = 0x0000;          /* i is now 0000000000000000 */
i |= 0x0010;         /* i is now 0000000000010000 */
```

  - The idiom of above is `i |= 1 << j`

- **Clearing a bit**
  - Is done using | or bitwise AND

```
i = 0x00ff;          /* i is now 0000000011111111 */
i &= ~0x0010;        /* i is now 0000000011101111 */
```

  - The idiom of above is `i &= ~(i << j)`

2. It swaps the elements between `x` and `y`.

<u>**Notes**</u>

- Preprocessor performs operations of statements in order from left to right

1 2 3

```
#define M(x,y)  ((x)^=(y), (y)^=(x), (x)^=(y))
```

New value of x

New value of y, using x from 1

New value of x, using y from 2, x from 1

3. `#define MK_COLOR(r,g,b) (long) ( (b | (g << 8)) | (b | (r << 16)))`

<u>**Rough Work**</u>

1. store `b` in bit 0

   `b`

2. store `g` in bit 8

   `b | g << 8`

3. store `r` in bit 16

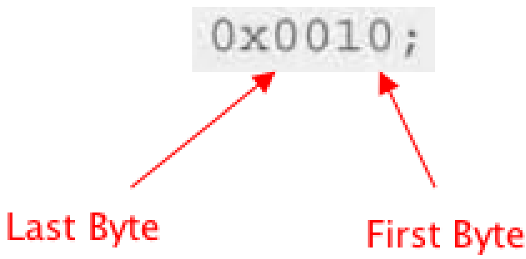
   `b | r << 16`

<u>**Correct Solution**</u>

`#define MK_COLOR(r,g,b) (long) ((r| (g << 8)) | (r| (b<< 16)))`

Notes

- First Byte is furthest from `0x` and first byte is closest to `0x`



4. - `GET_RED`

     `#define GET_RED(c) (long) (c & 0x007)`

   - `GET_GREEN`

     `#define GET_GREEN(c) (long) ((c >> 8) & 0x007)`

   - `GET_BLUE`

     `#define GET_BLUE(c) (long) ((c >> 16) & 0x007)`

Notes

- `0x0007` in binary is `0x0000000000001111`
- `c >> 4` shifts `c` to right by 4 bits and return overlapping value between c >> 4 and `0x0000000000001111` (`0x007`)
- Test code is below

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

#define MK_COLOR(r,g,b) (long) ( (r | (g << 8)) | (r | (b << 16)
))
#define GET_RED(c) (long) (c & 0x007)
#define GET_GREEN(c) (long) ((c >> 8) & 0x007)
#define GET_BLUE(c) (long) ((c >> 16) & 0x007)

int main() {
    long i, r = 4, g = 5, b = 6, r2, g2, b2;

    i = MK_COLOR(r,g,b);
```

```c

        r2 = GET_RED(i);
        g2 = GET_GREEN(i);
        b2 = GET_BLUE(i);

        printf("%ld\n", i);
        printf("%ld\n", r2);
        printf("%ld\n", g2);
        printf("%ld\n", b2);

        return 0;
    }
```

5.

```c
unsigned short swap_bytes(unsigned short i) {

}
```