CMP-245 Object Oriented Programming Lab
BS Fall 2018
Lab 04
Configuration C: PF >= 65

Issue Date: 11-Oct-2019
Marks: 26

Objective(s):

- Resolving issues related to pointer as data member.
- Focusing on Object's initialization and resource Allocation/de-allocation issues.
- Understanding Bitwise operators.

Bitwise Operators

'C' was originally designed to write system software as an alternative to assembler: compilers, kernels, device drivers, interpreters etc. So, this language needs access to raw hardware and individual bit values.
So, it is provided with bit manipulation operators by which we can deal with individual bits and bytes of memory. C++ as its extension also inherited all these operators in it. Typically, bitwise operations are substantially faster than division, several times faster than multiplication, and sometimes significantly faster than addition so they can be used as an alternative in low power processors like in embedded devices (Digital Watches, MP3 Players, Mobile phones, Camera etc). Bitwise operations play critical role particularly in low-level programming such as writing device drivers. Many situations, need to operate on the bits of a data word

- Register inputs or outputs
- Controlling attached devices
- Obtaining status of flags
- Doing some mathematical operations
- Data Compression

Following are the bitwise operators that are available in C/C++.

Operator Name	Operator Symbol
Left shift	<<
Right shift	>>
Bitwise AND	&
Bitwise OR	|
Exclusive OR	^
1's complement	~

These operators are applied to all kinds of *integer* types: Signed and Unsigned (char, short, int, long, long long)

Explanation with Programming Example

1) AND Operator (&)

This is a binary operator. It returns AND of the bits of Left-Hand Side Operand and Right-Hand Side Operand. Example:

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

```
int main()
{
 unsigned char a = 240 ; // 1 1 1 1 0 0 0 0
 unsigned char b = 170; //  1 0 1 0 1 0 1 0
 unsigned char c = a & b ;
 cout << (int)c ; //   1 0 1 0 0 0 0 0 = 160
 return 0;
}
```

2) OR Operator (|)

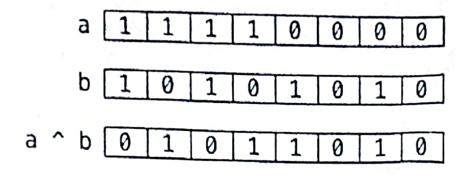
This is a binary operator. It returns OR of the bits of Left-Hand Side Operand and Right-Hand Side Operand. Example:

a	1	1	1	1	0	0	0	0
b	1	0	1	0	1	0	1	0
a | b	1	1	1	1	1	0	1	0

```
int main()
{
   unsigned char a = 240 ; // 1 1 1 1 0 0 0 0
   unsigned char b = 170; //  1 0 1 0 1 0 1 0
   unsigned char c = a | b ;
   cout << (int)c ; //   1 1 1 1 1 0 1 0 = 250
   return 0;
}
```

3) XOR Operator (^)

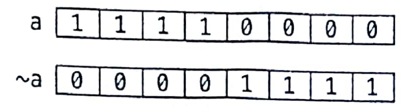
This is a binary operator. It returns XOR of the bits of Left Hand Side Operand and Right Hand Side Operand.
Here is a trick to remember its working its name says EXCLUSIVE OR which refers that exclusively first OR, second which means only one of them should be true.
So 1 XOR 0 or 0 XOR 1 is only true.... others are false. Example:



```
int main()
{
   unsigned char a = 240 ; // 1 1 1 1 0 0 0 0
   unsigned char b = 170; //  1 0 1 0 1 0 1 0
   unsigned char c = a ^ b ;
   cout << (int)c ; //   0 1 0 1 1 0 1 0 = 90
   return 0;
}
```

4) NOT Operator (~)

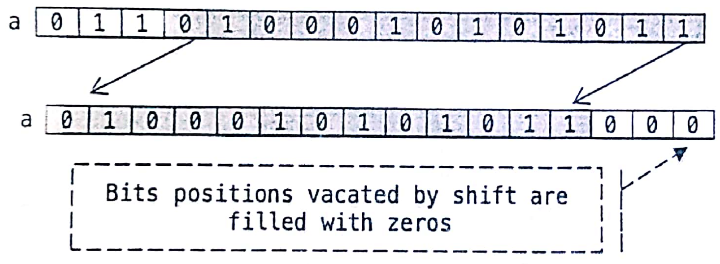
This is a unary operator. It reverses the bits of operand. Example:



```
int main()
{
   unsigned char a = 240 ; // 1 1 1 1 0 0 0 0
   unsigned char b = ~a;
   cout << (unsigned int)b ;  // answer is 15
   return 0;
}
```

5) SHIFT LEFT (<<)

This is a binary operator. It shifts the bits of operand on its Left-Hand side by a shift of operand on its Right-Hand Side in left direction and inserts 0's on its right-hand side. Example:

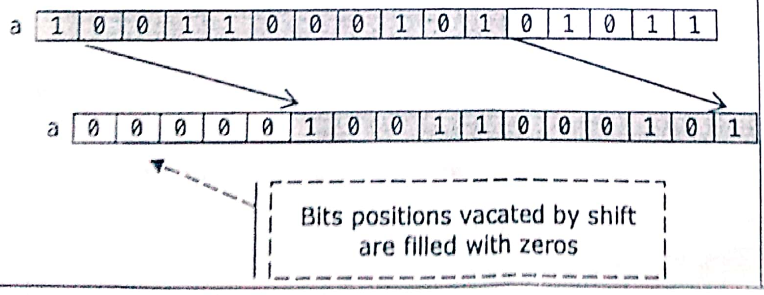

a 0 1 1 0 1 0 0 0 1 0 1 0 1 0 1 1
a 0 1 0 0 0 1 0 1 0 1 0 1 1 0 0 0
Bits positions vacated by shift are filled with zeros

```
int main()
{
   short int a = 26795;
   a = a << 3; // shift left
   3 bits
   return 0;
}
```

6) SHIFT RIGHT (>>)

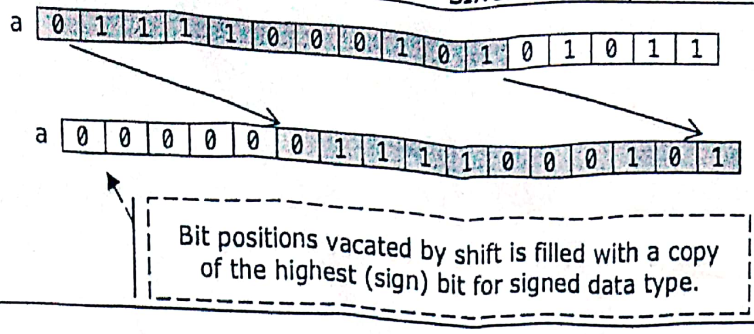
This is a binary operator. It shifts the bits of operand on its Left-Hand side by a shift of operand on its Right-Hand Side in right direction. In case of unsigned integer, the left-hand side is filled with 0's. In case of signed it is machine dependent.

UNSINGNED:


a 1 0 0 1 1 0 0 0 1 0 1 0 1 0 1 1
a 0 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1
Bits positions vacated by shift are filled with zeros

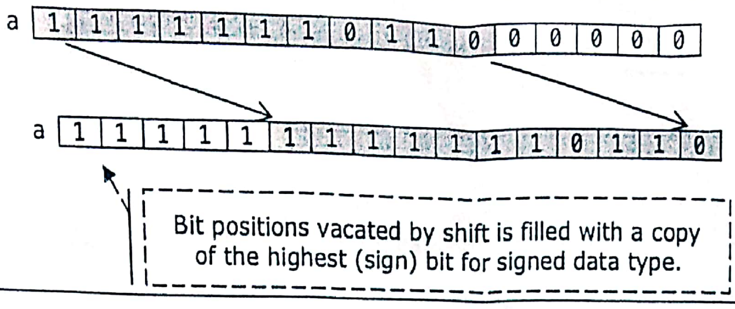
```
int main()
{
   unsigned short int a =
   39083;
   a = a >> 5;
   // shift right 5 bits
   return 0;
}
```

SINGNED: *Example-1*


a 0 1 1 1 1 0 0 0 1 0 1 0 1 0 1 1
a 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1
Bit positions vacated by shift is filled with a copy of the highest (sign) bit for signed data type.

```
int main()
{
   short int a = 30891;
   a = a >> 5;
   // shift right 5 bits
   return 0;
}
```

SINGNED: *Example-2*


a 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 0
a 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0
Bit positions vacated by shift is filled with a copy of the highest (sign) bit for signed data type.

```
int main()
{
   short int a = -320;
   a = a >> 5;
   // shift right 5 bits
   return 0;
}
```

Your Task

Bit Vector

[http://en.wikipedia.org/wiki/Bit_array http://www.cplusplus.com/reference/bitset/bitset/]
A bit array (also known as bitmap, bitset, bit string, or bit vector) is an array data structure that compactly stores bits.
Task: *Bit Array Implementation in C++.*
In this task we shall define an ADT BitArray, which will support the operations discussed above.

```
class BitArray
{
private:
    int capacity;
    unsigned char * data;
    int isValidBit(int i)
    {
        return i>=0 && i < capacity ;
    }
public:
    int getCapacity();
    BitArray(int n = 8)
    {
        capacity = n;
        int s = (int)ceil((float)capacity/8);
        data = new unsigned char[s];
        for (int i=0; i<s; i=i+1)
        {
            data[i] = data[i] & 0;
        }
    }
    BitArray(const BitArray & ref)        //.................................... (1)
    {
        // Complete yourself
    }
    void on( int bitNo);                  //.................................... (3)
    void off(int bitNo);                  //.................................... (..)
    int checkBitStatus(int bitNo);          //.................................. (2)
    void invert(int bitNo);                 //.................................. (2)
    void dump();                            //.................................. (4)

    BitArray AND(BitArray);                 //.................................. (4)
    BitArray OR(BitArray);                  //.................................. (..)

    void shiftLeft(int);                    //.................................. (5)
    void shiftRight(int);                   //.................................. (..)

    unsigned long long getUnSignedIntegeralValue();//........................... (2)
    void setItegralValue(unsigned long long);    //............................. (2)

    ~BitArray();                            //.................................. (1)
};
```

Sample Run:

```
int main()
{
    BitArray ba(17);
    ba.on(0);
    ba.on(2);
    ba.on(3);
    ba.on(8);
    ba.on(16);
    ba.dump();
    BitArray ba2(ba);
    ba.invert(2);
    ba.invert(6);
    cout<<'\n';
    ba.dump();
    cout<<'\n';
    ba2.dump();
    cout<<'\n';
    return 0;
}
```

Console Output

```
1 00000001 00001101
1 00000001 01001001
1 00000001 00001101
Program ended with exit code: 0
```

Hopefully, you know now that how to swim with bits but those of you, who want to do deep sea diving with bits: they should explore the following pointer.
http://graphics.stanford.edu/~seander/bithacks.html
You guys should also explore the bitset class in C++
http://www.cplusplus.com/reference/bitset/bitset/

Low-level programming is good for the programmer's soul.

-- John Carmack --