ECE 150 Bitwise and bit-shifting

Douglas Harder

December 2023

1 Bitwise and bit-shifting operations

The integer data types store either 8, 16, 32 or 64 bits. These can be used to represent integer values (signed or unsigned), but they can also be used as a collection of bits to be manipulated. For example, a single bit can be used to store a Boolean of either `true` (`1`) or `false` (`0`). The `bool` data type occupies one byte, or 8 bits. Similarly, two bits could be used to store values between `0` and `2` or between `0` and `3`. To access individual bits or groups of bits, we use bitwise and bit-shifting operations.

These bitwise and bit-shifting operations are restricted to integer data types including `char`, as it makes no sense to manipulate the bits of a floating-point number.

The operators include the unary `~` and the binary operators

```
&     &=
|     |=
^     ^=
<<    >>=
>>    >>=
```

2 Bitwise operations

We have already described three logical operators: the unary logical NOT, and the binary logical AND and OR. The bitwise operators perform similar operations but on the corresponding bits of the operands where `1` is interpreted as `true` and `0` as `false`.

There is one unary bitwise NOT operator, and three binary bitwise operators: `&`,`|` and `^`. For each of the binary operators, there is also an automatic operator: `&=`, `|=` and `^=`.

2.1 Bitwise NOT

The unary bitwise NOT operator parallels the unary logical NOT operator `!` and is represented by the tilde or `~`. It is also called the "complement operator", as it complements or switches the value of each bit. For example,

```
// Here, 'value' stores this value, and its complement
// flips all of the bits.
//    ~00001100001000100101010101011011
//     --------------------------------
//     11110011110111011010101010100100

unsigned int value{ 0b00001100001000100101010101011011 };
std::cout << ~value << std::endl;
```

This is most useful when you which to complement a mask:

```
//    ~00000000000000000000000011111111
//     --------------------------------
//     11111111111111111111111100000000

unsigned int mask{ 0x000000ff };
std::cout << ~mask << std::endl;
```

Not on the examination

Recall that the destructor of a class `Class_name` has the function name `~Class_name()`. You can think of the destructor as *not* the constructor.

2.2 Bitwise AND

The binary bitwise AND performs an AND on all the corresponding bits. This is most useful to determine if either a single bit is equal to `0` or `1`, to extract only some bits within a mask of a given number, and to set a bit to 0.

```
// The bits of an integer are numbered from
// right-to-left starting with 0
//  - The right-most bit is the least-significant bit
//  - In an unsigned integer,
//      left-most bit is the most-significant bit
//
```

```
//     00001100001000100101010101011011
//   & 00000000000000000000000001000000
//     --------------------------------
//     00000000000000000000000001000000
//                              ^

unsigned int value{ 0b00001100001000100101010101011011 };
unsigned int  bit6{ 0b00000000000000000000000001000000 };

if ( (value & bit6) == bit6 ) {
    std::cout << "Bit 6 is set to '1'" << std::endl;

    // Set Bit 6 to '0'
    //     00001100001000100101010101011011
    //     11111111111111111111111110111111
    //     --------------------------------
    //     00001100001000100101010100011011
    //                              ^
    value &= ~bit6;
} else {
    assert ( (value & bit6) == 0 );
    std::cout << "Bit 6 is set to '0'" << std::endl;
}

//     00001100001000100101010101011011
//   & 00000000111111110000000000000000
//     --------------------------------
//     00000000001000100000000000000000
//             ^^^^^^^^

// The bytes of an integer are also numbered
// from right-to-left starting with Byte 0
//  - The right-most byte is the least-significant byte
//  - The left-most byte is the most-significant byte
unsigned int byte2{ 0x00ff0000 };
std::cout << (value & byte2) << std::endl;
```

2.3 Bitwise XOR

The binary bitwise XOR performs an exclusive-or (XOR) on all the corresponding bits. If one, but not both operands are `true`, the result is `true`, otherwise, either the operands are either both `true` or they are both `false`, in which case, the result is `false`. In other words exclusively one or the other operand is `true`, but not both. The bitwise XOR operator `^` performs an exclusive-or on each of the corresponding bits.

This is most useful to flip the value of a bit, but it also has one very useful property:

```
((a ^ key) ^ key) == (a ^ (key ^ key)) == (a ^ 0) == a
```

meaning that if you XOR an unsigned integer with a known (but hidden) `key`, then `unsigned int b{a ^ key}` encrypts the integer `a`, and when you receive `b`, you can recover the original integer by applying this same operation a second time: `unsigned int text{b ^ key}`.

```
// The bits of an integer are numbered from
```

```
// right-to-left starting with 0
//
//      00001100001000100101010101011011
//    ^ 00000000000000000000000001000000
//      --------------------------------
//      00001100001000100101010100011011
//                                   ^

unsigned int value{ 0b00001100001000100101010101011011 };
unsigned int  bit6{ 0b00000000000000000000000001000000 };

value =^ bit6;
std::cout << "Flip the value of Bit 6" << std::endl;

//      01010101010101010101010101010101
//    ^ 00010000101100011101100110000001
//      --------------------------------
//      01000101111001001000110011010100
//    ^ 00010000101100011101100110000001
//      --------------------------------
//      01010101010101010101010101010101

unsigned int message{ 0x55555555 };
unsigned int key{ 0x10b1d981 };
unsigned int hidden{ message & key };
unsigned int revealed{ hidden & key };

std::cout << (message == revealed) << std::endl;
```

Not on the examination

There is no logical XOR because you can always use `!=` when comparing two Boolean values (`bool`).

2.4 Example: changing case

If you look at the codes for the ASCII characters in binary, you will note something interesting:

```
'A' 0b01000001     'B' 0b01000010     'C' 0b01000011     'D' 0b01000100
'a' 0b01100001     'b' 0b01100010     'c' 0b01100011     'd' 0b01100100
        ^                  ^                  ^                  ^

        v                  v                  v                  v
'W' 0b01010111     'X' 0b01011000     'Y' 0b01011001     'Z' 0b01011010
'w' 0b01110111     'x' 0b01111000     'y' 0b01111001     'z' 0b01111010
```

The only bit that changes between lower case and upper case is Bit 5. Thus, we may do the following:

```
char ch{ 'a' };  // or char ch{ 'M' };
char case_bit{ 0b00100000 };

if ( (ch & case_bit) == case_bit ) {
  std::cout << "The character is upper case" << std::endl;
```

```
} else {
  std::cout << "The character is lower case" << std::endl;
}

// Make the character upper case
ch |= case_bit;
// Make the character lower case
ch &= ~case_bit;
// Swap the case of the character
ch ^= case_bit;
```

3 Bit-shifting operations

Bit shifting operations `a << n` and `a >> n` both take a unsigned integer `n` as a second operand and then shift the bits of the first operand `a` that many to the left or to the right, respectively. Then `n` bits on the left and right, respectively, of the result will all be zero. For example:

```
10101100000101010011110101100011 << 9
  ______/                _____/
 /                      /
00101010011110101100011000000000
                       ^^^^^^^^^
```

```
10101100000101010011110101100011 >> 13
 \__________       \__________
            \                 \
00000000000001010110000010101001
^^^^^^^^^^^^^
```

There are also two automatic operators, `<<=` and `>>=` where the left-hand operand must be assignable and the right-hand operand is an unsigned integer.

This can be used to not necessarily have to hard code flags, as these are equivalent:

```
// A mask for Bit 2   000...0000100
unsigned int bit_2{ 4 };
unsigned int bit_2{ 1 << 2 };

// A mask for Bit 26
unsigned int bit_26{ 0x04000000 };
unsigned int bit_26{ 1 << 26 };
```

You can also walk through the bits of an unsigned integer with a mask:

```
unsigned int n{ 0x003ba7cd };

for ( unsigned int mask{ 1 }; mask != 0; m <<= 1 ) {
  if ( (n & mask) == mask ) {
    // Do something if the bit is '1'
  }
}
```

It is also useful for accessing one byte at a time:

```
unsigned int n{ 0x53a04bcf };

for ( unsigned int byte{ 0 }; byte < 4; ++byte ) {
  unsigned int byte{ (n >> (8*byte)) & 0xff };
  // Do something with Byte 'byte'
}
```

It is also useful for "rotating" the bits:

```
unsigned int n{ 0x53a04bcf };
// Rotate the bits 3 to the left
//       vvv
//       01010011101000000100101111001111
//         /                           /
//        /                           /
//       10011101000000100101111001111010
//                                    ^^^
assert( sizeof( unsigned int ) == 32 );
n = (n << 3) | (n >> (32 - 3));
```

It is also useful for create a sequence of ones, for $2^{10} = 10000000000_2$ and thus $2^{10} - 1 = 1111111111_2$. Thus, while we can hard code a fixed block of ones,

```
unsigned int mask{ 0x000000ff };
```

we cannot hard code an unknown bock of ones, so we must use:

```
unsigned int bits{ ... };
unsigned int mask{ (1 << bits) - 1 };
```

Not on the examination

If you think `a << -2` should be the same as `a >> 2`, you will find that the result is actually always zero. This is because the `-2` is interpreted as an integer, and `-2` as integer is `0b11111111111111111111111111111110`. However, this is then reinterpreted as an unsigned integer, so shifting `-2` to the left is the same as shifting 4294967294 to the left, a result that is definitely zero.

Not on the examination

It may be really easy to think that you can use `<< 1` instead of multiplication by two, and `<< 2` instead of multiplication by four. You may think that this is "faster" than calling an instruction that performs an integer multiplication.

Please don't do this: if you are using an integer data type to manipulate its bits, use bitwise and bit-shifting operations, but if you are using it to store integer values, use arithmetic operations.

The compiler will optimize the code as appropriate. For example, the following statement

```
a = 3*b;
```

was compiled as if it were

```
a = (b << 1);
a += b;
```

Let the compiler optimize the code; you should optimize the program so that it is most clearly understood by other programmers.

Not on the examination

We only discuss bit-shifting operations for unsigned integers. If the argument is a signed integer and you perform a right shift, the extra bits to the left are not filled with `0`, but rather with whatever the sign bit is. Thus, `0b1111111111101101` (storing -19) shifted to the right thrice yields `0b1111111111111101` (storing -3).

4 Exercises

You can try any one of these:

1. Write a function that returns `true` if Bit `k` of an argument `unsigned int n` is `1` and `false` otherwise.
2. Write a function that sets Bit `k` of an argument `unsigned int &n` to `1`.
3. Write a function that sets Bit `k` of an argument `unsigned int &n` to `0`.
4. Write a function that flips the value of Bit `k` of an argument `unsigned int &n`.
5. Write a function that counts the number of bits set to `1` in an argument `unsigned int n`.
6. Write a function that, for an argument `unsigned int n`, returns Bit `k0` to Bit `k1 - 1` (assert `k0 <= k1`) shifted to the right so that Bit `k0` is now at Bit `0` and all bits that were to the left of and including Bit `k1` are set to zero.
7. Write a function that returns rotates the bits of an argument `unsigned int &n` `k` bits to the left. Assert that `(0 <= k) && (k <= 32)` and in either boundary case, leave `n` unchanged.
8. Write a function that returns Bit `k` and `k - 1` that store a value between 0 and 3.
9. Write a function that sets Bit `k` and `k - 1` of an argument `unsigned int &n` to the last two bits of an argument `k`, assumed to store a value between 0 and 3.
10. Write a function that sets Bit `k` and `k - 1` of an argument `unsigned int &n` to the 0.
11. Write a function that sets Bit `k` and `k - 1` of an argument `unsigned int &n` to whatever value is stored there (interpreted as two bits) to that value plus one, so `00` becomes `01`, . . . , and `11` becomes `00`.
12. Write a function that returns Byte `0`, `1`, `2` or `3` of an argument `unsigned int n`, where the byte is passed as a second argument `unsigned int k`, asserting that `k` is in the correct range.
13. Write a function that sets Byte `0`, `1`, `2` or `3` of an argument `unsigned int &n` to all zeros.
14. Write a function that swaps bits `k0` and `k1` of an argument `unsigned int &n`.

This author cut-and-pasted some of these questions into ChatGPT and it returned a valid solution. Please don't do this. Instead, make an attempt, and if you cannot get it working, then ask ChatGPT (or any other such generative artificial intelligence tool) to give a hint as to why your code does not work.