APPENDIX M

Binary Numbers and Bitwise Operations

One of the many powers that C++ gives the programmer is the ability to work with the individual bits of an integer field. The purpose of this appendix is to give an overview of how integer data types are stored in binary and explain the bitwise operators that the C++ offers. Finally, we will look at bit fields, which allow us to treat the individual bits of a variable as separate entities.

Integer Forms

The integer types that C++ offers are as follows:

```
char
int
short
long
unsigned char
unsigned (same as unsigned int)
unsigned short
unsigned long
```

When you assign constant values to integers in C++, you may use decimal, octal, or hexadecimal. Placing a zero in the first digit creates an octal constant. For example, 0377 would be interpreted as an octal number. Hexadecimal constants are created by placing 0x or 0X (zero-x, not O-x) in front of the number. The number 0X4B would be interpreted as a hex number.

Binary Representation

Regardless of how you express constants, integer values are all stored the same way internally—in binary format. Let's take a quick review of binary number representation.

Let's assume we have a one-byte field. Figure M-1 shows the field broken into its individual bits.

Z13_GADD5003_10_SE_APPM.indd 1 25/01/19 2:56 AM

Figure M-1

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

High Order ----------------------------> Low Order

The leftmost bits are called the *high order* bits and the rightmost bits are called the *low order* bits. Bit number 7 is the highest order bit, so it is called the *most significant* bit.

Each of these bits has a value assigned to it. Figure M-2 shows the values of each bit.

Figure M-2

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Values –>	128	64	32	16	8	4	2	1

These values are actually powers of two. The value of bit 0 is 2^0, which equals 1. The value of bit 1 equals 2^1, which equals 2. Bit 2 has the value 2^2, which equals 4. This pattern of values continues to the last bit.

When a number is stored in this field, each bit may be set to either 1 or 0. Figure M-3 shows an example.

Figure M-3

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	0	1	1	0	0	1	1	0
Values –>	128	64	32	16	8	4	2	1

Here, bits 1, 2, 5, and 6 are set to 1. To calculate the overall value of this bit pattern, we add up all of the bit values of the bits that are set to 1.

Bit 1's value	2
Bit 2's value	4
Bit 5's value	32
Bit 6's value	64
	——
Overall Value	102

Thus the bit pattern `01100110` has the decimal value 102.

Negative Integer Values

One way that a computer can store a negative integer is to use the leftmost bit as a sign bit. When this bit is set to 1, it would indicate a negative number, and when it is set to 0 the number would be positive. The problem with this, however, is that we would have two bit patterns for the number 0. The pattern of all 0's would be for positive 0 and the pattern of a 1 followed by all 0's would be for negative 0. Because of this, most systems use two's complement representation for negative integers.

To calculate the two's complement of a number, first you must get what is known as the one's complement. This means changing each 1 to a 0, and each 0 to a 1. Next, add 1 to the resulting number. This gives you the two's complement. Here is how the computer stores the value –2.

2 is stored as	`00000010`
Get the one's complement	`11111101`
Add 1	`1`
The two's comp result is	`11111110`

Notice that the highest order bit for a negative number will always be set to 1, whereas the highest order bit for a positive number or a zero will always be set to 0.

Bitwise Operators

C++ provides operators that let you perform logical operations on the individual bits of integer values, and shift the bits right or left.

The Bitwise Negation Operator

The bitwise negation operator is the ~ symbol. It is a unary operator that performs a negation, or one's complement on each bit in the field. The expression

```
~val
```

returns the one's complement of `val`. It does not change the contents of `val`. It could be used in the following manner:

```
negval = ~val;
```

This will store the one's complement of `val` in `negval`.

The Bitwise AND Operator

The bitwise AND operator is the `&` symbol. This operator performs a logical AND operation on each bit of two operands. This means that it compares the two operands bit by bit. For each position, if both bits are 1, the result will be 1. If either or both bits are 0, the results will be 0. Here is an example:

```
andval = val & 0377;
```

The result of the AND operation will be stored in `andval`. There is also a combined assignment version of this operator. Here is an example:

```
val &= 0377;
```

This is the same as:

```
val = val & 0377;
```

The Bitwise OR Operator

The bitwise OR operator is the | symbol. This operator performs a logical OR operation on each bit of two operands. This means that it compares the two operands bit by bit. For each position, if either of the two bits is 1, the result will be 1. Otherwise, the results will be 0. Here is an example:

```
orval = val | 0377;
```

The result of the OR operation will be stored in `orval`. There is a combined assignment version of this operator. Here is an example:

```
val |= 0377;
```

This is the same as

```
val = val | 0377;
```

The Bitwise EXCLUSIVE OR Operator

The bitwise EXCLUSIVE OR operator is the ^ symbol. This operator performs a logical XOR operation on each bit of two operands. This means that it compares the two operands bit by bit. For each position, if one of the two bits is 1, but not both, the result will be 1. Otherwise, the results will be 0. Here is an example:

```
xorval = val ^ 0377;
```

The result of the XOR operation will be stored in `xorval`. There is also a combined assignment version of this operator. Here is an example:

```
val ^= 0377;
```

This is the same as:

```
val = val ^ 0377;
```

Using Bitwise Operators with Masks

Suppose we have the following variable declarations:

```
char value = 110, cloak = 2;
```

Figure M-4 illustrates the binary pattern for each of these two variables.

Figure M-4

`value ->`	0	1	1	0	1	1	1	0	= 110
`cloak ->`	0	0	0	0	0	0	1	0	= 2

The statement

```
value = value & cloak;
```

will perform a logical bitwise AND on the two variables `value` and `cloak`. The result will be stored in `value`. Remember a bitwise AND will produce a 1 only when both bits are set to 1. Figure M-5 shows the result of this AND operation.

Figure M-5

`value ->`	0	1	1	0	1	1	1	0
AND								
`cloak ->`	0	0	0	0	0	0	1	0
equals								
`result ->`	0	0	0	0	0	0	1	0

The 0's in the `cloak` variable "hide" the values that are in the corresponding positions of the `value` variable. This is called *masking*. When you mask a variable, the only bits that will "show through" are the ones that correspond with the 1's in the mask. All others will be turned off.

Turning Bits On

Sometimes you want to turn on selected bits in a variable and leave the rest alone. This operation can be performed with the bitwise OR operator. Let's see what happens when we OR the `value` variable used above with a new value for the `cloak` variable.

```
char value = 110, cloak = 16;
value = value | cloak;   // The cloak variable acts as the mask.
```

Figure M-6 illustrates this result of this OR operation.

Figure M-6

value –>	0	1	1	0	1	1	1	0
OR								
cloak –>	0	0	0	1	0	0	0	0
Equals								
result –>	0	1	1	1	1	1	1	0

This caused bit 4 of the `value` variable to be turned on and all the rest to be left alone.

Turning Bits Off

Suppose that instead of turning specific bits on, you wish to turn them off. Assume we have the following declaration:

```
char status = 127, cloak = 8;
```

Because binary 8 is stored as 1000, this sets bit 3 of `cloak` to 1, and all the rest to 0. If we wish to set bit 3 of `status` to 0, we must AND it with the negation of `cloak`. In other words, we must first get the one's complement of `cloak` and then AND it with `status`. Recall that the ~ symbol represents bitwise negation. So the statement would look like this:

```
status = status & ~cloak;
```

This works because the unary negation operation is done before the binary AND operation. Figure M-7 illustrates these steps.

Figure M-7

cloak –>	0	0	0	0	1	0	0	0
~cloak –>	1	1	1	1	0	1	1	1
status –>	0	1	1	1	1	1	1	1
AND								
~cloak –>	1	1	1	1	0	1	1	1

This is the result assigned as the new value of `status`

status –>	0	1	1	1	0	1	1	1

Bit 3 of `status` is now turned off, and all other bits are left unchanged.

Toggling Bits

To toggle a bit is to flip it off when it is on, and on when it is off. This can be done with the EXCLUSIVE OR operator. We will use the following variables to illustrate.

```
char status = 127, cloak = 8;
```

Our objective is to toggle bit 3 of `status`, so we will use the XOR operator.

```
status = status ^ cloak;
```

Figure M-8 illustrates this operation.

The operation has changed bit 3 of `status` from 1 to 0.

Figure M-8

`status ->`	0	1	1	1	1	1	1	1

XOR

`cloak ->`	0	0	0	0	1	0	0	0

This is the result assigned as the new value of `status`

`status ->`	0	1	1	1	0	1	1	1

Testing the Value of a Bit

To test the value of an individual bit, you can use the AND operator. For example, if we want to test the variable `bitvar` to see if bit 2 is on, we use a mask that has bit 2 turned on. Here is an example of the test:

```
if ((bitvar & 4) == 4)
   cout << "Bit 2 is on.\n";
```

Remember that ANDing a value with a mask produces a value that hides all of the bits but the ones turned on in the mask. Also recall that binary 4 is expressed as 100, with only bit 2 turned on. So, if bit 2 of `bitvar` is on, the expression `bitvar & 4` will return the value 4, causing the `if` test condition to be true and thus the message to be displayed.

The parentheses around `bitvar & 4` are necessary to ensure that the AND operation is done before the test for equality because the `==` operator has higher precedence than the `&` operator.

The Bitwise Left Shift Operator

The symbol for the bitwise left shift operator is two less than signs (`<<`). This operator takes two operands. The operand on the left is the one to be shifted, and the operand on the right is the number of places to shift. When the bit values are shifted left, the vacated positions to the right are filled with 0's and the bits that shift out of the field are lost.

Suppose we have the following variables:

```
char val = 2, shiftval;
```

The following statement will store in `shiftval` the value of `val` shifted left 2 places.

```
shiftval = val << 2;
```

Figure M-9 shows what is happening behind the scenes with the value in `val`.

Figure M-9

Before left shift	0	0	0	0	0	0	1	0
After left shift	0	0	0	0	1	0	0	0

Realize, however, that `val` itself is not being shifted. Only the variable `shiftval` is being changed. It is set to the value of `val` shifted left 2 places. If we wanted to shift `val` itself, we would need to say

```
val = val << 2;
```

Or, we could use the combined assignment version of the left shift operator, like this:

```
val <<= 2;
```

Shifting a number left by n places is the same as multiplying it by 2^n. So, this example is the same as:

```
val *= 4;
```

The bitwise shift will almost always work faster, however.

The Bitwise Right Shift Operator

The symbol for the bitwise right shift operator is two greater than signs (`>>`). Like the left shift operator, it takes two operands. The operand on the left is the one to be shifted, and the operand on the right is the number of places to shift. When the bit values are shifted right, and the variable is signed, what the vacated positions to the left are filled with depends on the machine. They could be filled with 0s, or with the value of the sign bit. If the variable is unsigned, the places will be filled with 0s. The bits that shift out of the field are lost. Suppose we have the following variables:

```
char val = 8, shiftval;
```

The following statement will store in `shiftval` the value of `val` shifted right 2 places.

```
shiftval = val >> 2;
```

Figure M-10 shows what is happening behind the scenes with the value in `val`.

Figure M-10

Before right shift	0	0	0	0	1	0	0	0
After right shift	0	0	0	0	0	0	1	0

As before, the value of `val` itself is not changed. The variable `shiftval` is being set to the value of `val` shifted right 2 places.

Shifting a number right by n places is the same as dividing it by 2^n (as long as the number is not negative). So, the example is the same as

```
val /= 4;
```

The bitwise shift will almost always work faster, however.

Bit Fields

C++ allows you to create data structures that use bits as individual variables. Bit fields must be declared as part of a structure. Here is an example.

```
struct {
   unsigned field1 : 1;
   unsigned field2 : 1;
   unsigned field3 : 1;
   unsigned field4 : 1;
   } fourbits;
```

The variable `fourbits` contains 4 bit fields: `field1`, `field2`, `field3`, and `field4`. Following the colon after each name is a number that tells how many bits each field should be made up of. In this example, each field is 1 bit in size. This structure is stored in memory in a regular unsigned `int`. As we are only using 4 bits, the remaining ones will go unused.

Values may be assigned to the fields just as if it were a regular structure. The following statement assigns the value 1 to the `field1` member:

```
fourbits.field1 = 1;
```

Because these fields are only 1 bit in size, we can only put a 1 or a 0 in each of them. If we wanted to expand the capacity of the bit fields we could make them larger, as in the following example:

```
struct {
   unsigned field1 : 1;
   unsigned field2 : 2;
   unsigned field3 : 3;
   unsigned field4 : 4;
   } mybits;
```

Here, `mybits.field1` is only 1 bit in size, but others are larger. `mybits.field2` occupies 2 bits, `mybits.field3` occupies 3 bits, and `mybits.field4` occupies 4 bits.

Table M-1 shows the size and the maximum value that can be stored in each field.

Table M-1

Field Name	Number of Bits	Maximum Value
`mybits.field1`	1	1
`mybits.field2`	2	3
`mybits.field3`	3	7
`mybits.field4`	4	15

This data structure uses a total of 10 bits. If you create a bit field structure that uses more bits than will fit in an `int`, the next `int` sized area will be used. Suppose we define the following bit field structure on a system that has 16 bit integers.

```
struct {
   unsigned tiny    : 1;
   unsigned small   : 4;
   unsigned big     : 6;
   unsigned bigger  : 8;
   unsigned biggest : 9;
   } flags;
```

This structure requires a total of 28 bits, so it will need to occupy more than one integer. It might seem that the bits would fit in two integers. However, because the compiler will not allow a field to straddle two different integers, three integers are used. Here is what occurs. The fields `tiny`, `small`, and `big` will occupy the first integer, leaving five unused bits in it. Because the fields `bigger` and `biggest`, totalling 17 bits, cannot fit in the same integer, `bigger` will be placed in its own second integer and `biggest` will be placed in a third one. There will be 8 unused bits in the second area, and 7 unused bits in the third.

You can force a field to be aligned with the next integer area by putting an unnamed bit field with a length of 0 before it. Here is an example:

```
struct {
   unsigned first  : 1;
                   : 0;
   unsigned second : 1;
                   : 0;
   unsigned third  : 2;
   } scattered;
```

The unnamed fields in the `scattered` structure with 0 width force the fields `second` and `third` to each be placed in the next `int` area.

You can also create unnamed fields with lengths other than 0. This way you can force gaps to exist at certain places. Here is an example.

```
struct {
   unsigned first  : 1;
                   : 2;
   unsigned second : 1;
                   : 3;
   unsigned third  : 2;
   } gaps;
```

This will cause a 2-bit gap to come between fields `first` and `second` of the `gaps` structure, and a 3-bit gap to come between fields `second` and `third`.

Bit fields are not very portable, however, when the physical order of the fields and the exact location of the boundaries are specified because some machines order the bit fields from left to right, while others order them from right to left.