Lecture 14: Bitwise Operators, Input/Output with Files and Exceptions Handling
Thinking about Bits
The byte is the lowest level at which we can access data; there's no "bit" type, and we can't ask for an individual bit. In fact, we can't even perform operations on a single bit -- every bitwise operator will be applied to, at a minimum, an entire byte at a time. This means we'll be considering the whole representation of a number whenever we talk about applying a bitwise operator. (Note that this doesn't mean we can't ever change only one bit at a time; it just means we have to be smart about how we do it.) Understanding what it means to apply a bitwise operator to an entire string of bits is probably easiest to see with the shifting operators. By convention, in C and C++ you can think about binary numbers as starting with the most significant bit to the left (i.e., 10000000 is 128, and 00000001 is 1). Regardless of underlying representation, you may treat this as true. As a consequence, the results of the left and right shift operators are not implementation dependent for unsigned numbers (for signed numbers, the right shift operator is implementation defined).
Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p &amp; q</th>
<th>p</th>
<th>q</th>
<th>p ^ q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

-----------------
A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011
Example to understand all the bitwise operators available in C++.

```cpp
#include <iostream>
using namespace std;

main() {
    unsigned int a = 60;       // 60 = 0011 1100
    unsigned int b = 13;       // 13 = 0000 1101
    int c = 0;
    c = a & b;                 // 12 = 0000 1100
    cout << "Line 1 - Value of c is : " << c << endl;
    c = a | b;                 // 61 = 0011 1101
    cout << "Line 2 - Value of c is: " << c << endl;
    c = a ^ b;                 // 49 = 0011 0001
    cout << "Line 3 - Value of c is: " << c << endl;
    c = ~a;                    // -61 = 1100 0011
    cout << "Line 4 - Value of c is: " << c << endl;
    c = a << 2;                // 240 = 1111 0000
    cout << "Line 5 - Value of c is: " << c << endl;
    c = a >> 2;                // 15 = 0000 1111
    cout << "Line 6 - Value of c is: " << c << endl;
    return 0;
}
```

Output:
Line 1 - Value of c is : 12
Line 2 - Value of c is: 61
Line 3 - Value of c is: 49
Line 4 - Value of c is: -61
Line 5 - Value of c is: 240
Line 6 - Value of c is: 15
The Bitwise operators supported by C++ language are listed in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Binary AND Operator copies a bit to the result if it exists in both operands.</td>
<td>(A &amp; B) will give 12 which is 0000 1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Binary OR Operator copies a bit if it exists in either operand.</td>
</tr>
<tr>
<td>^</td>
<td>Binary XOR Operator copies the bit if it is set in one operand but not both.</td>
<td>(A ^ B) will give 49 which is 0011 0001</td>
</tr>
<tr>
<td>~</td>
<td>Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.</td>
<td>(~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.</td>
<td>A &lt;&lt; 2 will give 240 which is 1111 0000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.</td>
<td>A &gt;&gt; 2 will give 15 which is 0000 1111</td>
</tr>
</tbody>
</table>
Bitwise Operators

When should you use bitwise operators?
Bitwise operators are good for saving space -- but many times, space is hardly an issue. And one problem with working at the level of the individual bits is that if you decide you need more space or want to save some time -- for instance, if we needed to store information about 9 cars instead of 8 -- then you might have to redesign large portions of your program. On the other hand, sometimes you can use bitwise operators to cleverly remove dependencies, such as by using ~0 to find the largest possible integer. And bit shifting to multiply by two is a fairly common operation, so it doesn't affect readability in the way that advanced use of bit manipulation can in some cases (for instance, using XOR to switch the values stored in two variables).

There are also times when you need to use bitwise operators: if you're working with compression or some forms of encryption, or if you're working on a system that expects bit fields to be used to store Boolean attributes.
C++ provides the following classes to perform output and input of characters to/from files:

- ofstream: Stream class to write on files
- ifstream: Stream class to read from files
- fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile;
    myfile.open ("example.txt");
    myfile << "Writing this to a file.\n";
    myfile.close();
    return 0;
}
This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead. But let's go step by step:

1. **Open a file**
   The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.
   In order to open a file with a stream object we use its member function open:
   ```cpp
   open (filename, mode);
   ```
   Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:
<table>
<thead>
<tr>
<th>Sr.No</th>
<th>Mode Flag &amp; Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>ios::app</strong></td>
</tr>
<tr>
<td></td>
<td>Append mode. All output to that file to be appended to the end.</td>
</tr>
<tr>
<td>2</td>
<td><strong>ios::ate</strong></td>
</tr>
<tr>
<td></td>
<td>Open a file for output and move the read/write control to the end of the file.</td>
</tr>
<tr>
<td>3</td>
<td><strong>ios::in</strong></td>
</tr>
<tr>
<td></td>
<td>Open a file for reading.</td>
</tr>
<tr>
<td>4</td>
<td><strong>ios::out</strong></td>
</tr>
<tr>
<td></td>
<td>Open a file for writing.</td>
</tr>
<tr>
<td>5</td>
<td><strong>ios::trunc</strong></td>
</tr>
<tr>
<td></td>
<td>If the file already exists, its contents will be truncated before opening the file.</td>
</tr>
</tbody>
</table>
You can combine two or more of these values by ORing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax –

```cpp
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc);
```

Similar way, you can open a file for reading and writing purpose as follows –

```cpp
fstream afile;
afile.open("file.dat", ios::out | ios::in);
```
Input/ Output with Files

Closing a File
When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.
Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```cpp
void close();
```
Input/ Output with Files

**Writing to a File**

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

**Reading from a File**

You read information from a file into your program using the stream extraction operator (>>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

**Read and Write Example**

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −
```cpp
#include <fstream>
#include <iostream>
using namespace std;

int main () {
    char data[100];
    // open a file in write mode.
ofstream outfile;
    outfile.open("afile.dat");
cout << "Writing to the file" << endl;
cout << "Enter your name: ";
cin.getline(data, 100);
    // write inputted data into the file.
outfile << data << endl;
cout << "Enter your age: ";
cin >> data;
cin.ignore();
    // again write inputted data into the file.
outfile << data << endl;
    // close the opened file.
outfile.close();
    // open a file in read mode.
ifstream infile;
infile.open("afile.dat");
cout << "Reading from the file" << endl;
infile >> data;
    // write the data at the screen.
cout << data << endl;
    // again read the data from the file and display it.
infile >> data;
cout << data << endl;
    // close the opened file.
infile.close();
return 0;    }
```

Output:
$ ./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

**File Position Pointers**
Both istream and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ostream.
The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream.
The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are –
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );
// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );
// position at end of fileObject
fileObject.seekg( 0, ios::end );
C++ Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

**throw** – A program throws an exception when a problem shows up. This is done using a throw keyword.

**catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

**try** – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
C++ Exception Handling

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows –

```cpp
try {
    // protected code
} catch( ExceptionName e1 ) {
    // catch block
} catch( ExceptionName e2 ) {
    // catch block
} catch( ExceptionName eN ) {
    // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.
C++ Exception Handling

Throwing Exceptions
Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs –

```cpp
double division(int a, int b) {
    if( b == 0 ) {
        throw "Division by zero condition!";
    }
    return (a/b);
}
```
C++ Exception Handling

Catching Exceptions
The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```cpp
try {
    // protected code
    catch( ExceptionName e ) {
        // code to handle ExceptionName exception
    }
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows –

```cpp
try {
    // protected code
} catch(...) {
    // code to handle any exception
}
```
The following is an example, which throws a division by zero exception and we catch it in catch block.

```cpp
#include <iostream>
using namespace std;
double division(int a, int b) {
    if (b == 0) {
        throw "Division by zero condition!";
    }
    return (a/b);
}
int main () {
    int x = 50;
    int y = 0;
    double z = 0;
    try {
        z = division(x, y);
        cout << z << endl;
    } catch (const char* msg) {
        cerr << msg << endl;
    }
    return 0;
}

Because we are raising an exception of type const char*, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result –

Division by zero condition!
```
C++ Standard Exceptions
C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below –

![Diagram showing the hierarchy of standard exceptions in C++](image)
Here is the small description of each exception mentioned in the above hierarchy –

<table>
<thead>
<tr>
<th>Sr.No</th>
<th>Exception &amp; Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>std::exception An exception and parent class of all the standard C++ exceptions.</td>
</tr>
<tr>
<td>2</td>
<td>std::bad_alloc This can be thrown by new.</td>
</tr>
<tr>
<td>3</td>
<td>std::bad_cast This can be thrown by dynamic_cast.</td>
</tr>
<tr>
<td>4</td>
<td>std::bad_exception This is useful device to handle unexpected exceptions in C++ program.</td>
</tr>
<tr>
<td>5</td>
<td>std::bad_typeid This can be thrown by typeid.</td>
</tr>
<tr>
<td>6</td>
<td>std::logic_error An exception that theoretically can be detected by reading the code.</td>
</tr>
<tr>
<td>7</td>
<td>std::domain_error This is an exception thrown when a mathematically invalid domain used.</td>
</tr>
<tr>
<td>8</td>
<td>std::invalid_argument This is thrown due to invalid arguments.</td>
</tr>
<tr>
<td>9</td>
<td>std::length_error This is thrown when a too big std::string is created.</td>
</tr>
<tr>
<td>10</td>
<td>std::out_of_range This can be thrown by the 'at' method, for example a std::vector and std::bitset&lt;&gt;::operator<a href=""></a>.</td>
</tr>
<tr>
<td>11</td>
<td>std::runtime_error An exception that theoretically cannot be detected by reading the code.</td>
</tr>
<tr>
<td>12</td>
<td>std::overflow_error This is thrown if a mathematical overflow occurs.</td>
</tr>
<tr>
<td>13</td>
<td>std::range_error This is occurred when you try to store a value which is out of range.</td>
</tr>
<tr>
<td>14</td>
<td>std::underflow_error This is thrown if a mathematical underflow occurs.</td>
</tr>
</tbody>
</table>
Define New Exceptions
You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way –

```cpp
#include <iostream>
#include <exception>
using namespace std;
struct MyException : public exception {
    const char * what () const throw () {
        return "C++ Exception";
    }
};
int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {
        //Other errors
    }
}
```

This would produce the following result –

```
MyException caught
C++ Exception
```

Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.