Bitwise and bit-shift operators

Outline

- In this presentation, we will:
  - Introduce bitwise logical operations
  - Contrast these with Boolean logical operations
  - Describe
    - The bitwise EXCLUSIVE OR operator in addition to bitwise AND and OR
    - The unary bitwise NOT or complement operator
  - Describe left and right shift operators

Logical operators

- We have seen two logical operators:
  - The binary logical AND operator and the binary logical OR operator
  - Their behavior is defined by the values of the operands:

| x   | y   | x && y | x || y |
|-----|-----|--------|--------|
| false | false | false  | false  |
| false | true  | false  | true   |
| true  | true  | true   | true   |
| true  | false | false  | true   |

- Recall that any zero value is false, while any non-zero value is true
  - true and false have the values 1 and 0, respectively

Primitive types

- Recall that primitive types are a fixed number of bits
  - Given any two bits, we could define

<table>
<thead>
<tr>
<th>b1</th>
<th>c2</th>
<th>b1 AND c2</th>
<th>b1 OR b2</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Bitwise AND operator

- There are three binary bitwise operators in C++
  - Given any two operands of the same type, the bitwise AND operator & compares the corresponding pairs of bits
  - Each result is 1 only if both bits are also 1

```cpp
00100100101010010100101001010100
& 0100101010111111000001
0000000101010101000000
```

Output:
```
m = 615074388
n = 1253003073
m & n = 11094592
```

Bitwise OR operator

- The second is bitwise OR operator |
  - Given any two operands of the same type a logical OR to each corresponding pair of bits
  - Each result is 0 only if both bits are also 0

```cpp
01010010101010010100101001010100
| 0100101010111111000001
011011110101100111110101010101
```

Output:
```
m = 615074388
n = 1253003073
m | n = 1856982869
```

Bitwise AND operator

- Like arithmetic operations, the bitwise AND of any pair of bits does not affect the operands

```cpp
#include <iostream>

int main()
```

```cpp
int m = 615074388;
int n = 1253003073;
unsigned int m1 = 0b01010010101010010100101001010100;
unsigned int n1 = 0b1001010101111110000001;
std::cout << "m = " << m1 << std::endl;
std::cout << "n = " << n1 << std::endl;
std::cout << "m & n = " << (m1 & n1) << std::endl;
return 0;
```
Bitwise EXCLUSIVE-OR operator

- The third is bitwise XOR operator
  - This has no equivalent binary logical operator
  - For this result to be true, one but not both operands must be true

<table>
<thead>
<tr>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_1 \text{ AND } b_2 )</th>
<th>( b_1 \text{ OR } b_2 )</th>
<th>( b_1 \text{ XOR } b_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Bitwise XOR operator

- Like arithmetic operations, the bitwise XOR of any pair of bits does not affect the operands

```cpp
#include <iostream>

int main()
{
    unsigned int m = 0b00100100101010010100101001010100;
    unsigned int n = 0b010010101110110100111101000001;

    std::cout << "m = " << m << std::endl;
    std::cout << "n = " << n << std::endl;
    std::cout << "m ^ n = " << (m ^ n) << std::endl;

    return 0;
}
```

Output:

```
m = 615074388
n = 1253035873
m ^ n = 1845888277
```

Bitwise XOR operator

- The third is bitwise XOR operator ^
  - This has no equivalent binary logical operator
  - If both bits have the same value, the result is 0, otherwise it is 1

\[
\begin{array}{c|c|c|c}
00101011010101010101010101010101 & 010011011101101111011011000001010101010101010101000111011010011101000001
\end{array}
\]

Automatic bitwise assignment

- For each binary bitwise operator, there is an automatic assignment operator:

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Automatic assignment</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = a &amp; 32</td>
<td>a &amp;= 32</td>
<td>auto bitwise AND</td>
</tr>
<tr>
<td>b = b</td>
<td>41</td>
<td>b</td>
</tr>
<tr>
<td>c = 2 ^ c</td>
<td>c ^= 2</td>
<td>auto bitwise XOR</td>
</tr>
</tbody>
</table>

- Note: there are no Boolean automatic assignment operators
  - The operators && and || do not exist in C++
Unary bitwise NOT operator

- A unary bitwise operator is the NOT operator ~
  - It is equivalent to applying the logical NOT operator ~ to each bit

```
~ 0100101010111101000000011011010100000101111110
101101010100001011000010111110
```

Application of bitwise operators

- Bitwise operators allow the manipulation of individual bits
  - Suppose this local variable has exactly one 1 bit
    unsigned int MASK(256); // @00000000000000000000000000000100
  - Suppose n is any unsigned integer value:
    unsigned int n();
    std::cin >> n;
    Bit 0
  - We can set the 8th bit of n to 1:
    n |= MASK;
  - We can set the 8th bit of n to 0:
    n &= ~MASK;
  - We can flip the 8th bit of n between 0 and 1:
    n ^= MASK;
  - We can have a condition that is true if the 8th bit is 1:
    if (n & MASK) {
      // Do something if the 8th bit is 1
    }

Bitwise AND operator

- Like arithmetic operations, the bitwise AND of any pair of bits does not affect the operands

```
#include <iostream>

int main()
```

```
unsigned int m(00000000000010001011000010111110);
std::cout << "m = " << m << std::endl;
std::cout << "n = " << ~m << std::endl;
return 0;
```

Output:
```
615874388
3679892907
```

Application of bitwise operators

```
#include <iostream>

int main()
```

```
unsigned short MASK_8;
unsigned short MASK_16;
unsigned short MASK_32;
unsigned short MASK_64;

int main()
```

```
unsigned short m(00000000000000000000000000000000);
unsigned short mask_8(00000000000000000000000000000100);
unsigned short mask_16(00000000000000000000000000010000);
unsigned short mask_32(00000000000000000000000001000000);
unsigned short mask_64(00000000000000000000000010000000);
```

```
std::cout << "Enter a positive integer: ";
std::cin >> m;
if (m & mask_8) {
  std::cout << "Bit 8 is '1'" << std::endl;
} else {
  std::cout << "Bit 8 is '0'" << std::endl;
}
```

```
if (m & mask_16) {
  std::cout << "Bit 16 is '1'" << std::endl;
} else {
  std::cout << "Bit 16 is '0'" << std::endl;
}
```

```
if (m & mask_32) {
  std::cout << "Bit 32 is '1'" << std::endl;
} else {
  std::cout << "Bit 32 is '0'" << std::endl;
}
```

```
if (m & mask_64) {
  std::cout << "Bit 64 is '1'" << std::endl;
} else {
  std::cout << "Bit 64 is '0'" << std::endl;
}
```

return 0;
```
Bitwise and bit-shift operators

- There are two operators that literally shift bits left or right:
  - The left-shift operator `<<` evaluates to the bits of the operand `op` shifted to the left by `n` bits
    ```c
    unsigned int op; // n is any non-negative integer
    unsigned int q1{ op << n }; // n is any non-negative integer
    ```
  - The right-shift operator `>>` evaluates to the bits of the operand `op` shifted to the right by `n` bits
    ```c
    unsigned int q2{ op >> n }; // n is any non-negative integer
    ```
- Any bits shifted beyond the last position are lost

Automatic bit-shift assignment

- There are two automatic bit-shift operators
  - Shift the bits in the operand `op` to the left by `n` bits
    ```c
    op <<= n;
    ```
  - Shift the bits in the operand `op` to the right by `n` bits
    ```c
    op >>= n;
    ```

Bitwise and bit-shift operators

- Examples:
  - If `op` is four bytes and has the value
    ```c
    00100100111100100011001010100
    ```
    - The result of `op >> 5` is
      ```c
      00000000000000000001001111001010
      ```
    - The result of `op >> 12` is
      ```c
      0000000000000000000000000000000000010111110010101
      ```
    - The result of `op << 8` is
      ```c
      11111100101011010101000000000000
      ```
    - The result of `op << 13` is
      ```c
      00101001110010101000000000000000
      ```

Application of bit-shift operators

- Bit-shift operators can be used to precisely read or place bits
  - In our next example, we will use bit shifting and bitwise `AND` to print a number to the screen in binary
Application of bit-shift operators

```cpp
#include <iostream>

int main()
{
    unsigned int n;
    std::cout << "Enter a positive integer: ";
    std::cin >> n;
    for (unsigned int k(1 << 31); k > 0; k >>= 1)
    {
        if (n & k)
        {
            std::cout << "1";
        }
        else
        {
            std::cout << "0";
        }
    }
    std::cout << std::endl;
    return 0;
}
```

Summary

• In this presentation, you now
  – Are aware of bitwise and bit-shifting operators
  – Understand the behavior of these operators
  – Understand the automatic operators corresponding to these
    • There are no &&= or ||= operators

Summary of operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Binary</th>
<th>Unary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+ - * / %</td>
<td>+ -</td>
</tr>
<tr>
<td>Comparison</td>
<td>&lt;= == !=</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>bitwise</td>
<td>&amp;</td>
<td>^</td>
</tr>
<tr>
<td>Bit shift</td>
<td>&lt;&lt; &gt; &gt;</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>Arithmetic auto-assignment</td>
<td>+= -= *= /= %=</td>
<td>++ --</td>
</tr>
<tr>
<td>Bitwise auto-assignment</td>
<td>&amp;=</td>
<td>= ^=</td>
</tr>
<tr>
<td>Bit-shift auto-assignment</td>
<td>&lt;&lt;= &gt;&gt;=</td>
<td></td>
</tr>
</tbody>
</table>

References

[1] No references?
Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using Consolas.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see https://www.rbg.ca/ for more information.

Disclaimer

These slides are provided for the ECE 150 Fundamentals of Programming course taught at the University of Waterloo. The material in it reflects the authors’ best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.