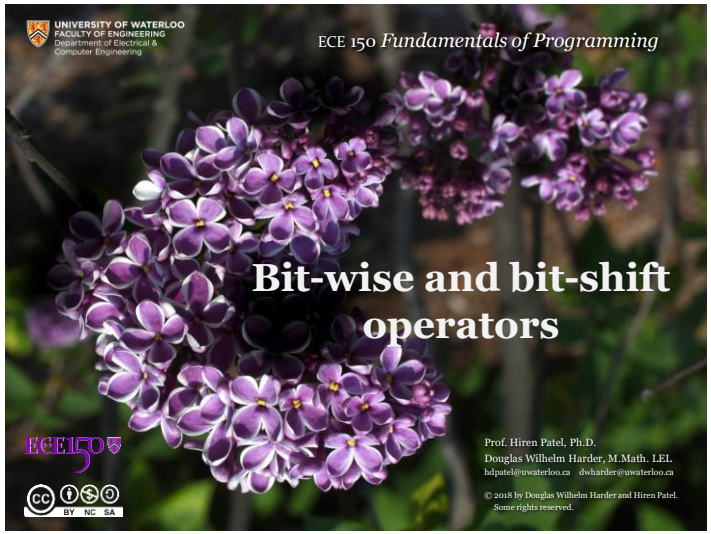

Outline

- In this presentation, we will:
  - Bitwise logical operations versus Boolean logical operations
  - Introduce
    - The binary EXCLUSIVE OR operator in addition to AND and OR
    - The unary NOT operator
  - Integrated development environments and on-line compilers

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

b_1	b_2	b_1 AND b_2	b_1 OR b_2
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

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

Bit-wise AND operator

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

```
  00100100101010010010010010101000
& 01001010101011110100111101000001
  00000000101010010010010010000000
```

  - The bit-wise AND of any pair of bits does not affect any other result

```
int m{615074388};
int n{1253003073};
std::cout << (m & n) << std::endl;
```


Bit-wise OR operator

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

```
  00100100101010010010010010101000
| 01001010101011110100111101000001
  01101110101011110100111101010101
```

```
int m{615074388};
int n{1253003073};
std::cout << (m | n) << std::endl;
```


Bit-wise EXCLUSIVE-OR operator

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

b_1	b_2	b_1 AND b_2	b_1 OR b_2	b_1 XOR b_2
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Bit-wise XOR operator

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

```
  00100100101010010010010010101000
^ 01001010101011110100111101000001
  01101110000000110000001010001010 1
```

```
int m{615074388};
int n{1253003073};
std::cout << (m ^ n) << std::endl;
```

Automatic bit-wise assignment

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

Assignment	Automatic assignment	Name
`a = a & 32`	`a &= 32`	auto AND
`b = b | 41`	`b |= 41`	auto OR
`c = 2 ^ c`	`c ^= 2`	auto XOR

n.b., there are no Boolean automatic assignment operators
  - The operators `&&=` and `||=` do not exist in C++

Unary bit-wise NOT operator

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

```
~ 01001010101011110100111101000001
  10110101010100001011000010111110
```

```
int n{1253003073};
std::cout << (~n) << std::endl;

std::cout << (-n) << std::endl;
std::cout << ((~n) + 1) << std::endl;
```


Application of bit-wise operators

- One significant application of bit-wise operators is the manipulation of individual bits within a primitive type

- Consider this local variable:

```
unsigned int MASK512{0b00000000000000000000001000000000};
// Also, either unsigned int MASK512{512};
//           or unsigned int MASK512{0x200};
```

- Given any other unsigned integer `m` where m_9 is the 9th bit:

Operation	Description
`m & MASK512`	Equal `0` if m_9 is `0` and `MASK512` if m_9 is `1`
`m | MASK512`	Equals `m` with m_9 set to `1`
`m ^ MASK512`	Equals `m` with the value of m_9 flipped
`m & (~MASK512)`	Equals `m` with m_9 set to `0`

Bit-shift operators

- There are two operators that literally shift bits left or right:
  - Return the bits of the operand `op` shifted to the left by `n` bits
    `op << n`
  - Return the bits of the operand `op` shifted to the right by `n` bits
    `op >> n`

- Any bits shifted beyond the last position are lost
- The amount to be shifted must be positive
  - The operand `n` will be interpreted as an unsigned integer

Bit-shift operators

- Examples:
  - If op is four bytes and has the value
    00100100111110010100111001010100
  - The result of `op >> 5` is
    00000001001001111100101001110010
  - The result of `op >> 12` is
    00000000000001001001111100101000
  - The result of `op << 8` is
    11111001010011100101010000000000
  - The result of `op << 13` is
    00101001110010101000000000000000

Automatic bit-shift assignment

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

Application of bit-shift operators

- There are two common applications of bit-shift operators:
  - A fast unsigned integer multiplication or division by 2^m:

```
unsigned int n{3235};
//                 3
// Multiplying n by 2
n = n << 3;

//               5
// Dividing n by 2  using integer division
//  - the remainder is discarded
n = n >> 5;
```

  - This does not work if n is signed and negative :

Application of bit-shift operators

- There are two common applications of bit-shift operators:
  - Creating a type with the n^{th} bit set to 1

```
// Create the unsigned integer 0b00…00100000
unsigned int m{1 << 5};
```

  - This is also a fast way of generating an integer equal to 2^nequal
    - The initialized value of m is $2^5 = 32$
- Note: if you are dealing with bits, don't think of `1 << 23` as having the value 8388608, but rather, just think of it as four bytes with a 1 in the 23rd location
  - Is 135217728 a power of two, and if so, which power of two?

Auto-assignment bit-wise and bit-shift operators

- As with addition, there are corresponding auto-assignment operators for both bit-wise and bit-shift operators

```
a = a & b;          a &= b;
a = a | b;          a |= b;
a = a ^ b;          a ^= b;
a = a << n;         a <<= n;
a = a >> n;         a >>= n;
```

- Very important: there are no auto-assignment operators for the logical operators && and ||

Applications of auto-assignment operators

- Common applications of auto-assignment operators include:
  - Bit manipulation

```
// The 9th bit is set to 1
unsigned int MASK_9{1 << 9};
unsigned int flags{0};
```

Operation	Description
`flags |= MASK9`	Set the 9th bit of `flags` to `1`
`flags ^= MASK9`	Flip the 9th bit of `flags`
`flags &= (~MASK9)`	Set the 9th bit of `flags` to `0`

Applications of auto-assignment operators

- Common applications of auto-assignment operators include:
  - Multiplication and integer division by powers of two

```
unsigned int m{53234};
//                  3
// Multiply 'm' by 2  = 8
m <<= 3;
// 'm' is now assigned 425872

//                 13
// Divide 'm' by 2   = 8192 using integer division
m >>= 13;
// 'm' is now assigned 51
```


Summary of operators

- To summarize our knowledge of operators

Operator	Binary	Unary
Arithmetic	`+ - * / %`	`+ -`
Comparison	`< <= == != >= >`	
Logical	`&& ||`	`!`
Bit-wise	`& | ^`	`~`
Bit-shift	`<< >>`	
Assignment	`=`	
Pointer		`* &`
Structure	`. ->`	
Arithmetic auto-assignment	`+= -= *= /= %=`	`++ --`
Bit-wise auto-assignment	`&= |= ^=`	
Bit-shift auto-assignment	`<<= >>=`	

Summary

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

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