**Bit-Wise Expressions**

Bit-wise expressions provide the mechanism for accessing individual bits within a type and creating new values by operating on the individual bits of the operands.  Each bit-wise expression has its own value and type.  The C++ bit-wise operators define the language facilities for accessing and manipulating the bits within the bytes of integral and enumerated types.  These operators follow the rules of precedence of the language.

This chapter reviews the precedence rules for the bit-wise operators and describing the bit-wise expressions that C++ supports on integral and enumerated types.  This chapter also shows how to declare bit fields in a class type.

Operator Precedence

Order of Evaluation

A compound expression evaluates according to rules defined through the precedence table below.  These rules cannot be changed.  The compiler evaluates the expression with the operator that has a higher precedence before evaluating any other expression.  Some operators associate from left to right, while other operators associate from right to left.  The bit-wise operators are highlighted below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Precedence | Class | Operator | Description | Associates |
| 18 | primary |  | name, literal |  |
| binary | :: | scope resolution |  |
| 17 | postfix | [] | array subscripting | left-to-right |
| (   ) | function call | left-to-right |
| . | direct selection | left-to-right |
| -> | indirect selection | left-to-right |
| ++ -- | postfix increment, decrement | left-to-right |
| typeid() | type name | left-to-right |
| const\_cast | constant type conversion | left-to-right |
| dynamic\_cast | dynamic type conversion | left-to-right |
| reinterpret\_cast | reinterpreted type conversion | left-to-right |
| static\_cast | static type conversion | left-to-right |
| 16 | prefix | ++ -- | prefix increment, decrement | right-to-left |
| unary | sizeof | size | right-to-left |
| ! | logical negation | right-to-left |
| ~ | bit-wise negation | right-to-left |
| - + | arithmetic negation, plus | right-to-left |
| & | address of | right-to-left |
| \* | indirection | right-to-left |
| new, new[] | dynamic memory allocation | right-to-left |
| delete, delete [] | dynamic memory deallocation | right-to-left |
| alignof(type) | alignment requirement | right-to-left |
| noexcept() | controls exception throwing | right-to-left |
| (type) | type casting | right-to-left |
| 15 | binary | .\* ->\* | object to member pointer pointer to member pointer | left-to-right |
| 14 | \* / % | multiplicative | left-to-right |
| 13 | + - | additive | left-to-right |
| 12 | << >> | left and right bit shifting | left-to-right |
| 11 | < > <= >= | relational | left-to-right |
| 10 | == != | equality, inequality | left-to-right |
| 9 | & | bit-wise logical and | left-to-right |
| 8 | ^ | bit-wise logical exclusive or | left-to-right |
| 7 | | | bit-wise logical or | left-to-right |
| 6 | && | logical and | left-to-right |
| 5 | || | logical or | left-to-right |
| 4 | ternary | ?: | conditional | right-to-left |
| 3 | binary | = += -= \*= /= %= | compound type-wise assignment | right-to-left |
| ~= &= ^= |= >>= <<= | compound bit-wise assignment | right-to-left |
| 2 | unary | throw | transfers control to exception handler | left-to-right |
| 1 | binary | , | sequencing | left-to-right |

To override the order defined in this table, we enclose the expressions to be evaluated first within parentheses.

Unary Expression

Bit-Wise Negation

The bit-wise operator (~) converts its operand to its one's complement.  The operand may be of integral or unscoped enumerated type.  The result of the operation is a prvalue.  For instance, applying the negation operator to the following bit pattern

|  |
| --- |
| 01011001 11001101 11101011 11100010 |

yields

|  |
| --- |
| 10100110 00110010 00010100 00011101 |

The following program yields the output shown on the right:

|  |  |
| --- | --- |
| // Bitwise Expressions - Negation  // negation.cpp  #include <iostream>  #include <iomanip>  using namespace std;  typedef unsigned char uc;  typedef signed char sc;  typedef unsigned short us;  typedef signed short ss;  typedef unsigned int ui;  typedef signed int si;  int main () {  ui u = 27u;  us s = (us)27u;  uc c = (uc)27;  si i = 27;  ss t = (ss)27;  sc d = (sc)27;  cout << "u:" << u << ",~u:" << setw(10) <<  ~u << ",~~u:" << setw(3) << ~~u << endl;  cout << "s:" << s << ",~s:" << setw(10) <<  (us)~s << ",~~s:" << setw(3) << ~~s << endl;  cout << "c:" << (ui)c << ",~c:" << setw(10)  << (ui)(uc)~c << ",~~c:" << setw(3) << ~~c  << endl;  cout << "i:" << i << ",~i:" << setw(10) <<  ~i << ",~~i:" << setw(3) << ~~i << endl;  cout << "t:" << t << ",~t:" << setw(10) <<  ~t << ",~~t:" << setw(3) << ~~t << endl;  cout << "d:" << (si)d << ",~d:" << setw(10)  << ~d << ",~~d:" << setw(3) << ~~d << endl;  } | u:27,~u:4294967268,~~u:27  s:27,~s: 65508,~~s:27  c:27,~c: 228,~~c:27  i:27,~i: -28,~~i:27  t:27,~t: -28,~~t:27  d:27,~d: -28,~~d:27 |

Negation is self-inverting: the negation of the negation of a value is the original value.

The negation of an unsigned variable is the largest number that an unsigned type can store less its original value.

Good Design Practice

Representation of negative integers is not defined in the language standard and different platforms may use different rules.  Since the result of a bit-wise negation on a signed integral value is not fully portable, it is better to limit the use of bit-wise negation expressions to unsigned integral types.

Binary Expressions

Binary expressions consist of two operands and one operator.  They include bit-shifting, bit-wise and, bit-wise or and bit-wise exclusive or operators.  They evaluate to prvalues.

Bit-Shifting

Bit-shifting expressions shift the bit pattern of the left operand towards its left or right end.  The right operand specifies the number of bit positions by which to shift the pattern.  The left operand is of integral or unscoped enumeration type;  The right operand is of integral or unscoped enumeration type and non-negative in value.  The expression evaluates to the result, is of the same type as the left operand, and is a prvalue.

Left-Shift Operator

The left-shift operator (<<) shifts the bits in the left operand to the left by the value of the right operand and fills the vacated bits with 0 values.

For example,

|  |  |
| --- | --- |
| // Bitwise Expressions - Left Shift  // left.cpp  #include <iostream>  int main () {  unsigned u = 27u; // 0...00011011  std::cout << u << " << " << 2 << " = " << (u << 2)  << std::endl;  } | 27 << 2 = 108 |

For an unsigned left operand, the result is the left operand multiplied by 2 to the power of the right operand.

Right-Shift Operator

The right-shift operator (>>) shifts the bits in the left operand to the right by the value of the right operand and fills the vacated bits with the value of the highest order bit.

For example,

|  |  |
| --- | --- |
| // Bitwise Expressions - Right Shift  // right.cpp  #include <iostream>  #include <climits>  int main () {  unsigned u, v;  int w;  u = 27u; // 0...0011011  v = 1u << sizeof(unsigned) \* CHAR\_BIT - 1;  // 10..0000000  w = -1; // 1...1111111  std::cout << u << " >> " << 2 << " = "  << (u >> 2) << std::endl;  std::cout << v << " >> " << 8 << " = "  << (v >> 8) << std::endl;  std::cout << w << " >> " << 2 << " = "  << (w >> 2) << std::endl;  } | 27 >> 2 = 6  2147483648 >> 8 = 8388608  -1 >> 2 = -1 |

CHAR\_BIT holds the number of bits in a byte and is defined in climits.

For an unsigned left operand, the result is the left operand divided by 2 to the power of the right operand.

Multiplying and Dividing by Powers of 2

Bit-shifting expressions provide fast multiplication or division by powers of 2.  That is,

|  |
| --- |
| 27 << 0 is equal to 27 \* 1 = 27 // 0...00011011  27 << 1 is equal to 27 \* 2 = 54 // 0...00110110  27 << 2 is equal to 27 \* 4 = 108 // 0...01101100  27 << 3 is equal to 27 \* 8 = 216 // 0...11011000 |

and

|  |
| --- |
| 27 >> 0 is equal to 27 / 1 = 27 // 0...00011011  27 >> 1 is equal to 27 / 2 = 13 // 0...00001101  27 >> 2 is equal to 27 / 4 = 6 // 0...00000110  27 >> 3 is equal to 27 / 8 = 3 // 0...00000011 |

Bit-Wise and

The bit-wise 'and' operator (&) compares every bit of the left and right operands and returns a value of the same type that consists of the bit by bit results of the comparison.  The operands are of integral or unscoped enumeration type.  The rules for an 'and' comparision are:

* if both bit values are 1, the resultant bit value is 1
* otherwise, the resultant bit value is 0

Masking

We can use bit-wise 'and' expressions to extract individual bits from a variable.  Consider the following two operands:

|  |
| --- |
| 01011001 11001101 11101011 11100010  00000000 00000000 00001111 00000000 |

The bit-wise 'and' operator applied to these two operands yields the following result

|  |
| --- |
| 00000000 00000000 00001011 00000000 |

That is, we retrieve the 8th-11th bit values (the rightmost bit being bit 0) of the left operand.  The right operand ignores all but bits 8 through 11.  We call the operand that defines the bits to extract the *mask*.

The following example yields the output shown on the right

|  |  |
| --- | --- |
| // Bit-wise and  // and.cpp  #include <iostream>  int main () {  unsigned char u = 27u; // 00011011 = 27  unsigned char v = 14u; // 00001110 = 14 - mask  // -------------  // 00001010 = 10  std::cout << (unsigned)u << " & " << (unsigned)v  << " = " << (u & v) << std::endl;  } | 27 & 14 = 10 |

The parentheses are necessary since the insertion operator (<<) is of higher precedence than the bit-wise 'and' operator.

Oddness Test

The bit-wise 'and' operator with a right operand of 1 returns the oddness of the left operand:

|  |
| --- |
| bool odd = (bool)(value & 1) |

odd has the value true if value is odd, false if value is even.

Fast Remainder

The bit-wise 'and' operator can extract the remainder of a division by powers of 2 more efficiently than the modulus operator.  The bit-wise 'and' expression on a left operand with the right operand set to one less than the divisor gives the remainder directly

|  |
| --- |
| unsigned char u = 27u; // 00011011  27 & (1 - 1) is equal to 00000000 or 0 which is 27 % 1  27 & (2 - 1) is equal to 00000001 or 1 which is 27 % 2  27 & (4 - 1) is equal to 00000011 or 3 which is 27 % 4  27 & (8 - 1) is equal to 00000011 or 3 which is 27 % 8 |

Bit-Wise or

The bit-wise 'or' operator (|) compares every bit of the left and right operands and returns a value of the same type that is the bit by bit result of the comparison.  The operands are of integral or unscoped enumeration type.  The rules for an 'or' comparision are:

* if either bit value is 1, the resultant bit value is 1
* otherwise, the resultant bit value is 0

Turn-on a Bit

A bit-wise 'or' expression can turn on an individual bit or a set of bits in a variable.  Consider the following two operands:

|  |
| --- |
| 01011001 11001101 11101011 11100010  00000000 00000000 00001111 00000000 |

The bit-wise 'or' operator applied to these operands yields the result

|  |
| --- |
| 01011001 11001101 11101111 11100010 |

The following program produces the output shown on the right:

|  |  |
| --- | --- |
| // Bit-wise or  // or.cpp  #include <iostream>  int main () {  unsigned char u = 27u; // 00011011 = 27  unsigned char v = 14u; // 00001110 = 14  // -------------  // 00011111 = 31  std::cout << (unsigned)u << " | " << (unsigned)v  << " = " << (u | v) << std::endl;  } | 27 | 14 = 31 |

Bit-Wise Exclusive or

The bit-wise 'exclusive or' operator (^) compares every bit of the left and right operands for exclusivity.  The operands are of integral or unscoped enumeration type.  The rules for an 'exclusive-or' comparison are:

* if the bit values are different, the resultant bit value is 1
* if the bit values are the same, the resultant bit value is 0

Flipping Selected Bits

A bit-wise 'exclusive or' expression to flip an individual bit or a set of bit values in a variable.  Consider the following two operands:

|  |
| --- |
| 01011001 11001101 11101011 11100010  00000000 00000000 00001111 00000000 |

The bit-wise 'exclusive-or' operator applied to these operands yields the result:

|  |
| --- |
| 01011001 11001101 11100100 11100010 |

The following program produces the output shown on the right:

|  |  |
| --- | --- |
| // Bitwise Exclusive or  // xor.cpp  #include <iostream>  int main () {  unsigned char u = 27u; // 00011011  unsigned char v = 14u; // 00001110  // --------  // 00010101  std::cout << (unsigned)u << " ^ " << (unsigned)v  << " = " << (u ^ v) << std::endl;  std::cout << (unsigned)u << " ^ " << (unsigned)v  << " ^ " << (unsigned)v << " = "  << (u ^ v ^ v) << std::endl;  } | 27 ^ 14 = 21  27 ^ 14 ^ 14 = 27 |

The 'exclusive or' operator is self-inverting.

Assignment Expressions

An assignment expression copies from the right operand to the left one.  The left operand must be a modifiable lvalue.  Assignment expressions associate from right to left, which enables cascading.  They may be simple or compound.

Compound Assignment

The compound operators (>>= <<= ~= &= |= ^=) perform the operation specified by the first character(s) in the operator on the two operands before copying the result into the left operand.

Bit Fields

A member declaration of a class type can include a bit field.  C++ lets us allocate a specific number of bits to a data member, provided that that member is of integral or unscoped enumeration type.  The declaration of a bit field takes the form

|  |
| --- |
| *Type identifier* : *constant-expression*; |

*constant-expression* is positive-valued and specifies the number of bits to be allocated.  C++ does not allow a non-const reference to a bit field.

To include bit-padding between adjacent data members, we insert an unnamed bit field that specifies the number of padding bits:

|  |
| --- |
| class A {  unsigned a : 5;  unsigned : 3; // padding  unsigned b : 6;  }; |

To ensure alignment of the next member at an alignment boundary, we insert a bit field of 0:

|  |
| --- |
| class A {  unsigned a : 5;  unsigned : 0; // align b at alignment boundary  unsigned b : 6;  }; |

Bit-field allocation is implementation defined.  Packing strategies are typically implementation-dependent and not portable.  Each platform has

1. its own constraints on the alignment of bit fields and members
2. its own maximum size that a bit field cannot exceed
3. its own method of packing several bit fields