Left Shift and Right Shift Operators (>> and <<)

Visual Studio 2015

The latest version of this topic can be found at Left Shift and Right Shift Operators (>> and <<).

The bitwise shift operators are the right-shift operator (>>), which moves the bits of `shift_expression` to the right, and the left-shift operator (<<), which moves the bits of `shift_expression` to the left. [1]

Syntax

```
      shift-expression << additive-expression
shift-expression >> additive-expression
```

Remarks

Important

The following descriptions and examples are valid on Windows for X86 and x64 architectures. The implementation of left-shift and right-shift operators is significantly different on Windows RT for ARM devices. For more information, see the "Shift Operators" section of the Hello ARM blog post.

Left Shifts

The left-shift operator causes the bits in `shift-expression` to be shifted to the left by the number of positions specified by `additive-expression`. The bit positions that have been vacated by the shift operation are zero-filled. A left shift is a logical shift (the bits that are shifted off the end are discarded, including the sign bit). For more information about the kinds of bitwise shifts, see Bitwise shifts.

The following example shows left-shift operations using unsigned numbers. The example shows what is happening to the bits by representing the value as a bitset. For more information, see bitset Class.

C++

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

```cpp
int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1};   // the bitset representation of 4
    cout << bitset1 << endl;  // 0000000000000100

    unsigned short short2 = short1 << 1;     // 4 left-shifted by 1 = 8
    bitset<16> bitset2{short2};
    cout << bitset2 << endl;  // 0000000000001000

    unsigned short short3 = short1 << 2;     // 4 left-shifted by 2 = 16
    bitset<16> bitset3{short3};
    cout << bitset3 << endl;  // 0000000000010000
}
```

If you left-shift a signed number so that the sign bit is affected, the result is undefined. The following example shows what happens in Visual C++ when a 1 bit is left-shifted into the sign bit position.

C++

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

int main() {
    short short1 = 16384;
    bitset<16> bitset1{short2};
    cout << bitset1 << endl;  // 0100000000000000

    short short3 = short1 << 1;
    bitset<16> bitset3{short3};  // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl;  // 100000000000000

    short short4 = short1 << 14;
    bitset<16> bitset4{short4};  // 4 left-shifted by 14 = 0
    cout << bitset4 << endl;  // 000000000000000
}
```

Right Shifts

The right-shift operator causes the bit pattern in `shift-expression` to be shifted to the right by the number of positions specified by `additive-expression`. For unsigned numbers, the bit positions that have been vacated by the shift operation are zero-filled. For signed numbers, the sign bit is used to fill the vacated bit positions. In other words, if the number is positive, 0 is used, and if the number is negative, 1 is used.

Important

The result of a right-shift of a signed negative number is implementation-dependent. Although Visual C++ uses the sign bit to fill vacated bit positions, there is no guarantee that other implementations also do so.

This example shows right-shift operations using unsigned numbers:

C++

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

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl;     // 0000010000000000

    unsigned short short12 = short11 >> 1;  // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl;     // 0000001000000000

    unsigned short short13 = short11 >> 10;  // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl;     // 0000000000000001

    unsigned short short14 = short11 >> 11;  // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl;     // 0000000000000000}
}
```

The next example shows right-shift operations with positive signed numbers.

C++

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

int main() {
    short short1 = 1024;
    bitset<16> bitset1{short1};
    cout << bitset1 << endl;     // 0000010000000000

    short short2 = short1 >> 1;  // 512
    bitset<16> bitset2{short2};
    cout << bitset2 << endl;     // 0000001000000000

    short short3 = short1 >> 11;  // 0
    bitset<16> bitset3{short3};
    cout << bitset3 << endl;     // 0000000000000000
}
```

The next example shows right-shift operations with negative signed integers.

C++

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

int main() {
    short neg1 = -16;
    bitset<16> bn1{neg1};
```

```cpp
    cout << bn1 << endl;  // 1111111111110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2{neg2};
    cout << bn2 << endl;  // 1111111111111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3{neg3};
    cout << bn3 << endl;  // 1111111111111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4{neg4};
    cout << bn4 << endl;  // 1111111111111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5{neg5};
    cout << bn5 << endl;  // 1111111111111111
}
```

Shifts and Promotions

The expressions on both sides of a shift operator must be integral types. Integral promotions are performed according to the rules described in Integral Promotions. The type of the result is the same as the type of the promoted `shift-expression`.

In the following example, a variable of type `char` is promoted to an `int`.

C++

```cpp
#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1;  // 194
    cout << typeid(promoted1).name() << endl;  // int

    auto promoted2 = char1 << 10;  // 99328
    cout << typeid(promoted2).name() << endl;   // int
}
```

Additional Details

The result of a shift operation is undefined if `additive-expression` is negative or if `additive-expression` is greater than or equal to the number of bits in the (promoted) `shift-expression`. No shift operation is performed if `additive-expression` is 0.

C++

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

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl;    // 00000000000000000000000000000100

    unsigned int int2 = int1 << -3;  // C4293: '<<' : shift count negative or too big, undefined behavior
    unsigned int int3 = int1 >> -3;  // C4293: '>>' : shift count negative or too big, undefined behavior

    unsigned int int4 = int1 << 32;  // C4293: '<<' : shift count negative or too big, undefined behavior

    unsigned int int5 = int1 >> 32;  // C4293: '>>' : shift count negative or too big, undefined behavior

    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl;    // 00000000000000000000000000000100 (no change)}
}
```

Footnotes

1 The following is the description of the shift operators in the C++ ISO specification (INCITS/ISO/IEC 14882-2011[2012]), sections 5.8.2 and 5.8.3.

The value of `E1 << E2` is `E1` left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. Otherwise, if `E1` has a signed type and non-negative value, and $E1 \times 2^{E2}$ is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; otherwise, the behavior is undefined.

The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a non-negative value, the value of the result is the integral part of the quotient of $E1/2^{E2}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

See Also

Expressions with Binary Operators
C++ Built-in Operators, Precedence and Associativity