Document number: *N3990*

Date: *2014-05-08*

Project: Programming Language C++, Evolution Working Group

Reply-to: "Daniel Gutson <daniel.gutson@tallertechnologies.com>,
Angel Bustamante <angel.bustamante@tallertechnologies.com>"

Adding Standard Circular Shift operators for computer integers

I. Motivation

Overview

C and C++ languages have the standard set of bit-wise operations, including *OR, AND, XOR, LEFT/RIGHT SHIFT, NOT*. However, circular shift (left and right rotate) isn't included in the language.

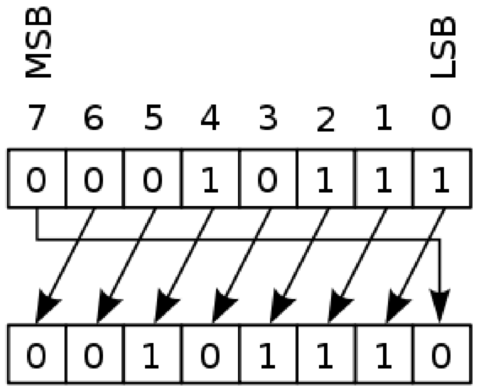
Rotating a computer integer is similar to shifting, but when a bit falls off one end of the register, it is moved to other end (as if they were connected end-to-end, conceptually). Rotation is used in encryption and decryption, so we want it to be fast. The obvious C/C++ (not safe) implementation for 32-bit integer rotate is:

```
uint32_t rotate_left32(uint32_t x, uint32_t n)
{
  if (n == 0) return x;
  return (x << n) | (x >> (32u-n));
}

uint32_t rotate_right32(uint32_t x, uint32_t n)
{
  if (n == 0) return x;
  return (x >> n) | (x << (32u-n));
}

//---------------------------
//Or the branchless version - using NDEBUG for gcc
uint32_t rotate_left32(uint32_t x, uint32_t n)
{
  assert(n<32u);
  return (x << n) | (x >> (-n&31u));
}

uint32_t rotate_right32(uint32_t x, uint32_t n)
{
  assert(n<32u);
  return (x >> n) | (x << (-n&31u));
}
```

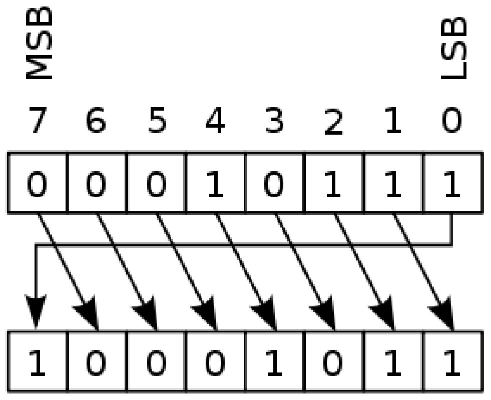
The complexity of a Rotation operation is similar to that of the bit-wise operators and, in many instruction sets, it normally requires a single assembly instruction. In addition, modern compilers are able to recognize this code and translate it into a rotate instruction, if there is one in the processor's instruction set.
The following code is the x86-64 rotate instruction generated by GCC (only works on version 4.7 and newer versions):

```
rotate_left32:
              movl  %edi,  %eax
              movb  %sil,  %cl
              roll  %cl,   %eax
              ret
```

On the other hand, there are two types of circular shift: with and without carry. That would not imply implementing a total of 4 operators, since it has no sense to perform signed shifts; the compiler should know whether we are shifting a signed or unsigned variable.
As the Standard specifies for bit shifting (5.8.2), if the left-hand operator is signed and it has a negative value, the behavior is undefined. The same will occur with circular shift operators.



Left-Rotate



Right-Rotate

II. Proposal

Enable this feature by specifying four new Standard operators to perform the rotation of a computer integer:

```
int32_t data;
const unsigned int i_pivot = 8u;

data = data >>> i_pivot;     // Rotate right
data = data <<< i_pivot;     // Rotate left

data >>>= i_pivot;     // Rotate and assign
data <<<= i_pivot;     // Rotate and assign
```

Consider the operators >>> and <<< as suggestions. Another symbol combination might be used.

The circular-shift operators should be overloadable in order to allow other classes to define their own semantics. An example of this can be a String or a Circular Buffer class:

```
class CircularBuffer
{
public:
        CircularBuffer operator>>> (const CircularBuffer & other);
        CircularBuffer operator<<< (const CircularBuffer & other);

        CircularBuffer operator>>>= (const CircularBuffer & other);
        CircularBuffer operator<<<= (const CircularBuffer & other);
...
};
```

This would increase the portability of algorithmic and cryptographic libraries implemented in C++. With regards to endianness, there should not be any difference in the semantics of the circular shift. The bit rotation would take place in the in the same way without taking care of the byte order.

III. Applications of the proposal

The implementation of the Bit Circular Shift can be useful in several development fields, for instance:

- Convert a 16-bit word between big-endian and little-endian representation: right or left circular shift by 8.
- Generate random bitset with even number of bits set: t = rand(); result = t XOR cshift(t,1).
- In-place, stable, and in linear time: move all elements of some array with even positions to the beginning and all elements with odd positions - to the end. One of possible algorithms is described in this paper: "In-Situ, Stable Merging by way of the Perfect Shuffle" (section 7). It generates all possible binary necklaces and uses them as starting points of cycle-leader algorithm where each next position is computed from previous one by circular shift. This application is closely related to multiplication by 2 (mod (2^N - 1)) mentioned in Henrik's answer.
- Micro-optimization. Suppose you need to unpack four 2-bit words from a single byte. You could do this by shifting each sub-word to rightmost position, then applying AND operation with proper mask. (No need to shift the first sub-word or mask the last one). All this needs 6 CPU instructions. If you circularly shift the byte by 4, two middle sub-words become the first and the last one, and also need only one instruction each. So using circular shift decreases number of needed instructions to 5.
- Cryptography applications receive significant speed-up when machine instruction set contains rotation instructions. For example, Twofish Cipher uses circular shifts extensively.

IV. Considerations: Why an operator?

This proposal may make readers think of several alternatives to achieve the bit circular shift. For instance, another choice for implementing this feature could be provided by the standard library (e.g. a function template rotate_left/right specialized for integer values).

The following snippet of code illustrates a possible implementation of such template functions:

```
template<class T>
inline T rotate_right(const T& x, const unsigned int n);

template<class T>
inline T rotate_left(const T& x, const unsigned int n);
```

Trade offs

Even though using a template function is performant, having a standard *operator* would keep it simple and more attractive due to the following:

- An operator is easily overloadable:
  - *By overloading a standard operator on a class, the developers can exploit intuition of the users of that class. The is to reduce both the learning curve and the defect rate.*
  - *Example: A class representing a circular buffer might overload the operators <<<, >>>, <<<= and >>>=.*
- Assignment Optimization
  - Having the assignment "y = rotate_right<T>(x, n)" is optimization dependant but
  - "y >>>(x, n) " is not.
- Other languages already implement this operator.
  - *Java implements >>, >>> and also the methods Integer.rotateRight() and Integer.rotateLeft().*
  - *Javascript implements these operators.*
- Having an operator completes the Semantics of bit shifting.

V. Impact On the Standard

This proposal is an extension of the core language and it does not require changes to any standard classes or functions. Moreover, the proposal does not depend on any other library extensions.

The modifications on the ISOCPP Standard C++ should be done in the chapter 5 (Expressions) and chapter 13 (Overloading).

Add new item to Chapter **5 Expressions:**
5.20 Circular Shift Operators **[expr.rotate]**

1 - The rotate operators <<< and >>> group left-to-right.

rotate-expression:
additive-expression
rotate-expression <<< *additive-expression*
rotate-expression >>> *additive-expression*

The operands shall be of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand.
If the left-hand operand is signed and has a negative value, the behavior is undefined.
(5.20/1)

2 - The value of E1 <<< E2 is E1 left-shifted E2 bit positions; vacated bits are not zero-filled; instead, when a bit falls off the left end of the register, it is used to fill the vacated bit position at the right end.

(5.20/2)

3 - The value of E1 >>> E2 is E1 right-shifted E2 bit positions; vacated bits are not zero-filled; instead, when a bit falls off the right end of the register, it is used to fill the vacated bit position at the left end.
(5.20/3)

Add two new items into to the list of Compound Assignment Operators.

5.17 Assignment and compound assignment operators [expr.ass]

The assignment operator (=) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand.
[...]

assignment-expression:
conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression
assignment-operator: one of

```
=     *=    /=    %=    +=    -=    >>=   <<=   &=    ˆ=    |=
>>>=  <<<=
```

(5.17/1)

Add four new items into to Chapter **13 Overloading:**

13.5 Overloaded Operators **[over.oper]**

```
operator-function-id:
        operator operator
operator:
        one of
                new     delete  new[]   delete[]
                +       -       *       /       %       ˆ       &       |       ~
                !       =       <       >       +=      -=      *=      /=      %=
                ˆ=      &=      |=      <<      >>      >>=     <<=     ==      !=
                <=      >=      &&      ||      ++      --      ,       ->*     ->
                <<<     >>>     >>>=    <<<=    ( )     [ ]
```

(13.5/1)

VI. Technical Specifications

None identified.

VII. Acknowledgements

- Thanks to Daniel Krügler for his valuable input and suggestions on the proposal.
- Thanks to Pablo Miguel Oliva for the additional comments on the proposal and the reviews.
- Thanks to Sebastián Davalle for reviewing the proposal.

VIII. References

- ISOCPP Standard C++ - ISO/IEC JTC1 SC22 WG21 N 3690
- CPP: Bit Shifting Operators that Wrap
- Poor optimization of portable rotate idiom
- Variable rotate optimization
- Applications of Bit Circular Shift