

What are bitwise shift (bit-shift) operators and how do they work?

I've been attempting to learn C in my spare time, and other languages (C#, Java, etc.) have the same concept (and often the same operators) ...

What I'm wondering is, at a core level, what does bit-shifting (<<, >>, >>>) do, what problems can it help solve, and what gotchas lurk around the bend? In other words, an absolute beginner's guide to bit shifting in all its goodness.

[operators](#) [bit-manipulation](#) [bit-shift](#) [binary-operators](#)

edited Jun 3 '15 at 7:13



Alexis King

26k 9 76 142

asked Sep 26 '08 at 19:47



John Rudy

22.7k 11 55 97

The functional or non-functional cases in which you would use bitshifting in 3GL's are few. – [Troy DeMonbreun](#) Sep 26 '08 at 20:19

10 After reading these answers you may want to look at these links: graphics.stanford.edu/~seander/bithacks.html & jji.de/bitwizardry/bitwizardrypage.html – [claws](#) Jun 15 '10 at 15:28

It's important to note that bit-shifting is extremely easy and fast for computers to do. By finding ways to use bit-shifting in you program, you can greatly reduce memory usage and execution times. – [Hoytman](#) Aug 23 '16 at 22:56

8 Answers

The bit shifting operators do exactly what their name implies. They shift bits. Here's a brief (or not-so-brief) introduction to the different shift operators.

The Operators

- >> is the arithmetic (or signed) right shift operator.
- >>> is the logical (or unsigned) right shift operator.
- << is the left shift operator, and meets the needs of both logical and arithmetic shifts.

All of these operators can be applied to integer values (`int` , `long` , possibly `short` and `byte` or `char`). In some languages, applying the shift operators to any datatype smaller than `int` automatically resizes the operand to be an `int` .

Note that <<< is not an operator, because it would be redundant. Also note that C and C++ do not distinguish between the right shift operators. They provide only the >> operator, and the right-shifting behavior is implementation defined for signed types.

Left shift (<<)

Integers are stored, in memory, as a series of bits. For example, the number 6 stored as a 32-bit `int` would be:

```
00000000 00000000 00000000 00000110
```

Shifting this bit pattern to the left one position (`6 << 1`) would result in the number 12:

```
00000000 00000000 00000000 00001100
```

As you can see, the digits have shifted to the left by one position, and the last digit on the right is filled with a zero. You might also note that shifting left is equivalent to multiplication by powers of 2. So `6 << 1` is equivalent to `6 * 2` , and `6 << 3` is equivalent to `6 * 8` . A good optimizing compiler will replace multiplications with shifts when possible.

Non-circular shifting

Please note that these are *not* circular shifts. Shifting this value to the left by one position (`3,758,096,384 << 1`):

```
11100000 00000000 00000000 00000000
```

results in 3,221,225,472:

```
11000000 00000000 00000000 00000000
```

The digit that gets shifted "off the end" is lost. It does not wrap around.

Logical right shift (>>>)

A logical right shift is the converse to the left shift. Rather than moving bits to the left, they simply move to the right. For example, shifting the number 12:

```
00000000 00000000 00000000 00001100
```

to the right by one position (12 >>> 1) will get back our original 6:

```
00000000 00000000 00000000 00000110
```

So we see that shifting to the right is equivalent to division by powers of 2.

Lost bits are gone

However, a shift cannot reclaim "lost" bits. For example, if we shift this pattern:

```
00111000 00000000 00000000 00000110
```

to the left 4 positions (939,524,102 << 4), we get 2,147,483,744:

```
10000000 00000000 00000000 01100000
```

and then shifting back ((939,524,102 << 4) >>> 4) we get 134,217,734:

```
00001000 00000000 00000000 00000110
```

We cannot get back our original value once we have lost bits.

Arithmetic right shift (>>)

The arithmetic right shift is exactly like the logical right shift, except instead of padding with zero, it pads with the most significant bit. This is because the most significant bit is the *sign* bit, or the bit that distinguishes positive and negative numbers. By padding with the most significant bit, the arithmetic right shift is sign-preserving.

For example, if we interpret this bit pattern as a negative number:

```
10000000 00000000 00000000 01100000
```


we have the number -2,147,483,552. Shifting this to the right 4 positions with the arithmetic shift (-2,147,483,552 >> 4) would give us:

```
11111000 00000000 00000000 00000110
```

or the number -134,217,722.

So we see that we have preserved the sign of our negative numbers by using the arithmetic right shift, rather than the logical right shift. And once again, we see that we are performing division by powers of 2.


edited Jun 7 at 7:24



Peter Cordes

74.7k 11 125 215

answered Sep 26 '08 at 20:46



Derek Park

36.9k 11 47 70

- 225

The answer should make it more clear that this is a Java-specific answer. There is no >>> operator in C/C++ or C#, and whether or not >> propagates the sign is implementation defined in C/C++ (a major potential gotcha) – [Michael Burr](#) Oct 20 '08 at 6:33
- 34

The answer is totally incorrect in the context of C language. There's no meaningful division into "arithmetic" and "logical" shifts in C. In C the shifts work as expected on unsigned values and on positive signed values - they just shift bits. On negative values, right-shift is implementation defined (i.e. nothing can be said about what it does in general), and left-shift is simply prohibited - it produces undefined behavior. – [Ant](#) Jun 8 '10 at 22:19
- 10

Audrey, there is certainly a difference between arithmetic and logical right shifting. C simply leaves the choice implementation defined. And left shift on negative values is definitely not prohibited. Shift 0xff000000 to the left one bit and you'll get 0xfe000000. – [Derek Park](#) Jul 9 '10 at 23:09
- 9

A good optimizing compiler will substitute shifts for multiplications when possible. What? Bitshifts are orders of magnitude faster when it comes down to the low level operations of a CPU, a good optimizing compiler would do the *exact* opposite, that is, turning ordinary multiplications by powers of two into bit shifts. – [Mahn](#) Jun 14 '13 at 11:45
- 36

@Mahn, you're reading it backwards from my intent. Substitute Y for X means to replace X with Y. Y is the substitute for X. So the shift is the substitute for the multiplication. – [Derek Park](#) Jan 27 '14 at 22:13

Let's say we have a single byte:

```
0110110
```

Applying a single left bitshift gets us:

```
1101100
```

The leftmost zero was shifted out of the byte, and a new zero was appended to the right end of the byte.

The bits don't rollover; they are discarded. That means if you left shift 1101100 and then right shift it, you won't get the same result back.

Shifting left by N is equivalent to multiplying by 2^N .

Shifting right by N is (if you are using [ones' complement](#)) is the equivalent of dividing by 2^N and rounding to zero.

Bitshifting can be used for insanely fast multiplication and division, provided you are working with a power of 2. Almost all low-level graphics routines use bitshifting.

For example, way back in the olden days, we used mode 13h (320x200 256 colors) for games. In Mode 13h, the video memory was laid out sequentially per pixel. That meant to calculate the location for a pixel, you would use the following math:

```
memoryOffset = (row * 320) + column
```

Now, back in that day and age, speed was critical, so we would use bitshifts to do this operation.

However, 320 is not a power of two, so to get around this we have to find out what is a power of two that added together makes 320:

```
(row * 320) = (row * 256) + (row * 64)
```

Now we can convert that into left shifts:

```
(row * 320) = (row << 8) + (row << 6)
```

For a final result of:

```
memoryOffset = ((row << 8) + (row << 6)) + column
```

Now we get the same offset as before, except instead of an expensive multiplication operation, we use the two bitshifts...in x86 it would be something like this (note, it's been forever since I've done assembly (editor's note: corrected a couple mistakes and added a 32-bit example)):

```
mov ax, 320; 2 cycles
mul word [row]; 22 CPU Cycles
mov di, ax; 2 cycles
add di, [column]; 2 cycles
; di = [row]*320 + [column]

; 16-bit addressing mode limitations:
; [di] is a valid addressing mode, but [ax] isn't, otherwise we could skip the last
mov
```

Total: 28 cycles on whatever ancient CPU had these timings.

Vrs

```
mov ax, [row]; 2 cycles
mov di, ax; 2
shl ax, 6; 2
shl di, 8; 2
add di, ax; 2      (320 = 256+64)
add di, [column]; 2
; di = [row]*(256+64) + [column]
```

12 cycles on the same ancient CPU.

Yes, we would work this hard to shave off 16 CPU cycles.

In 32 or 64-bit mode, both versions get a lot shorter and faster. Modern out-of-order execution CPUs like Intel Skylake (see <http://agner.org/optimize/>) have very fast hardware multiply (low latency and high throughput), so the gain is much smaller. AMD Bulldozer-family is a bit slower, especially for 64-bit multiply. On Intel CPUs, and AMD Ryzen, two shifts are slightly lower latency but more instructions than a multiply (which may lead to lower throughput):

```
imul edi, [row], 320    ; 3 cycle latency from [row] being ready
add edi, [column]       ; 1 cycle latency (from [column] and edi being ready).
; edi = [row]*(256+64) + [column], in 4 cycles from [row] being ready.
```

vs.

```
mov edi, [row]
shl edi, 6              ; row*64. 1 cycle latency
lea edi, [edi + edi*4]   ; row*(64 + 64*4). 1 cycle latency
add edi, [column]       ; 1 cycle latency from edi and [column] both being ready
; edi = [row]*(256+64) + [column], in 3 cycles from [row] being ready.
```

Compilers will do this for you: See how [gcc](#), [clang](#), and [MSVC](#) all use `shift+lea` when optimizing `return 320*row + col; .`

The most interesting thing to note here is that **x86 has a shift-and-add instruction (LEA)** that can do small left shifts and add at the same time, with the performance as and add instruction. ARM is even more powerful: one operand of any instruction can be left or right shifted for free. So scaling by a compile-time-constant that's known to be a power-of-2 can be even more efficient than a multiply.

OK, back in the modern days... something more useful now would be to use bitshifting to store two 8-bit values in a 16-bit integer. For example, in C#:

```
// Byte1: 11110000
// Byte2: 00001111

Int16 value = ((byte)(Byte1 >> 8) | Byte2));

// value = 0000111111110000;
```

In C++, compilers should do this for you if you used a struct with two 8-bit members, but in practice don't always.

edited Jun 9 at 4:56 answered Sep 26 '08 at 19:55

 **Peter Cordes** 74.7k 11 125 215  **FlySwat** 102k 60 226 297

6 Expanding this, on Intel processors (and a lot of others) it's faster to do this: `int c, d; c=d<<2; Than this: c=4*d; Sometimes, even "c=d<<2 + d<<1" is faster than "c=6*d"!!` I used these tricks extensively for graphic functions in the DOS era, I don't think they're so useful anymore... – [Joe Pineda](#) Sep 26 '08 at 20:44

@Joe: If these tricks aren't useful anymore, does that mean that compiler writers are using them now? – [James Thompson](#) Oct 25 '11 at 0:48

4 @James: not quite, nowadays it's rather the video-card's firmware which includes code like that, to be executed by the GPU rather than the CPU. So theoretically you don't need to implement code like this (or like Carmack's black-magic inverse root function) for graphic functions :-). – [Joe Pineda](#) Aug 29 '12 at 2:03

2 @JoePineda @james The compiler writers are definitely using them. If you write `c=4*d` you will get a shift. If you write `k = (n<0)` that may be done with shifts too: `k = (n>>31)&1` to avoid a branch. Bottom line, this improvement in cleverness of compilers means it's now unnecessary to use these tricks in the C code, and they compromise readability and portability. Still very good to know them if you're writing e.g. SSE vector code; or any situation where you need it fast and there's a trick which the compiler isn't using (e.g. GPU code). – [greggo](#) Oct 30 '14 at 14:17

1 Another good example: very common thing is `if(x >= 1 && x <= 9)` which can be done as `if((unsigned)(x-1) <=(unsigned)(9-1))` Changing two conditional tests to one can be a big speed advantage; especially when it allows predicated execution instead of branches. I used this for years (where justified) until I noticed abt 10 years ago that compilers had started doing this transform in the optimizer, then I stopped. Still good to know, since there are similar situations where the compiler can't make the transform for you. Or if you're working on a compiler. – [greggo](#) Oct 30 '14 at 14:28

Bitwise operations, including bit shift, are fundamental to low-level hardware or embedded programming. If you read a specification for a device or even some binary file formats, you will see bytes, words, and dwords, broken up into non-byte aligned bitfields, which contain various values of interest. Accessing these bit-fields for reading/writing is the most common usage.

A simple real example in graphics programming is that a 16-bit pixel is represented as follows:

bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	Blue					Green						Red				

To get at the green value you would do this:

```
#define GREEN_MASK 0x7E0
#define GREEN_OFFSET 5

// Read green
uint16_t green = (pixel & GREEN_MASK) >> GREEN_OFFSET;
```

Explanation

In order to obtain the value of green ONLY, which starts at offset 5 and ends at 10 (i.e. 6-bits long), you need to use a (bit) mask, which when applied against the entire 16-bit pixel, will yield only the bits we are interested in.

```
#define GREEN_MASK 0x7E0
```

The appropriate mask is 0x7E0 which in binary is 0000011111100000 (which is 2016 in decimal).

```
uint16_t green = (pixel & GREEN_MASK) ...;
```

To apply a mask, you use the AND operator (&).

```
uint16_t green = (pixel & GREEN_MASK) >> GREEN_OFFSET;
```

After applying the mask, you'll end up with a 16-bit number which is really just a 11-bit number since its MSB is in the 11th bit. Green is actually only 6-bits long, so we need to scale it down using a right shift (11 - 6 = 5), hence the use of 5 as offset (`#define GREEN_OFFSET 5`).

Also common is using bit shifts for fast multiplication and division by powers of 2:

```
i <<= x; // i *= 2^x;
i >>= y; // i /= 2^y;
```

edited Aug 8 '15 at 18:11



Peter Mortensen

11.6k 17 79 106

answered Sep 26 '08 at 22:22



robottobor

6,658 8 31 35

1 0x7e0 is the same as 11111100000 which is 2016 in decimal. – Saheed Mar 31 '15 at 22:20

Bit Masking & Shifting

Bit shifting is often used in low level graphics programming. For example a given pixel color value encoded in a 32-bit word.

```
Pixel-Color Value in Hex:    B9B9B900
Pixel-Color Value in Binary: 10111001 10111001 10111001 00000000
```

For better understanding, the same binary value labeled with what sections represents what color part.

```
Pixel-Color Value in Binary: 10111001 10111001 10111001 00000000
                             Red      Green    Blue      Alpha
```

Let's say for example we want to get the green value of this pixels color. We can easily get that value by *masking* and *shifting*.

Our mask:

```
color :      Red      Green    Blue      Alpha
           10111001 10111001 10111001 00000000
green_mask : 00000000 11111111 00000000 00000000
```

```
masked_color = color & green_mask
```

```
masked_color: 00000000 10111001 00000000 00000000
```

The logical `&` operator ensures that only the values where the mask is 1 are kept. The last thing we now have to do, is to get the correct integer value by shifting all those bits to the right by 16 places (*logical right shift*).

```
green_value = masked_color >>> 16
```

Et voilà, we have the integer representing the amount of green in the pixels color:

```
Pixels-Green Value in Hex:    000000B9
Pixels-Green Value in Binary: 00000000 00000000 00000000 10111001
Pixels-Green Value in Decimal: 185
```

This is often used for encoding or decoding image formats like `jpg`, `png`, ...

edited Dec 27 '15 at 23:25

answered Mar 31 '15 at 10:49



Basti Funck

777 8 24

One gotcha is that the following is implementation dependent (according to the ANSI standard):

```
char x = -1;
x >> 1;
```

x can now be 127 (01111111) or still -1 (11111111).

In practice, it's usually the latter.

edited Sep 27 '08 at 0:56

answered Sep 26 '08 at 20:07



AShelly

23.8k 9 63 113

4 If I recall it correctly, the ANSI C standard explicitly says this is implementation-dependent, so you need to check your compiler's documentation to see how it's implemented if you want to right-shift signed integers on your code. – Joe Pineda Sep 26 '08 at 20:46

Yes, I just wanted to emphasize the ANSI standard itself says so, it's not a case where vendors are simply not following the standard or that the standard says nothing about this particular case. – Joe Pineda Sep 27 '08 at 0:17

1 @AShelly Its arithmetic vs logical right shift. – abc Apr 27 '13 at 4:26

Note that in the Java implementation, the number of bits to shift is mod'd by the size of the

source.

For example:

```
(long) 4 >> 65
```

equals 2. You might expect shifting the bits to the right 65 times would zero everything out, but it's actually the equivalent of:

```
(long) 4 >> (65 % 64)
```

This is true for <<, >>, and >>>. I have not tried it out in other languages.

answered Aug 28 '15 at 13:16



[Patrick Monkelban](#)

106 1 4

I am writing tips and tricks only, may find useful in tests/exams.

1. $n = n * 2$: $n = n << 1$
2. $n = n / 2$: $n = n >> 1$
3. Checking if n is power of 2 (1,2,4,8,...): check $!(n \& (n-1))$
4. Getting x^{th} bit of n : $n \mid= (1 << x)$

edited Jul 6 at 0:04



[EJP](#)

236k 22 184 305

answered Oct 11 '16 at 22:43



[Ravi Prakash](#)

51 4

There must be a few more that you know by now? – [ryyker](#) Jun 6 at 13:31

Be aware of that only 32 bit version of PHP is available on the Windows platform.

Then if you for instance shift << or >> more than by 31 bits, results are unexpectable. Usually the original number instead of zeros will be returned, and it can be a really tricky bug.

Of course if you use 64 bit version of PHP (unix), you should avoid shifting by more than 63 bits. However, for instance, MySQL uses the 64-bit BIGINT, so there should not be any compatibility problems.

UPDATE: From PHP7 Windows, php builds are finally able to use full 64bit integers: *The size of an integer is platform-dependent, although a maximum value of about two billion is the usual value (that's 32 bits signed). 64-bit platforms usually have a maximum value of about 9E18, except on Windows prior to PHP 7, where it was always 32 bit.*

edited Sep 26 at 20:42

answered Oct 23 '15 at 14:28



[lukyer](#)

2,103 16 20

protected by [Robert Harvey](#) ♦ Mar 7 '13 at 18:17

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site (the [association bonus](#) does not count).

Would you like to answer one of these [unanswered questions](#) instead?