O.3 — Bit manipulation with bitwise operators and bit masks

BY ALEX ON SEPTEMBER 8TH, 2015 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 5TH, 2020

In the previous lesson on bitwise operators (**O.2 -- Bitwise operators**), we discussed how the various bitwise operators apply logical operators to each bit within the operands. Now that we understand how they function, let's take a look at how they're more commonly used.

Bit masks

In order to manipulate individual bits (e.g. turn them on or off), we need some way to identify the specific bits we want to manipulate. Unfortunately, the bitwise operators don't know how to work with bit positions. Instead they work with bit masks.

A **bit mask** is a predefined set of bits that is used to select which specific bits will be modified by subsequent operations.

Consider a real-life case where you want to paint a window frame. If you're not careful, you risk painting not only the window frame, but also the glass itself. You might buy some masking tape and apply it to the glass and any other parts you don't want painted. Then when you paint, the masking tape blocks the paint from reaching anything you don't want painted. In the end, only the non-masked parts (the parts you want painted) get painted.

A bit mask essentially performs the same function for bits -- the bit mask blocks the bitwise operators from touching bits we don't want modified, and allows access to the ones we do want modified.

Let's first explore how to define some simple bit masks, and then we'll show you how to use them.

Defining bit masks in C++14

The simplest set of bit masks is to define one bit mask for each bit position. We use 0s to mask out the bits we don't care about, and 1s to denote the bits we want modified.

Although bit masks can be literals, they're often defined as symbolic constants so they can be given a meaningful name and easily reused.

Because C++14 supports binary literals, defining these bit masks is easy:

```
#include <cstdint>

constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7
```

Now we have a set of symbolic constants that represents each bit position. We can use these to manipulate the bits (which we'll show how to do in just a moment).

Defining bit masks in C++11 or earlier

Because C++11 doesn't support binary literals, we have to use other methods to set the symbolic constants. There are two good methods for doing this. Less comprehensible, but more common, is to use hexadecimal. If you need

a refresher on hexadecimal, please revisit lesson **4.12 -- Literals**.

```
constexpr std::uint_fast8_t mask0{ 0x1 }; // hex for 0000 0001
constexpr std::uint_fast8_t mask1{ 0x2 }; // hex for 0000 0010
constexpr std::uint_fast8_t mask2{ 0x4 }; // hex for 0000 0100
constexpr std::uint_fast8_t mask3{ 0x8 }; // hex for 0000 1000
constexpr std::uint_fast8_t mask4{ 0x10 }; // hex for 0001 0000
constexpr std::uint_fast8_t mask5{ 0x20 }; // hex for 0010 0000
constexpr std::uint_fast8_t mask6{ 0x40 }; // hex for 0100 0000
constexpr std::uint_fast8_t mask7{ 0x80 }; // hex for 1000 0000
```

This can be a little hard to read. One way to make it easier is to use the left-shift operator to shift a bit into the proper location:

```
constexpr std::uint_fast8_t mask0{ 1 << 0 }; // 0000 0001
constexpr std::uint_fast8_t mask1{ 1 << 1 }; // 0000 0010
constexpr std::uint_fast8_t mask2{ 1 << 2 }; // 0000 0100
constexpr std::uint_fast8_t mask3{ 1 << 3 }; // 0000 1000
constexpr std::uint_fast8_t mask4{ 1 << 4 }; // 0001 0000
constexpr std::uint_fast8_t mask5{ 1 << 5 }; // 0010 0000
constexpr std::uint_fast8_t mask6{ 1 << 6 }; // 0100 0000
constexpr std::uint_fast8_t mask7{ 1 << 7 }; // 1000 0000
```

Testing a bit (to see if it is on or off)

Now that we have a set of bit masks, we can use these in conjunction with a bit flag variable to manipulate our bit flags.

To determine if a bit is on or off, we use *bitwise AND* in conjunction with the bit mask for the appropriate bit:

```
#include <cstdint>
#include <iostream>

int main()
{
    constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
    constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
    constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
    constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
    constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
    constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
    constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
    constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7

    std::uint_fast8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

    std::cout << "bit 0 is " << ((flags & mask0) ? "on\n" : "off\n");
    std::cout << "bit 1 is " << ((flags & mask1) ? "on\n" : "off\n");

    return 0;
}
```

This prints:

```
bit 0 is on
bit 1 is off
```

Setting a bit

To set (turn on) a bit, we use bitwise OR equals (operator |=) in conjunction with the bit mask for the appropriate bit:

```
#include <cstdint>
#include <iostream>

int main()
{
    constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
    constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
    constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
    constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
    constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
    constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
    constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
    constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7

    std::uint_fast8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

    std::cout << "bit 1 is " << ((flags & mask1) ? "on\n" : "off\n");

    flags |= mask1; // turn on bit 1

    std::cout << "bit 1 is " << ((flags & mask1) ? "on\n" : "off\n");

    return 0;
}
```

This prints:

```
bit 1 is off
bit 1 is on
```

We can also turn on multiple bits at the same time using *Bitwise OR*:

```
flags |= (mask4 | mask5); // turn bits 4 and 5 on at the same time
```

Resetting a bit

To clear a bit (turn off), we use *Bitwise AND* and *Bitwise NOT* together:

```
#include <cstdint>
#include <iostream>

int main()
{
    constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
    constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
    constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
    constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
    constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
    constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
    constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
    constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7

    std::uint_fast8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

    std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");

    flags &= ~mask2; // turn off bit 2
```

```
20
21      std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");
22
23      return 0;
24  }
```

This prints:

```
bit 2 is on
bit 2 is off
```

We can turn off multiple bits at the same time:

```
1  flags &= ~(mask4 | mask5); // turn bits 4 and 5 off at the same time
```

Flipping a bit

To toggle a bit state, we use *Bitwise XOR*:

```
1   #include <cstdint>
2   #include <iostream>
3
4   int main()
5   {
6       constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
7       constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
8       constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
9       constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
10      constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
11      constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
12      constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
13      constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7
14
15      std::uint_fast8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags
16
17      std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");
18      flags ^= mask2; // flip bit 2
19      std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");
20      flags ^= mask2; // flip bit 2
21      std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");
22
23      return 0;
24  }
```

This prints:

```
bit 2 is on
bit 2 is off
bit 2 is on
```

We can flip multiple bits simultaneously:

```
1  flags ^= (mask4 | mask5); // flip bits 4 and 5 at the same time
```

Bit masks and std::bitset

std::bitset supports the full set of bitwise operators. So even though it's easier to use the functions (test, set, reset, and flip) to modify individual bits, you can use bitwise operators and bit masks if you want.

Why would you want to? The functions only allow you to modify individual bits. The bitwise operators allow you to modify multiple bits at once.

```
#include <cstdint>
#include <iostream>
#include <bitset>

int main()
{
    constexpr std::bitset<8> mask0{ 0b0000'0001 }; // represents bit 0
    constexpr std::bitset<8> mask1{ 0b0000'0010 }; // represents bit 1
    constexpr std::bitset<8> mask2{ 0b0000'0100 }; // represents bit 2
    constexpr std::bitset<8> mask3{ 0b0000'1000 }; // represents bit 3
    constexpr std::bitset<8> mask4{ 0b0001'0000 }; // represents bit 4
    constexpr std::bitset<8> mask5{ 0b0010'0000 }; // represents bit 5
    constexpr std::bitset<8> mask6{ 0b0100'0000 }; // represents bit 6
    constexpr std::bitset<8> mask7{ 0b1000'0000 }; // represents bit 7

    std::bitset<8> flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

    std::cout << "bit 1 is " << (flags.test(1) ? "on\n" : "off\n");
    std::cout << "bit 2 is " << (flags.test(2) ? "on\n" : "off\n");
    flags ^= (mask1 | mask2); // flip bits 1 and 2

    std::cout << "bit 1 is " << (flags.test(1) ? "on\n" : "off\n");
    std::cout << "bit 2 is " << (flags.test(2) ? "on\n" : "off\n");
    flags |= (mask1 | mask2); // turn bits 1 and 2 on

    std::cout << "bit 1 is " << (flags.test(1) ? "on\n" : "off\n");
    std::cout << "bit 2 is " << (flags.test(2) ? "on\n" : "off\n");
    flags &= ~(mask1 | mask2); // turn bits 1 and 2 off

    std::cout << "bit 1 is " << (flags.test(1) ? "on\n" : "off\n");
    std::cout << "bit 2 is " << (flags.test(2) ? "on\n" : "off\n");

    return 0;
}
```

This prints:

```
bit 1 is off
bit 2 is on
bit 1 is on
bit 2 is off
bit 1 is on
bit 2 is on
bit 1 is off
bit 2 is off
```

Making bit masks meaningful

Naming our bit masks “mask1” or “mask2” tells us what bit is being manipulated, but doesn’t give us any indication of what that bit flag is actually being used for.

A best practice is to give your bit masks useful names as a way to document the meaning of your bit flags. Here’s an example from a game we might write:

```
#include <cstdint>
#include <iostream>
```

```
3
4   int main()
5   {
6           // Define a bunch of physical/emotional states
7       constexpr std::uint_fast8_t isHungry{   1 << 0 }; // 0000 0001
8       constexpr std::uint_fast8_t isSad{      1 << 1 }; // 0000 0010
9       constexpr std::uint_fast8_t isMad{      1 << 2 }; // 0000 0100
10      constexpr std::uint_fast8_t isHappy{    1 << 3 }; // 0000 1000
11      constexpr std::uint_fast8_t isLaughing{ 1 << 4 }; // 0001 0000
12      constexpr std::uint_fast8_t isAsleep{   1 << 5 }; // 0010 0000
13      constexpr std::uint_fast8_t isDead{     1 << 6 }; // 0100 0000
14      constexpr std::uint_fast8_t isCrying{   1 << 7 }; // 1000 0000
15
16      std::uint_fast8_t me{}; // all flags/options turned off to start
17      me |= isHappy | isLaughing; // I am happy and laughing
18      me &= ~isLaughing; // I am no longer laughing
19
20      // Query a few states
21      // (we'll use static_cast<bool> to interpret the results as a boolean value)
22      std::cout << "I am happy? " << static_cast<bool>(me & isHappy) << '\n';
23      std::cout << "I am laughing? " << static_cast<bool>(me & isLaughing) << '\n';
24
25      return 0;
26  }
```

Here's the same example implemented using std::bitset:

```
1   #include <iostream>
2   #include <bitset>
3
4   int main()
5   {
6           // Define a bunch of physical/emotional states
7       std::bitset<8> isHungry{    0b0000'0001 };
8       std::bitset<8> isSad{       0b0000'0010 };
9       std::bitset<8> isMad{       0b0000'0100 };
10      std::bitset<8> isHappy{     0b0000'1000 };
11      std::bitset<8> isLaughing{  0b0001'0000 };
12      std::bitset<8> isAsleep{    0b0010'0000 };
13      std::bitset<8> isDead{      0b0100'0000 };
14      std::bitset<8> isCrying{    0b1000'0000 };
15
16
17      std::bitset<8> me{}; // all flags/options turned off to start
18      me |= isHappy | isLaughing; // I am happy and laughing
19      me &= ~isLaughing; // I am no longer laughing
20
21      // Query a few states (we use the any() function to see if any bits remain set)
22      std::cout << "I am happy? " << (me & isHappy).any() << '\n';
23      std::cout << "I am laughing? " << (me & isLaughing).any() << '\n';
24
25      return 0;
26  }
```

Two notes here: First, std::bitset doesn't have a nice function that allows you to query bits using a bit mask. So if you want to use bit masks rather than positional indexes, you'll have to use *Bitwise AND* to query bits. Second, we make use of the any() function, which returns true if any bits are set, and false otherwise to see if the bit we queried remains on or off.

When are bit flags most useful?

Astute readers may note that the above examples don't actually save any memory. 8 booleans would normally take 8 bytes. But the above examples use 9 bytes (8 bytes to define the bit masks, and 1 bytes for the flag variable)!

Bit flags make the most sense when you have many identical flag variables. For example, in the example above, imagine that instead of having one person (me), you had 100. If you used 8 Booleans per person (one for each possible state), you'd use 800 bytes of memory. With bit flags, you'd use 8 bytes for the bit masks, and 100 bytes for the bit flag variables, for a total of 108 bytes of memory -- approximately 8 times less memory.

For most programs, the amount of memory using bit flags saved is not worth the added complexity. But in programs where there are tens of thousands or even millions of similar objects, using bit flags can reduce memory use substantially. It's a useful optimization to have in your toolkit if you need it.

There's another case where bit flags and bit masks can make sense. Imagine you had a function that could take any combination of 32 different options. One way to write that function would be to use 32 individual Boolean parameters:

```
1 void someFunction(bool option1, bool option2, bool option3, bool option4, bool option5, bool op
```

Hopefully you'd give your parameters more descriptive names, but the point here is to show you how obnoxiously long the parameter list is.

Then when you wanted to call the function with options 10 and 32 set to true, you'd have to do so like this:

```
1 someFunction(false, false, false, false, false, false, false, false, false, true, false, false,
```

This is ridiculously difficult to read (is that option 9, 10, or 11 that's set to true?), and also means you have to remember which parameters corresponds to which option (is setting the "edit flag" the 9th, 10th, or 11th parameter?) It may also not be very performant, as every function call has to copy 32 booleans from the caller to the function.

Instead, if you defined the function using bit flags like this:

```
1 void someFunction(std::bitset<32> options);
```

Then you could use bit flags to pass in only the options you wanted:

```
1 someFunction(option10 | option32);
```

Not only is this much more readable, it's likely to be more performant as well, since it only involves 2 operations (one *Bitwise OR* and one parameter copy).

This is one of the reasons OpenGL, a well regarded 3d graphic library, opted to use bit flag parameters instead of many consecutive Boolean parameters.

Here's a sample function call from OpenGL:

```
1 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // clear the color and the depth buffer
```

GL_COLOR_BUFFER_BIT and GL_DEPTH_BUFFER_BIT are bit masks defined as follows (in gl2.h):

```
1 #define GL_DEPTH_BUFFER_BIT               0x00000100
2 #define GL_STENCIL_BUFFER_BIT             0x00000400
3 #define GL_COLOR_BUFFER_BIT               0x00004000
```

Bit masks involving multiple bits

Although bit masks often are used to select a single bit, they can also be used to select multiple bits. Lets take a look at a slightly more complicated example where we do this.

Color display devices such as TVs and monitors are composed of millions of pixels, each of which can display a dot of color. The dot of color is composed from three beams of light: one red, one green, and one blue (RGB). By varying the intensity of the colors, any color on the color spectrum can be made. Typically, the amount of R, G, and B for a given pixel is represented by an 8-bit unsigned integer. For example, a red pixel would have R=255, G=0, B=0. A purple pixel would have R=255, G=0, B=255. A medium-grey pixel would have R=127, G=127, B=127.

When assigning color values to a pixel, in addition to R, G, and B, a 4th value called A is often used. "A" stands for "alpha", and it controls how transparent the color is. If A=0, the color is fully transparent. If A=255, the color is opaque.

R, G, B, and A are normally stored as a single 32-bit integer, with 8 bits used for each component:

32-bit RGBA value			
bits 31-24	bits 23-16	bits 15-8	bits 7-0
RRRRRRRR	GGGGGGGG	BBBBBBBB	AAAAAAAA
red	green	blue	alpha

The following program asks the user to enter a 32-bit hexadecimal value, and then extracts the 8-bit color values for R, G, B, and A.

```
#include <cstdint>
#include <iostream>

int main()
{
    constexpr std::uint_fast32_t redBits{ 0xFF000000 };
    constexpr std::uint_fast32_t greenBits{ 0x00FF0000 };
    constexpr std::uint_fast32_t blueBits{ 0x0000FF00 };
    constexpr std::uint_fast32_t alphaBits{ 0x000000FF };

    std::cout << "Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): ";
    std::uint_fast32_t pixel{};
    std::cin >> std::hex >> pixel; // std::hex allows us to read in a hex value

    // use Bitwise AND to isolate red pixels,
    // then right shift the value into the lower 8 bits
    // (we're not using brace initialization to avoid a static_cast)
    std::uint_fast8_t red = (pixel & redBits) >> 24;
    std::uint_fast8_t green = (pixel & greenBits) >> 16;
    std::uint_fast8_t blue = (pixel & blueBits) >> 8;
    std::uint_fast8_t alpha = pixel & alphaBits;

    std::cout << "Your color contains:\n";
    std::cout << std::hex; // print the following values in hex
    std::cout << static_cast<int>(red) << " red\n";
    std::cout << static_cast<int>(green) << " green\n";
    std::cout << static_cast<int>(blue) << " blue\n";
    std::cout << static_cast<int>(alpha) << " alpha\n";

    return 0;
}
```

This produces the output:

```
Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): FF7F3300
Your color contains:
ff red
7f green
```

```
33 blue
0 alpha
```

In the above program, we use a *bitwise AND* to query the set of 8 bits we're interested in, and then we *right shift* them into an 8-bit value so we can print them back as hex values.

Summary

Summarizing how to set, clear, toggle, and query bit flags:

To query bit states, we use *bitwise AND*:

```
if (flags & option4) ... // if option4 is set, do something
```

To set bits (turn on), we use *bitwise OR*:

```
flags |= option4; // turn option 4 on.
flags |= (option4 | option5); // turn options 4 and 5 on.
```

To clear bits (turn off), we use *bitwise AND* with *bitwise NOT*:

```
flags &= ~option4; // turn option 4 off
flags &= ~(option4 | option5); // turn options 4 and 5 off
```

To flip bit states, we use *bitwise XOR*:

```
flags ^= option4; // flip option4 from on to off, or vice versa
flags ^= (option4 | option5); // flip options 4 and 5
```

Quiz time

Question #1

Given the following program:

```
#include <cstdint>

int main()
{
    constexpr std::uint_fast8_t option_viewed{ 0x01 };
    constexpr std::uint_fast8_t option_edited{ 0x02 };
    constexpr std::uint_fast8_t option_favorited{ 0x04 };
    constexpr std::uint_fast8_t option_shared{ 0x08 };
    constexpr std::uint_fast8_t option_deleted{ 0x80 };

    std::uint_fast8_t myArticleFlags{};

    return 0;
}
```

a) Write a line of code to set the article as viewed.

Show Solution

b) Write a line of code to check if the article was deleted.

Show Solution

c) Write a line of code to clear the article as a favorite.

Show Solution

1d) Extra credit: why are the following two lines identical?

```
myflags &= ~(option4 | option5); // turn options 4 and 5 off
myflags &= ~option4 & ~option5; // turn options 4 and 5 off
```

Show Solution

O.4 -- Converting between binary and decimal

Index

O.2 -- Bitwise operators

C++ TUTORIAL | PRINT THIS POST

294 comments to O.3 — Bit manipulation with bitwise operators and bit masks

« Older Comments 1 2 3 4

Cadmiral
February 6, 2020 at 2:16 pm · Reply

Hi, thank you for the tutorial! Q: In your section and example in "Resetting a bit", I'm trying to figure out how the program came to the conclusions of on/off. I got 0000 0100 for the first comparison, and 0000 0001 after the second. I'm just trying to understand, first are my results correct? second, how are they interpreted as true/false ? Thanks!

nascardriver

February 7, 2020 at 8:45 am · Reply

on means the bit 1, off means the bit is 0

Any non-zero integer is `true`, so 0000 0100 is true, 0000 0001 is true, only 0000 0000 is `false`.

giang
January 17, 2020 at 7:17 am · Reply

I don't understand why in 1 example you 8-bits variable and binary assignment and in another example, you use 32-bits variable and hexadecimal assignment, why not binary too?. What's is the main difference between these two? And Why binary in 1 example and hexadecimal in other, what are the differences with using binary and hexadecimal in these examples because I think these are 2 very different types? Thanks very much

nascardriver
January 17, 2020 at 7:23 am · Reply

We use what's needed. If we need 32 bits, eg. for a color, we use a 32 bit integer. With colors, we don't care about the exact bits, but about the values of each color component. For this, we don't need binary values.
When we use flags, it's important to access bits individually, so we use a binary literal.

kavin
January 5, 2020 at 4:33 am · Reply

In, "Bit masks and std::bitset" section i get this warning/error in line 28.
(28,63): error C2220: the following warning is treated as an error
(28,63): warning C4245: 'argument': conversion from 'int' to 'unsigned __int64', signed/unsigned mismatch

```
{
      std::cout << "bit 1 is " << (flags.test(1) ? "on\n" : "off\n");
   std::cout << "bit 2 is " << (flags.test(2) ? "on\n" : "off\n");
   flags &= ~(mask1 | mask2); // turn bits 1 and 2 of
}
```

How do i get rid of the warning? I assume it has got to do with storing the result of ~(mask1 | mask2); being stored in flags by &= operator (or)
constexpr std::uint_fast8_t getting converted to unsigned_int64 ? why shd it get converted into 64 bit value when we have specifically defined them to be 8 bit value?

If i remove ~ operator and write,
flags &= ~(mask1 | mask2);
then the code compiles without error. I assume ~ produce a negative value . The -ve value is causing error?

Could you explain me please how to make the code compile?

nascardriver
January 5, 2020 at 7:02 am · Reply

Hi!

This happened because we mixed `std::uint_fast8_t` and `std::bitset`. Those types are mostly compatible, but `~` and `|` caused the `std::uint_fast8_t` to be promoted to an `int`. Creating a `std::bitset` from an `int`

can cause problems (Because `std::bitset` uses `unsigned` integers), so the compiler issued a warning/error.

I replaced `std::uint_fast8_t` with `std::bitset` in the example. This gets rid of the error, thanks for pointing it out!
There is a solution without having to change the types, but we haven't covered that yet.

TheDoctor
December 30, 2019 at 3:47 am · Reply

I have another question.
Well, this lesson is way harder than I thought.
In the last section (bit mask involving multiple bits) you're doing this-

```
1 constexpr unsigned int redBits{ 0xFF000000 };
```

Also you're storing 32 bits value in unsigned int pixel.

Since the minimum guaranteed size of int is 2 bytes (16 bits), how are you doing all that?

I'm sorry if this is a stupid question but I even searched last 2 comments pages and couldn't find anything related.

nascardriver
December 30, 2019 at 5:33 am · Reply

Good point! The code won't compile if an `int` is less than 32 bits wide (The integer literal (0xFF000000) will be a wider type, eg. unsigned long.). I updated the lesson to use guaranteed-width types.

TheDoctor
December 30, 2019 at 6:01 am · Reply

I was scratching my head with it, thanks for your prompt reply.
One last suggestion as the whole point of bit manipulation is generally to save space using uint_least_32_t could help impart better habits to user when dealing with bit manipulation. I may be wrong and you may have better reasons for using fast version, and in that case, I beg your pardon.

TheDoctor
December 29, 2019 at 8:33 am · Reply

```
1 std::bitset<4> var_bit{0b0101};
2 constexpr unsigned char mask2_char{0b0100};
3 var_bit &= ~(mask2_char);
```

Above code gives "error: negative integer implicitly coverted to unsigned type"
I scanned through comments for possible solution and found a similar problem but couldn't understand it properly.

TheDoctor
December 29, 2019 at 11:29 pm · Reply

So this is what I thought of, changing line 3 of above code to this solves it,

```
1 var_bit &= static_cast<unsigned char>(~(mask2_char))
```

Here are my thoughts;
- mask2_char was originally as expected an unsigned char type
- ~ operator implicitly typecasts it to int
- this int is then tried to type cast into unsigned char by &= operator with which compiler is unhappy as this implicit typecast (int to unsigned char) could lead to data loss.

Above para is all hypothesis as I'm also just a learner, it would be nice if someone would correct or approve it.

But I still have one question since var_bit is type std::bitset<4>, is it also first typecasted to unsigned char by &= operator (to do operation with the other unsigned char operand on right hand) and again typecasted back to std::bitset<4>?

nascardriver
December 30, 2019 at 5:17 am · Reply

Your explanation is correct, only the reason for the error is different. The compiler doesn't complain about a narrowing conversion, it complains about the loss of the sign.

~0b0100 = 0b1011 (assuming a 4 bit int)
The first bit is set, which means that the integer is negative. The negative `int` is then converted to an `unsigned long long` (More about this type later) by `&=`.
The `unsigned long long` can't represent negative numbers, so you get a warning (or error).
The compiler is able to know that the number is negative, because you made `mask2_char` `constexpr`. The compiler performs the negation at compile-time and catches the error before anything bad can happen at run-time. Without `constexpr`, you might not have noticed.

`std::bitset` is a custom type, it doesn't follow the same rules as built-in types. It doesn't undergo integral promotion (conversion to int) when you use it with arithmetic operators. The right side of its operators (eg. `&=`) has to be a `std::bitset` as well. `std::bitset`s are created from `unsigned long long`s. You gave it an `int`, so the `int` has to be converted to `unsigned long long` and then to `std::bitset` before it can be used with `&=`. This happens automatically.

If there's anything you didn't understand, don't hesitate to ask.

TheDoctor
December 30, 2019 at 5:58 am · Reply

Yes, now I'm starting to get the hang of it, you're awesome, man.
a.) By 'unsigned long long' you mean 'unsigned long long int', right?
So &= operator converts what on the right to first 'unsigned long long' and then to std::bitset.
b.) Is that why in the following code line 3 works and line 4 don't?

```
1 std::bitset<4> var_bit{0b0001};
2 constexpr unsigned char mask2_char{0b0100};
3 std::cout<<(var_bit|=mask2_char)<<'\n';
4 std::cout<<(var_bit | mask2_char);<<'\n';
```

nascardriver
December 30, 2019 at 6:51 am · Reply

a)
Correct. Just like with `short` or `long`, the `int` can (and usually is) be omitted.

b)
Ignoring the wild semicolon, this doesn't work because of a concept that is covered later in the tutorials. In case you're coming back to this question later,
`operator|=` is a member function. When you use it, there's no template deduction, because the type is already known from the left operand.
`operator|` is a non-member template function. Implicit conversions aren't considered during template deduction, so both operands (left and right) have to be the same.
`std::bitset<4>` is not the same as `unsigned char`, so no matching `operator|` is found.

TheDoctor
December 30, 2019 at 7:19 am · Reply

Ah, sorry for that wild semicolon, it was a typo.
I'll come back later after covering further sections.
As usual, your help has been very helpful.

Charan
December 13, 2019 at 8:32 am · Reply

I read the comments and saw that char is used to define the mask because char is of size 8. If our inputs deal with higher sized values(eg: there are 12 qualities like happy,laughing...), is it that we need to define the mask with int or long?

nascardriver
December 13, 2019 at 8:39 am · Reply

You'd need a wider type, yes. Rather than using the built-in types (int, long), you should use a type with guaranteed width, eg. `std::int_fast16_t` from lesson 4.6.

Charan
December 13, 2019 at 8:54 am · Reply

Thanks for a quick reply:). I think it will be useful to add why we used char there to avoid confusion. I did not really get two char variables can be or,and,not operated before looking at the comments .

nascardriver
December 13, 2019 at 9:03 am · Reply

I added a comment to the first example that uses `char`, because it's easy to forget that `char` is an integral type just like `int`.

koe
December 20, 2019 at 11:19 am · Reply

Since it is somewhat ambiguous whether an int like 'std::int_fast16_t' will actually be 16 bits (it might be 32 bits), would you recommend favoring 'std::bitset<16>' to guarantee it's 16 bits every time? The integers with guaranteed size, like int32_t, we were told not to use since they may not be supported on all architectures.

nascardriver
December 21, 2019 at 2:40 am · Reply

Wasting a couple of bits is no problem. I don't know if std::bitset has a guaranteed size, it might just limit what you can use.

koe
December 21, 2019 at 5:22 pm · Reply

Is there a possibility of type conversion errors when using 'std::int_fast16_t' since it might actually be 32 bits?

Ex. I tried to think of

```
1 std::int_fast8_t x{1}
2 char ch{static_cast<char>(x)} //but actually it's 16 bits or 32 bits
```

nascardriver
December 22, 2019 at 2:02 am · Reply

`static_cast` casts the value, the bits don't matter.

Anderson
December 7, 2019 at 12:07 pm · Reply

Hi!

You know when we configure our character output to print out hexadecimal or binary figures like this:

```
1 std::cout << std::hex;
2 std::cout << maroon;
3 //Printing some more color values here...
```

How exactly do we tell the compiler to exit the hexadecimal output mode ?

Thanks.

nascardriver
December 8, 2019 at 4:25 am · Reply

You can use `std::dec` to switch back to decimal mode

```
1 std::cout << std::dec;
```

Anderson
December 6, 2019 at 4:03 pm · Reply

Hi!

Is it possible to create flags with binary digits using C++ 11 or prior
like this ?

```
unsigned char flags{ 0b0000'0101 };
```

instead of using hexadecimal digits for the flag value.

nascardriver
December 7, 2019 at 5:00 am · Reply

You can write a `constexpr` function that converts a binary string to a binary number. I'm sure there are plenty of examples for this on the internet.
You don't need binary literals though, set up your flags and use bitwise arithmetic instead.

```
constexpr unsigned char PLAYER_JUMPING{ 1 << 0 }; // 0001
// ...
constexpr unsigned char PLAYER_SHOOTING{ 1 << 2 }; // 0100
// ...

unsigned char flags{ PLAYER_JUMPING | PLAYER_SHOOTING }; // 0101
```

Anderson
December 7, 2019 at 9:32 am · Reply

Thank you nascardriver.

I understand you could use bitwise arithmetic to form binary values as an alternative to using binary numbers. But what if I wanted to represent something like " 0b0101'0000 " (notice the multiple 1's) in bitwise arithmetic.

Do you think I can do something like this ?:

```
unsigned char completedTasks{ 101 << 4} //Is this equivalent to 0101 0000
```

Thank you once more.

nascardriver
December 8, 2019 at 1:18 am · Reply

Please edit your comments instead of deleting and re-posting them. Code tags work after refreshing the page.

If you're working with flags, you'll only have single bit flags. As I showed above, you can use a bitwise or to combine flags.
Your example doesn't work, `101` is a decimal number. You can use a decimal `5` (Binary 101) and shift left by 4, or set the value bit by bit

```
unsigned char completedTask{ (1 << 6) | (1 << 4) };
```

Anderson
December 8, 2019 at 4:05 am · Reply

Thank you nascardriver, it works. :)

Hana

October 27, 2019 at 12:59 pm · Reply

Do you have a chapter on bit fields?

nascardriver
October 28, 2019 at 4:06 am · Reply

Hi Hana!

No, there's currently no lesson about bit fields. If you have a specific question about them, feel free to ask anyway.

Jim
October 25, 2019 at 2:20 am · Reply

Hi, Alex or nascar
I have a question, why do we need to use bit masks in the program that extracts RGB colors from a hexadecimal value, when it can be done like this:

```
#include <iostream>

int main()
{
    unsigned int pixel{0xFF7F3300};

    // use Bitwise AND to isolate red pixels, then right shift the value into the lower 8 b
    unsigned char red = pixel >> 24;
    unsigned char green = pixel >> 16;
    unsigned char blue = pixel >> 8;
    unsigned char alpha = pixel;

    std::cout << "Your color contains:\n";
    std::cout << std::hex; // print the following values in hex
    std::cout << static_cast<int>(red) << " red\n";
    std::cout << static_cast<int>(green) << " green\n";
    std::cout << static_cast<int>(blue) << " blue\n";
    std::cout << static_cast<int>(alpha) << " alpha\n";

    return 0;
}
```

nascardriver
October 25, 2019 at 2:53 am · Reply

Hi Jim

There are several ways of extracting the individual values from an rgba number. I like yours, although it requires the reader to know how the conversion from `unsigned int` to `unsigned char` works. There are also purely arithmetic solutions, eg.

```
// not bitwise operations, no casts
unsigned int red{ pixel / 0x1000000 };
unsigned int green{ (pixel / 0x10000) % 0x100 };
unsigned int blue{ (pixel / 0x100) % 0x100 };
unsigned int alpha{ pixel % 0x100 };

// gets rid of the cast while printing too
```

Since those solutions, including yours, defeat the purpose of the code in the lesson, which is to teach bitwise operations, I'll leave the lesson as is.

Louis Cloete
August 20, 2019 at 10:42 am · Reply

Hi Alex!

Shouldn't the bit masks be declared constexpr rather than const?

Alex
August 21, 2019 at 8:19 pm · Reply

Yup. They've been updated. Thanks!

avidlearner
August 17, 2019 at 6:02 pm · Reply

There might be a slight error on "Setting a bit" line 18: should be mask1, not mask4.

Alex
August 18, 2019 at 4:06 pm · Reply

Indeed. Thanks again!

Pablo Asenjo Navas-Parejo
August 7, 2019 at 4:06 pm · Reply

Hi Alex or Nascar. Could you tell me why does this happen? If I initialize @eightflags to @mask3 and then I assing @eightflags the value of @eightflags & @mask3, it equals false, but if I assing @eighflags the value of @mask3 & @mask3, it equals true. I think this shouldn't happen since @eightflags is supposed to be equal to @mask3.

Code:

```
#include <iostream>

int main()
{
    using namespace std;

    const unsigned char mask0 = 0;      // 0000 0000
    const unsigned char mask1 = 1 << 1; // 0000 0010
    const unsigned char mask2 = 1 << 2; // 0000 0100
    const unsigned char mask3 = 1 << 3; // 0000 1000
    const unsigned char mask4 = 1 << 4; // 0001 0000
    const unsigned char mask5 = 1 << 5; // 0010 0000
    const unsigned char mask6 = 1 << 6; // 0100 0000
    const unsigned char mask7 = 1 << 7; // 1000 0000

    bool eightflags = mask3; //eightflags equals true
    cout << eightflags; //couts 1
    eightflags &= mask3; /* eightflags equals false but should equal true as
    0000 1000
    0000 1000
```

```
    &--------
    0000 1000
    */
    cout << eightflags; //couts 0
    system("pause");
    return 0;
}
```

Thank you!

nascardriver
August 7, 2019 at 11:50 pm · Reply

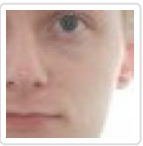
A `bool` is `false` (0) or `true` (1), it doesn't preserve the integer you assigned to it.

Vishal
July 8, 2019 at 1:59 am · Reply

Hey Alex, I learn C, C++, Java, HTML, CSS3, javascript, python. I almost understand all programming concepts of these modern programming languages but until now I don't build any cool stuff. I don't know what to do when I learn some new language. Alex please give me some suggestions to how I make or create some great stuff and what should I do when I learne some new programming language.

Daniel DeAnda
July 10, 2019 at 1:11 pm · Reply

Hey Vishal, I've found this website (https://www.linuxtrainingacademy.com/projects/) particularly useful as a wellspring of ideas for things to try out when you learn a new programming language. The links within suggest everything from simple manipulation programs to complex and massive projects, so I'm sure there will be something for every level of experience and comfort. The projects are almost all language-agnostic too, so you might even challenge your skills by trying to code the same thing in many of the languages you mentioned. Happy coding, and welcome to the community!



Grego
July 1, 2019 at 5:00 am · Reply

Hello Alex & Nascardriver,
This may be a stupid question, but why are contants in this lesson assigned to and not initialized?
I looked back to char variables and constants lessons and could see both of them showing proper initialization.
What makes this type of a constant not need an initialisation? Am I just missing something obvious?

Alex
July 1, 2019 at 1:04 pm · Reply

These lessons are still in the process of being rewritten/updated, so they aren't yet compliant with all modern best practices. The constants here are being initialized (not assigned) via copy initialization rather than the now-preferred uniform/brace initialization.

learning
June 22, 2019 at 5:25 am · Reply

```cpp
#include <iostream>
int main()
{
    const unsigned int redBits = 0xFF000000;
    const unsigned int greenBits = 0x00FF0000;
    const unsigned int blueBits = 0x0000FF00;
    const unsigned int alphaBits = 0x000000FF;

    std::cout << "Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): ";
    unsigned int pixel;
    std::cin >> std::hex >> pixel; // std::hex allows us to read in a hex value

    // use bitwise AND to isolate red pixels, then right shift the value into the range 0-2
    unsigned char red = (pixel & redBits) >> 24;
    unsigned char green = (pixel & greenBits) >> 16;
    unsigned char blue = (pixel & blueBits) >> 8;
    unsigned char alpha = pixel & alphaBits;

    std::cout << "Your color contains:\n";
    std::cout << static_cast<int>(red) << " of 255 red\n";
    std::cout << static_cast<int>(green) << " of 255 green\n";
    std::cout << static_cast<int>(blue) << " of 255 blue\n";
    std::cout << static_cast<int>(alpha) << " of 255 alpha\n";

    return 0;
}
```

what's wrong with this code? gives a warning:

```
Warning C4244 'initializing': conversion from 'unsigned int' to 'unsigned char', possible lo
```

nascardriver
June 22, 2019 at 5:29 am · Reply

```
(pixel & redBits) >> 24
```

is an `int`, `red` is a `char`. Converting from `int` to `char` could cause data loss. You're doing it on purpose and know that you're not losing data. Add a cast to silence the warning.

Sagar Pohekar
April 6, 2019 at 9:27 pm · Reply

Hello,
Thanks for the tutorial.
BTW, you said you are in process of replacing compile time/symbolic constants from 'const' to 'constexpr', so this tutorial is good place to use 'constexpr'

```cpp
#define CPP14 TRUE
//#define CPP11 TRUE //just for testing

#ifdef CPP14
constexpr unsigned char option0 = 0b0000'0001; // represents bit 0
constexpr unsigned char option1 = 0b0000'0010; // represents bit 1
```

```
constexpr unsigned char option2 = 0b0000'0100; // represents bit 2
constexpr unsigned char option3 = 0b0000'1000; // represents bit 3

constexpr unsigned char option4 = 0b0001'0000; // represents bit 4
constexpr unsigned char option5 = 0b0010'0000; // represents bit 5
constexpr unsigned char option6 = 0b0100'0000; // represents bit 6
constexpr unsigned char option7 = 0b1000'0000; // represents bit 7

#elifdef CPP11

constexpr unsigned char option0 = 0x1; // hex for 0000 0001
constexpr unsigned char option1 = 0x2; // hex for 0000 0010
constexpr unsigned char option2 = 0x4; // hex for 0000 0100
constexpr unsigned char option3 = 0x8; // hex for 0000 1000

constexpr unsigned char option4 = 0x10; // hex for 0001 0000
constexpr unsigned char option5 = 0x20; // hex for 0010 0000
constexpr unsigned char option6 = 0x40; // hex for 0100 0000
constexpr unsigned char option7 = 0x80; // hex for 1000 0000

#else

const unsigned char option0 = 1 << 0; // 0000 0001
const unsigned char option1 = 1 << 1; // 0000 0010
const unsigned char option2 = 1 << 2; // 0000 0100
const unsigned char option3 = 1 << 3; // 0000 1000
const unsigned char option4 = 1 << 4; // 0001 0000
const unsigned char option5 = 1 << 5; // 0010 0000
const unsigned char option6 = 1 << 6; // 0100 0000
const unsigned char option7 = 1 << 7; // 1000 0000

#endif
```

Thanks.

Alireza
February 18, 2019 at 7:38 am · Reply

Hello and thank you so much for the Bit flags tutorial. This is very useful.

question: Why have you used a char variable to use bit flags, Why haven't you used a bool variable or other ones to teach bit flags ?
Does it mean especial ?

Why the following program gives errors ?

```
#include <iostream>
#include <bitset>
#include "contestsman.h"
using std::cout;

const int o0   = 0b0000'000'1;
const int o1   = 0b0000'001'0;
const int o2   = 0b0000'010'0;
const int o3   = 0b0000'100'0;
const int o4   = 0b0001'000'0;
const int o5   = 0b0010'000'0;
const int o6   = 0b0100'000'0;
const int o7   = 0b1000'000'0;

int main()
{
```

```
    std::bitset<8> flags (o1) ;

    flags.flip(o1);
    flags.set(o2);

    cout << flags.test(o1) << "\n";         // works right
    cout << flags.test(o2) << "\n";         // works right
    cout << flags.test(o3) << "\n";         // gives error

    return 0;
}
```

Imagine I write this program, which variable does bitset do ?

```
#include <iostream>
#include <bitset>
#include "contestsman.h"
using std::cout;

const int o0   = 0b0000'000'1;
const int o1   = 0b0000'001'0;
const int o2   = 0b0000'010'0;
const int o3   = 0b0000'100'0;
const int o4   = 0b0001'000'0;
const int o5   = 0b0010'000'0;
const int o6   = 0b0100'000'0;
const int o7   = 0b1000'000'0;

int main()
{

    int flag {0};

    flag |= o2;
    flag &= ~o2;
    flag ^= o0;

    return 0;
}
```

Alex
February 19, 2019 at 7:53 pm · Reply

std::bitset functions take bit positions, not bit masks. o1 works because it evaluates to bit 2, o2 works because it evaluates to bit 4, but o3 doesn't work because it evaluates to bit 8, which is out of range for a bitset of size 8.

Alireza
February 20, 2019 at 11:04 am · Reply

Thanks for replying,
So the inputed number in std::bitset as the size is not bits' number ; it is their values and must be equal to our flags.

Sorry but you haven't answered it yet:

Why have you used a char variable to use bit flags, Why haven't you used a bool variable or other ones to teach bit flags ?

and one more question (important):

Even though the value 128 is equal to 1000 0000 in binary, and 255 is equal to 1111 1111 in binary, so what's difference between std::bitset<128> and std::bitset<255> ?

Alex
February 21, 2019 at 6:12 pm · Reply

> So the inputed number in std::bitset as the size is not bits' number ; it is their values and must be equal to our flags.

I read this a bunch of times and I'm not sure what you're trying to say.

```
1 std::bitset<8> flags (o1); // 8 is the size
```

This line defines a std::bitset with a size of 8 bits, and initializes those bits to 0b'0000'0010. This is fine.

```
1 flags.set(o2);
```

This is invalid, as o2 is decimal value 8, and 8 is outside the range of our bitset. You actually meant:

```
1 flags.set(2); // these functions take a bit position, not a bit pattern
```

I use a char here because I want 8 bits, and unsigned because we should always use unsigned variables when dealing with bits. Bool is typically used for logical operations, not bitwise operations. It's bad practice to use bool in a bitwise context. I'm also not sure how well defined it is for use with bitwise operations.

std::bitset<128> should define a bit field that holds 128 bits, and std::bitset<256> should define a bit field that holds 256 bits.

Alireza
February 26, 2019 at 8:58 am · Reply

Thank you so much,
I guess I understood. Thanks again.

a700
February 17, 2019 at 3:06 am · Reply

Hi, can you explain how this goes?
if (myflags & option4)
std::cout << "myflags has option 4 set";

nascardriver
February 17, 2019 at 6:40 am · Reply

```
1 myflags & option4
```

bitwise-ands myflags with ...10000, ie. the result is all 0, and the fifth bit from the right is the same as in @myflags.
If @myflags doesn't have this bit set, the result is 0.
Non-zero integers evaluate to true, 0 evaluates to false.
If the fifth bit from the right is set in @myflags, the condition is true.

Senna
February 15, 2019 at 2:47 pm · Reply

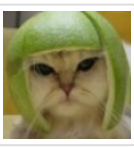
Hi,
Why this webpage is broken down these days ?
Is it temporary ?
Don't stop supporting it, please.
There's no source as simple, full contents as like learncpp.

nascardriver
February 16, 2019 at 4:29 am · Reply

Hi Senna!

Quoting the index: "Feb 15: [Site News] Our server died yesterday and had to be restored from backups. We're in the process of getting everything restored and reconfigured. If you find anything broken, please let us know here. Sorry for the inconvenience."

If you want to continue reading while learncpp is down, you can do so on archive.org
http://web.archive.org/web/20190214123506/https://www.learncpp.com/



Alex
February 16, 2019 at 10:49 am · Reply

Yup, there was a hardware failure on the server. The site has been moved to a new server and should be up and running full-time (at least, until the next catastrophe). :)

« Older Comments 1 2 3 4