3.8a — Bit flags and bit masks

BY ALEX ON SEPTEMBER 8TH, 2015 | LAST MODIFIED BY ALEX ON FEBRUARY 19TH, 2018

Note: This is a tough lesson. If you find yourself stuck, you can safely skip this lesson and come back to it later.

Bit flags

The smallest addressable unit of memory is a byte. Since all variables need to have unique addresses, this means variables must be at least one byte in size.

For most variable types, this is fine. However, for boolean values, this is a bit wasteful. Boolean types only have two states: true (1), or false (0). This only requires one bit to store. However, if a variable must be at least a byte, and a byte is typically 8 bits, that means a boolean is using 1 bit and leaving the other 7 unused.

In the majority of cases, this is fine -- we're usually not so hard-up for memory that we need to care about 7 wasted bits. However, in some storage-intensive cases, it can be useful to "pack" 8 individual boolean values into a single byte for storage efficiency purposes. This is done by using the bitwise operators to set, clear, and query individual bits in a byte, treating each as a separate boolean value. These individual bits are called **bit flags**.

Bit counting

When talking about individual bits, we typically count from right to left, starting with 0 (not 1). So given the bit pattern 0000 0111, bits 0 through 2 are 1, and bits 3 through 7 are 0.

So although we are typically used to counting starting with 1, in this lesson we'll generally count starting from 0.

Defining bit flags in C++14

In order to work with individual bits, we need to have a way to identify the individual bits within a byte, so we can manipulate those bits (turn them on and off). This is typically done by defining a symbolic constant to give a meaningful name to each bit used. The symbolic constant is given a value that represents that bit.

Because C++14 supports binary literals, this is easiest in C++14:

```
// Define 8 separate bit flags (these can represent whatever you want)
const unsigned char option0 = 0b0000'0001; // represents bit 0
const unsigned char option1 = 0b0000'0010; // represents bit 1
const unsigned char option2 = 0b0000'0100; // represents bit 2
const unsigned char option3 = 0b0000'1000; // represents bit 3
const unsigned char option4 = 0b0001'0000; // represents bit 4
const unsigned char option5 = 0b0010'0000; // represents bit 5
const unsigned char option6 = 0b0100'0000; // represents bit 6
const unsigned char option7 = 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 flags 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 **2.8 -- Literals**.

```
// Define 8 separate bit flags (these can represent whatever you want)
const unsigned char option0 = 0x1; // hex for 0000 0001
const unsigned char option1 = 0x2; // hex for 0000 0010
const unsigned char option2 = 0x4; // hex for 0000 0100
const unsigned char option3 = 0x8; // hex for 0000 1000
const unsigned char option4 = 0x10; // hex for 0001 0000
const unsigned char option5 = 0x20; // hex for 0010 0000
const unsigned char option6 = 0x40; // hex for 0100 0000
const unsigned char option7 = 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:

```
// Define 8 separate bit flags (these can represent whatever you want)
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
```

Using bit flags to manipulate bits

The next thing we need is a variable that we want to manipulate. Typically, we use an unsigned integer of the appropriate size (8 bits, 16 bits, 32 bits, etc… depending on how many options we have).

```
// Since we have 8 options above, we'll use an 8-bit value to hold our options
// Each bit in myflags corresponds to one of the options defined above
unsigned char myflags = 0; // all bits turned off to start
```

Variable myflags, defined above, will hold the actual bits that we'll turn on and off. How do we turns those bits on and off? We use our bit flags (options).

Turning individual bits on

To set a bit (turn on), we use bitwise OR equals (operator |=):

```
myflags |= option4; // turn option 4 on
```

Let's unpack this one to show how it works in more detail.

myflags |= option4 is the equivalent of myflags = (myflags | option4).

Evaluating the portion between the parenthesis:

```
myflags = 0000 0000 (we initialized this to 0)
option4 = 0001 0000
-------------------
result  = 0001 0000
```

So we get myflags = 0001 0000. In other words, we just turned the 4th bit on.

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

```
myflags |= (option4 | option5); // turn options 4 and 5 on at the same time
```

Turning individual bits off

To clear a bit (turn off), we use bitwise AND with an inverse bit pattern:

```
myflags &= ~option4; // turn option 4 off
```

This works similarly to the above. Let's say myflags was initially set to 0001 1100 (options 2, 3, and 4 turned on).

myflags &= ~option4; is the equivalent of myflags = (myflags & ~option4).

```
myflags  = 0001 1100
~option4 = 1110 1111
--------------------
result   = 0000 1100
```

So 0000 1100 gets assigned back to myflags. In other words, we just turned off bit 4 (and left the other bits alone).

We can turn off multiple bits at the same time:

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

Flipping individual bits

To toggle a bit state, we use bitwise XOR:

```
myflags ^= option4; // flip option4 from on to off, or vice versa
myflags ^= (option4 | option5); // flip options 4 and 5 at the same time
```

Determining if a bit is on or off

To query a bit state, we use bitwise AND:

```
if (myflags & option4)
    std::cout << "myflags has option 4 set";
if !(myflags & option5)
    std::cout << "myflags does not have option 5 set";
```

Bit flags in real life

In OpenGL (a 3d graphics library), some functions take one or more bit flags as a parameter:

```
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 defined as follows (in gl2.h):

```
#define GL_DEPTH_BUFFER_BIT               0x00000100
#define GL_STENCIL_BUFFER_BIT             0x00000400
#define GL_COLOR_BUFFER_BIT               0x00004000
```

Here’s a less abstract example for a game we might write:

```
#include <iostream>
int main()
{
        // Define a bunch of physical/emotional states
    const unsigned char isHungry = 1 << 0; // 0000 0001
    const unsigned char isSad = 1 << 1; // 0000 0010
    const unsigned char isMad = 1 << 2; // 0000 0100
    const unsigned char isHappy = 1 << 3; // 0000 1000
    const unsigned char isLaughing = 1 << 4; // 0001 0000
    const unsigned char isAsleep = 1 << 5; // 0010 0000
    const unsigned char isDead = 1 << 6; // 0100 0000
    const unsigned char isCrying = 1 << 7; // 1000 0000

    unsigned char me = 0; // all flags/options turned off to start
    me |= isHappy | isLaughing; // I am happy and laughing
    me &= ~isLaughing; // I am no longer laughing

    // Query a few states (we'll use static_cast<bool> to interpret the results as a boolean value rath
er than an integer)
    std::cout << "I am happy? " << static_cast<bool>(me & isHappy) << '\n';
    std::cout << "I am laughing? " << static_cast<bool>(me & isLaughing) << '\n';

    return 0;
}
```

Why are bit flags useful?

Astute readers will note that the above myflags example actually doesn’t save any memory. 8 booleans would normally take 8 bytes. But the above example uses 9 bytes (8 bytes to define the bit flag options, and 1 bytes for the bit flag)! So why would you actually want to use bit flags?

Bit flags are typically used in two cases:

1) When you have many sets of identical bitflags.

Instead of a single myflags variable, consider the case where you have two myflags variables: myflags1 and myflags2, each of which can store 8 options. If you defined these as two separate sets of booleans, you'd need 16 booleans, and thus 16 bytes. However, using bit flags, the memory cost is only 10 (8 bytes to define the options, and 1 byte for each myflags variable). With 100 myflag variables, your memory cost would be 108 bytes instead of 800. The more identical variables you need, the more substantial your memory savings.

Let's take a look at a more concrete example. Imagine you're creating a game where there are monsters for the player to fight. When a monster is created, it may be resistant to certain types of attacks (chosen at random). The different type of attacks in the game are: poison, lightning, fire, cold, theft, acid, paralysis, and blindness.

In order to track which types of attacks the monster is resistant to, we can use one boolean value per resistance (per monster). That's 8 booleans per monster.

With 100 monsters, that would take 800 boolean variables, using 800 bytes of memory.

However, using bit flags:

```
const unsigned char resistsPoison     = 1 << 0;
const unsigned char resistsLightning = 1 << 1;
const unsigned char resistsFire       = 1 << 2;
const unsigned char resistsCold       = 1 << 3;
const unsigned char resistsTheft      = 1 << 4;
const unsigned char resistsAcid       = 1 << 5;
const unsigned char resistsParalysis = 1 << 6;
const unsigned char resistsBlindness = 1 << 7;
```

Using bit flags, we only need one byte to store the resistances for a single monster, plus a one-time setup fee of 8 bytes for the options.

With 100 monsters, that would take 108 bytes total, or 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.

2) 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:

```
void someFunction(bool option1, bool option2, bool option3, bool option4, bool option5, bool option6, bo
ol option7, bool option8, bool option9, bool option10, bool option11, bool option12, bool option13, bool
 option14, bool option15, bool option16, bool option17, bool option18, bool option19, bool option20, boo
l option21, bool option22, bool option23, bool option24, bool option25, bool option26, bool option27, bo
ol option28, bool option29, bool option30, bool option31, bool option32);
```

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:

```
someFunction(false, false, false, false, false, false, false, false, false, true, false, false, false, f
alse, false, false, false, false, false, false, false, false, false, false, false, false, false, false,
false, false, false, true);
```

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:

```
void someFunction(unsigned int options);
```

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

```
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 opted to use bitflag parameters instead of many consecutive booleans.

Also, if you have unused bit flags and need to add options later, you can just define the bit flag. There's no need to change the function prototype, which is good for backwards compatibility.

An introduction to std::bitset

All of this bit flipping is exhausting, isn't it? Fortunately, the C++ standard library comes with functionality called std::bitset that helps us manage bit flags.

To create a std::bitset, you need to include the bitset header, and then define a std::bitset variable indicating how many bits are needed. The number of bits must be a compile time constant.

```
#include <bitset>

std::bitset<8> bits; // we need 8 bits
```

If desired, the bitset can be initialized with an initial set of values:

```
#include <bitset>

std::bitset<8> bits(option1 | option2) ; // start with option 1 and 2 turned on
std::bitset<8> morebits(0x3) ; // start with bit pattern 0000 0011
```

Note that our initialization value is interpreted as binary. Since we pass in the value 3, the std::bitset will start with the binary value for 3 (0000 0011).

std::bitset provides 4 key functions:

- test() allows us to query whether a bit is a 0 or 1
- set() allows us to turn a bit on (this will do nothing if the bit is already on)
- reset() allows us to turn a bit off (this will do nothing if the bit is already off)
- flip() allows us to flip a bit from a 0 to a 1 or vice versa

Each of these functions takes a bit-position parameter indicating which bit should be operated on. The position of the rightmost bit is 0, increasing with each successive bit to the left. Giving descriptive names to the bit indices can be useful here (either by assigning them to const variables, or using enums, which we'll introduce in the next chapter).

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

// Note that with std::bitset, our options correspond to bit indices, not bit patterns
const int option0 = 0;
const int option1 = 1;
const int option2 = 2;
const int option3 = 3;
const int option4 = 4;
const int option5 = 5;
const int option6 = 6;
const int option7 = 7;

int main()
{
    std::bitset<8> bits(0x2); // we need 8 bits, start with bit pattern 0000 0010
    bits.set(option4); // set bit 4 to 1 (now we have 0001 0010)
    bits.flip(option5); // flip bit 5 (now we have 0011 0010)
    bits.reset(option5); // set bit 5 back to 0 (now we have 0001 0010)

    std::cout << "Bit 4 has value: " << bits.test(option4) << '\n';
    std::cout << "Bit 5 has value: " << bits.test(option5) << '\n';
    std::cout << "All the bits: " << bits << '\n';

    return 0;
}
```

This prints:

```
Bit 4 has value: 1
Bit 5 has value: 0
All the bits: 00010010
```

Note that sending the bitset variable to std::cout prints the value of all the bits in the bitset.

Remember that the initialization value for a bitset is treated as binary, whereas the bitset functions use bit positions!

std::bitset also supports the standard bit operators (operator|, operator&, and operator^), so you can still use those if you wish (they can be useful when setting or querying multiple bits at once).

We recommend using std::bitset instead of doing all the bit operations manually, as bitset is more convenient and less error prone.

(h/t to reader "Mr. D")

Bit masks

The principles for bit flags can be extended to turn on, turn off, toggle, or query multiple bits at once, in a bit single operation. When we bundle individual bits together for the purpose of modifying them as a group, this is called a **bit mask**.

Let's take a look at a sample program using bit masks. In the following program, we ask the user to enter a number. We then use a bit mask to keep only the low 4 bits, which we print the value of.

```cpp
#include <iostream>

int main()
{
    const unsigned int lowMask = 0xF; // bit mask to keep low 4 bits (hex for 0000 0000 0000 1111)

    std::cout << "Enter an integer: ";
    int num;
    std::cin >> num;

    num &= lowMask; // remove the high bits to leave only the low bits

    std::cout << "The 4 low bits have value: " << num << '\n';

    return 0;
}
```

```
Enter an integer: 151
The 4 low bits have value: 7
```

151 is 1001 0111 in binary. lowMask is 0000 1111 in 8-bit binary. 1001 0111 & 0000 1111 = 0000 0111, which is 7 decimal.

Although this example is pretty contrived, the important thing to note is that we modified multiple bits in one operation!

An RGBA color example

Now lets take a look at a more complicated example.

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.

```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 0-255
    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;
}
```

This produces the output:

```
Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): FF7F3300
Your color contains:
255 of 255 red
127 of 255 green
51 of 255 blue
0 of 255 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 to move them to the range of 0-255 for storage and printing.

Note: RGBA is sometimes stored as ARGB instead, with the alpha channel being stored in the most significant byte rather than the least significant.

Summary

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

To query bit states, we use bitwise AND:

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

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

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

To clear bits (turn off), we use bitwise AND with an inverse bit pattern:

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

To toggle bit states, we use bitwise XOR:

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

Quiz

1) Given the following program:

```
int main()
{
    unsigned char option_viewed = 0x01;
    unsigned char option_edited = 0x02;
    unsigned char option_favorited = 0x04;
    unsigned char option_shared = 0x08;
    unsigned char option_deleted = 0x80;

    unsigned char myArticleFlags = 0;

    return 0;
}
```

1a) Write a line of code to set the article as viewed.
1b) Write a line of code to check if the article was deleted.
1c) Write a line of code to clear the article as a favorite.
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
```

Quiz answers

1a) **Hide Solution**

myArticleFlags |= option_viewed;

1b) **Hide Solution**

if (myArticleFlags & option_deleted) …

1c) **Hide Solution**

myArticleFlags &= ~option_favorited;

1d) **Hide Solution**

De Morgan’s law says that if we distribute a NOT, we need to flip ORs and ANDs to the other. So ~(option4 | option5) becomes ~option4 & ~option5.

3.x -- Chapter 3 comprehensive quiz

Index

3.8 -- Bitwise operators