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

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::uint8_t mask0{ 0b0000'0001 }; // represents bit 0
constexpr std::uint8_t mask1{ 0b0000'0010 }; // represents bit 1
constexpr std::uint8_t mask2{ 0b0000'0100 }; // represents bit 2
constexpr std::uint8_t mask3{ 0b0000'1000 }; // represents bit 3
constexpr std::uint8_t mask4{ 0b0001'0000 }; // represents bit 4
constexpr std::uint8_t mask5{ 0b0010'0000 }; // represents bit 5
constexpr std::uint8_t mask6{ 0b0100'0000 }; // represents bit 6
constexpr std::uint8_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.

The first method is to use hexadecimal literals.

Related content

We talk about hexadecimal in lesson 5.2 -- Literals.

Here's how hexadecimal converts to binary:

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Therefore, we can define bit masks using hexadecimal like this:

```
constexpr std::uint8_t mask0{ 0x01 }; // hex for 0000 0001
constexpr std::uint8_t mask1{ 0x02 }; // hex for 0000 0010
constexpr std::uint8_t mask2{ 0x04 }; // hex for 0000 0100
constexpr std::uint8_t mask3{ 0x08 }; // hex for 0000 1000
constexpr std::uint8_t mask4{ 0x10 }; // hex for 0001 0000
constexpr std::uint8_t mask5{ 0x20 }; // hex for 0010 0000
constexpr std::uint8_t mask6{ 0x40 }; // hex for 0100 0000
constexpr std::uint8_t mask7{ 0x80 }; // hex for 1000 0000
```

Sometimes leading hexadecimal 0s will be omitted (e.g. instead of `0x01` you'll just see `0x1`). Either way, this can be a little hard to read if you're not familiar with hexadecimal to binary conversion.

An easier method is to use the left-shift operator to shift a single bit into the proper location:

```
constexpr std::uint8_t mask0{ 1 << 0 }; // 0000 0001
constexpr std::uint8_t mask1{ 1 << 1 }; // 0000 0010
constexpr std::uint8_t mask2{ 1 << 2 }; // 0000 0100
constexpr std::uint8_t mask3{ 1 << 3 }; // 0000 1000
constexpr std::uint8_t mask4{ 1 << 4 }; // 0001 0000
constexpr std::uint8_t mask5{ 1 << 5 }; // 0010 0000
constexpr std::uint8_t mask6{ 1 << 6 }; // 0100 0000
constexpr std::uint8_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()
{
	[[maybe_unused]] constexpr std::uint8_t mask0{ 0b0000'0001 }; // represents bit 0
	[[maybe_unused]] constexpr std::uint8_t mask1{ 0b0000'0010 }; // represents bit 1
	[[maybe_unused]] constexpr std::uint8_t mask2{ 0b0000'0100 }; // represents bit 2
	[[maybe_unused]] constexpr std::uint8_t mask3{ 0b0000'1000 }; // represents bit 3
	[[maybe_unused]] constexpr std::uint8_t mask4{ 0b0001'0000 }; // represents bit 4
	[[maybe_unused]] constexpr std::uint8_t mask5{ 0b0010'0000 }; // represents bit 5
	[[maybe_unused]] constexpr std::uint8_t mask6{ 0b0100'0000 }; // represents bit 6
	[[maybe_unused]] constexpr std::uint8_t mask7{ 0b1000'0000 }; // represents bit 7

	std::uint8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

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

	return 0;
}
```

This prints:

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

Let's examine how this works.

In the case of `flags & mask0`, we have `0000'0101` & `0000'0001`. Let's line these up:

```
0000'0101 &
0000'0001
---------
0000'0001
```

We are then casting `0000'0001` to a `bool`. Since any non-zero number converts to `true` and this value has a non-zero digit, this evaluates to `true`.

In the case of `flags & mask1`, we have `0000'0101` & `0000'0010`. Let's line these up:

```
0000'0101 &
0000'0010
---------
0000'0000
```

Since a zero value converts to `false` and this value has only zero digits, this evaluates to `false`.

Setting a bit

To set (turn on) a bit (to value 1), we use bitwise OR equals (operator |=) in conjunction with the bit mask for the appropriate bit:

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

int main()
{
    [[maybe_unused]] constexpr std::uint8_t mask0{ 0b0000'0001 }; // represents bit 0
    [[maybe_unused]] constexpr std::uint8_t mask1{ 0b0000'0010 }; // represents bit 1
    [[maybe_unused]] constexpr std::uint8_t mask2{ 0b0000'0100 }; // represents bit 2
    [[maybe_unused]] constexpr std::uint8_t mask3{ 0b0000'1000 }; // represents bit 3
    [[maybe_unused]] constexpr std::uint8_t mask4{ 0b0001'0000 }; // represents bit 4
    [[maybe_unused]] constexpr std::uint8_t mask5{ 0b0010'0000 }; // represents bit 5
    [[maybe_unused]] constexpr std::uint8_t mask6{ 0b0100'0000 }; // represents bit 6
    [[maybe_unused]] constexpr std::uint8_t mask7{ 0b1000'0000 }; // represents bit 7

    std::uint8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

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

    flags |= mask1; // turn on bit 1

    std::cout << "bit 1 is " << (static_cast<bool>(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 reset (clear) a bit (to value 0), we use *Bitwise AND* and *Bitwise NOT* together:

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

int main()
{
    [[maybe_unused]] constexpr std::uint8_t mask0{ 0b0000'0001 }; // represents bit 0
    [[maybe_unused]] constexpr std::uint8_t mask1{ 0b0000'0010 }; // represents bit 1
    [[maybe_unused]] constexpr std::uint8_t mask2{ 0b0000'0100 }; // represents bit 2
    [[maybe_unused]] constexpr std::uint8_t mask3{ 0b0000'1000 }; // represents bit 3
    [[maybe_unused]] constexpr std::uint8_t mask4{ 0b0001'0000 }; // represents bit 4
    [[maybe_unused]] constexpr std::uint8_t mask5{ 0b0010'0000 }; // represents bit 5
    [[maybe_unused]] constexpr std::uint8_t mask6{ 0b0100'0000 }; // represents bit 6
    [[maybe_unused]] constexpr std::uint8_t mask7{ 0b1000'0000 }; // represents bit 7

    std::uint8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

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

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

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

    return 0;
}
```

This prints:

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

We can turn off multiple bits at the same time:

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

Key insight

Some compilers may complain about a sign conversion with this line:

```
flags &= ~mask2;
```

Because the type of `mask2` is smaller than `int`, `operator~` causes operand `mask2` to undergo integral promotion to type `int`. Then the compiler complains that we're trying to use `operator&=` where the left operand is unsigned and the right operand is signed.

If this is the case, try the following:

```
flags &= static_cast<std::uint8_t>(~mask2);
```

We discuss integral promotion in lesson 10.2 -- Floating-point and integral promotion.

Flipping a bit

To toggle (flip) a bit state (from 0 to 1 or from 1 to 0), we use *Bitwise XOR*:

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

int main()
{
    [[maybe_unused]] constexpr std::uint8_t mask0{ 0b0000'0001 }; // represents bit 0
    [[maybe_unused]] constexpr std::uint8_t mask1{ 0b0000'0010 }; // represents bit 1
    [[maybe_unused]] constexpr std::uint8_t mask2{ 0b0000'0100 }; // represents bit 2
    [[maybe_unused]] constexpr std::uint8_t mask3{ 0b0000'1000 }; // represents bit 3
    [[maybe_unused]] constexpr std::uint8_t mask4{ 0b0001'0000 }; // represents bit 4
    [[maybe_unused]] constexpr std::uint8_t mask5{ 0b0010'0000 }; // represents bit 5
    [[maybe_unused]] constexpr std::uint8_t mask6{ 0b0100'0000 }; // represents bit 6
    [[maybe_unused]] constexpr std::uint8_t mask7{ 0b1000'0000 }; // represents bit 7

    std::uint8_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

    std::cout << "bit 2 is " << (static_cast<bool>(flags & mask2) ? "on\n" : "off\n");
    flags ^= mask2; // flip bit 2
    std::cout << "bit 2 is " << (static_cast<bool>(flags & mask2) ? "on\n" : "off\n");
    flags ^= mask2; // flip bit 2
    std::cout << "bit 2 is " << (static_cast<bool>(flags & mask2) ? "on\n" : "off\n");

    return 0;
}
```

This prints:

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

We can flip multiple bits simultaneously:

```
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 <bitset>
#include <iostream>

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

int main()
{
	// Define a bunch of physical/emotional states
	[[maybe_unused]] constexpr std::uint8_t isHungry   { 1 << 0 }; // 0000 0001
	[[maybe_unused]] constexpr std::uint8_t isSad      { 1 << 1 }; // 0000 0010
	[[maybe_unused]] constexpr std::uint8_t isMad      { 1 << 2 }; // 0000 0100
	[[maybe_unused]] constexpr std::uint8_t isHappy    { 1 << 3 }; // 0000 1000
	[[maybe_unused]] constexpr std::uint8_t isLaughing { 1 << 4 }; // 0001 0000
	[[maybe_unused]] constexpr std::uint8_t isAsleep   { 1 << 5 }; // 0010 0000
	[[maybe_unused]] constexpr std::uint8_t isDead     { 1 << 6 }; // 0100 0000
	[[maybe_unused]] constexpr std::uint8_t isCrying   { 1 << 7 }; // 1000 0000

	std::uint8_t me{}; // 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)
	std::cout << std::boolalpha; // print true or false instead of 1 or 0
	std::cout << "I am happy? " << static_cast<bool>(me & isHappy) << '\n';
	std::cout << "I am laughing? " << static_cast<bool>(me & isLaughing) << '\n';

	return 0;
}
```

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

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

int main()
{
	// Define a bunch of physical/emotional states
	[[maybe_unused]] constexpr std::bitset<8> isHungry   { 0b0000'0001 };
	[[maybe_unused]] constexpr std::bitset<8> isSad      { 0b0000'0010 };
	[[maybe_unused]] constexpr std::bitset<8> isMad      { 0b0000'0100 };
	[[maybe_unused]] constexpr std::bitset<8> isHappy    { 0b0000'1000 };
	[[maybe_unused]] constexpr std::bitset<8> isLaughing { 0b0001'0000 };
	[[maybe_unused]] constexpr std::bitset<8> isAsleep   { 0b0010'0000 };
	[[maybe_unused]] constexpr std::bitset<8> isDead     { 0b0100'0000 };
	[[maybe_unused]] constexpr std::bitset<8> isCrying   { 0b1000'0000 };


	std::bitset<8> me{}; // 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 use the any() function to see if any bits remain set)
	std::cout << std::boolalpha; // print true or false instead of 1 or 0
	std::cout << "I am happy? " << (me & isHappy).any() << '\n';
	std::cout << "I am laughing? " << (me & isLaughing).any() << '\n';

	return 0;
}
```

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 separate booleans values would normally take 8 bytes. But the examples above (using std::uint8_t) use 9 bytes -- 8 bytes to define the bit masks, and 1 byte 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 saved using bit flags 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:

```
void someFunction(bool option1, bool option2, bool option3, bool option4, bool option5, bool option6, bool 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, bool option21, bool option22, bool
option23, bool option24, bool option25, bool option26, bool option27, bool 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, 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 argument corresponds to which option (is setting the "edit flag" the 9th, 10th, or 11th parameter?).

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

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

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

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

This is much more readable.

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:

```
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):

```
#define GL_DEPTH_BUFFER_BIT               0x00000100
#define GL_STENCIL_BUFFER_BIT             0x00000400
#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. Each dot of color is the result of combining three beams of light: one red, one green, and one blue (RGB). The intensity of these lights are varied to produce different colors.

Typically, the intensity 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::uint32_t redBits{ 0xFF000000 };
	constexpr std::uint32_t greenBits{ 0x00FF0000 };
	constexpr std::uint32_t blueBits{ 0x0000FF00 };
	constexpr std::uint32_t alphaBits{ 0x000000FF };

	std::cout << "Enter a 32-bit RGBA color value in hexadecimal (e.g. FF7F3300): ";
	std::uint32_t pixel{};
	std::cin >> std::hex >> pixel; // std::hex allows us to read in a hex value

	// use Bitwise AND to isolate the pixels for our given color,
	// then right shift the value into the lower 8 bits
	const std::uint8_t red{ static_cast<std::uint8_t>((pixel & redBits) >> 24) };
	const std::uint8_t green{ static_cast<std::uint8_t>((pixel & greenBits) >> 16) };
	const std::uint8_t blue{ static_cast<std::uint8_t>((pixel & blueBits) >> 8) };
	const std::uint8_t alpha{ static_cast<std::uint8_t>(pixel & alphaBits) };

	std::cout << "Your color contains:\n";
	std::cout << std::hex; // print the following values in hex

	// reminder: std::uint8_t will likely print as a char
	// we static_cast to int to ensure it prints as an integer
	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

Do not use `std::bitset` in this quiz. We're only using `std::bitset` for printing.

Given the following program:

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

int main()
{
    [[maybe_unused]] constexpr std::uint8_t option_viewed{ 0x01 };
    [[maybe_unused]] constexpr std::uint8_t option_edited{ 0x02 };
    [[maybe_unused]] constexpr std::uint8_t option_favorited{ 0x04 };
    [[maybe_unused]] constexpr std::uint8_t option_shared{ 0x08 };
    [[maybe_unused]] constexpr std::uint8_t option_deleted{ 0x10 };

    std::uint8_t myArticleFlags{ option_favorited };

    // Place all lines of code for the following quiz here

    std::cout << std::bitset<8>{ myArticleFlags } << '\n';

    return 0;
}
```

a) Add a line of code to set the article as viewed.
Expected output:

```
00000101
```

Show Solution

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

Show Solution

c) Add a line of code to clear the article as a favorite.
Expected output (Assuming you did quiz (a)):

```
00000001
```

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