O.1 — Bit flags and bit manipulation via std::bitset

ALEX DECEMBER 12, 2023

On modern computer architectures, the smallest addressable unit of memory is a byte. Since all objects need to have unique memory addresses, this means objects must be at least one byte in size. For most variable types, this is fine. However, for Boolean values, this is a bit wasteful (pun intended). Boolean types only have two states: true (1), or false (0). This set of states only requires one bit to store. However, if a variable must be at least a byte, and a byte is 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 (we're better off optimizing for understandability and maintainability). However, in some storage-intensive cases, it can be useful to "pack" 8 individual Boolean values into a single byte for storage efficiency purposes.

Doing these things requires that we can manipulate objects at the bit level. Fortunately, C++ gives us tools to do precisely this. Modifying individual bits within an object is called **bit manipulation**.

Author's note

Bit manipulation is used a lot in certain programming contexts (e.g. graphics, encryption, compression, optimization), but not as much in general programming.

Because of that, this entire chapter is optional reading. Feel free to skip or skim it and come back later.

Bit flags

Up to this point, we've used variables to hold single values:

```
int foo { 5 }; // assign foo the value 5 (probably uses 32 bits of storage)
std::cout << foo; // print the value 5
```

However, instead of viewing objects as holding a single value, we can instead view them as a collection of individual bits. When individual bits of an object are used as Boolean values, the bits are called **bit flags**.

As an aside...

In computing, a **flag** is a value that signals when some condition exists in a program. With a bit flag, a value of `true` means the condition exists.

Analogously, in the United States, many mailboxes have small (usually red) physical flags attached to the side. When outgoing mail is waiting to be picked up by the carrier, the flag is raised to signal that there is outgoing mail.

To define a set of bit flags, we'll typically use an unsigned integer of the appropriate size (8 bits, 16 bits, 32 bits, etc... depending on how many flags we have), or std::bitset.

```
#include <bitset> // for std::bitset

std::bitset<8> mybitset {}; // 8 bits in size means room for 8 flags
```

Best practice

Bit manipulation is one of the few times when you should unambiguously use unsigned integers (or std::bitset).

In this lesson, we'll show how to do bit manipulation the easy way, via std::bitset. In the next set of lessons, we'll explore how to do it the more difficult but versatile way.

Bit numbering and bit positions

Given a sequence of bits, we typically number the bits from right to left, starting with 0 (not 1). Each number denotes a **bit position**.

```
76543210  Bit position
00000101  Bit sequence
```

Given the bit sequence 0000 0101, the bits that are in position 0 and 2 have value 1, and the other bits have value 0.

Manipulating bits via std::bitset

In lesson 5.3 -- Numeral systems (decimal, binary, hexadecimal, and octal) (https://www.learncpp.com/cpp-tutorial/numeral-systems-decimal-binary-hexadecimal-and-octal/) we already showed how to use a std::bitset to print values in binary. However, this isn't the only useful thing std::bitset can do.

std::bitset provides 4 key member functions that are useful for doing bit manipulation:

- 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 value from a 0 to a 1 or vice versa.

Each of these functions takes the position of the bit we want to operate on as their only argument.

Here's an example:

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

int main()
{
    std::bitset<8> me{ 0b0000'0101 }; // we need 8 bits, start with bit pattern 0000 0101
    me.set(3);   // set bit position 3 to 1 (now we have 0000 1101)
    me.flip(4);  // flip bit 4 (now we have 0001 1101)
    me.reset(4); // set bit 4 back to 0 (now we have 0000 1101)

    std::cout << "All the bits: " << me << '\n';
    std::cout << "Bit 3 has value: " << me.test(3) << '\n';
    std::cout << "Bit 4 has value: " << me.test(4) << '\n';

    return 0;
}
```

This prints:

```
All the bits: 00001101
Bit 3 has value: 1
Bit 4 has value: 0
```

A reminder

We introduced member functions in lesson 5.9 -- Introduction to std::string (https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/). With normal functions, we call `function(object)`. With member functions, we call `object.function()`.

We covered the `0b` binary literal prefix and the `'` digit separator in lesson 5.3 -- Numeral systems (decimal, binary, hexadecimal, and octal) (https://www.learncpp.com/cpp-tutorial/numeral-systems-decimal-binary-hexadecimal-and-octal/).

Giving our bits names can help make our code more readable:

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

int main()
{
    [[maybe_unused]] constexpr int  isHungry   { 0 };
    [[maybe_unused]] constexpr int  isSad      { 1 };
    [[maybe_unused]] constexpr int  isMad      { 2 };
    [[maybe_unused]] constexpr int  isHappy    { 3 };
    [[maybe_unused]] constexpr int  isLaughing { 4 };
    [[maybe_unused]] constexpr int  isAsleep   { 5 };
    [[maybe_unused]] constexpr int  isDead     { 6 };
    [[maybe_unused]] constexpr int  isCrying   { 7 };

    std::bitset<8> me{ 0b0000'0101 }; // we need 8 bits, start with bit pattern 0000 0101
    me.set(isHappy);      // set bit position 3 to 1 (now we have 0000 1101)
    me.flip(isLaughing);  // flip bit 4 (now we have 0001 1101)
    me.reset(isLaughing); // set bit 4 back to 0 (now we have 0000 1101)

    std::cout << "All the bits: " << me << '\n';
    std::cout << "I am happy: " << me.test(isHappy) << '\n';
    std::cout << "I am laughing: " << me.test(isLaughing) << '\n';

    return 0;
}
```

Related content

We cover `[maybe_unused]` in lesson 1.4 -- Variable assignment and initialization (https://www.learncpp.com/cpp-tutorial/variable-assignment-and-initialization/).

In lesson 13.2 -- Unscoped enumerations (https://www.learncpp.com/cpp-tutorial/unscoped-enumerations/), we show how enumerators make an even better collection of named bits.

What if we want to get or set multiple bits at once

std::bitset doesn't make this easy. In order to do this, or if we want to use unsigned integer bit flags instead of std::bitset, we need to turn to more traditional methods. We'll cover these in the next couple of lessons.

The size of std::bitset

One potential surprise is that std::bitset is optimized for speed, not memory savings. The size of a std::bitset is typically the number of bytes

needed to hold the bits, rounded up to the nearest `sizeof(size_t)`, which is 4 bytes on 32-bit machines, and 8-bytes on 64-bit machines.

Thus, a `std::bitset<8>` will typically use either 4 or 8 bytes of memory, even though it technically only needs 1 byte to store 8 bits. Thus, std::bitset is most useful when we desire convenience, not memory savings.

Querying `std::bitset`

There are a few other member functions that are often useful:

- size() returns the number of bits in the bitset.
- count() returns the number of bits in the bitset that are set to `true`. This can be used to determine if all bits are 0 or any bits are 1.
- all() returns a Boolean indicating whether all bits are set to `true`.
- any() returns a Boolean indicating whether any bits are set to `true`.
- none() returns a Boolean indicating whether no bits are set to `true`.

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

int main()
{
    std::bitset<8> bits{ 0b0000'1101 };
    std::cout << bits.size() << " bits are in the bitset\n";
    std::cout << bits.count() << " bits are set to true\n";

    std::cout << std::boolalpha;
    std::cout << "All bits are true: " << bits.all() << '\n';
    std::cout << "Some bits are true: " << bits.any() << '\n';
    std::cout << "No bits are true: " << bits.none() << '\n';

    return 0;
}
```

This prints:

```
8 bits are in the bitset
3 bits are set to true
All bits are true: false
Some bits are true: true
No bits are true: false
```


Next lesson
O.2 Bitwise operators

Back to table of contents

Previous lesson
6.x Chapter 6 summary and quiz

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/ are connected to your provided email address.

110 COMMENTS

Newest