

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

👤 ALEX 🕒 AUGUST 6, 2022

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**.

Bit manipulation is also useful in encryption and compression algorithms.

### Author's note

This entire chapter is optional reading. Feel free to skip it and come back later.

### Bit flags

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

```
1 | 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 acts as a signal for some function or process. Analogously, in real life, a mailbox flag is used to signal that there is something inside the mailbox, so the mailbox doesn't have to be opened to check.

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.

```
1 | #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 [4.16 -- Numeral systems \(decimal, binary, hexadecimal, and octal\)](https://www.learncpp.com/cpp-tutorial/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 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:

```
1 | #include <bitset>
   | #include <iostream>
   |
   | int main()
   | {
   |     std::bitset<8> bits{ 0b0000'0101 }; // we need 8 bits, start with bit pattern 0000
   |     0101
   |     bits.set(3); // set bit position 3 to 1 (now we have 0000 1101)
   |     bits.flip(4); // flip bit 4 (now we have 0001 1101)
   |     bits.reset(4); // set bit 4 back to 0 (now we have 0000 1101)
   |
   |     std::cout << "All the bits: " << bits << '\n';
   |     std::cout << "Bit 3 has value: " << bits.test(3) << '\n';
   |     std::cout << "Bit 4 has value: " << bits.test(4) << '\n';
   |
   |     return 0;
2 | }
```

This prints:

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

## A reminder

If you need a refresher on what the `0b` prefix or the `'` separator is, review [4.16 -- Numeral systems \(decimal, binary, hexadecimal, and octal\)](https://www.learncpp.com/cpp-tutorial/numeral-systems-decimal-binary-hexadecimal-and-octal/) (<https://www.learncpp.com/cpp-tutorial/numeral-systems-decimal-binary-hexadecimal-and-octal/>).

# 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.


 [Next lesson](#)  
0.2 [Bitwise operators](#)



 [Back to table of contents](#)



 [Previous lesson](#)  
5.x [Chapter 5 summary and quiz](#)

B U URL INLINE CODE C++ CODE BLOCK HELP!


Leave a comment...

 Name\*

 Email\* 

 Find a mistake? [Leave a comment!](#) 

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

Notify me about replies: 

POST COMMENT

83 COMMENTS

Newest ▾

