



D

Bit Manipulation

D.1 Introduction

This appendix discusses the manipulation of bits in integer data. We discuss the **bitwise operators** that allow you to access and manipulate the individual bits in bytes of data. We also present **bit fields**—structures that can be used to specify the exact number of bits a variable occupies in memory.

C++ provides extensive **bit-manipulation** capabilities for getting down to the “bits-and-bytes” hardware level. Bit manipulation is typically used in programs that need to communicate directly with hardware. It’s also used in encryption data compression and other algorithms.¹ We introduce each **bitwise operator** and discuss how to save memory by using **bit fields**.

D.2 Bitwise Operators

All data is represented internally by computers as sequences of bits. Each bit can assume the value 0 or the value 1. On most systems, a sequence of eight bits forms a **byte**—the standard storage unit for a variable of type `char`. Other data types are stored in larger numbers of bytes. Bitwise operators manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both signed and unsigned)—typically, **unsigned integers**.

The bitwise operator discussions in this section show the binary representations of the integer operands. For a detailed explanation of the binary (base-2) number system, see online Appendix B. Some of these programs might not work on your system without modification due to the **machine-dependent nature of bitwise manipulations**.

The bitwise operators are:

- **bitwise AND (&)**,
- **bitwise inclusive OR (|)**,
- **bitwise exclusive OR (^)**,
- **left shift (<<)**,
- **right shift (>>)** and
- **bitwise complement (~)**—also known as the **one’s complement**.

1. “Bit manipulation.” Wikipedia. Wikimedia Foundation. Accessed April 4, 2023. https://en.wikipedia.org/wiki/Bit_manipulation.

D.2 Appendix D Bit Manipulation

We've been using `&`, `<<` and `>>` for other purposes—this is a classic example of operator overloading. Detailed discussions of each bitwise operator appear in the following examples. The bitwise operators are summarized in the following table.

Operator	Name	Description
<code>&</code>	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
<code> </code>	bitwise inclusive OR	The bits in the result are set to 1 if one or both of the corresponding bits in the two operands is 1.
<code>^</code>	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
<code><<</code>	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand. Fills from the right with 0 bits.
<code>>></code>	right shift with sign extension	Shifts the bits of the first operand right by the number of bits specified by the second operand. The method of filling from the left is machine-dependent.
<code>~</code>	bitwise complement	All 0 bits are set to 1, and all 1 bits are set to 0.

Printing a Binary Representation of an Integral Value

When using the bitwise operators, it's useful to illustrate their precise effects by printing values in their binary representation. The program of Fig. D.1 prints an unsigned integer in its binary representation in groups of eight bits each.

```
1 // figE_01.cpp
2 // Printing an unsigned integer in bits.
3 #include <format>
4 #include <iostream>
5
6 void displayBits(unsigned); // prototype
7
8 int main() {
9     unsigned inputValue{0}; // integral value to print in binary
10
11     std::cout << "Enter an unsigned integer: ";
12     std::cin >> inputValue;
13     displayBits(inputValue);
14 }
15
16 // display bits of an unsigned integer value
17 void displayBits(unsigned value) {
18     const unsigned SHIFT{8 * sizeof(unsigned) - 1};
19     const unsigned MASK{static_cast<const unsigned>(1 << SHIFT)};
20
21     std::cout << std::format("{:10d} = ", value);
22 }
```

Fig. D.1 | Printing an unsigned integer in bits. (Part I of 2.)

```

23 // display bits
24 for (unsigned i{1}; i <= SHIFT + 1; ++i) {
25     std::cout << (value & MASK ? '1' : '0');
26     value <<= 1; // shift value left by 1
27
28     if (i % 8 == 0) { // output a space after 8 bits
29         std::cout << ' ';
30     }
31 }
32
33 std::cout << "\n";
34 }

```

```

Enter an unsigned integer: 65000
65000 = 00000000 00000000 11111101 11101000

```

```

Enter an unsigned integer: 29
29 = 00000000 00000000 00000000 00011101

```

Fig. D.1 | Printing an unsigned integer in bits. (Part 2 of 2.)

Function `displayBits` (lines 17–34) uses the **bitwise AND** operator to combine variable `value` with constant `MASK`. The **bitwise AND** operator is often used with an operand called a **mask**—an integer value with specific bits set to 1. Masks hide some bits in a value while selecting other bits. In `displayBits`, line 19 initializes constant `MASK` with `1 << SHIFT`. The value of constant `SHIFT` was calculated in line 18 by

```
8 * sizeof(unsigned) - 1
```

This expression multiplies an `unsigned` object’s number of bytes by 8 (the number of bits in a byte) to get the total number of bits required to store an `unsigned` object, then subtracts 1. The bit representation of `1 << SHIFT` on a computer that represents `unsigned` objects in four bytes of memory is

```
10000000 00000000 00000000 00000000
```

The **left-shift operator** shifts the value 1 from the low-order (rightmost) bit to the high-order (leftmost) bit in `MASK` and fills with 0 bits from the right. Line 25 prints a 1 or a 0 for the current leftmost bit of `value`.

Assume `value` contains 65000, which in bits is

```
00000000 00000000 11111101 11101000
```

When you combine `value` and `MASK` using `&`, all the bits except the high-order bit in `value` are “masked off” (hidden) because any bit “ANDed” with 0 yields 0.

If the leftmost bit is 1, `value & MASK` evaluates to

```

00000000 00000000 11111101 11101000 (value)
10000000 00000000 00000000 00000000 (MASK)
-----
00000000 00000000 00000000 00000000 (value & MASK)

```

D.4 Appendix D Bit Manipulation

The value 0 is treated as false, and `displayBits` displays a 0. Then line 26 shifts `value` left by one bit with the expression `value <<= 1`. These steps are repeated for each bit in `value`. Eventually, a 1 bit shifts into the leftmost bit position. Consider the following bitwise AND:

```
11111101 11101000 00000000 00000000    (value)
10000000 00000000 00000000 00000000    (MASK)
-----
10000000 00000000 00000000 00000000    (value & MASK)
```

Both left bits are 1s, so the `&` expression's result is nonzero (true), and `displayBits` displays a 1. The following table summarizes combining two bits with the bitwise AND (`&`) operator.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

The program of Fig. D.2 demonstrates the **bitwise AND operator**, the **bitwise inclusive OR operator**, the **bitwise exclusive OR operator** and the **bitwise complement operator**. Function `displayBits` is identical to the one in Fig. D.1.

```
1 // figE_02.cpp
2 // Bitwise AND, inclusive OR,
3 // exclusive OR and complement operators.
4 #include <format>
5 #include <iostream>
6
7 void displayBits(unsigned); // prototype
8
9 int main() {
10 // demonstrate bitwise &
11 unsigned number1{2179876355};
12 unsigned mask{1};
13 std::cout << "The result of combining the following\n";
14 displayBits(number1);
15 displayBits(mask);
16 std::cout << "using the bitwise AND operator & is\n";
17 displayBits(number1 & mask);
18
19 // demonstrate bitwise |
20 number1 = 15;
21 unsigned setBits{241};
22 std::cout << "\nThe result of combining the following\n";
23 displayBits(number1);
24 displayBits(setBits);
```

Fig. D.2 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 1 of 3.)

```

25     std::cout << "using the bitwise inclusive OR operator | is\n";
26     displayBits(number1 | setBits);
27
28     // demonstrate bitwise exclusive OR
29     number1 = 139;
30     unsigned number2{199};
31     std::cout << "\nThe result of combining the following\n";
32     displayBits(number1);
33     displayBits(number2);
34     std::cout << "using the bitwise exclusive OR operator ^ is\n";
35     displayBits(number1 ^ number2);
36
37     // demonstrate bitwise complement
38     number1 = 21845;
39     std::cout << "\nThe one's complement of\n";
40     displayBits(number1);
41     std::cout << "is\n";
42     displayBits(~number1);
43 }
44
45 // display bits of an unsigned integer value
46 void displayBits(unsigned value) {
47     const unsigned SHIFT{8 * sizeof(unsigned) - 1};
48     const unsigned MASK{static_cast<const unsigned>(1 << SHIFT)};
49
50     std::cout << std::format("{:10d} = ", value);
51
52     // display bits
53     for (unsigned i{1}; i <= SHIFT + 1; ++i) {
54         std::cout << (value & MASK ? '1' : '0');
55         value <<= 1; // shift value left by 1
56
57         if (i % 8 == 0) { // output a space after 8 bits
58             std::cout << ' ';
59         }
60     }
61
62     std::cout << "\n";
63 }

```

The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
1 = 00000000 00000000 00000000 00000001

The result of combining the following
15 = 00000000 00000000 00000000 00001111
241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
255 = 00000000 00000000 00000000 11111111

Fig. D.2 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 2 of 3.)

```

The result of combining the following
  139 = 00000000 00000000 00000000 10001011
  199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
  76 = 00000000 00000000 00000000 01001100

The one's complement of
  21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010

```

Fig. D.2 | Bitwise AND, inclusive OR, exclusive OR and complement operators. (Part 3 of 3.)

Bitwise AND Operator (&)

In Fig. D.2, line 11 uses 2179876355 (10000001 11101110 01000110 00000011) to initialize variable `number1`, and line 12 uses 1 (00000000 00000000 00000000 00000001) to initialize variable `mask`. When `mask` and `number1` are combined using the **bitwise AND operator (&)** in the expression `number1 & mask` (line 17), the result is 00000000 00000000 00000000 00000001. All the bits except the low-order bit in variable `number1` are “masked off” (hidden) by “ANDing” with constant `MASK`.

Bitwise Inclusive OR Operator (|)

The **bitwise inclusive OR operator** sets specific bits to 1 in an operand. In Fig. D.2, line 20 assigns 15 (00000000 00000000 00000000 00001111) to variable `number1`, and line 21 uses 241 (00000000 00000000 00000000 11110001) to initialize variable `setBits`. When `number1` and `setBits` are combined using the **bitwise inclusive OR operator (|)** in the expression `number1 | setBits` (line 16), the result is 255 (00000000 00000000 00000000 11111111). The following table summarizes combining two bits with the **bitwise inclusive-OR operator**.

Bit 1	Bit 2	Bit 1 Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Bitwise Exclusive OR (^)

The **bitwise exclusive OR operator (^)** sets each bit in the result to 1 if exactly one of the corresponding bits in its two operands is 1. In Fig. D.2, lines 29–30 give variables `number1` and `number2` the values 139 (00000000 00000000 00000000 10001011) and 199 (00000000 00000000 00000000 11000111), respectively. When these variables are combined with the **bitwise exclusive OR operator** in the expression `number1 ^ number2` (line 35), the result is 00000000 00000000 00000000 01001100. The following table summarizes combining two bits with the **bitwise exclusive OR operator**.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Bitwise Complement (~)

The **bitwise complement operator** (~) sets all 1 bits in its operand to 0 in the result and all 0 bits to 1. This is referred to as “taking the **one’s complement** of the value.” In Fig. D.2, line 38 assigns variable `number1` the value 21845 (00000000 00000000 01010101 01010101). When the expression `~number1` evaluates, the result is (11111111 11111111 10101010 10101010).

Bitwise Shift Operators

Figure D.3 demonstrates the **left-shift** (<<) and the **right-shift** (>>) operators. Function `displayBits` is identical to the one in Fig. D.1.

```

1 // figE_3.cpp
2 // Using the bitwise shift operators.
3 #include <format>
4 #include <iostream>
5
6 void displayBits(unsigned); // prototype
7
8 int main() {
9     unsigned number1{960};
10
11     // demonstrate bitwise left shift
12     std::cout << "The result of left shifting\n";
13     displayBits(number1);
14     std::cout << "8 bit positions using the left-shift operator is\n";
15     displayBits(number1 << 8);
16
17     // demonstrate bitwise right shift
18     std::cout << "\nThe result of right shifting\n";
19     displayBits(number1);
20     std::cout << "8 bit positions using the right-shift operator is\n";
21     displayBits(number1 >> 8);
22 }
23
24 // display bits of an unsigned integer value
25 void displayBits(unsigned value) {
26     const unsigned SHIFT{8 * sizeof(unsigned) - 1};
27     const unsigned MASK{static_cast<const unsigned>(1 << SHIFT)};
28

```

Fig. D.3 | Bitwise shift operators. (Part I of 2.)

```

29     std::cout << std::format("{:10d} = ", value);
30
31     // display bits
32     for (unsigned i{1}; i <= SHIFT + 1; ++i) {
33         std::cout << (value & MASK ? '1' : '0');
34         value <<= 1; // shift value left by 1
35
36         if (i % 8 == 0) { // output a space after 8 bits
37             std::cout << ' ';
38         }
39     }
40
41     std::cout << "\n";
42 }

```

```

The result of left shifting
960 = 00000000 00000000 00000011 11000000
8 bit positions using the left-shift operator is
245760 = 00000000 00000011 11000000 00000000

The result of right shifting
960 = 00000000 00000000 00000011 11000000
8 bit positions using the right-shift operator is
3 = 00000000 00000000 00000000 00000011

```

Fig. D.3 | Bitwise shift operators. (Part 2 of 2.)

Left-Shift Operator

The **left-shift operator** (<<) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with 0s; bits shifted off the left are lost. In Fig. D.3, line 9 initializes variable `number1` with the value 960 (00000000 00000000 00000011 11000000). The result of left-shifting `number1` eight bits in the expression `number1 << 8` (line 15) is 245760 (00000000 00000011 11000000 00000000).

Right-Shift Operator

The **right-shift operator** (>>) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an `unsigned` integer causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost. In Fig. D.3, the result of right-shifting `number1` in the expression `number1 >> 8` (line 21) is 3 (00000000 00000000 00000000 00000011).



The result of shifting a value is undefined if the right operand is negative or if the right operand is greater than or equal to the number of bits in which the left operand is stored.



The result of right-shifting a signed value is machine-dependent. Some machines fill with zeros. Others use the sign bit.

Bitwise Assignment Operators

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These **bitwise assignment operators** are shown in the following table. Each modifies its left operand.

Bitwise assignment operators

<code>&=</code>	Bitwise AND assignment operator.
<code> =</code>	Bitwise inclusive OR assignment operator.
<code>^=</code>	Bitwise exclusive OR assignment operator.
<code><<=</code>	Left-shift assignment operator.
<code>>>=</code>	Right-shift with sign extension assignment operator.

D.3 Bit Fields

You can specify the number of bits in which an integral type or enum type member of a class or a structure is stored. Such a member is referred to as a **bit field** and enables **better memory utilization** by storing data in the minimum number of bits required. Bit field members must have an integral or enum type.

The following structure definition contains three unsigned bit fields—`face`, `suit` and `color`—used to represent a card from a deck of 52 cards:

```
struct BitCard {
    unsigned face : 4;
    unsigned suit : 2;
    unsigned color : 1;
};
```

You declare a bit field by following an integral type or enum type member with a colon (`:`) and an **integer constant** representing the **bit field's width**—the number of bits in which the member is stored.

`BitCard`'s definition indicates that `face` is stored in four bits, `suit` in 2 bits and `color` in one bit. The number of bits for each is based on the member's desired range of values.

- Member `face` stores values between 0 (Ace) and 12 (King)—four bits can store a value between 0 and 15.
- Member `suit` stores values between 0 and 3 (0 = Diamonds, 1 = Hearts, 2 = Clubs, 3 = Spades)—two bits can store a value between 0 and 3.
- Member `color` stores either 0 (Red) or 1 (Black)—one bit can store either 0 or 1.

The program in Figs. D.4–D.6 creates array `deck` containing `BitCard` structures (line 23 of Fig. D.4). The constructor inserts the 52 cards in the `deck` array, and the function `deal` prints the 52 cards. Bit fields are accessed via the dot (`.`) operator (lines 11–13 and 21–26 of Fig. D.5).

```
1 // Fig. D.4: DeckOfCards.h
2 // Definition of class DeckOfCards that
3 // represents a deck of playing cards.
4 #include <array>
5
```

Fig. D.4 | Definition of class `DeckOfCards` that represents a deck of playing cards. (Part 1 of 2.)

```

6 // BitCard structure definition with bit fields
7 struct BitCard {
8     unsigned face : 4; // 4 bits; 0-15
9     unsigned suit : 2; // 2 bits; 0-3
10    unsigned color : 1; // 1 bit; 0-1
11 };
12
13 // DeckOfCards class definition
14 class DeckOfCards {
15 public:
16     static const int faces{13};
17     static const int colors{2}; // black and red
18     static const int numberOfCards{52};
19
20     DeckOfCards(); // constructor initializes deck
21     void deal() const; // deals cards in deck
22 private:
23     std::array<BitCard, numberOfCards> deck; // represents deck of cards
24 };

```

Fig. D.4 | Definition of class DeckOfCards that represents a deck of playing cards. (Part 2 of 2.)

```

1 // Fig. D.5: DeckOfCards.cpp
2 // Member-function definitions for class DeckOfCards that simulates
3 // the shuffling and dealing of a deck of playing cards.
4 #include <format>
5 #include <iostream>
6 #include "DeckOfCards.h" // DeckOfCards class definition
7
8 // no-argument DeckOfCards constructor initializes deck
9 DeckOfCards::DeckOfCards() {
10     for (size_t i{0}; i < deck.size(); ++i) {
11         deck[i].face = i % faces; // faces in order
12         deck[i].suit = i / faces; // suits in order
13         deck[i].color = i / (faces * colors); // colors in order
14     }
15 }
16
17 // deal cards in deck
18 void DeckOfCards::deal() const {
19     for (size_t k1{0}, k2{k1 + deck.size() / 2};
20         k1 < deck.size() / 2 - 1; ++k1, ++k2) {
21         std::cout << std::format("Card: {:3d}", deck[k1].face)
22             << std::format(" Suit: {:2d}", deck[k1].suit)
23             << std::format(" Color: {:2d}", deck[k1].color)
24             << std::format(" Card: {:3d}", deck[k2].face)
25             << std::format(" Suit: {:2d}", deck[k2].suit)
26             << std::format(" Color: {:2d}\n", deck[k2].color);
27     }
28 }

```

Fig. D.5 | Member-function definitions for class DeckOfCards.

```

1 // figE_06.cpp
2 // Card shuffling and dealing program.
3 #include "DeckOfCards.h" // DeckOfCards class definition
4
5 int main() {
6     DeckOfCards deckOfCards; // create DeckOfCards object
7     deckOfCards.deal(); // deal the cards in the deck
8 }

```

```

Card: 0 Suit: 0 Color: 0   Card: 0 Suit: 2 Color: 1
Card: 1 Suit: 0 Color: 0   Card: 1 Suit: 2 Color: 1
Card: 2 Suit: 0 Color: 0   Card: 2 Suit: 2 Color: 1
Card: 3 Suit: 0 Color: 0   Card: 3 Suit: 2 Color: 1
Card: 4 Suit: 0 Color: 0   Card: 4 Suit: 2 Color: 1
Card: 5 Suit: 0 Color: 0   Card: 5 Suit: 2 Color: 1
Card: 6 Suit: 0 Color: 0   Card: 6 Suit: 2 Color: 1
Card: 7 Suit: 0 Color: 0   Card: 7 Suit: 2 Color: 1
Card: 8 Suit: 0 Color: 0   Card: 8 Suit: 2 Color: 1
Card: 9 Suit: 0 Color: 0   Card: 9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0  Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0  Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0  Card: 12 Suit: 2 Color: 1
Card: 0 Suit: 1 Color: 0   Card: 0 Suit: 3 Color: 1
Card: 1 Suit: 1 Color: 0   Card: 1 Suit: 3 Color: 1
Card: 2 Suit: 1 Color: 0   Card: 2 Suit: 3 Color: 1
Card: 3 Suit: 1 Color: 0   Card: 3 Suit: 3 Color: 1
Card: 4 Suit: 1 Color: 0   Card: 4 Suit: 3 Color: 1
Card: 5 Suit: 1 Color: 0   Card: 5 Suit: 3 Color: 1
Card: 6 Suit: 1 Color: 0   Card: 6 Suit: 3 Color: 1
Card: 7 Suit: 1 Color: 0   Card: 7 Suit: 3 Color: 1
Card: 8 Suit: 1 Color: 0   Card: 8 Suit: 3 Color: 1
Card: 9 Suit: 1 Color: 0   Card: 9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0  Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0  Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0  Card: 12 Suit: 3 Color: 1

```

Fig. D.6 | Bit fields used to store a deck of cards.

Unnamed Bit Fields

An **unnamed bit field** can be used as **padding** in the structure. For example, the structure definition uses an unnamed three-bit field as padding—nothing can be stored in those three bits. Member `b` is stored in another storage unit:

```

struct Example {
    unsigned a : 13;
    unsigned   : 3; // align to next storage-unit boundary
    unsigned b : 4;
};

```




Zero-Width Unnamed Bit Fields

An **unnamed bit field with a zero width** is used to align the next bit field on a new storage-unit boundary. For example, the following structure definition uses an unnamed 0-bit field to skip the remaining bits (as many as there are) of the storage unit in which `a` is stored and align `b` on the next storage-unit boundary:

```
struct Example {  
    unsigned a : 13;  
    unsigned : 0; // align to next storage-unit boundary  
    unsigned b : 4;  
};
```

Bit Field Notes

When using bit fields, keep the following in mind:

- SE  • Bit-field manipulations are machine-dependent. Some computers allow bit fields to cross storage unit boundaries, whereas others do not.
- Err  • Attempting to take the address of a bit field is a compilation error. The & operator may not be used with bit fields because a pointer can designate only a particular byte in memory. On the other hand, bit fields can start in the middle of a byte.
- Perf  • Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine-language operations to access only portions of an addressable storage unit.