MODERN C++ DESIGN PATTERNS

Bit-Twiddling

by Prasanna Ghali

C++ Integer Types

- Microsoft compiler is 32-/64-bit compiler while GCC and Clang are 64-bit compiler
- `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

Type Name	Number of Bytes
`char`	1
`short`	2
`int`	4
`long`	4/8
`long long`	8
`size_t`	4/8

Modern C++ Integer Types (1/2)

- Declared in <cstdint> since C++11

Signed Type Name	Number of Bytes (GCC/Clang)
intmax_t	8
int8_t/int16_t/int32_t/int64_t	1/2/4/8
int_least8_t/int_least16_t/ int_least32_t/int_least64_t	1/2/4/8
int_fast8_t/int_fast16_t/ int_fast32_t/int_fast64_t	1/8/8/8

4/4 for Microsoft

Modern C++ Integer Types (2/2)

- Declared in `<cstdint>` since C++11

Unsigned Type Name	Number of Bytes (in GCC/Clang)
`uintmax_t`	8
`uint8_t/uint16_t/uint32_t/uint64_t`	1/2/4/8
`uint_least8_t/uint_least16_t/uint_least32_t/uint_least64_t`	1/2/4/8
`uint_fast8_t/uint_fast16_t/uint_fast32_t/uint_fast64_t`	1/8/8/8

4/4 for Microsoft

Modern C++ Integer Literals

- Integer literals allow values of integer type to be directly used in expressions

```
uint64_t d = 42ul;
uint64_t o = 052Ul;
uint64_t x = 0x2auL; // 0x2AUL
uint64_t b = 0b0010'1010UL; // 0b00101010UL
```

0b prefix for integer literals is new since C++14

digit separator makes large values readable

Deciding On Integer Type (1/2)

- What is most optimal integer type to store n digit decimal number?
- Number of bits N to store n digit decimal value: $N = \left\lceil \frac{n}{\log_{10} 2} \right\rceil$ or $N = \left\lceil \frac{n \ln 10}{\ln 2} \right\rceil$

Deciding On Integer Type (2/2)

- What is most optimal integer type to store decimal number d?
- Number of bits N to store decimal value d:
  $N = \left\lceil \frac{\log_{10} d}{\log_{10} 2} \right\rceil$ or $N = \left\lceil \frac{\ln d}{\ln 2} \right\rceil$

Why Bit Twiddling?

- Not all data may be byte-aligned: common in compression, encryption, or internal hardware design
- Not all addresses point to RAM; GPIO pin in embedded device is individual bit interfacing with external device
- Bit twiddling can lead to faster operations
- Discrete information can be encoded into single unit for more efficient storage

Terminology: Bit

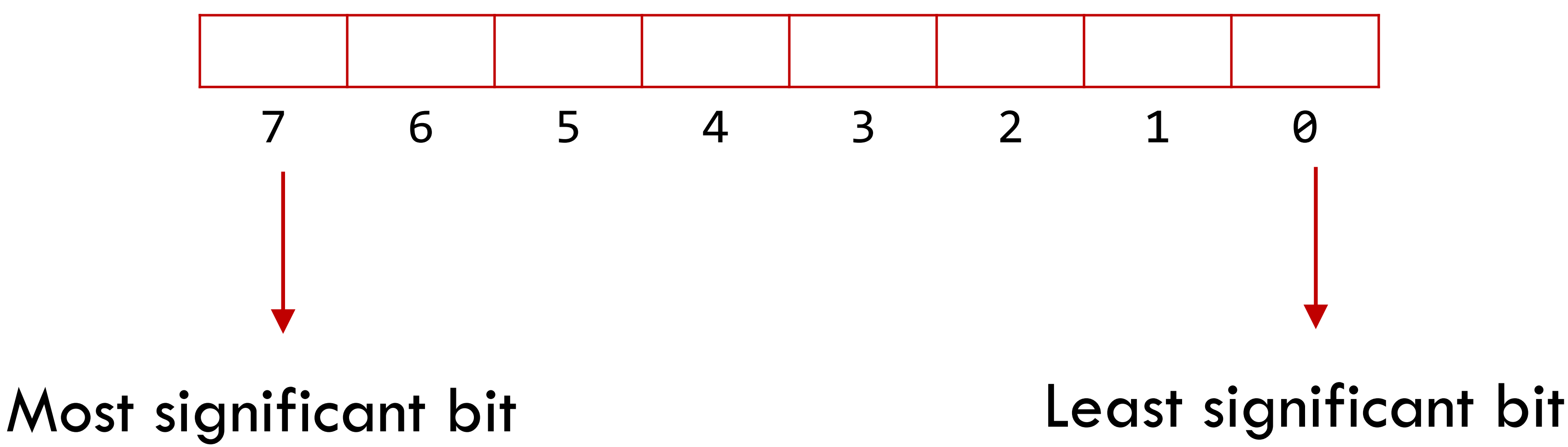
- Unit of data storage in execution environment large enough to hold object that may have one of two values

Terminology: Byte

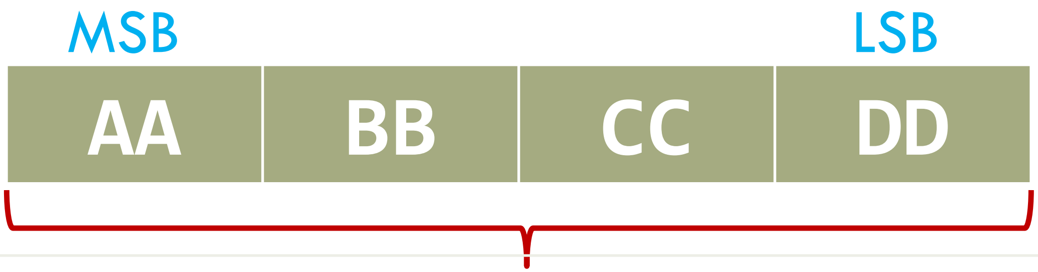
- Memory available to C++ program consists of one or more sequences of contiguous *bytes*
  - Every byte has unique address
  - Byte is at least large enough to contain any member of basic execution character set
  - Byte is made up contiguous sequence of *at least* 8 bits
  - Number of bits is *implementation defined*; reported by `CHAR_BIT` macro in `<climits>`
  - Byte in Windows and POSIX [all versions of Linux and variants] has 8 bits
  - Usually programmers use `char` to represent member of character set and `unsigned char` to represent 8-bit value
  - Since C++20, `std::byte` provides concept of byte

Numbering Bits

- By convention, each bit is numbered according to power of 2 it represents
- Single byte of memory is 8 bits, numbered 0 to 7:

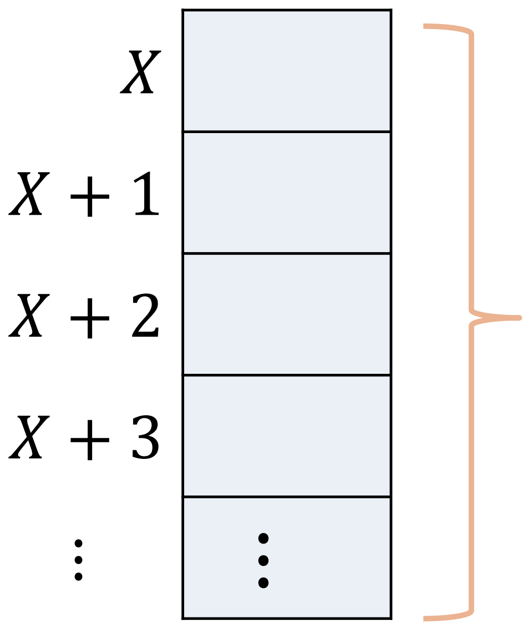


Terminology: Endianness (1/2)



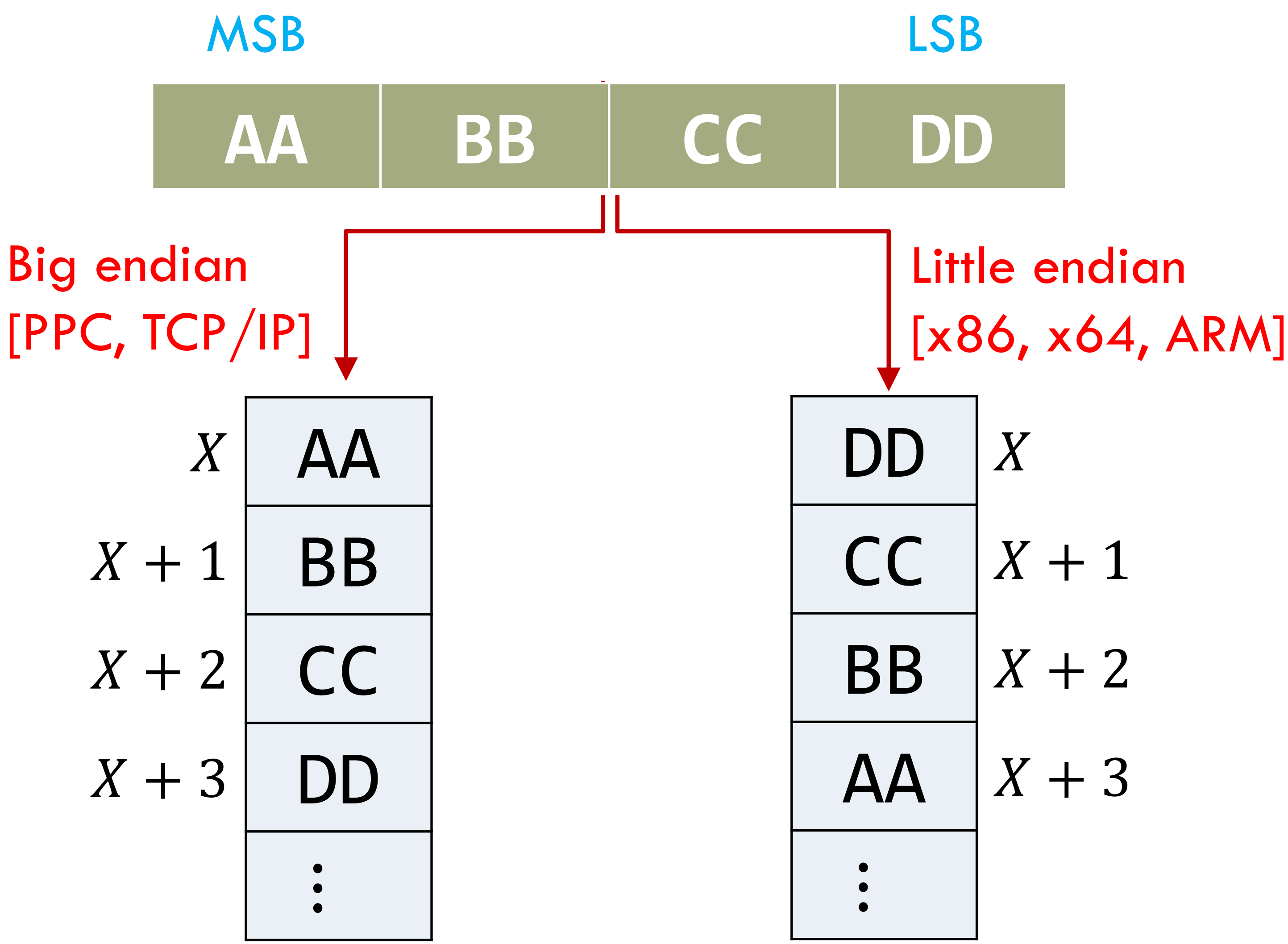
32-bit CPU register containing value `0xAABBCCDD`

How is this 32-bit value stored in memory?



Byte-sized memory locations with addresses X, $X + 1$, and so on

Terminology: Endianness (2/2)



Determining Endianness (1/2)


```
bool bigendian() { // pre C++20
  uint32_t val{0x01234567U};
  return (0x67 != *reinterpret_cast<uint8_t *>(&val))
         ? true : false;
}
```

```
#include <bit> // since C++20
bool bigendian() {
  return (std::endian::native == std::endian::big)
         ? true : false;
}
```

Determining Endianness (2/2)


```
#if __cplusplus >= 202002L // since C++20
  #include <bit>
#endif
bool bigendian() {
#if __cplusplus >= 202002L // since C++20
  return (std::endian::native == std::endian::big)
         ? true : false;
#else // pre C++20
  uint32_t val{0x01234567U};
  return (0x67 != *reinterpret_cast<uint8_t *>(&val))
         ? true : false;
#endif
}
```

Terminology: Bit Twiddling

- Flag: Bit or bits in value that represent something useful that you want to *include* or *omit* or *amend*
- Mask: Bit pattern used to *access* specific bits in a value
- You use mask to read flag value or write new value to flag

Bitwise Operators

- Operands must be integral types

Operator	Description
~	One's complement
<<	Left shift
>>	Right shift
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

same precedence

high to low precedence order

Left to right associative order for binary operators
Right-to-left associative order for unary operator

Bitwise Operators: Caution

- If operand is signed and its value is negative, sign bit is handled by bitwise operators in machine-dependent manner
  - Right shift operator's behavior with value of sign bit is implementation-dependent
  - Therefore, only unsigned types must be used with bitwise operators
- If operand is small integer, it is promoted to larger integral type

One's Complement Operator: ~

- Used to *invert* bits of operand
- Modern C++ provides alternative keyword `compl`

```
uint8_t b {0b0010'0001};

// promotion to uint32_t:  0b0000'0000'0000'0000'0000'0000'0010'0001
// inversion of bits:      0b1111'1111'1111'1111'1111'1111'1101'1110
b = ~b; // truncated to uint8_t: 0b1101'1110

// 0xabcdef:       0b0000'0000'1010'1011'1100'1101'1110'1111
// compl 0xabcdef: 0b1111'1111'0101'0100'0011'0010'0001'0000
uint32_t i = compl 0xabcdef;
```

Bitwise OR Operator: |

- Used to *set* one or more bits
- Modern C++ provides alternative keyword `bitor`

```
uint16_t p {0x9105}, q {3};

// p:                  0b1001'0001'0000'0101
// q:                  0b0000'0000'0000'0011
// r = p | q:          0b1001'0001'0000'0111
// 0xFF00:             0b1111'1111'0000'0000
// s = r bitor 0xFF00: 0b1111'1111'0000'0111
uint16_t r = p | q, s = r bitor 0xFF00;
```

Bitwise AND Operator: &

- Used to *clear* one or more bits
- Modern C++ provides alternative keyword `bitand`

```
uint16_t p {0x9105}, q {2};

// p:                      0b1001'0001'0000'0010
// q:                      0b0000'0000'0000'0110
// r = p & q:              0b0000'0000'0000'0010
// 0xFF00:                 0b1111'1111'0000'0000
// s = r bitand 0xFF00:    0b0000'0000'0000'0000
uint16_t r = p & q, s = r bitand 0xFF00;
```

Bitwise XOR Operator: ^

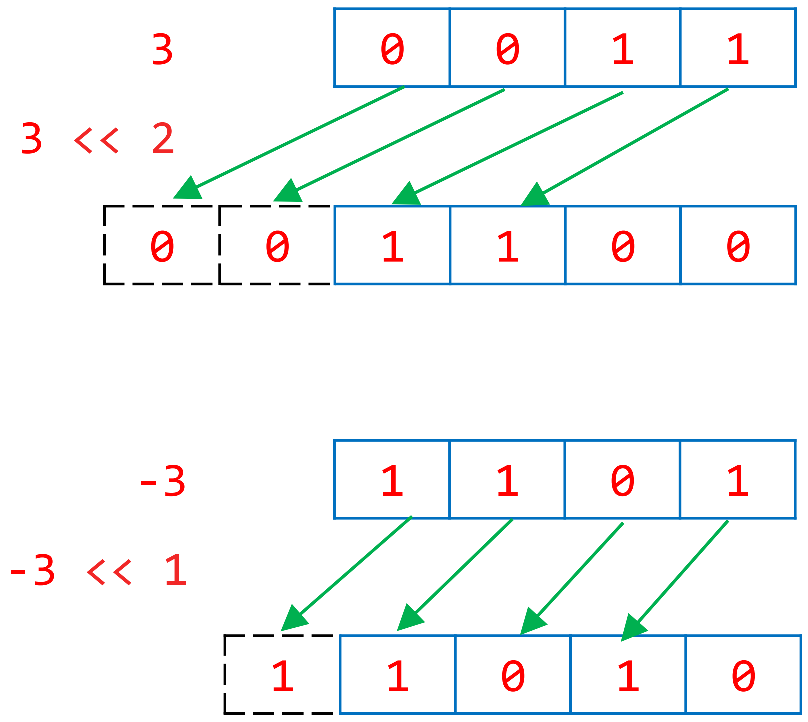
- Used to *toggle* one or more bits
- Alternative keyword xor
- Useful for permuting and counting bits; hence used in compression and encoding algorithms

```
uint16_t p {0x9105}, q {3};

// p:              0b1001'0001'0000'0101
// q:              0b0000'0000'0000'0011
// r = p ^ q:      0b1001'0001'0000'0110
// 0xFF00:         0b1111'1111'0000'0000
// s = r ^ 0xFF00: 0b0110'1110'0000'0110
uint16_t r = p ^ q, s = r xor 0xFF00;
```

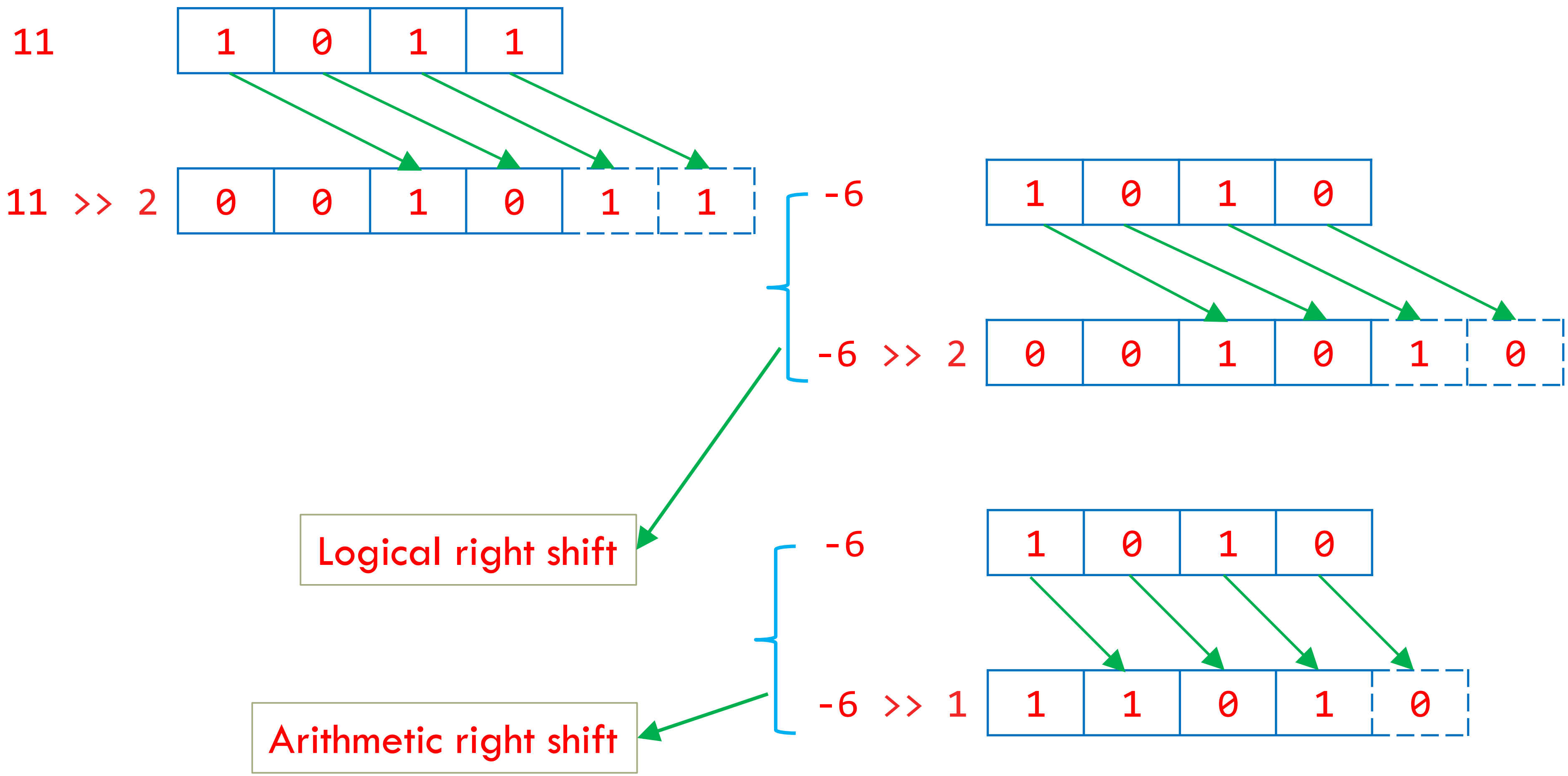
Left Shift Operator: <<

- Moves binary digits of memory word to left



Right Shift Operator: >>

- Moves binary digits of memory word right



Shifts and Division

- For 2's complement arithmetic, division and multiplication by powers of 2 is right and left shift, respectively
  - True for unsigned types and for *multiplication with signed types* [with most C/C++ compilers]
  - Division with signed types rounds to zero [as one would expect], but right shift is division that rounds to $-\infty$

```
uint32_t ua {1};
uint32_t uc {ua >> 1}; // uc == 0
uint32_t ud {ua / 2};  // ud == 0
uint32_t ue {ua << 1}; // ue == 2
uint32_t uf {ua * 2};  // uf == 2
```

```
int32_t a {-1};
int32_t c {a >> 1}; // c == -1
int32_t d {a / 2};  // d == 0
int32_t e {a << 1}; // e == -2
int32_t f {a * 2};  // f == -2
```

Shifts and Masking

- Commonly used for positioning mask

```
std::cout << "Enter bit position to toggle: ";
uint32_t bit;
std::cin >> bit;
uint32_t mask {1U << bit};

uint32_t ui{0x19abcdef};
std::cout << std::hex << "0x" << (ui^mask) << "\n";
```

Lots More Useful Operations …

- Encode multiple values into single variable
- How to extract particular value from encoded variable
- How to replace old value in encoded variable with new value
- To conditionally set or clear bits in variable using mask
- …

The `std::bitset` Type (1/3)

- Class template with template non-type parameter for creating fixed size container for bitwise data manipulation

The std::bitset Type (2/3)


```
// pass/fail quiz results for 30 students
uint32_t quiz{};  // start with all 30 students failing
quiz |= 1U << 27; // student with id 27 has passed

// print status of id 27
std::cout << "ID 27: " << (quiz & (1U << 27) ? "P" : "F") << '\n';

quiz &= ~(1U << 27); // later fail id 27 due to poor attendance
// print status of id 27
std::cout << "ID 27: " << (quiz & (1U << 27) ? "P" : "F") << '\n';
```

```
#include <bitset>

// pass/fail quiz results for 30 students
std::bitset<30> quiz; // start with all 30 students failing
quiz.set(27);         // student with id 27 has passed
std::cout << "ID 27: " << (quiz[27] ? "P" : "F") << '\n';
quiz.reset(27);       // id 27 fails
std::cout << "ID 27: " << (quiz[27] ? "P" : "F") << '\n';
```

The `std::bitset` Type (3/3)

- See *bitset.cpp* for defining, initializing, and using `bitset`s