ECE 150 Integers

Douglas Harder

December 2023

1 Integers

An integer is one of four primitive data types in C++, including also characters, floating-point numbers and Boolean values.
An integer represents what you would expect: the integers you have learned about since elementary school, including negative integers, 0 and positive integers. However, integers can be arbitrarily large, but computer memory is limited. Thus, there are a number of different types that store integers. Integers are stored in memory as binary numbers, so while an integer may appear to be base 10 both as literals and as output, any literal integer is converted to a binary representation before it is stored, and when an integer is printed to the screen, the binary representation is reverted back to the decimal representation. The three integer data types are `short`, `int` and `long`, the first and last being abbreviations for *short integers* and *long integers*, respectively. The `short` type occupies two (2) bytes, `int` occupies four (4) bytes, and `long` occupies eight (8) bytes in memory.

Not on the examination

The Microsoft compiler only uses four (4) bytes for `long`. To get an eight-byte integer, you must use `long long`; that is a *long long integer*. To see how many bytes a type occupies, you can always execute this statement:

```
std::cout << sizeof( short ) << std::endl;
std::cout << sizeof( int )   << std::endl;
std::cout << sizeof( long )  << std::endl;
```

In some cases, you need both positive and negative integers; for example, if you are calculating differences between values. In other cases, you may only need 0 and positive integers; for example, if you are counting items. Consequently, C and C++ allow for both signed and unsigned integer types. Signed integer types are exactly those you see above `short`, `int` and `long`. Unsigned types are these three prefixed by the keyword `unsigned`, so `unsigned short`, `unsigned int` and `unsigned long`.

2 Unsigned integers

We will first consider all possible integers that can be stored in unsigned integers. When counting in decimal, we perform a carry whenever we add one to nine ($1 + 9 = 10$), as there are ten digits being used. In binary, we have only two digits: 0 and 1, so thus, we perform a carry whenever we add one to one ($1 + 1 = 10$, and so also $1 + 1 + 1 = 11$). Thus, when counting we have the following equivalent representations:

0	0	8	1000	16	10000	24	11000	32	100000
1	1	9	1001	17	10001	25	11001	33	100001
2	10	10	1010	18	10010	26	11010	34	100010
3	11	11	1011	19	10011	27	11011	35	100011
4	100	12	1100	20	10100	28	11100	36	100100
5	101	13	1101	21	10101	29	11101	37	100101
6	110	14	1110	22	10110	30	11110	38	100110
7	111	15	1111	23	10111	31	11111	39	100111

Thus, if I had 42 items, I could also say I have 101010_2. The subscript "2" emphasizes that this is a binary number, and not one-hundred and one thousand and ten. We often refer to decimal digits as simply digits, but binary digits (there are only two) are referred to as bits (from *bi*nary dig*its*).

Not on the examination

While it looks like binary numbers are much longer than decimal numbers (1957 in binary is 11110100101_2), but in reality, it isn't significantly worse: if a number is written using n decimal digits, then in binary, it will be approximately $\frac{\ln(10)}{\ln(2)}n \approx 3.3n$ binary digits.

First, looking at the above table, powers of two have a simple representation: 2^n is $1\underbrace{0\cdots0}_{n \text{ zeros}}$. Also notice that $2^n - 1$ is therefore $\underbrace{1\cdots1}_{n \text{ ones}}$.
Thus, we have the following:

Type	Bits	Maximum (binary)	Maximum (decimal)
`short`	16	$\underbrace{11\cdots11}_{16 \text{ ones}}$	$2^{16} - 1 = 65535$ less than 1% more than 65500
`int`	32	$\underbrace{11\cdots11}_{32 \text{ ones}}$	$2^{32} - 1 = 4294967295$ less than 1% more than 4.29 billion
`long`	64	$\underbrace{11\cdots11}_{64 \text{ ones}}$	$2^{64} - 1 = 18446744073709551615$ less than 1% more than 1.84 quintillian

Not on the examination

Do not memorize the decimal numbers: it is sufficient to remember 65500, just over 4 billion and just under 2 quintillion.

2.1 Adding unsigned integers

If we are adding two signed integers of the same type, then we perform addition in a similar manner as with decimal numbers, recalling that $1 + 1 = 10$, which results in a carry, and $1 + 1 + 1 = 11$, which also results in a carry. For example, here we see the addition $13077 + 12598 = 25675$ performed in binary.

```
  11    1  11 1
  0011001100010101
+ 0011000100110110
  0110010001001011
```

Note what happens if we add 1 to the largest unsigned integer:

```
 1111111111111111
  1111111111111111
+ 0000000000000001
  0000000000000000
```

The result can only be stored to 16 digits, so we must ignore the last **carry**. If there is an extra carry, the answer is wrong. You can try this out:

```
int main() {
    unsigned short n{ 65535 };

    std::cout << n << std::endl;
    n += 1;
    std::cout << n << std::endl;

    return 0;
}
```

This prints the incorrect answer, because the answer is too large to be stored as a `unsigned short`:

```
65535
0
```

2.2 Subtracting an unsigned integer from another

Suppose local variables `m` and `n` are unsigned integers of the same type. How do we calculate `m - n`? There are two problems:

1. If you subtract 1 from 1000000, you must perform many different *borrows* before you get that the answer is 111111. Borrows are actually quite difficult to perform in hardware.

2. If you subtract a larger number from a smaller number, the result will be negative, but unsigned integers cannot store negative numbers. How do we detect this?

To subtract one number from another, we will perform the following algorithm: we will take what is called the two's complement of the number being subtracted, and then instead of adding, we will add the two's complement of the second number to the first.

To find the two's complement of an unsigned integer, switch all the bits and then add 1 to the result. For example, if we were subtracting `unsigned short n{150}`, then we would find the two's complement as follows:

```
  0000000010010110
  1111111101101001
+ ________________1
  1111111101101010
```

Observe what happens when we add the two's complement of 15 to 57:

```
  0000000010010110
  1111111101101001
+ ________________1
  1111111101101010
```

Observe what happens when we add the two's complement of 150 to 616:

```
 1111111  11 1
  0000001001101000
+ 1111111101101010
  0000000111010010
```

The result is the binary representation of $466 = 616 - 150$. Once again, the last carry is ignored, but in this case, the carry indicates that the result is actually valid. If we tried adding the two's complement of 150 onto 42, we get a different picture:

```
          11 1 1
  0000000000101010
+ 1111111101101010
  1111111110010100
```

Now the result is a very large number: specifically, 65428. If you consider this carefully, you will notice that this is $42 - 150 + 2^16$.

To understand what is happening, let subtract 1 from 0. The two's complement of `1` is `1111111111111111`, so `0 - 1` may be calculated as:

```
  0000000000000000
+ 1111111111111111
  1111111111111111
```

You may recognize this as the largest integer that can be stored as a short. On the other hand, if you add `1` to the largest integer, you get the following:

```
  1111111111111111
  1111111111111111
+ 0000000000000001
  0000000000000000
```

Thus, adding `1` to the largest integer results in `0`. What is happening is straightforward. If the result of a calculation is outside the range from 0 to $2^{16}-1$, the calculation is performed modulo 2^{16}. Thus, subtracting `2` from `0` results in the second-largest 16-bit number, or `1111111111111110`, and so on. We describe this by saying that the calculation *wraps around* the given range. The same happens for `unsigned int` and `unsigned long`, only that we are now using 32 bits and 64 bits, respectively.

In conclusion, with unsigned integers, you can store values from 0 to $2^{16}-1$, $2^{32}-1$ and $2^{64}-1$ using `unsigned short`, `unsigned int`, and `unsigned long`, respectively. Calculations will wrap, meaning if you add two numbers that have a result equal to or greater than 2^{16}, 2^{32} or 2^{64}, respectively, then the result will wrap around to `0` and the answer will not be the mathematical answer you would expect from integer arithmetic. The answer will, however, be correct modulo 2^{16}, 2^{32} or 2^{64}, respectively. Similarly, when you are subtracting one unsigned integer from another, you add the two's complement representation of the number being subtracted onto the number being subtracted from. If, however, you subtract a larger number from a smaller number, the calculation will wrap around towards the largest possible integer that can be stored. For example, if `m < n` and they are both `unsigned int`, then the `unsigned int` that will be calculated when performing, for example, `m -= n;` will have the numeric value of calculating $m - n + 2^{32}$.

In this section, we saw the following:

1. If we wanted to add two unsigned integers, we would simply add them, but if a carry occurred when adding the most significant bits, the calculation resulted a mathematically incorrect answer: the correct answer cannot be stored in the given number of bits.

2. If we wanted to subtract one unsigned integer from another, we add the two's complement of the subtrahend (the unsigned integer being subtracted), and if the resulting calculation does not result in a carry in the most significant bit, the result is, again, invalid (the result should be a negative number).

3 Signed integers

We have already discussed how to subtract one unsigned integer from another: the subtrahend is converted to its two's complement representation, and the two are then added. This ability to use addition instead of subtraction is exceptionally efficient, as addition can easily be performed in a computer. Subtraction is more difficult, as you don't know how many "borrows" you must make (subtract 1 from 10000000), while the addition, each carry will only affect the next sum to the left.

If we want to store positive integers, we will proceed as follows:

1. If the leading bit is `0`, the number is `0` or positive and the remaining bits (63 for `long`, 31 for `int` and 15 for `short` represents the binary representation of the number. Thus, we can store number from 0 to $2^{63} - 1$ for `long`, 0 to $2^{31} - 1$ for `int` and 0 to $2^{15} - 1$ for `ishort`.

2. If the leading bit is `1`, the number is negative, and the bits stored are the two's complement representation of the absolute value of that number.

Thus, if we see `1111111110110100`, this is a negative `short` and taking the two's complement, we get `0000000001001100` (flip the bits and add `1`). This is the binary representation of 76, and thus the original number is our representation of -76. If you wanted to store -150 as an `int`, you would take the two's complement of the representation and that is what you would store:

```
00000000000000000000000010010110
11111111111111111111111101101001
                               + 1
11111111111111111111111101101010
```

This is the number that would be stored if we declared `int crse{ -150 };`.

Thus, we will look at some oddities. We will use `short`, but this also works for `int` and `long`. Let us look at the two's complement representation of -1, -2 and then -3:

```
0000000000000001
1111111111111110
              + 1
1111111111111111

0000000000000010
1111111111111101
              + 1
1111111111111110

0000000000000011
1111111111111100
              + 1
1111111111111101
```

On the other hand, the most negative `short` is

```
1000000000000000
0111111111111111
               + 1
1000000000000000
```

Thus, a `1` followed by fifteen zeros is the two's complement representation of a `1` followed by fifteen zeros, so this is the two's complement representation of -2^{15}. This means that using this approach, we can represent integers from -2^{15} up to $2^{15} - 1$. This leads to at least one oddity: the largest negative number does not have an absolute value in this representation. Similarly, `int` stores integers from -2^{31} up to $2^{31} - 1$, and `long` stores integers from -2^{63} up to $2^{63} - 1$.

Type	Bits	Minimum (binary)	Maximum (binary)	Range (decimal)
`short`	16	$1\underbrace{00\cdots00}_{\text{15 zeros}}$	$0\underbrace{11\cdots11}_{\text{15 ones}}$	$-2^{15}, \ldots, 2^{15} - 1$ $= -32768, \ldots, 32767$ approximately ± 32700
`int`	32	$1\underbrace{00\cdots00}_{\text{31 zeros}}$	$0\underbrace{11\cdots11}_{\text{31 ones}}$	$-2^{31}, \ldots, 2^{31} - 1$ $= -2147483648, \ldots, 2147483647$ approximately ± 2.14 billion
`long`	64	$1\underbrace{00\cdots00}_{\text{63 zeros}}$	$0\underbrace{11\cdots11}_{\text{63 ones}}$	$-2^{63}, \ldots, 2^{63} - 1$ $= -2147483648, \ldots, 2147483647$ approximately ± 922 quadrillion

Not on the examination

Do not memorize the decimal numbers: it is sufficient to remember just under ± 33 thousand, just over ± 2 billion and just under ± 1 quintillion.

Observe what happens if we add one to the largest representable integer:

```
 111111111111111
 0111111111111111
+0000000000000001
 1000000000000000
```

We get the largest negative integer. Adding `1` to this value does, of course, produce the second-largest negative integer, because `1000000000000001` is negative, and the two's complement is `0111111111111111`, which is one less than $2^3 1$.

On the other hand, subtracting `1` from the largest negative integer results in the following: the two's complement of `1` is

```
1
 1000000000000000
+1111111111111111
 0111111111111111
```

Thus, subtracting 1 from the largest negative number results in the largest positive number. These two issues with fixed-width integers are described as *overflow* and *underflow*, respectively.

To add two signed integers, just add them. To subtract a signed integer from another signed integer, take the two's complement of the subtrahend (the integer being subtracted) and add the two.

4 Bit-wise operations

Any integer can be thought of as a sequence of 16, 32 or 64 zeros or ones, and a zero can be interpreted as `false` and a one as `true`. Given two integers of the same size (the same number of bytes), we can perform the logical operations of AND and OR on each pair of corresponding bits. This is performed with the bit-wise logical operators `&` and `|` for a bit-wise AND and bit-wise OR, respectively. Here we can see the result of the AND operation on two 32-bit integers:

```
  11110111000111100100000101000010
& 10111110111011001100000111110011
  10110110000011000100000101000010
```

The only time that the result of the calculation results in a bit equalling `1` is when the corresponding bits in the two operands are also `1`. Here we can see the result of the OR operation on the same two 32-bit integers:

```
  11110111000111100100000101000010
| 10111110111011001100000111110011
  11111111111111101100000111110011
```

The only time that the result of the calculation results in a bit equalling `0` is when the corresponding bits in the two operands are also `0`.

There is one another bit-wise binary operation that has no parallel with the logical operators: that of the *exclusive or*, or XOR: the exclusive or of two operands is `true` if and only if exactly one of the two operands is `true`, and thus, for the bit-wise XOR, the exclusive or of two bits is `1` if and only if exactly one of the two corresponding bits of the operands is `1`. Here we can see the result of the XOR operation on the same two 32-bit integers:

```
  11110111000111100100000101000010
^ 10111110111011001100000111110011
  01001001111100101000000010110001
```

The only time that the result of the calculation results in a bit equalling `1` is when the one of the corresponding bits in the two operands is `0` and the other corresponding bit is `1`.

For each of these binary bit-wise operators, there is a corresponding auto-assignment operator:

```
m |= n;      // m = m|n;
m &= n;      // m = m&n;
m ^= n;      // m = m^n;
```

Remember that there is no auto-assignment operator for the two binary logical operators: you cannot perform `m &&= n;` or `m` —= n;—.

Not on the examination

Note that the bit-wise exclusive or operator is the caret symbol `^`. Many programming languages use that character for exponentiation, so `x^5` represents `x*x*x*x*x`. Because exponentiation cannot be performed as a single machine instruction, C and C++ does not support such an operation, and leaves that for a function `unsigned int pow( unsigned int base, unsigned int exp );`

The final operation is the bit-wise equivalent of the unary `!` operator: the unary complement operator `~`. This flips all the bits of the operand. Thus,

```
~ 10111110111011001100000111110011
  01000001000100110011111000001100
```

Note that if the `unsigned int` has exactly one `1`, then performing bit-wise operations may be summarized as follows, where we will assume that `mask` has been defined as follows:

```
//                    11
// This is equal to 2   = 2048
//  - If the least significant bit is 'bit 0',
//    then this number has 'bit 11' set to '1'
unsigned int mask{ 0b00000000000000000000100000000000 };
```

1. If `(n & mask) == 0`, then bit 11 of `n` is zero (`0`).
2. If `(n & mask) == mask` or `(n & mask) != 0`, then bit 11 of `n` is one (`1`).
3. The operation `n |= mask` sets bit 11 of `n` to `1`.
4. The operation `n &= ~mask` sets bit 11 of `n` to `0`.
5. The operation `n ^= mask` flips the value of bit 11, so if it was `0` it is now `1`, and if it was `1` it is now `0`.

5 Bit-shifting operations

Given an unsigned integer `n`, the operation `n << k` moves the bits to the left by `k` places, and the left-most `k` bits are lost. The integer `k` must be non-negative (so `0` or greater). Note that if you left-shift a number greater than or equal to the number of bits of that data type, the result is `0`.

```
10111110111011001100000111110011 << 3
11110111011001100000111110011000

10111110111011001100000111110011 << 7
01110110011000001111100110000000

10111110111011001100000111110011 << 29
01100000000000000000000000000000
```

Similarly, for an unsigned integer `n`, `n >> k` moves the bits to the right by `k` places, and the right-most `k` bits are lost.

```
10111110111011001100000111110011 >> 3
00010111110111011001100000111110

10111110111011001100000111110011 >> 12
00000000000010111110111011001100

10111110111011001100000111110011 >> 30
00000000000000000000000000000010
```

For each of these binary bit-shifting operators, there is a corresponding auto-assignment operator:

```
m <<= n;     // m = m << n;
m >>= n;     // m = m >> n;
```

Not on the examination

For bit-shifting, we have always emphasized that we are using unsigned integers. This is because there is a slight variation for signed integers: for the right shift of a signed integer, the sign bit is repeated. For example, -150 is stored as the two's complement of `0x10010110`. Assuming it is stored in a signed `int`, we have:

```
11111111111111111111111101101010 >> 3
11111111111111111111111111101101

11111111111111111111111101101010 << 24
01101010000000000000000000000000
```

Note that right-shifted three times leaves the number negative, however, left-shifting may result in a positive number.
There are a few common uses for using bit shifting: given an unsigned integer that contains just a single bit set to `1`, bit shifting can move that `1` around within that integer:

```
00000000000000000001000000000000 >> 3
00000000000000000000001000000000

00000000000000000001000000000000 << 7
00000000000100000000000000000000
```

Specifically, `1 << k` sets bit `k` to 1, and this happens to be the integer 2^k:

```
00000000000000000000000000000001 << 4
00000000000000000000000000010000

00000000000000000000000000000001 << 10
00000000000000000000010000000000
```

The first number is the binary representation of $2^4 = 16$ and the second that of $2^{10} = 1024$. Note also that `(1 << 4) - 1` and `(1 << 10) - 1` result in the calculations:

```
  11111111111111111111111111111111
    00000000000000000000000000010000
+ 11111111111111111111111111111111
    00000000000000000000000000001111

  11111111111111111111
    00000000000000000000010000000000
+ 11111111111111111111111111111111
    00000000000000000000001111111111
```

Thus, `(1 << k) - 1` is a convenient way of creating a block of `k` bits set to 1.

Not on the examination

A common interview question is "How do you know if an unsigned integer has exactly one 1?" the quick answer is `(n != 0) && ((n & (n - 1)) == 0)`. If `n` has only one 1, then `n - 1` will have a 0 in that location and because all other bits of `n` are already zero, `n & (n - 1)` must be zero. If `n` has at least two ones, then subtracting 1 will not affect the most significant bit, so `n & (n - 1)` will not be zero.

You can use bit-shifting operations to walk through an integer data type. Remember that an array is indexed form 0 to `capacity - 1`. You can think of an `unsigned int` as being an array of 32 bits, only the index goes from the right-most bit (the least-significant bit) being 0 to the left-most bit (the most-significant bit).

For example, a for loop that goes through an array of capacity 32 looks like the following:

```
for ( std::size_t k{ 0 }; k < 32; ++k ) {
```

```
    // Do something with array[k]...
}
```

You can go through all the bits in an `unsigned int n{...};` as follows

```
for ( unsigned int mask{ 1 }; mask != 0; mask <<= 1 ) {
    // Do something with n&mask
}
```

This will go through all the bits of `n` one by one and you can determine if the corresponding bit of `n` is `1` if `(n&mask) != 0`. If you want to know what the actual bit number is, you can also use:

```
for ( unsigned int mask{ 1 }, bit{ 0 }; mask != 0; mask <<= 1, ++bit ) {
    // Do something with n&mask which is
    // accessing bit 'bit' (where 'bit' goes from '0' to '31'
}
```