Integer overflow

Overflow is a phenomenon where operations on 2 or more numbers exceeds the maximum (or goes below the minimum "Underflow") value the data type can have. Usually, it is thought that integral types are very large and people don't take into account the fact that sum of two numbers or their multiplication can be larger than the range. But in things like scientific and mathematical computation, this can happen. For example, an unhandled arithmetic overflow in the engine steering software was the primary cause of the crash of the maiden flight of the Ariane 5 rocket. The software had been considered bug-free since it had been used in many previous flights; but those used smaller rockets which generated smaller accelerations than Ariane 5's.

This article will tell how this problem can be tackled.

In this article, we will only deal with integral types (and not with types like float and double)

In order to understand how to tackle this problem we will first know how numbers are stored.

About integers:

If the size of a data type is n bytes, it can store 2^{8n} different values. This is called the data type's range.

If size of an unsigned data type is n bytes, it ranges from 0 to 2^{8n-1}

If size of a signed data type is n bytes, it ranges from -2^{8n-1} to $2^{8n-1} - 1$

So, a short (usually 2 bytes) ranges from -32768 to 32767 and an unsigned short ranges from 0 to 65535

Consider a short variable having a value of 250.

It is stored in the computer like this (in binary format): 00000000 11111010

Complement of a number is a number with its bits toggled.

It is denoted by tilde ~. For e.g., ~250 is 11111111 00000101

Negative numbers are stored using 2's complement system.

According to this system, $-n = \sim n + 1$

-250 is stored as 11111111 00000110

Two's Complement

Two's complement is a clever way of storing integers so that common math problems are very simple to implement.

To understand, you have to think of the numbers in binary.

It basically says,

1. For zero, use all 0's.
2. For positive integers, start counting up, with a maximum of $2^{8n-1} - 1$
3. For negative integers, do exactly the same thing, but switch the role of 0's and 1's and count down (so instead of starting with 0000, start with 1111 - that's the "complement" part).

Let's try it with a mini-byte of 4 bits (we'll call it a nibble - 1/2 a byte).

0000 - zero

0001 - one

0010 - two

0011 - three

0100 to 0111 - four to seven

That's as far as we can go in positives. $2^3 - 1 = 7$.

For negatives:

1111 - negative one

1110 - negative two

1101 - negative three

1100 to 1000 - negative four to negative eight

Note that you get one extra value for negatives ($1000 = -8$) that you don't for positives. This is because 0000 is used for zero. This can be considered as Number Line of computers.

Distinguishing between positive and negative numbers

Doing this, the first bit gets the role of the "sign" bit, as it can be used to distinguish between non-negative and negative decimal values. If the MSB is 1, then the binary can be said to be negative, where as if the MSB (the leftmost bit) is 0, you can say the decimal value is non-negative.

"Sign-magnitude" negative numbers just have the sign bit flipped of their positive counterparts, but this approach has to deal with interpreting 1000 (one 1 followed by all 0s) as "negative zero" which is confusing.

"Ones' complement" negative numbers are just the bit-complement of their positive counterparts, which also leads to a confusing "negative zero" with 1111 (all ones).

You will likely not have to deal with Ones' Complement or Sign-Magnitude integer representations unless you are working very close to the hardware.

The basic problem that you are trying to solve with two's complement representation is the problem of storing negative integers.

First, consider an unsigned integer stored in 4 bits. You can have the following

0000 = 0

0001 = 1

0010 = 2

...

1111 = 15

These are unsigned because there is no indication of whether they are negative or positive.

Sign Magnitude and Excess Notation

To store negative numbers, you can try a number of things. First, you can use sign magnitude notation which assigns the first bit as a sign bit to represent +/- and the remaining bits to represent the magnitude. So, using 4 bits again and assuming that 1 means - and 0 means + then you have

0000 = +0

0001 = +1

0010 = +2

...

1000 = −0

1001 = −1

1111 = −7

So, you see the problem there? We have positive and negative 0. The bigger problem is adding and subtracting binary numbers. The circuits to add and subtract using sign magnitude will be very complex.

What is

0010

1001 +

?

Another system is excess notation. You can store negative numbers; you get rid of the two zeros problem but addition and subtraction remain difficult.

So along comes two's complement. Now you can store positive and negative integers and perform arithmetic with relative ease. There are a number of methods to convert a number into two's complement. Here's one.

Convert Decimal to Two's Complement

1. Convert the number to binary (ignore the sign for now) e.g., 5 is 0101 and −5 is also 0101
2. If the number is positive then you are done. e.g., 5 is 0101 in binary using two's complement notation.
3. If the number is negative then

3.1 find the complement (invert 0's and 1's) e.g., −5 is 0101 so finding the complement is 1010

3.2 Add 1 to the complement 1010 + 1 = 1011. Therefore, −5 in two's complement is 1011.

So, what if you wanted to do 2 + (−3) in binary? 2 + (−3) is −1. What would you have to do if you were using sign magnitude to add these numbers? 0010 + 1101 = ?

Using two's complement consider how easy it would be.

2 = 0010

−3 = 1101

−1 = 1111

Converting Two's Complement to Decimal

Converting 1111 to decimal:

The number starts with 1, so it's negative, so we find the complement of 1111, which is 0000.

1. Add 1 to 0000, and we obtain 0001.
2. Convert 0001 to decimal, which is 1.
3. Apply the sign = −1

Recall how it works for decimals:

$$2345$$

is a way of writing

$$2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$$

In the same way, binary is a way of writing numbers using just 0 and 1 following the same general idea, but replacing those 10s above with 2s. Then in binary,

$$1111$$

is a way of writing

$$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

and if you work it out, that turns out to equal 15 (base 10). That's because it is

$$8 + 4 + 2 + 1 = 15$$

This is all well and good for positive numbers. It even works for negative numbers if you're willing to just stick a minus sign in front of them, as humans do with decimal numbers. That can even be done in computers, sort of, but I haven't seen such a computer since the early 1970's.

For computers it turns out to be more efficient to use a complement representation for negative numbers. And here's something that is often overlooked. Complement notations involve some kind of reversal of the digits of the number, even the implied zeroes that come before a normal positive number. That's awkward, because the question arises: all of them? That could be an infinite number of digits to be considered.

Fortunately, computers don't represent infinities. Numbers are constrained to a particular length (or width, if you prefer). So, let's return to positive binary numbers, but with a particular size. I'll use 8 digits ("bits") for these examples. So, our binary number would really be

$$00001111$$

or

$$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

To form the 2's complement negative, we first complement all the (binary) digits to form

11110000

and add 1 to form

11110001

but how are we to understand that to mean -15?

The answer is that we change the meaning of the high-order bit (the leftmost one). This bit will be a 1 for all negative numbers. The change will be to change the sign of its contribution to the value of the number it appears in. So now our 11110001 is understood to represent

$$-1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Notice that "−" in front of that expression? It means that the sign bit carries the weight -2^7, that is -128 (base 10). All the other positions retain the same weight they had in unsigned binary numbers.

Working out our -15, it is

$$-128 + 64 + 32 + 16 + 1 = -15$$

10000000 00000000 (−32768) has no positive counterpart.

Its negative is the number itself.

11100010 01110101 will be read as 57973 if data type is unsigned while it will be read as −7563 if data type is signed. If you add 65536 (which is the range) to −7563, you get 57973.

Overflow

Consider a data type var_t of 1 byte (range is 256)

signed var_t a, b;

unsigned var_t c, d;

If c is 200 (11001000) and d is 100 (01100100), (c + d) is 300 (00000001 00101100), which is more than the maximum value 255 (11111111). 00000001 00101100 is more than a byte, so the higher byte will be rejected and (c + d) will be read as 44.

So, 200+100=44! This is absurd! (Note that 44 = 300−256). This is an example of an unsigned overflow, where the value couldn't be stored in the available no. of bytes.

In such overflows, the result is moduloed by range (here, 256).

If a is 100 (01100100) and b is 50 (00110010), (a + b) is 150 (10010110), which is more than the maximum value 127. Instead, (a + b) will be read as −106 (note that −106 = 150−256).

This is an example of a signed overflow, where result is moduloed by range (256).

Detecting overflow

Division and modulo can never generate an overflow.

Addition overflow

Overflow can only occur when sign of numbers being added is the same (which will always be the case in unsigned numbers)
signed overflow can be easily detected by seeing that its sign is opposite to that of the operands.

Let us analyze overflow in unsigned integer addition.

Consider 2 variables a and b of a data type with size n and range R.
Let + be actual mathematical addition and a $ b be the addition that the computer does.

If a + b <= R−1, (a $ b) = (a + b)
As a and b are unsigned, (a $ b) is more than or equal to both a and b.

If a + b >= R, (a $ b) = a + b − R
as R is more than both a and b, (a−R) and (b−R) are negative
So, (a + b – R) < a and (a + b – R) < b
Therefore, (a $ b) is less than both a and b.

This difference can be used to detect unsigned addition overflow.

a−b can be treated as a + (−b) hence subtraction can be taken care of in the same way.

Multiplication overflow

There are two ways to detect an overflow:

1. If a * b > max, then a > max/b (max is R−1 if unsigned and R/2 −1 if signed).
2. Let there be a data type of size n and range R called var_t and a data type of size 2n called var2_t.

Let there be 2 variables of var_t called a and b. Range of var2_t will be R*R, which will always be more than the product of a and b. hence if var2_t(a)*var2_t(b) > R overflow has happened.

Truncation: (The value is moduloed by Range)

This happens when a shorter is assigned from a longer variable.

For e.g., `short a; long b = 70000; a = b;` Only the lower bits are copied and the value's meaning is translated.

```
short int a;
long int b = 70000;
a = b; // It's stored as b % (1<<16)
cout << b % (1 << 16);
```

`short a; int b = 57973; a = b;` will also show this behavior become −7563.
Similar behavior will be shown if int is replaced by unsigned short.

Type conversion:

consider `unsigned int a = 4294967290; int b = −6; cout << (a==b);` This returns 1.
Whenever an operation is performed between an unsigned and a signed variable of the same type, operands are **converted to unsigned.**
Whenever an operation is performed between a long type and a short type, operands are **converted to the long type.**
The above code returned 1 as a and b were converted to unsigned int and then compared.
If we used __int64 (a 64-bit type) instead of unsigned int and 18446744073709551610 instead of 4294967290, the result would have been the same.

Type promotion:

Whenever an operation is performed on two variables of a type shorter than int, the type of both variables is **converted to int.**

For e.g., `short a = 32000, b = 32000; cout << a + b << endl;` would display 64000, which is more than the max value of short. The reason is that a and b were converted to int and a + b would return an int, which can have a value of 64000.

In C++

Signed integer overflow is undefined behavior, while unsigned integer overflow is well-defined; the value wraps around.

In other words, the value is modulo divided by 2^n , where n is the number of bits in the data type.

For a 32-bit unsigned int

4294967295 + 1 = 4294967296 % 2^{32} = 0

it results in 0.

For a signed int

2147483647 + 1 = ?, from the C++ language standpoint, it's undefined.

However, most implementations employ 2's complement to implement signed integer types. A toy, signed 4-bit data type, implemented using 2's complement may be used to explain what happened in this case. In this type

POS_MAX = 7 = 0111

NEG_MAX = −8 = 1000

The type can hold 2^4 = 16 states, 8 positives (0 to 7) and 8 negatives (−1 to −8).

POS_MAX + 1 = 0111 + 1 = 1000

Since the first bit is set, it's a negative number, to find the actual value, do the inverse the two's complement (subtract 1 and flip bits)

1000 − 1 = 0111

~0111 = 1000 = 8

Thus, the final value is −8. All this is not defined by the language but this is what happened in this case, specifically.

That's why overflow can sometimes cause TLE

Consider the following loop:

```
for (int i{1}; i * i <= N; i++)
{
    // Some logic
}
```

The value (i * i) will overflow the signed int datatype which is undefined behavior in C++ and yields a **negative value** which is always less than N making the loop condition always true and the loop is then infinite which cause TLE.