Integer OverFlow

- Phenomenon where operations on two numbers exceeds the maximum, or falls below the minimum, value the data type can hold
- Unhandled arithmetic overflow in the engine steering software was primary cause of the crash of the maiden flight of the Ariane 5 rocket
- This tutorial only deals with integral types, but this can happen with any data type
- Occurs when an arithmetic operation attempts to create a numeric value that is outside the range that can be represented with a given number of bits
- This can be either higher than the maximum value or lower than the minimum representable value

About Integers

- If size of a data type is n bytes, it can store 2^{8n} different values, this is called the data type's range
- If the size of an unsigned data type is n bytes, it ranges from 0 to $2^{8n-1} - 1$
- Thus, short (usually two bytes) ranges from -32768 to 32767 and an unsigned short ranges from 0 to 65535
- Chars hold one byte of data and ranges from 0 to 255 with its unsigned counterpart ranging from -128 to 127

Example

Consider a char variable having a value of 250, it will be stored in the computer in the following binary format: `0b11111010`, where `0b` signifies a binary sequence. The complement of a number is a number is a number with its bit toggled and is denoted by ~. For example, ~250 is `0b00000101`. Negative numbers are stored in a computer using the 2's complement system. According to this, $-n = \sim n + 1$, thus -250 will be stored as `0b00000110`.

It is worth noting that for a given data type, the maximum negative number has no positive counterpart. To demonstrate this, take the following char with the bit sequence `0b10000000`(-128). Computing its 2's complement will yield the exact same number (i.e. using $-n = \sim n + 1$).

Let's look at another binary sequence `0b11110101`. This byte will be read as 245 if the data type is an unsigned char while it will read as -11 if the data type is signed. If you subtract 256 (which is the range of the data type) from 245, you will get -11 (i.e. $245 - 256 = -11$).

OverFlow

- Division (except for the `INT_MIN` and -1 special case) and modulo can never generate overflow

Addition overflow

- Addition overflow can only happen when the sign of numbers being added is the same

Again, consider a char of one byte (range is 256):

```
unsigned char x, y;
signed char s, t;
```

Unsigned integer wrap If `x` is 200 (`0b11001000`) and `y` is 100 (`0b01100100`) then `x + y` computes to 300 (`0b0000000100101100`). This is, however, higher than the max value of a char: 255 (`0b11111111`), thus, the higher byte will be rejected and `x + y` will be read as 44 (`0b00101100`). This is an example of unsigned overflow where the value couldn't be stored in the available number of bits. In such overflows, the result is equivalent to performing the modulus operator (%) over the range, i.e. 44 = 300%256. This is sometimes called "modulo wrapping". 256 modulus operation is stripping every bit before the last 8 bits.

Signed integer overflow If `s` is 100 (`0b01100100`) and `t` is 50 (`0b00110010`), then `s + t` computes to 150 (`0b10010110`), which is more than the maximum value of a signed char (127). Instead, `s + t` will be represented as -106 ($150 - 256 = -106$). This is an example of signed overflow, where the result is equivalent to wrapping around the range of the datatype. If `s` and `t` were -100 and -50 respectively, adding their result would be equal to $-150 + 256 = 106$.

How to detect integer overflow

- The only safe way is to check for overflow before it occurs

Unsafe methods to check for overflow

Unsigned addition

- If both `x` are `y` are unsigned ints, when added, their values can't be less than either individual operand

```
unsigned int x, y;
unsigned int result = x + y;
bool overflow = result < x || result < y;
```

Signed addition

- Easily detected by seeing that its sign is opposite to that of the operands

```
signed int x, y;
signed int result = x + y;
bool overflow = (x > 0 && y > 0 && result < 0) || (x < 0 && y < 0 && result > 0);
```

Multiplication

- Can't have division by zero

```
signed int x, y;
signed int result = x * y;
bool overflow = (x != 0 && y == result / x);
```

Safe method to check for overflow

- Once there has been overflow, undefined behaviour (behaviour that is prescibed to be unpredictable) has occurred your program can do anything, thus, rendering the tests inconclusive
- To create a conforming program, it is required to test for overflow before generating overflow

Addition

```
signed int x, y;  // result = x + y
bool overflow = (x > 0 && y > INT_MAX - x) || (x < 0 && y < INT_MIN - x);
```

Subtraction

```
signed int x, y;  // result = x - y
bool overflow = (y > 0 && x < INT_MIN + y) || (y < 0 && x > INT_MAX + y);
```

Multiplication

- Requires checking for `INT_MIN` and -1 edge cases separately

```
signed int x, y;  // result = x * y
bool overflow = false;
if (x == -1 && y == INT_MIN) || (y == -1 && x == INT_MIN) {
    overflow = true;
}
else if (x > INT_MAX / y || x < INT_MIN / y) {
    overflow = true;
}
```

Definition variations and ambiguity

- For an unsigned type, when the ideal results of an operation is outside the type's representable range and the result is obtained by wrapping, then this event is commonly defined as overflow
- In contrast, the C++ standard defines this event is not an overflow and states:

    A comutation involving unsigned operands can never overflow, because a result that cannot be represented by the

> resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type

- Howeverm both standards state that signed integer overflow is undefined behaviour. Again from the C99 standard:

> An example of undefined behaviour is the behaviour on integer overflow

- Unsigned integer: modulo power of two
- Signed integer: undefined behaviour
- Compiler may have optimized the code for such undefined behaviour cases with surprising effects