October 13, 2018

Arithmetic Overflow and Underflow

Arithmetic overflow happens when an arithmetic operation results in a value that is outside the range of values representable by the expression’s type. For example, the following C++ code prints 0:

```
uint16_t x = 65535;
x++;

std::cout << x;
```

x is an unsigned 16 bit integer, which can represent values between 0 and 65535. If x is 65535 and we increment it, the value becomes 65536 but that value cannot be represented by a uint16_t. This is an overflow. In this case, C++ wraps the value around and x becomes 0.

Similarly, an underflow occurs when an arithmetic operation generates a result that is below the smallest representable value of the expression’s type:

```
uint16_t x = 0;
x--;

std::cout << x;
```

The above prints 65535, as the result of decrementing 0 is -1, which cannot be represented by an uint16_t.

Before we digging into overflow behavior, we need to understand how computers represent numbers.

Number Representations

Arbitrarily large integers

Python provides support for arbitrarily large integers: unlike C++, where the bit width (number of bits used to represent a number) is fixed, we can have integers of any size:

```
print(10**100)
```

Prints

1000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000

Why don't all languages provide such support? The answer is performance. The underlying hardware the code runs on uses fixed-width integers, so performing arithmetic on fixed-width integer types becomes a single CPU instruction. On the other hand, supporting arbitrarily large integers usually involves writing code to determine how many bits a given value or the result of an arithmetic operation needs and convert that into an array of fixed-width integers large enough to hold that value. The added overhead of this is non-trivial, so unlike Python, most other mainstream languages offer only fixed-width integers and support arbitrarily large integers only explicitly, via libraries.

Unsigned integers

Unsigned integers are represented as a sequence of N bits, thus being able to represent numbers between 0 and 2^N-1. An unsigned 8-bit integer can store any value between 0 and 255, an unsigned 16-bit integer can store any value between 0 and 65535, an unsigned 32-bit integer between 0 and 4294967295, and an unsigned 64-bit integer between 0 and 18446744073709551615.

Unsigned integer representation is trivial.

Signed integers

Signed integers are usually represented in two's complement.

Positive numbers are encoded the same as unsigned binary numbers described above. Negative numbers are encoded as two's complement of their absolute value. For example, an 8-bit representation of -3 is 2^8-3.

The most significant bit is always 1 for negative numbers and 0 for positive numbers or 0.

With this representation, N bits can encode a signed integer between 2^{N-1}-1 and -2^N. So 8 bits can encode an integer between -128 and 127.

Handling Overflow

If the result of an arithmetic operation cannot fit the type, there are several approaches we can take and different programming languages employ different strategies. These are:

- Exceptions
- Wrap-around
- Saturation

All of these approaches have their pros and cons.

Exceptions

The safest approach is to treat an arithmetic overflow as an exception. This usually gets rid of security vulnerabilities and treats any overflow as an exceptional scenario. In this case an exception is thrown (or an error returned) whenever an arithmetic operation overflows or underflows.

This is usually desirable from a security/safety perspective, but the trade-off is in performance: the downside of this approach is that all arithmetic operations need to be checked for overflow (underlying hardware usually does not do this natively) and exceptions need to be handled by callers.

Wrap-around

The default behavior in C++, wrap-around simply continues from the smallest possible value in case of overflow or from the largest possible value in case of underflow. For unsigned integers, this is equivalent to modulo arithmetic. For example, for an int8_t, which can represent values between -128 and 127, wrap-around would make 127 + 1 be -128 and similarly -128 - 1 be 127.

This is usually the most efficient way to perform arithmetic as no checking is involved. Most hardware uses wrap-around as it can simply discard overflowing bits to achieve the result. The two's complement representation of 127 is 01111111. The two's complement representation of 128 is 10000000. With this representation, adding 1 to 127 naturally makes it 128.

This is also the most unsafe implementation as it can lead to unexpected behavior and exploitable security holes[1].

Saturation

Saturation means clamping the value within the allowed range, so on overflow, we would simply stop at the largest representable value. On underflow, we would stop at the smallest representable value. In our 8-bit signed integer example, we would now have 127 + 1 be 127 and -128 - 1 be -128. There are several advantages with this approach: for one, the resulting values on overflow and underflow are the closest to the "real" values we would get if operating without constraints. A lot of physical systems naturally lend themselves to saturation. Imagine, for example, a thermostat which can only operate within a range of temperature.

The downsides of this approach are results which might be surprising and the fact that properties of arithmetic operations like associativity no longer hold: (120 + 10) + (-10) is 117, but 120 + (10 + (-10)) is 120.

Detecting Overflow and Underflow

Let's now see how we can tell whether an arithmetic operation overflow while operating only within the range of values representable by a given type.

For a type which can represent any value between some MIN and MAX, we observe that an addition overflow means a + b > MAX, while an underflow means a + b < MIN (note a and b can

be negative, so adding them could produce a value that would be under our minimum representable value).

We can detect overflow and underflow by checking, if b >= 0, that a > MAX - b, otherwise with b < 0, that a < MIN - b.

The reason this works is that, if b is greater than or equal to 0, we can safely subtract it from MAX (if it were negative, subtracting it would cause an overflow). So with this in mind, we are simply saying that a + b > MAX is equivalent to a > MAX - b (subtracting b on both sides). We also observe that a + b can never underflow if b is greater than or equal to 0 because, regardless how small a is, adding a positive number to it will make it larger not smaller.

If b is less than 0, then by the same logic we cannot possibly overflow - regardless how large a is, adding b to it would make it smaller. In this case we only need to check for underflow. Here we observe that subtracting a negative number from MIN is safe - it will increase MIN. So by subtracting b on both sides of a + b < MIN, we get a < MIN - b.

The following code implements these two checks:

```
#include <limits>

template <typename T>
constexpr bool AdditionOverflows(const T& a, const T& b) {
    return (b >= 0) && (a > std::numeric_limits<T>::max() - b);
}

template <typename T>
constexpr bool AdditionUnderflows(const T& a, const T& b) {
    return (b < 0) && (a < std::numeric_limits<T>::min() - b);
}
```

Detecting overflow or underflow for subtraction is very similar, as subtracting b from a is the equivalent of adding -b to a, thus we only need to adjust the checks. a - b > MAX means a > MAX + b if b is negative (so we don't cause an overflow during the check), while a - b < MIN means a < MIN + b if b is greater than or equal to 0:

```
template <typename T>
constexpr bool SubtractionOverflows(const T& a, const T& b) {
    return (b < 0) && (a > std::numeric_limits<T>::max() + b);
}

template <typename T>
constexpr bool SubtractionUnderflows(const T& a, const T& b) {
    return (b >= 0) && (a < std::numeric_limits<T>::min() + b);
}
```

Detecting overflow for multiplication is more interesting. a * b > MAX can happen if b >= 0, a >= 0, and a > MAX / b or when b < 0, a < 0, and a < MAX / b (dividing a * b > MAX on both sides by b, a negative number, flips the sign of the inequality).

Underflow can happen only when one of the numbers is negative and the other one isn't. So if b >= 0, a < 0, and a < MIN / b or if b < 0, a >= 0, and a > MIN / b.

We can implement the checks as follows:

```
template <typename T>
constexpr bool MultiplicationOverflows(const T& a, const T& b) {
    return ((b >= 0) && (a >= 0) && (a > std::numeric_limits<T>::max() / b))
        || ((b < 0) && (a < 0) && (a < std::numeric_limits<T>::max() / b));
}

template <typename T>
constexpr bool MultiplicationUnderflows(const T& a, const T& b) {
    return ((b >= 0) && (a < 0) && (a < std::numeric_limits<T>::min() / b))
        || ((b < 0) && (a >= 0) && (a > std::numeric_limits<T>::min() / b));
}
```

Note integer division cannot possibly underflow. The single overflow that can happen is due to the fact that in two's complement representation, we can represent one more negative number than positives, as 0 is, in a sense, positive with this representation (the sign bit is not set for 0). An 8-bit signed integer can represent 128 positive values (0 to 127) and 128 negative values (-1 to -128). Overflow can only happen when we change the sign of the smallest possible value we can represent by dividing it with -1. -128 / -1 becomes 128, which is an overflow. This is the only case we need to check for:

```
template <typename T>
constexpr bool DivisionOverflows(const T& a, const T& b) {
    return (a == std::numeric_limits<T>::min()) && (b == -1)
        && (a != 0);
}
```

Note that unsigned integers can never overflow, so once we confirm that a is the smallest possible value and b is -1, we also check to ensure a is not 0.

We are explicitly not looking at division by 0, which is part of the same safe arithmetic topic. This post focuses on overflow and underflow only.

Handling Overflow and Underflow

Now that we can detect overflows and underflows, we can implement a couple of policies to handle them. Wrap-around is the default behavior in C++, so let's look at the other two possibilities. We will implement a couple of types templated on an integer type T, with overflow and underflow handlers:

```
template <typename T>
struct Policy {
    static constexpr T OnOverflow() { /* ... */ }
    static constexpr T OnUnderflow() { /* ... */ }
};
```

The throwing policy looks like this:

```
struct ArithmeticException : std::exception {};
struct ArithmeticOverflowException : ArithmeticException {};
struct ArithmeticUnderflowException : ArithmeticException {};

template <typename T>
struct ThrowingPolicy {
    static constexpr T OnOverflow() {
        throw new ArithmeticOverflowException{};
    }

    static constexpr T OnUnderflow() {
        throw new ArithmeticUnderflowException{};
    }
};
```

The saturation policy is:

```
template <typename T>
struct SaturationPolicy {
    static constexpr T OnOverflow() {
        return std::numeric_limits<T>::max();
    }

    static constexpr T OnUnderflow() {
        return std::numeric_limits<T>::min();
    }
};
```

Safe Arithmetic

Now that we have all the required pieces, we can create a type that wraps an integer type and implements all the arithmetic operations checking for overflow or underflow. The type is templated on a policy for handling overflows and underflows:

```
template <typename T, template<typename> typename Policy>
struct Integer
{
    T value;

    constexpr Integer<T, Policy> operator+(
        const Integer<T, Policy>& other) const {
        if (AdditionOverflows(value, other.value))
            return { Policy<T>::OnOverflow() };

        if (AdditionUnderflows(value, other.value))
            return { Policy<T>::OnUnderflow() };

        return { value + other.value };
    }

    constexpr Integer<T, Policy> operator-(
        const Integer<T, Policy>& other) const {
```

```
        if (SubtractionOverflows(value, other.value))
            return { Policy<T>::OnOverflow() };

        if (SubtractionUnderflows(value, other.value))
            return { Policy<T>::OnUnderflow() };

        return { value - other.value };
    }

    constexpr Integer<T, Policy> operator*(
        const Integer<T, Policy>& other) const {
        if (MultiplicationOverflows(value, other.value))
            return { Policy<T>::OnOverflow() };

        if (MultiplicationUnderflows(value, other.value))
            return { Policy<T>::OnUnderflow() };

        return { value * other.value };
    }

    constexpr Integer<T, Policy> operator/(
        const Integer<T, Policy>& other) const {
        if (DivisionOverflows(value, other.value))
            return { Policy<T>::OnOverflow(); }

        return { value / other.value };
    }
};
```

Now we can wrap an integer type with this and perform safe arithmetic:

```
Integer<int8_t, ThrowingPolicy> a{ 64 };
Integer<int8_t, ThrowingPolicy> b{ 2 };

// Throws
Integer<int8_t, ThrowingPolicy> result = a * b;
```

This is a simple implementation for illustrative purposes. The Integer type currently only defines addition, subtraction, multiplication, and division. A complete implementation would handle multiple other operators, like pre and post increment, implicit casting from T etc.

The generic overflow and underflow checks can be specialized for unsigned types so that we don't redundantly check for b < 0 for a type which cannot represent negative numbers. Similarly, we wouldn't worry, for example, about addition underflowing for an unsigned type.

We can also extend our safe arithmetic to not only rely on the standard numeric_limits, but also allow users to clamp values between user-defined minimum and maximum values.

For a production-ready safe arithmetic library, I recommend you check out David LeBlanc's SafeInt.

Summary

This post covered arithmetic overflow and underflow, and ways to handle it. We looked at:

- What arithmetic overflow and underflow are
- Integer representations:
  - Unsigned
  - Two's complement
- Ways to deal with overflow/underflow:
  - Exceptions
  - Wrap-around
  - Saturation
- How to detect overflow/underflow
- Implementing a simple Integer wrapper that performs safe arithmetic

---

[1] An example of how an attacker can exploit integer overflow is the following SSH1 vulnerability.

- « Notes on Encoding Text
- Clean Code: Types »