4.5 — Unsigned integers, and why to avoid them

ALEX SEPTEMBER 1, 2021

Unsigned integers

In the previous lesson (4.4 -- Signed integers), we covered signed integers, which are a set of types that can hold positive and negative whole numbers, including 0.

C++ also supports unsigned integers. Unsigned integers are integers that can only hold non-negative whole numbers.

Defining unsigned integers

To define an unsigned integer, we use the *unsigned* keyword. By convention, this is placed before the type:

```
unsigned short us;
unsigned int ui;
unsigned long ul;
unsigned long long ull;
```

Unsigned integer range

A 1-byte unsigned integer has a range of 0 to 255. Compare this to the 1-byte signed integer range of -128 to 127. Both can store 256 different values, but signed integers use half of their range for negative numbers, whereas unsigned integers can store positive numbers that are twice as large.

Here's a table showing the range for unsigned integers:

Size/Type	Range
1 byte unsigned	0 to 255
2 byte unsigned	0 to 65,535
4 byte unsigned	0 to 4,294,967,295
8 byte unsigned	0 to 18,446,744,073,709,551,615

An n-bit unsigned variable has a range of 0 to (2^n)-1.

When no negative numbers are required, unsigned integers are well-suited for networking and systems with little memory, because unsigned integers can store more positive numbers without taking up extra memory.

Remembering the terms signed and unsigned

New programmers sometimes get signed and unsigned mixed up. The following is a simple way to remember the difference: in order to differentiate negative numbers from positive ones, we use a negative sign. If a sign is not provided, we assume a number is positive. Consequently, an integer with a sign (a signed integer) can tell the difference between positive and negative. An integer without a sign (an unsigned integer) assumes all values are positive.

Unsigned integer overflow

What happens if we try to store the number 280 (which requires 9 bits to represent) in a 1-byte (8-bit) unsigned integer? The answer is overflow.

Author’s note

Oddly, the C++ standard explicitly says “a computation involving unsigned operands can never overflow”. This is contrary to general programming consensus that integer overflow encompasses both signed and unsigned use cases (cite). Given that most programmers would consider this overflow, we’ll call this overflow despite C++’s statements to the contrary.

If an unsigned value is out of range, it is divided by one greater than the largest number of the type, and only the remainder kept.

The number 280 is too big to fit in our 1-byte range of 0 to 255. 1 greater than the largest number of the type is 256. Therefore, we divide 280 by 256, getting 1 remainder 24. The remainder of 24 is what is stored.

Here’s another way to think about the same thing. Any number bigger than the largest number representable by the type simply “wraps around” (sometimes called “modulo wrapping”, or more obscurely, “saturation”). 255 is in range of a 1-byte integer, so 255 is fine. 256, however, is outside the range, so it wraps around to the value 0. 257 wraps around to the value 1. 280 wraps around to the value 24.

Let’s take a look at this using 2-byte integers:

```cpp
#include <iostream>

int main()
{
    unsigned short x{ 65535 }; // largest 16-bit unsigned value possible
    std::cout << "x was: " << x << '\n';

    x = 65536; // 65536 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    x = 65537; // 65537 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    return 0;
}
```

What do you think the result of this program will be?

```
x was: 65535
x is now: 0
x is now: 1
```

It’s possible to wrap around the other direction as well. 0 is representable in a 2-byte unsigned integer, so that’s fine. -1 is not representable, so it wraps around to the top of the range, producing the value 65535. -2 wraps around to 65534. And so forth.

```cpp
#include <iostream>

int main()
{
    unsigned short x{ 0 }; // smallest 2-byte unsigned value
possible
    std::cout << "x was: " << x << '\n';

    x = -1; // -1 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    x = -2; // -2 is out of our range, so we get wrap-around
    std::cout << "x is now: " << x << '\n';

    return 0;
}
```

```
x was: 0
x is now: 65535
x is now: 65534
```

The above code triggers a warning in some compilers, because the compiler detects that the integer literal is out-of-range for the given type. If you want to compile the code anyway, temporarily disable “Treat warnings as errors”.

As an aside…

Many notable bugs in video game history happened due to wrap around behavior with unsigned integers. In the arcade game Donkey Kong, it’s not possible to go past level 22 due to an overflow bug that leaves the user with not enough bonus time to complete the level.

In the PC game Civilization, Gandhi was known for often being the first one to use nuclear weapons, which seems contrary to his expected passive nature. Players believed this was a result of Gandhi’s aggression setting was initially set at 1, but if he chose a democratic government, he’d get a -2 modifier. This would cause his aggression to overflow to 255, making him maximally aggressive! However, more recently Sid Meier (the game’s author) clarified that this wasn’t actually the case.

The controversy over unsigned numbers

Many developers (and some large development houses, such as Google) believe that developers should generally avoid unsigned integers.

This is largely because of two behaviors that can cause problems.

First, consider the subtraction of two unsigned numbers, such as 3 and 5. 3 minus 5 is -2, but -2 can’t be represented as an unsigned number.

```cpp
#include <iostream>

int main()
{
 unsigned int x{ 3 };
 unsigned int y{ 5 };

 std::cout << x - y <<
'\n';
 return 0;
}
```

On the author’s machine, this seemingly innocent looking program produces the result:

```
4294967294
```

This occurs due to -2 wrapping around to a number close to the top of the range of a 4-byte integer. Another common unwanted wrap-around happens when an unsigned integer is repeatedly decremented by 1 (using the `--` operator). You'll see an example of this when loops are introduced.

Second, unexpected behavior can result when you mix signed and unsigned integers. In a mathematical operation in C++ (e.g. arithmetic or comparison), if one signed and one unsigned integer are used, the signed integer will be converted to unsigned. And because unsigned integers can not store negative numbers, this can result in loss of data.

Consider the following program demonstrating this:

```cpp
#include <iostream>

int main()
{
    signed int s { -1 };
    unsigned int u { 1 };

    if (s < u) // -1 is implicitly converted to 4294967295, and 4294967295 < 1 is false
        std::cout << "-1 is less than 1\n";
    else
        std::cout << "1 is less than -1\n"; // this statement executes

    return 0;
}
```

This program is well formed, compiles, and is logically consistent to the eye. But it prints the wrong answer. And while your compiler should warn you about a signed/unsigned mismatch in this case, it will also generate similar warnings for other cases that do not suffer from this problem (e.g. when both numbers are positive), making it hard to detect when there is an actual problem.

Additionally, there are other problematic cases that are essentially undetectable. Consider the following:

```cpp
void doSomething(unsigned int x)
{
    // Run some code x times

    std::cout << "x is " << x << '\n';
}

int main()
{
    doSomething(-1);

    return 0;
}
```

The author of doSomething() was expecting someone to call this function with only positive numbers. But the caller is passing in *-1* -- clearly a mistake, but one made none-the-less. What happens in this case?

The signed argument of *-1* gets implicitly converted to an unsigned parameter. -1 isn't in the range of an unsigned number, so it wraps around to some large number (probably 4294967295). Then your program goes ballistic. Worse, there's no good way to guard against this condition from happening. C++ will freely convert between signed and unsigned numbers, but it won't do any range checking to make sure you don't overflow your type.

All of these problems are commonly encountered, produce unexpected behavior, and are hard to find, even using automated tools designed to detect problem cases.

Given the above, the somewhat controversial best practice that we'll advocate for is to avoid unsigned types except in specific circumstances.

Best practice

Favor signed numbers over unsigned numbers for holding quantities (even quantities that should be non-negative) and mathematical operations. Avoid mixing signed and unsigned numbers.

Related content

Additional material in support of the above recommendations (also covers refutation of some common counter-arguments):

1. Interactive C++ panel (see 12:12-13:08, 42:40-45:26, and 1:02:50-1:03:15)
2. Subscripts and sizes should be signed
3. Unsigned integers from the libtorrent blog

Converting unsigned numbers to signed numbers

To convert an unsigned number to a signed number, you can use the `static_cast` operator:

```
unsigned int u { 5u };
int s { static_cast<int>(u) }; // convert u into an
int
```

If the number is too large to fit in the range of the int, undefined behavior will result.

So when should you use unsigned numbers?

There are still a few cases in C++ where it's okay / necessary to use unsigned numbers.

First, unsigned numbers are preferred when dealing with bit manipulation (covered in chapter O (That's a capital 'o', not a '0'). They are also useful when well-defined wrap-around behavior is required (useful in some algorithms like encryption and random number generation).

Second, use of unsigned numbers is still unavoidable in some cases, mainly those having to do with array indexing. We'll talk more about this in the lessons on arrays and array indexing.

Also note that if you're developing for an embedded system (e.g. an Arduino) or some other processor/memory limited context, use of unsigned numbers is more common and accepted (and in some cases, unavoidable) for performance reasons.

Next lesson
4.6 Fixed-width integers and size_t

Back to table of contents

Previous lesson
4.4 Signed integers

X

X