4.4 — Signed integers

An **integer** is an integral type that can represent positive and negative whole numbers, including 0 (e.g. -2, -1, 0, 1, 2). C++ has *4* primary fundamental integer types available for use:

Type	Minimum Size	Note
short int	16 bits	
int	16 bits	Typically 32 bits on modern architectures
long int	32 bits	
long long int	64 bits	

The key difference between the various integer types is that they have varying sizes -- the larger integers can hold bigger numbers.

A reminder

C++ only guarantees that integers will have a certain minimum size, not that they will have a specific size. See lesson 4.3 -- Object sizes and the sizeof operator for information on how to determine how large each type is on your machine.

As an aside…

Technically, the `bool` and `char` types are considered to be integral types (because these types store their values as integer values). For the purpose of the next few lessons, we'll exclude these types from our discussion.

Signed integers

When writing negative numbers in everyday life, we use a negative sign. For example, *-3* means "negative 3". We'd also typically recognize *+3* as "positive 3" (though common convention dictates that we typically omit plus prefixes).

This attribute of being positive, negative, or zero is called the number's **sign**.

By default, integers in C++ are **signed**, which means the number's sign is stored as part of the number. Therefore, a signed integer can hold both positive and negative numbers (and 0).

In this lesson, we'll focus on signed integers. We'll discuss unsigned integers (which can only hold non-negative numbers) in the next lesson.

Related content

In binary representation, a single bit (called the **sign bit**) is used to store the sign of the number. The non-sign bits (called the **magnitude bits**) determine the magnitude of the number.

We discuss how the sign bit is used when representing numbers in binary in lesson O.4 -- Converting integers between binary and decimal representation.

Defining signed integers

Here is the preferred way to define the four types of signed integers:

```
short s;      // prefer "short" instead of "short int"
int i;
long l;       // prefer "long" instead of "long int"
long long ll; // prefer "long long" instead of "long long int"
```

Although *short int*, *long int*, or *long long int* will work, we prefer the short names for these types (that do not use the *int* suffix). In addition to being more typing, adding the *int* suffix makes the type harder to distinguish from variables of type *int*. This can lead to mistakes if the short or long modifier is inadvertently missed.

The integer types can also take an optional *signed* keyword, which by convention is typically placed before the type name:

```
signed short ss;
signed int si;
signed long sl;
signed long long sll;
```

However, this keyword should not be used, as it is redundant, since integers are signed by default.

Best practice

Prefer the shorthand types that do not use the `int` suffix or `signed` prefix.

Signed integer ranges

As you learned in the last section, a variable with *n* bits can hold 2^n possible values. But which specific values? We call the set of specific values that a data type can hold its **range**. The range of an integer variable is determined by two factors: its size (in bits), and whether it is signed or not.

By definition, an 8-bit signed integer has a range of -128 to 127. This means a signed integer can store any integer value between -128 and 127 (inclusive) safely.

As an aside…

Math time: an 8-bit integer contains 8 bits. 2^8 is 256, so an 8-bit integer can hold 256 possible values. There are 256 possible values between -128 to 127, inclusive.

7 bits are used to hold the magnitude of the number, and 1 bit is used to hold the sign.

Here's a table containing the range of signed integers of different sizes:

Size/Type	Range
8 bit signed	-128 to 127
16 bit signed	-32,768 to 32,767
32 bit signed	-2,147,483,648 to 2,147,483,647
64 bit signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

For the math inclined, an n-bit signed variable has a range of $-(2^{n-1})$ to $2^{n-1}-1$.

For the non-math inclined… use the table. :)

Overflow

What happens if we try to assign the value *140* to an 8-bit signed integer? This number is outside the range that an 8-bit signed integer can hold. The number 140 requires 9 bits to represent (8 magnitude bits and 1 sign bit), but we only have 8 bits (7 magnitude bits and 1 sign bit) available in an 8-bit signed integer.

The C++20 standard makes this blanket statement: "If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined". Colloquially, this is called **overflow**.

Therefore, assigning value 140 to an 8-bit signed integer will result in undefined behavior.

If an arithmetic operation (such as addition or multiplication) attempts to create a value outside the range that can be represented, this is called **integer overflow** (or **arithmetic overflow**). For signed integers, integer overflow will result in undefined behavior.

```
#include <iostream>

int main()
{
    // assume 4 byte integers
    int x { 2'147'483'647 }; // the maximum value of a 4-byte signed integer
    std::cout << x << '\n';

    x = x + 1; // integer overflow, undefined behavior
    std::cout << x << '\n';

    return 0;
}
```

On the author's machine, the above printed:

```
2147483647
-2147483648
```

However, because the second output is the result of undefined behavior, the value output may vary on your machine.

For advanced readers

We cover what happens when unsigned integers overflow in lesson 4.5 -- Unsigned integers, and why to avoid them.

In general, overflow results in information being lost, which is almost never desirable. If there is *any* suspicion that an object might need to store a value that falls outside its range, use a type with a bigger range!

Integer division

When dividing two integers, C++ works like you'd expect when the quotient is a whole number:

```
#include <iostream>

int main()
{
    std::cout << 20 / 4 << '\n';
    return 0;
}
```

This produces the expected result:

```
5
```

But let's look at what happens when integer division causes a fractional result:

```
#include <iostream>

int main()
{
    std::cout << 8 / 5 << '\n';
    return 0;
}
```

This produces a possibly unexpected result:

```
1
```

When doing division with two integers (called **integer division**), C++ always produces an integer result. Since integers can't hold fractional values, any fractional portion is simply dropped (not rounded!).

Taking a closer look at the above example, 8 / 5 produces the value 1.6. The fractional part (0.6) is dropped, and the result of 1 remains. Alternatively, we can say 8 / 5 equals 1 remainder 3. The remainder is dropped, leaving 1.

Similarly, -8 / 5 results in the value -1.

Warning

Be careful when using integer division, as you will lose any fractional parts of the quotient. However, if it's what you want, integer division is safe to use, as the results are predictable.

If fractional results are desired, we show a method to do this in lesson 6.2 -- Arithmetic operators.

Quiz time

Question #1

What would the range of a 5-bit signed integer be?

Show Solution

Question #2

a) What is the result of 13 / 5?

Show Solution

b) What is the result of -13 / 5?

Show Solution