2.4 — Integers

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 29TH, 2018

An **integer** type (sometimes called an integral type) variable is a variable that can only hold non-fractional numbers (e.g. -2, -1, 0, 1, 2). C++ has *five* different fundamental integer types available for use:

Category	Type	Minimum Size	Note
character	char	1 byte	
integer	short	2 bytes	
	int	2 bytes	Typically 4 bytes on modern architectures
	long	4 bytes	
	long long	8 bytes	C99/C++11 type

Char is a special case, in that it falls into both the character and integer categories. We'll talk about the special properties of char later. In this lesson, you can treat it as a normal integer.

The key difference between the various integer types is that they have varying sizes -- the larger integers can hold bigger numbers. Note that C++ only guarantees that integers will have a certain minimum size, not that they will have a specific size. See lesson **2.3 -- variable sizes and the sizeof operator** for information on how to determine how large each type is on your machine.

Defining integers

Defining some integers:

```
char c;
short int si; // valid
short s;      // preferred
int i;
long int li; // valid
long l;      // preferred
long long int lli; // valid
long long ll;      // preferred
```

While *short int*, *long int*, and *long long int* are valid, the shorthand versions *short*, *long*, and *long long* should be preferred. 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.

Identifying integer

Because the size of char, short, int, and long can vary depending on the compiler and/or computer architecture, it can be instructive to refer to integers by their size rather than name. We often refer to integers by the number of bits a variable of that type is allocated (e.g. "32-bit integer" instead of "long").

Integer ranges and sign

As you learned in the last section, a variable with n bits can store 2^n different 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 its **sign**, which can be "signed" or "unsigned".

A **signed** integer is a variable that can hold both negative and positive numbers. To explicitly declare a variable as signed, you can use the *signed* keyword:

```
signed char c;
signed short s;
signed int i;
signed long l;
signed long long ll;
```

By convention, the keyword “signed” is placed before the variable’s data type.

A 1-byte signed integer has a range of -128 to 127. Any value between -128 and 127 (inclusive) can be put in a 1-byte signed integer safely.

Sometimes, we know in advance that we are not going to need negative numbers. This is common when using a variable to store the quantity or size of something (such as your height -- it doesn’t make sense to have a negative height!). An **unsigned** integer is one that can only hold positive values. To explicitly declare a variable as unsigned, use the *unsigned* keyword:

```
unsigned char c;
unsigned short s;
unsigned int i;
unsigned long l;
unsigned long long ll;
```

A 1-byte unsigned integer has a range of 0 to 255.

Note that declaring a variable as unsigned means that it can not store negative numbers, but it can store positive numbers that are twice as large.

Now that you understand the difference between signed and unsigned, let’s take a look at the ranges for different sized signed and unsigned variables:

Size/Type	Range
1 byte signed	-128 to 127
1 byte unsigned	0 to 255
2 byte signed	-32,768 to 32,767
2 byte unsigned	0 to 65,535
4 byte signed	-2,147,483,648 to 2,147,483,647
4 byte unsigned	0 to 4,294,967,295
8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
8 byte unsigned	0 to 18,446,744,073,709,551,615

For the math inclined, an n-bit signed variable has a range of $-(2^{n-1})$ to $2^{n-1}-1$. An n-bit unsigned variable has a range of 0 to $(2^n)-1$. For the non-math inclined… use the table. ☺

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 typically 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.

Default signs and integer best practices

So what happens if we do not declare a variable as signed or unsigned?

Category	Type	Default Sign	Note
character	char	Signed or Unsigned	Usually signed
integer	short	Signed	
	int	Signed	
	long	Signed	
	long long	Signed	

All integer variables except char are signed by default. Char can be either signed or unsigned by default (but is usually signed for conformity).

Generally, the signed keyword is not used (since it’s redundant), except on chars (when necessary to ensure they are signed).

Best practice is to avoid use of *unsigned* integers unless you have a specific need for them, as unsigned integers are more prone to unexpected bugs and behaviors than signed integers.

Rule: Favor signed integers over unsigned integers

Overflow

What happens if we try to put a number outside of the data type's range into our variable? **Overflow** occurs when bits are lost because a variable has not been allocated enough memory to store them.

In lesson **2.1 -- Fundamental variable definition, initialization, and assignment**, we mentioned that data is stored in binary format.

In binary (base 2), each digit can only have 2 possible values (0 or 1). We count from 0 to 15 like this:

Decimal Value	Binary Value
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

As you can see, the larger numbers require more bits to represent. Because our variables have a fixed number of bits, this puts a limit on how much data they can hold.

Overflow examples

Consider a hypothetical unsigned variable that can only hold 4 bits. Any of the binary numbers enumerated in the table above would fit comfortably inside this variable (because none of them are larger than 4 bits).

But what happens if we try to assign a value that takes more than 4 bits to our variable? We get overflow: our variable will only store the 4 least significant (rightmost) bits, and the excess bits are lost.

For example, if we tried to put the decimal value 21 in our 4-bit variable:

Decimal Value	Binary Value
21	10101

21 takes 5 bits (`10101`) to represent. The 4 rightmost bits (`0101`) go into the variable, and the leftmost (`1`) is simply lost. Our variable now holds `0101`, which is the decimal value 5.

Note: At this point in the tutorials, you're not expected to know how to convert decimal to binary or vice-versa. We'll discuss that in more detail in section **3.7 -- Converting between binary and decimal**.

Now, let's take a look at an example using actual code, assuming a short is 16 bits:

```
#include <iostream>

int main()
{
    unsigned short x = 65535; // largest 16-bit unsigned value possible
    std::cout << "x was: " << x << std::endl;
    x = x + 1; // 65536 is out of our range -- we get overflow because x can't hold 17 bits
    std::cout << "x is now: " << x << std::endl;
    return 0;
}
```

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

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

What happened? We overflowed the variable by trying to put a number that was too big into it (65536), and the result is that our value "wrapped around" back to the beginning of the range.

> For advanced readers, here's what's actually happening behind the scenes: the number 65,535 is represented by the bit pattern `1111 1111 1111 1111` in binary. 65,535 is the largest number an unsigned 2 byte (16-bit) integer can hold, as it uses all 16 bits. When we add 1 to the value, the new value should be 65,536. However, the bit pattern of 65,536 is represented in binary as `1 0000 0000 0000 0000`, which is 17 bits! Consequently, the highest bit (which is the 1) is lost, and the low 16 bits are all that is left. The bit pattern `0000 0000 0000 0000` corresponds to the number 0, which is our result.

Similarly, we can overflow the bottom end of our range as well, resulting in "wrapping around" to the top of the range.

```
#include <iostream>

int main()
{
    unsigned short x = 0; // smallest 2-byte unsigned value possible
    std::cout << "x was: " << x << std::endl;
    x = x - 1; // overflow!
    std::cout << "x is now: " << x << std::endl;
    return 0;
}
```

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

Overflow results in information being lost, which is almost never desirable. If there is *any* suspicion that a variable might need to store a value that falls outside its range, use a larger variable!

Also note that the results of overflow are only predictable for unsigned integers. Overflowing signed integers or non-integers (e.g. floating point numbers) may result in different results on different systems.

Rule: Do not depend on the results of overflow in your program.

Integer division

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

```
#include <iostream>

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

This produces a possibly unexpected result:

1

When doing division with two integers, C++ 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.

Rule: Be careful when using integer division, as you will lose any fractional parts of the result


2.4a -- Fixed-width integers and the unsigned controversy


Index


2.3 -- Variable sizes and the sizeof operator

Share this:

C++ TUTORIAL | PRINT THIS POST

150 comments to 2.4 — Integers

« Older Comments 1 2

Callaghan Ebojoh
July 10, 2018 at 4:05 am · Reply

What happens when you try to compile a program if you declare the iostream header file before #include "stdafx.h"
- When running a program with erros (more errors are generated running when the iostream header file is included first)
- Is there any particular reason for this

> Alex