4.4 — Signed integers

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON DECEMBER 22ND, 2019

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

Type	Minimum Size	Note
short	2 bytes	
int	2 bytes	Typically 4 bytes on modern architectures
long	4 bytes	
long long	8 bytes	C99/C++11 type

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.

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 are **signed**, which means the number’s sign is preserved. 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.

Defining signed integers

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

```
short s;
int i;
long l;
long long ll;
```

All of the integers (except int) can take an optional *int* suffix:

```
short int si;
long int li;
long long int lli;
```

This suffix should not be used. 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 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 whether it is signed or not.

By definition, a 1-byte 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: a 1-byte integer contains 8 bits. 2^8 is 256, so a 1-byte integer can hold 256 different values. There are 256 different values between -128 to 127, inclusive.

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

Size/Type	Range
1 byte signed	-128 to 127
2 byte signed	-32,768 to 32,767
4 byte signed	-2,147,483,648 to 2,147,483,647
8 byte 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. :)

Integer overflow

What happens if we try to assign the value *280* to a 1-byte signed integer? This number is outside the range that a 1-byte signed integer can hold. The number 280 requires 9 bits (plus 1 sign bit) to be represented, but we only have 7 bits (plus 1 sign bit) available in a 1-byte signed integer.

Integer overflow (often called *overflow* for short) occurs when we try to store a value that is outside the range of the type. Essentially, the number we are trying to store requires more bits to represent than the object has available. In such a case, data is lost because the object doesn't have enough memory to store everything.

In the case of signed integers, which bits are lost is not well defined, thus signed integer overflow leads to undefined behavior.

> **Warning**
>
> Signed integer overflow will result in undefined behavior.

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:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << 20 / 4;
6     return 0;
7 }
```

This produces the expected result:

5

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

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << 8 / 5;
6     return 0;
7 }
```

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.

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.

4.5 -- Unsigned integers, and why to avoid them

Index

4.3 -- Object sizes and the sizeof operator

C++ TUTORIAL | PRINT THIS POST

213 comments to 4.4 — Signed integers

« Older Comments 1 2 3

Air Paul
February 1, 2020 at 10:38 am · Reply

It is given in this tutorial that the capacity of the signed integer is 256 ranging from -128 to 127, a/c to me it should be from -127 to 127, means 255 numbers but 256 representations as because zero can be represented as 00000000, 10000000.

nascardriver
February 2, 2020 at 1:12 am · Reply

8 bit binary 1000 0000 is -128 decimal. Only 0000 0000 is 0.

Air Paul
February 2, 2020 at 8:17 am · Reply

Why? as for the 8-bit representation of a binary number, the first digit will be used for deciding its sign and the rest of the seven digits will be used for the number itself. So, 10000000 represents 0. If I'm wrong please correct me with a brief explanation.

nascardriver
February 2, 2020 at 8:35 am · Reply

Negative numbers are represented in two's complement. This is covered in lesson O.4.

Air Paul
February 3, 2020 at 6:32 am · Reply

Okay, I thought we're using sign-bit representation. Thanks!!!

Kermit
January 14, 2020 at 6:24 am · Reply

hmm just for my satisfaction xD

```
#include <iostream>

using namespace std;

int main()
{
    /*
    bool:           1 bytes
    char:           1 bytes
    wchar_t:        2 bytes
    char16_t:       2 bytes
    char32_t:       4 bytes
    short:          2 bytes
    int:            4 bytes
    long:           4 bytes
    long long:      8 bytes
    float:          4 bytes
    double:         8 bytes
    long double:    8 bytes
    */
    /*
    Size/Type            Range // these are for signed integers
    1 byte signed        -128 to 127
    2 byte signed        -32,768 to 32,767
    4 byte signed        -2,147,483,648 to 2,147,483,647
    8 byte signed        -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807


    Size/Type            Range // these are for unsigned integers
    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
    */


    int signedint{ 2147483647 };                                    // signed int can hold upto 2147483647. they are of 4 bytes.
    unsigned int unsignedint{ 4294967295 };                          // unsigned int can hold upto  4,294,967,295, they are of 4 bytes.
    long long signed_long_long{ 9223372036854775807 };               // signed 'long' is 4 bytes but twice 'long long' are of 8 bytes, single one can hold upto 2147483647 and double can hold 9,223,372,036,854,775,807.
    unsigned long long unsigned_long_long{ 18446744073709551615 };   // unsigned 'long long' are of 8 bytes which can hold 18,446,744,073,709,551,615.
    short shortx{ 30000 };                                          // signed short integer can hold upto 30k, they are of 2 bytes
    unsigned shortx2{ 65535 };                                      // unsigned short integer can hold upto 60k, they are of 2 bytes

    cout << "Simple Program that tells about Integer Ranges.\n\n";
```

```
    cout << "Signed Int (4 byte) can hold:\t\t\t" << signedint << '\n';
    cout << "Unsigned Int (4 byte) can hold:\t\t\t" << unsignedint << '\n';
    cout << "Signed long Integer (4 byte) can hold:\t\t2147483647\n";
    cout << "Unsigned long Integer (4 byte) can hold:\t4294967295\n";
    cout << "Signed long long integer (8 byte) can hold:\t" << signed_long_long << '\n';
    cout << "Unsigned long long Integer (8 byte) can hold:\t" << unsigned_long_long <<
'\n';
    cout << "Signed short integer (2 byte) can hold: \t" << "upto " << shortx << '\n';
    cout << "Unsigned short integer (2 byte) can hold:\t" << shortx2 << '\n';
    return 0;
}
```

nascardriver
January 14, 2020 at 6:54 am · Reply

Those are the maximum sizes on your system, but they can vary, causing your code to not compile (Because the numbers are too large for the type). A portable solution involves using `std::numeric_limits`

```
#include <iostream>
#include <limits>

int main()
{
  std::cout << std::numeric_limits<int>::max() << '\n';

  return 0;
}
```

HZ
December 21, 2019 at 1:05 pm · Reply

An editorial suggestion: "The number 280 requires 9 bits to represent (plus the sign bit), but we only have 7 bits (plus the sign bit) available in a 1-byte integer."

nascardriver
December 22, 2019 at 2:53 am · Reply

Lesson updated, thanks!

Ayberk
September 15, 2019 at 8:24 am · Reply

What is the difference between long and int on modern architectures? They have the same size.

Alex
September 19, 2019 at 11:55 am · Reply

If long and int have the same size on your architecture then they function identically.

Samira Ferdi
May 28, 2019 at 6:39 pm · Reply

Hi Alex and Nascardriver!

I have 3 questions here:

1. signed integer can hold negative number, but what about the negative sign?
2. can string type overflow?
3. How to understand the difference between overflow and wrap-around (modulo wrapping)?

nascardriver
May 29, 2019 at 2:10 am · Reply

1.
The internal representation of integers is implementation defined. Most compilers use two's complement (covered in the lesson about binary conversion).

2.
You could run out of memory.

3.
overflow: You add a number, the addition is performed, all binary digits that don't fit into the type are dropped.
wrap around: You add a number, if it doesn't fit into your type, its remainder is stored.

signed numbers overflow (I don't know if this is well defined), unsigned numbers explicitly wrap around.

Samira Ferdi
June 10, 2019 at 5:10 pm · Reply

Thank you for reply!

1. the range of type is -2^(n-1) to 2^(n-1)-1. Why n-1?
2. I don't get it

nascardriver
June 11, 2019 at 2:20 am · Reply

Your total number of values is `2^n`, where `n` is the number of bits. We need 1 sign bit, leaving `n-1` bits for the number, thus we have `2^(n-1)` negative- and `2^(n-1)` non-negative numbers. The non-negative numbers include `0`, so the maximum value is `2^(n-1) - 1`. The negative numbers remain unchanged, the minimum value is `-(2^(n-1))`. Binary numbers are covered in lesson O.3.7.

David Sitner
February 28, 2019 at 6:42 pm · Reply

Alex,
Firstly, this site is AMAZING! Thank you.
Secondly as a simple editing note, do you mean to use the word integral in the paragraph below? It seems like you meant to use integer instead.

Pretty simple, right? We can infer that operator sizeof returns an integral value -- but what type of integer is that value? An int? A short? The answer is that sizeof (and many functions that return a size or length value) return a value of type “size_t”. size_t is an unsigned, integral value that is typically used to represent the size or length of objects.

Alex

March 1, 2019 at 5:01 pm · Reply

Yes, I think it makes more sense to use "integer" than "integral" here. Thanks for the feedback!

Alireza
January 27, 2019 at 12:37 pm · Reply

Hello dear,
What is the ' size_t ' exactly ?
I've read the above texts, but I haven't understood what the 'size_t' does exactly.
Can it be a data type ?

```
1    size_t a=4294967295;
2    std::cout << a+1;
```

Output is 0, it's a 4-byte unsigned type.

nascardriver
January 29, 2019 at 8:19 am · Reply

Hi!
See my answer here
https://www.learncpp.com/cpp-tutorial/24-integers/comment-page-3/#comment-390187

> it's a 4-byte unsigned type
On your system, yes. It might not be somewhere else.

nascardriver
January 20, 2019 at 2:42 am · Reply

> Is that [] considered overflow
Yes

> Is that technically considered overflow
No. Unsigned types don't overflow, they explicitly wrap around.

Alex
January 23, 2019 at 8:57 pm · Reply

Integer overflow is defined by Wikipedia as, "[occuring] when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits."

Based on that definition, unsigned types do overflow. They just exhibit guaranteed wrap-around behavior when they do.

nascardriver
January 24, 2019 at 5:31 am · Reply

The range of representable values for the unsigned type is 0 to 2 N − 1 (inclusive); arithmetic for the unsigned type is performed modulo 2 N . [Note: Unsigned arithmetic does not overflow. Overflow for signed arithmetic yields undefined behavior (7.1). — end note]

N4791 § 6.7.1 2

Jeremy
January 12, 2019 at 6:34 pm · Reply

So if i understand correctly (from reading multiple definitions of size_t), size_t returns the maximum width an integer data type can store in bytes in memory? and this is used to understand the maximum value that an integer can be stored can be across unique systems before overflowing occurs giving us an undesired result? and is mainly used for portability?

nascardriver
January 13, 2019 at 6:21 am · Reply

@std::size_t doesn't return anything, it's a type, like int, float, etc.
It can store the number that describes the size of the largest type.
Let's say your compiler allows an object to be 4294967295 bytes (maximum on a 32bit system) in size.
@std::size_t can store that number.
This is unrelated to integers. You can check how large an integer is using @sizeof

```
1 std::cout << sizeof(int) << '\n';
```

which probably print 4.
@std::size_t is required, because you need a type that can store sizes. Looking at @sizeof, you need a type that can store the return value. This is @std::size_t.

Jeremy
January 14, 2019 at 12:07 am · Reply

Okay, i got that one really wrong.

So i think i understand it better. size_t is an integer data type that can store the largest object in size(in bytes) that the compiler will allow. It's useful as a return value for sizeof because it's unsigned(memory can only be positive integer value) and can work with the largest objects allowed so it's guaranteed to safely return a value of all data types. Plus it is also useful for array indexing and for loops in place of int as it can store the largest value in bytes so there is less chance of overflow?

Is this correct? Is there a way to check for integer overflow just in case or is that not really needed if using size_t?

nascardriver
January 15, 2019 at 7:41 am · Reply

Right.
Be careful when using it as an iterator in loops. It's less likely to overflow, but if you don't watch out, it might underflow (go below 0).

> Is there a way to check for integer overflow just in case or is that not really needed if using size_t?
If you're working close to min/max values, you should verify that the over/underflow won't occur _before_ doing the calculation.

cren

January 20, 2019 at 3:49 pm · Reply

Hi,
I have a question on size_t. I am sure my system is 64 bits, I expected to see std::cout << sizeof(size_t); print out 8 on console.

But when I ran above line in Eclipse, the output is 8, the output from code blocks is 4, what I am doing wrong here?

Thanks!

nascardriver
January 21, 2019 at 4:13 am · Reply

On a 64 bit system, you can compile in 32 or 64 bit. If you're using g++ or clang++, the flag you're looking for is -m64
If you're using something else or don't know how to set compiler flags, you'll have to search through code::block's settings.

cren
January 21, 2019 at 6:34 am · Reply

Thank you for the reply.

Yes, if I run g++ with -m64 from command line. it works OK.
In CodeBlocks, I set /Build options/Compiler settings/Compiler Flags/"Target x_86_64(64bit) [-m64]" to true.
It turn out compile error - "sorry, unimplemented: 64-bit mode not compiled in".

Are there other settings need to turn on? Looks like CodeBlocks issue.

nascardriver
January 21, 2019 at 9:10 am · Reply

It looks like your Code::Blocks is using a different compiler or version. If it works on the command line, it should work in your IDE (Unless the compiler is different).
I don't have Code::Blocks so I won't be of much help from here on.

Jeremy
January 12, 2019 at 5:05 pm · Reply

I understand that using and unsigned short and going from it's max limit of 65,535 (1111 1111 1111 1111) to 65,536 (1 0000 0000 0000 000) results in only being able to keep the right most 16 bits (1 [0000 0000 0000 0000]) subsequently returning a 0.

But how do we end up with the binary value of 1111 1111 1111 1111 (65,535) from the binary value of 0 if we have and unsigned short assigned as -1?

Why does it “wrap around” the top of the range like that?

nascardriver
January 13, 2019 at 5:55 am · Reply

Hi Jeremy!

Lesson 3.7 covers binary representation of integers (You're looking for two's complement).

Jeremy
January 20, 2019 at 2:43 am · Reply

Thanks. I finally made it to 3.7

So if i understand correctly, it's not that it wraps around, it's just -1 is signed and represented as 1111 1111 1111 1111 in binary when using two's complement.

So from the overflow example above:

```
1 unsigned short x = 0; // smallest 2-byte unsigned value possible
2 std::cout << "x was: " << x << std::endl;
3 x = x - 1; // overflow!
```

Is that technically considered overflow since the compiler is just storing -1 (1111 1111 1111 1111) into memory? It's only since the data type x in the example is declared unsigned that the compiler interprets the stored binary value of 1111 1111 1111 1111 as 65,535?

EDIT** I get why the example is considered overflow now, it's that we overflowed/stepped out of our range when we went below 0 decrementing by 1. It's not that we directly assigned a negative number to an unsigned data type/

Two's complement is really making my brain melt. But i think i am starting to get it.

nascardriver
January 20, 2019 at 2:45 am · Reply

After editing a comment, the syntax highlighter will work after refreshing the page.
My reply to your original comment is https://www.learncpp.com/cpp-tutorial/24-integers/comment-page-3/#comment-391303

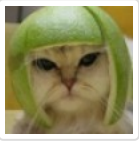
Dimitri
January 8, 2019 at 2:17 am · Reply

Hi, dear teacher!

Please correct

"per the table above, 4,294,967,295 bytes" - it's not bytes

Thanks for great site!



Alex
January 9, 2019 at 4:58 pm · Reply

Thanks for pointing out the typo!

Bastiaan
December 27, 2018 at 8:25 am · Reply

Hello,
I just have one quick question:
Why would we use long if the normal int is as well 4 bytes? Is it because the size of int is unsure? If so should we then stop using int, or is int fine for the learning part of C++?
Thanks!

*edit: With the next chapter I see that the long double can be(and is with my labtop) as long as the double. So am I right to assume that there is more difference between the variable?
again Thanks!

nascardriver
December 27, 2018 at 8:54 am · Reply

> normal int
There is no such thing as a "normal" int. The size of an int is decided by your compiler. "long" suggests that the compiler should use a larger type, but it doesn't have to.
Your compiler might use the same type for long int/int and long double/double, another compiler or other compiler settings might behave differently.

Bastiaan
December 27, 2018 at 9:03 am · Reply

OK, thank you!

Yanal
December 21, 2018 at 6:05 am · Reply

Hi
I am wondering about the value for 1 byte that it should be -127 instead of 128.

1 byte signed -128 to 127
I think it should be from -127 to +127.
-111 1111 TO +111 1111
-127 TO 127
can you please show how you get these ranges and tell me which bit is used for sign ?
Thanks

nascardriver
December 21, 2018 at 6:13 am · Reply

-127 to 127 (inclusive) means that there are 255 possible numbers. There must be 2^8 = 256 numbers, making your statement impossible.
0000'0000 to 0111'1111 (inclusive) are used to represent non-negative numbers (0 to 127 (inclusive))
1000'0000 to 1111'1111 (inclusive) are used to represent negative numbers (-1 to -128 (inclusive))

Lesson 3.7 goes into more detail about binary representations of integers.

Clapfish
November 2, 2018 at 12:58 pm · Reply

Since I'm coding a 32bit Console application, that means size_t = 4, as you also demonstrated in this lesson.

Shouldn't that then mean long long integers (being 64bit) can't be used in such an application?

I tested it however, and it seems like they can…

Clapfish
November 3, 2018 at 1:03 am · Reply

On further investigation, interestingly:

```
#include <iostream>
int main()
{
    unsigned long long x { 9223372036854775807 };
    std::cout << x;
    return 0;
}
```

… compiles and runs fine. However, if I go over this number (which, as far as I understand it, should be half of the range available to this variable):

```
#include <iostream>
int main()
{
    unsigned long long x { 9223372036854775808 };
    std::cout << x;
    return 0;
}
```

… I get the following compile error:

error: integer constant is so large that it is unsigned
    unsigned long long x { 9223372036854775808 };
                           ^

Is this because of the 32bit limitation, and size_t being 4?

Using an implicitly 'signed' long long seems to work fine though:

```
#include <iostream>
int main()
{
    long long x { 9223372036854775807 };
    std::cout << x;
    return 0;
}
```

… and …

```
#include <iostream>
int main()
{
    long long x { -9223372036854775807 };
    std::cout << x;
    return 0;
}
```

… both compile without issue.

This, to me, suggests that the full 8 byte range of long long is available to this 32bit console application, but only half the range of the 8 byte 'unsigned' long long is available (4 bytes worth).

Is it because the 'negative version' of the number can be stored in 4 bytes, just like the positive version, but with the 'two's complement flipping' of the bit values? I suppose that would explain it, because going over this number (either positive or negative) would then need that extra bit which is unavailable in this 32bit application…

I'd really appreciate an explanation of what's going on here!

Cheers :)

nascardriver
November 3, 2018 at 4:50 am · Reply

Hi!

> Shouldn't that then mean long long integers (being 64bit) can't be used in such an application?
No. You can use variables as big as you like, but using 64bit variables in a 32bit application is slower than using using 64bit variables in a 64bit application or using 32bit variables in a 32bit application.

> integer constant is so large that it is unsigned
> Is this because of the 32bit limitation, and size_t being 4?
No. @x is an unsigned long long, but the number you're initializing @x with is a long, or at least that's what it's supposed to be, but it can't, because it's too large.
C++ determines the type of the value before it considers the type of the variable you're initializing with said value.
You need to explicitly make the number an unsigned long long by appending "ull"

```
1 unsigned long long x { 9223372036854775808ull };
```

This is covered in lesson 2.8

Clapfish
November 3, 2018 at 10:51 am · Reply

Hi nascardriver,

Many thanks for your reply! I've tested the 'ull' appendage and that works as you suggest.

I'm still a little confused though, since the lesson material seems to suggest that 8 byte objects shouldn't be usable when size_t is 4:

"size_t is an unsigned, integral value that is typically used to represent the size or length of objects, and it is defined to be big enough to hold the size of the largest object createable on your system."

and

"By definition, any object larger than the largest value size_t can hold is considered ill-formed (and will cause a compile error), as the sizeof operator would not be able to return the size without overflow."

Isn't using a long long, of size 8 bytes, considered 'ill-formed' (as per the above statement) in a 32-bit application where size_t = 4, and therefore shouldn't I be getting a compile error as the statement suggests?

That's why I wanted to test it in the first place, and being able to use long long integers made me want to investigate further...

Cheers!

nascardriver
November 3, 2018 at 11:03 am · Reply

I see why you're confused as the quoted text can be understood two ways. What it means to say is that the size of any object in bytes cannot exceed the maximum value @std::size_t can store.
Let's the maximum value of @std::size_t is (2^64)-1 = 18446744073709551615, which is the case in 64bit applications. Then the maximum size of an object is (2^64)-1 bytes = 16EiB.

Clapfish
November 3, 2018 at 12:21 pm · Reply

Thanks for the clarification! I think I see now...

So, in a 64-bit application the maximum size of a single object is over 18 billion Gigabytes?

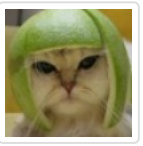
Can one even make an object that large? To me that seems to be a limit with no practical application or meaning, at least from my current (very possibly naive) position...

nascardriver
November 4, 2018 at 1:52 am · Reply

> size of a single object is over 18 billion Gigabytes?
Yes

> Can one even make an object that large?
If you have enough and fast memory, sure.

> that seems to be a limit with no practical application or meaning
It doesn't mean that you're supposed to create objects that large.
64 bit memory can address (2^64)-1 bytes. We're far from reaching that limit, but theoretically a 64 bit system could have that much memory, meaning that it'd be possible to create an object that large (assuming nothing else is taking up memory). When an object of that size can be created, there needs to be a variable that can store the size of that object. Hello @std::size_t.



Alex
November 5, 2018 at 10:01 am · Reply

Good feedback. I've updated the lesson text to try and clarify this better. Let me know if it's still unclear.

Clapfish
November 6, 2018 at 2:03 pm · Reply

Thanks Alex, looks less ambiguous now!

vbot
September 26, 2018 at 6:09 am · Reply

Hi,

So, there are no any compiler-errors / warnings when an int overflow occurs, just undesired values, right?

I did find some information on methods concerning detecting / testing for overflows though, what do you think about those in general? How common are such techniques?

Thanks! :)

nascardriver
September 26, 2018 at 6:17 am · Reply

As long as you're using the proper types and clamping the values that should be clamped, integers overflow aren't a problem.

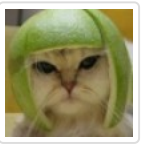
vbot
September 26, 2018 at 6:49 am · Reply

Ok, Thanks! :)

lucusgod
August 24, 2018 at 5:45 am · Reply

"size_t is guaranteed to be unsigned and at least 16 bits, but on most systems will be equivalent to the largest unsigned integer that your architecture supports. For 32-bit applications, size_t will typically be a 32 bit unsigned integer, and for a 64-bit application, size_t will typically be a 64-bit unsigned integer."

Largest unsigned integer on 32-bit platform is 64 bits(long long), but the size of size_t is 32 bits?



Alex
August 25, 2018 at 8:47 am · Reply

Good catch. I've updated the lesson to remove the reference to the largest unsigned integer your architecture supports, as the introduction of long long makes this untrue on many 32-bit systems.

nascardriver
August 4, 2018 at 6:12 am · Reply

Hi Aakash!

16 bytes would be huge, it should be 16 bits (or 2 bytes). What worries me is the 16, I can't find any source stating this number.

Alex
August 7, 2018 at 12:13 pm · Reply

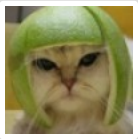
My understanding is that the 16 bit minimum was defined in the C definition and inherited by C++.

Aakash
August 3, 2018 at 11:14 pm · Reply

"Much like an integer can vary in size depending on the system, size_t also varies in size. size_t is guaranteed to be unsigned and at least 16 bites,"

at end, it must be "bytes" resulting in

"Much like an integer can vary in size depending on the system, size_t also varies in size. size_t is guaranteed to be unsigned and at least 16 bytes,"



Alex
August 7, 2018 at 9:47 am · Reply

Typo fixed. This is why I am not a professional editor. :) Thanks!

« Older Comments 1 2 3