4.5 — Unsigned integers, and why to avoid them

BY ALEX ON APRIL 23RD, 2019 | LAST MODIFIED BY NASCARDRIVER ON DECEMBER 22ND, 2019

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:

```
1 unsigned short us;
2 unsigned int ui;
3 unsigned long ul;
4 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

Trick question: What happens if we try to store the number 280 (which requires 9 bits to represent) in a 1-byte unsigned integer? You might think the answer is "overflow!". But, it's not.

By definition, unsigned integers cannot overflow. Instead, if a 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"). 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:

```
#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 1-byte integer, so that's fine. -1 is not representable, so it wraps around to the top of the range, producing the value 255. -2 wraps around to 254. And so forth.

```
#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
```

Author's note

In common language, unsigned integer wrap around is sometimes incorrectly called “overflow” since the cause is identical to signed integer overflow.

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 a bug that leaves the user with not enough bonus time to complete the level. In the PC game Civilization, Gandhi was known for being the first one to use nuclear weapons, which seems contrary to his normally passive nature. Gandhi’s aggression setting was normally set at 1, but if he chose a democratic government, he’d get a -2 modifier. This wrapped around his aggression setting to 255, making him maximally aggressive!

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.

```
#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. A common unwanted wrap-around happens when an unsigned integer is repeatedly decremented with 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 the above example, even if one of the operands (x or y) is signed, the other operand (the unsigned one) will cause the signed one to be promoted to an unsigned integer, and the same behavior will result!

Consider the following snippet:

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

```
}

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*. 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.

If you need to protect a function against negative inputs, use an assertion or exception instead. Both are covered later.

Some modern programming languages (such as Java) and frameworks (such as .NET) either don't include unsigned types, or limit their use.

New programmers often use unsigned integers to represent non-negative data, or to take advantage of the additional range. Bjarne Stroustrup, the designer of C++, said, "Using an unsigned instead of an int to gain one more bit to represent positive integers is almost never a good idea".

Warning

Avoid using unsigned numbers, except in specific cases or when unavoidable.

Don't avoid negative numbers by using unsigned types. If you need a larger range than a signed number offers, use one of the guaranteed-width integers shown in the next lesson (**4.6 -- Fixed-width integers and size_t**).

If you do use unsigned numbers, avoid mixing signed and unsigned numbers where possible.

So where is it reasonable to use unsigned numbers?

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

First, unsigned numbers are preferred when dealing with bit manipulation (covered in chapter O).

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.

4.6 -- Fixed-width integers and size_t

Index

4.4 -- Signed integers

C++ TUTORIAL | PRINT THIS POST

64 comments to 4.5 — Unsigned integers, and why to avoid them

Air Paul
February 5, 2020 at 9:15 pm · Reply

-1 is not representable, so it wraps around to the top of the range, producing the value 255. Please explain how we get 255!!!

nascardriver
February 7, 2020 at 8:40 am · Reply

-1 is 1111 1111, that's how two's complement works (See lesson about binary numbers). But we have an unsigned number, so 1111 1111 is 255.

Air Paul
February 7, 2020 at 9:21 am · Reply

Can you explain how we got 255, while representing -1, through modulo wrapping?

nascardriver
February 8, 2020 at 1:44 am · Reply

When you use modulus, you go in circles, like an analog clock. After the maximum, you go back to the start. If you turn the clock counter-clockwise, you go back to the maximum.

Air Paul
February 8, 2020 at 9:18 am · Reply

I got the point!

Bruno
January 29, 2020 at 1:06 pm · Reply

So 280/256=1,09375. Remainder 09375 -> 0+9+3+7+5=24.
Is it always like that?
Why is it like that?
Why does it get 1 greater of the type size and divides it?
Why does it add the remainder?

Can i ask more fucking questions XD?(Reminded me of Dr. Ken Jeong on Wired :p)
Sorry for the all the question. Specially the stupid math one.

nascardriver
January 30, 2020 at 2:05 am · Reply

No, it's not always like that, you picked some lucky numbers. Change something and it doesn't work anymore.

Bruno
January 30, 2020 at 12:18 pm · Reply

So there's no logic to how the numbers wrap? It's just random?

nascardriver
January 31, 2020 at 12:22 am · Reply

It's well defined, but your division and summation example was a lucky pick. Modulus is covered in lesson 5.3.

Bruno
February 1, 2020 at 4:29 pm · Reply

Oh! ok. Ty my good Sir! :)
Btw, those numbers were in this lesson. That's why i got confused.

Tjark
February 3, 2020 at 12:29 am · Reply

It has to do with the binary representation. 280 in binary is 100011000. This number has 9 bits, but we only have 8 bits to store the number. So only the last 8 bits are taken which is 00011000. This is 24 in decimal. Stripping every bit before the last 8 bits is the same as a 256 modulus operation.

Bruno
February 3, 2020 at 11:51 am · Reply

Nice! I almost got everything :). Ty so much for all the explanations! I gotta get to modulus as i don't fully understand it yet. All in its time.

HZ
December 21, 2019 at 1:51 pm · Reply

An editorial suggestion: "In the above example, even if one of the operands (x or y) is signed, the other operand (the unsigned one) will cause the signed one to be promoted to an unsigned integer, and the same behavior will result!"

nascardriver
December 22, 2019 at 2:57 am · Reply

Thanks, lesson updated!

OLGD
December 8, 2019 at 12:52 pm · Reply

Phenomenal example @Gandhi

Benjamin
November 27, 2019 at 7:14 am · Reply

I actually wound up here as a fairly novice and self-taught programmer, looking for information for bit field restrictions on signed integers. Although this topic does not cover that, I read through anyway, all the way down to the last published comment, because I almost exclusively use unsigned variable types, char being the only exception. I do program professionally in very memory constrained "embedded" environments. My problem being self-taught is the assumptions most tutorials make on their audience. "Modern" vs historic computing is no less or more memory constrained. Modern computing is more diverse, and there are systems with terabytes of RAM and systems with bytes. Some programmers learn on a memory constrained system and some learn on a memory abundant system. I genuinely appreciate your tutorial and your time in writing it. Although your intent is to evangelize avoiding their use, you do not clearly show how a signed variable will be a solution for the pitfalls you point out related to unsigned variables. Most of your support for your argument is quoting industry professionals. The value in your article is for those who use unsigned types to better understand their behavior.

nascardriver
November 27, 2019 at 7:30 am · Reply

Hi Benjamin,

thanks for your feedback! I totally agree with what you said. Arduino and SBCs come to my mind right away, both systems are often used by beginners and can have very limited resources.
I've marked this lesson to be updated to illustrate use cases for unsigned integers and to show solutions to unsigned problems.

If you're looking for something specific, like restrictions of signed bit fields, cppreference should be your first stop.
https://en.cppreference.com/w/cpp/language/bit_field#Notes

Benjamin
November 27, 2019 at 9:43 am · Reply

Thank you for pointing me to the notes. The ambiguity of behavior of signed types in bit fields is another use case for unsigned types. Here is my example which is for a local time offset from UTC... I was thinking of using a signed int for hour since it can be -12 to +14, but given the unclear behavior, I will continue with the way I have it. Separating the sign from the rest of the type also allows me to apply the sign across all of the values. A sledgehammer is not the appropriate tool to apply a finishing nail to cornice molding and likewise, the computer system is often sized appropriate for the task.

```
1 struct localOffset {
2     unsigned int sign : 1;// 0-1
3     unsigned int hour : 4;// 0-15 use html forms validation to restrict to -12 to
4 +14
```

```
    unsigned int minute : 6;// 0-63 use html forms validation to restrict to 0-59
    unsigned int driftSign : 1;
    unsigned int second : 4;// 0-15 use html forms validation to restrict to -15 to +15 - allows a little drift in seconds
};
```

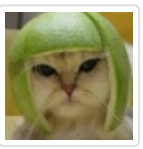
Indigoandblack
November 15, 2019 at 10:41 am · Reply

"Many modern programming languages (such as Java and C#) either don't include unsigned types, or limit their use."

Incorrect - C# most certainly includes unsigned types.

I'm an embedded software engineer. Unsigned types are indispensable in this general "field", and I assuredly use unsigned types to avoid the unnecessary annoyance of having to test sign in a method or function taking a value that will never and cannot be negative.

Please consider the breadth of modern software development before issuing myopic umbrella opinions.



Alex
November 15, 2019 at 4:02 pm · Reply

I said C#, but I meant .NET. Lesson corrected. Thanks for pointing out the error.

The opinion isn't myopic -- it's actually fantastically well supported by literature and expert opinion as a *general programming* best practice (I posted a video link in the comment below -- here's another one: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es-expressions-and-statements -- see ES.100 through ES.107)

There will always be cases (of which embedded programming is often mentioned as a significant one) for which a given recommendation might not be the best option. If you work in these areas, use your judgement, and find some good resources specific to your area that you can layer on top of the general recommendations.

Alain
November 14, 2019 at 1:10 pm · Reply

Sorry, but your insistence on not using unsigned integers is pure FUD, and does not make any sense, because :

- all your examples apply to signed integers as well; what you complain about is the fact that typical integer types have finite size, nothing else. But that's a fact of life, your code should avoid or handle this. Period. With signed integers you cannot handle over/underflow.

- if you ever run into over/underflow using signed integers, what you get is undefined behavior, which is *much* worse than mere wrap-around. Especially because the compiler may have "optimized" your code for such cases, with surprising effects to you -- even though there is nothing to be surprised about once you've entered undefined behavior.

On the last point, see https://en.wikipedia.org/wiki/Undefined_behavior and the blog posts by Chris Lattner (Ref #1 on wikipedia last time I checked).

nascardriver
November 15, 2019 at 3:19 am · Reply

> all your examples apply to signed integers as well
The examples show undesired wrap arounds when going negative, which doesn't happen with signed integers.

> what you complain about is the fact that typical integer types have finite size
> if you ever run into over/underflow using signed integers, what you get is undefined behavior
Size doesn't matter here. If you need a large integer, using a fixed/least width unsigned integer with wrap around handling is fine. Most of the time we don't use an int to its max and using an unsigned int instead would introduce unnecessary causes of issues.

Alex
November 15, 2019 at 3:33 pm · Reply

I'll just leave this here: https://www.youtube.com/watch?v=Puio5dly9N8 (start at 42:40)

flow
January 13, 2020 at 1:11 am · Reply

hi alex! in case you didnt know already, you can rightclick on a youtube video to get a link with a timestamp or simply add &t=numberofseconds yourself

https://www.youtube.com/watch?v=Puio5dly9N8&t=2560

amazing site btw, am really glad i found it

BooGDaaN
November 8, 2019 at 7:33 am · Reply

Can you update this lesson using Uniform Initialization?
In this way, we can get used to this recommended type of initialization.

nascardriver
November 9, 2019 at 5:02 am · Reply

Done, thanks for pointing it out!
There are still several lessons that don't use brace initialization or break rules that were introduced before. Feel free to point them out when you see one and I'll make sure it gets updated.

BooGDaaN
November 10, 2019 at 12:27 pm · Reply

Sure! Here can also be updated std::endl with '\n' from the second code paragraph.

Dennis
October 31, 2019 at 9:30 am · Reply

I'm confused, what exactly is the difference between integer overflow and modulo wrapping?

nascardriver
November 1, 2019 at 3:39 am · Reply

For unsigned integers and since C++20 also signed integers, nothing.
Before that, an overflowing signed integer caused undefined behavior.
When the lessons say "underflow" or "overflow" in an unsigned context, it's the same as a wrap.

Alex
November 1, 2019 at 10:56 pm · Reply

Unfortunately, signed integer overflow remains undefined behavior, even in C++20 (unless this was changed at the last minute). Citations: https://en.wikipedia.org/wiki/C%2B%2B20 and https://news.ycombinator.com/item?id=17190864

Unsigned to signed conversions do modulo wrap in C++20 though.

nascardriver
November 2, 2019 at 3:23 am · Reply

Thanks for double checking Alex, I must've mixed it up. I checked in the latest working draft, and you're right, signed integers can still overflow into UB.

Brandon
September 11, 2019 at 2:34 pm · Reply

Shouldn't it be "An n-[byte] unsigned variable has a range of 0 to (2n)-1"?

Alex
September 16, 2019 at 1:01 pm · Reply

Nope, it's correct as written. The n is measured in bits.

Jose
August 2, 2019 at 11:07 am · Reply

If unsigned integers give so much problems why do they even exist in the first place?

Alex
August 3, 2019 at 9:53 pm · Reply

I presume for two reasons:
1) Back when the language was created, computers had very little memory, so saving memory counted for a lot more than it does today.
2) Computer science wasn't as mature a field back when the language was created, so they didn't have 50+ years of mistakes and best practices to make well informed decisions based off of.

Edwin Martens
August 22, 2019 at 2:00 am · Reply

Unsigned types are actually VERRY usefull !
Imagine a program that needs a lot of boolean variables, each taking up a complete byte (at least)

you're MUCH better of storing those bools in one or more unsigned ints and bitmask them in and out ! this way you can store 32 booleans in just ONE unsigned int !

It would be a great loss to remove this

p.s. It could also be the case that your program controlls a piece of hardware where you literaly are sending 0's and 1's to some device.
a negative number would be complete nonsense in this case.

as for videogames, that is where wraparround can come in verry handy. Using unsigned char for an angle gives you limitless rotation in both directions, without any additional code !

scott pelger
October 2, 2019 at 5:48 pm · Reply

i agree that unsigned ints are very useful but i don't think your examples fit the bill. you could easily use the bits in a signed integer to represents the bools in your example. a bit is a bit is a bit no matter if it is used to represent a signed or an unsigned.
also, regarding the wraparound to represent degrees...i get the idea but there are 360 of them in a full rotation, not 255. but i suppose if you considered 65535 and scaled that to 360 then you have something there.
the real use for unsigned integers is in the embedded arena. every register is an unsigned int as are addresses and various other things.

Kostiantyn Cherkas
July 24, 2019 at 11:25 pm · Reply

First of all - You are doing a beautiful job, thank you! I'm not sure about this - "Don’t avoid negative numbers by using unsigned types", may I ask you to give some clarifications.

nascardriver
July 25, 2019 at 8:26 am · Reply

Say you want to store a person's age. It can't be negative, so you might be tempted to use and unsigned integer. Unsigned integers are prone to trouble.
I can't think of an example without using loops. I suppose Alex shows an example in some lesson once loops have been covered. If he doesn't leave another comment.

Alex
July 25, 2019 at 2:17 pm · Reply

See lesson 5.7, quiz question 3.

daniel jo
June 15, 2019 at 10:52 am · Reply

To prevent a overflow you could use the brackets for example int name{ value }. Corrrect?

nascardriver
June 15, 2019 at 11:03 am · Reply

Brace initialization prevents narrowing casts (Loss of precision), but can't prevent overflows.

daniel jo
June 15, 2019 at 11:42 am · Reply

oh..... thanks for responding

Hugo
June 7, 2019 at 10:34 pm · Reply

My question is regarding this. it is mentioned above that "The signed argument of -1 gets implicitly converted to an unsigned parameter". shouldn't that be 1 then

nascardriver
June 8, 2019 at 12:03 am · Reply

No. The bits stay the same, only the interpretation changes, causing an underflow to probably `4294967295`.
This will make more sense after your read lesson O.3.7 about binary-decimal conversion.

ata021
June 4, 2019 at 5:35 am · Reply

```
1  int main()
2  {
3      uint32_t x = 1;
4      int32_t  y = -1;
5      if (y<x) { cout << y << " < " << x << "\n";  }
6      else     { cout << y << " >= " << x << "\n"; }
7
8      return 0;
9  }
```

result: -1 why i have this result?
by changing datatyp to
  uint16_t x = 1;
  int16_t y = -1;
it will work

nascardriver
June 4, 2019 at 8:31 am · Reply

Signed types get converted to unsigned when compared to unsigned. You should've gotten a compiler warning for your code.

ata021
June 5, 2019 at 4:59 am · Reply

i understad, the compiler will make Implicit integer type conversion, but why it works for 16 bit and it shows -1<+1 and not for 32 bit? and what about using 8 bit ?
ih should be the same,because in all cases will convert from signed tpye to unsigned ?

nascardriver
June 5, 2019 at 5:33 am · Reply

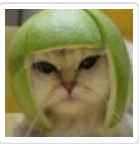
Sorry, I missed that.
Integral types smaller than int will be promoted to an int when used in arithmetic or comparison.
On your system, a signed int can store the maximum number of a uint16, so promotion to signed int is used (If your int was too small, the uint16 would've been promoted to unsigned int).
You're left with a comparison of 2 signed ints.

alfonso
May 25, 2019 at 11:22 pm · Reply

280 is 0000 0001 0001 1000 on 16 bits

But on 8 bit, only last 8 bits are stored, 0001 1000, aka 24

So, still it looks like overflow to me. I want to see that definition that says why it is not overflow an why. Keeping 24 is as a (useless) 'random' value as in other cases of overflow.



Alex
May 29, 2019 at 12:52 pm · Reply

From C11 6.2.5/9: "The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same. A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type."

Keanu
May 22, 2019 at 1:37 pm · Reply

```
#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;
}
```

Trying to compile this code gives me the following errors:
C2220 warning treated as error - no 'object' file generated (line 8)
C4305 '=': truncation from 'int' to 'unsigned short' (line 8)
C4309 '=': truncation of constant value (line 8)

error C4305 and C4309 are the same for line 11, what does this error/warning mean?

nascardriver
May 23, 2019 at 5:16 am · Reply

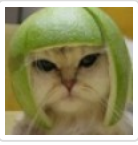
The comments in the code already say what the error means.
65536 and 65537 don't fit into an unsigned short. The largest value you can store in an unsigned short is 65535.
The value of @x after line 8/11 is 0/1, I'm not sure if these values are well defined. Since you wouldn't expect a different value to be assigned than you wrote, you get a warning. You're compiler is treating warnings as errors, so you get an error.

Keanu
May 23, 2019 at 8:38 am · Reply

Clear as day, thanks for clarifying.

Dude
May 15, 2019 at 11:57 pm · Reply

Not sure if it's a small typo in the comment under Unsigned integer overflow line 11. Should it be 65537 instead?



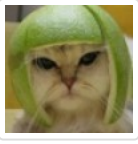
Alex
May 17, 2019 at 8:52 pm · Reply

Yup. Thanks for pointing that out!

noname
May 3, 2019 at 9:08 am · Reply

You have a syntactic error in your first program. Line 6.
std::cout << "x was: " << x << '\n'

You need a semicolon there.



Alex
May 6, 2019 at 1:47 pm · Reply

Correct indeed. Fixed!

Louis Cloete
April 26, 2019 at 1:54 pm · Reply

In paragraph 2: C++ also *supposed* unsigned ... should be supports?

Last paragraph before red box: Unfortunately, *do* to ... should be due to.

Alex
April 30, 2019 at 4:55 pm · Reply

Thanks! Clearly my proofreading skills leave something to be desired. :)

Louis Cloete
May 1, 2019 at 12:29 pm · Reply

It is a pleasure. At least you are one of the few people who are still worrying about correct spelling and grammar. That's much appreciated by me, because I hate to have to decipher incorrect spelling and grammar and then still not be sure if I guessed the correct possibility. I know you won't mind if I point out errors, so I do that with pleasure. ;-)

Alex
May 6, 2019 at 12:58 pm · Reply

I really do appreciate it. It helps make the site better for everyone.

Merlin
April 26, 2019 at 3:06 am · Reply

```
1  #include <iostream>
2  #include <cstdint>
3
4  int main()
5  {
6      std::cout << "Please enter a number: \n";
7      std::int8_t myInt{0};
8      std::cin >> myInt;
9
10     std::cout << "Is " << myInt << " an integer?";
11
12
13     return 0;
14 }
```

If I type in "65" it outputs a "6" instead of an "A". Is this undefined behaviour or is there something behind? This did only occur when the user inputs a number.

nascardriver
April 26, 2019 at 3:21 am · Reply

@std::cin and @std::cout treat @std::int8_t as a char. Your code is no different than if @myInt was a char.