4.6 — Fixed-width integers and size_t

BY ALEX ON NOVEMBER 25TH, 2011 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 2ND, 2020

In the previous lessons on integers, we covered that C++ only guarantees that integer variables will have a minimum size -- but they could be larger, depending on the target system.

Why isn't the size of the integer variables fixed?

The short answer is that this goes back to C, when computers were slow and performance was of the utmost concern. C opted to intentionally leave the size of an integer open so that the compiler implementors could pick a size for int that performs best on the target computer architecture.

Doesn't this suck?

By modern standards, yes. As a programmer, it's a little ridiculous to have to deal with types that have uncertain ranges. A program that uses more than the minimum guaranteed ranges might work on one architecture but not on another.

Fixed-width integers

To help with cross-platform portability, C99 defined a set of **fixed-width integers** (in the stdint.h header) that are guaranteed to have the same size on any architecture.

These are defined as follows:

Name	Type	Range	Notes
std::int8_t	1 byte signed	-128 to 127	Treated like a signed char on many systems. See note below.
std::uint8_t	1 byte unsigned	0 to 255	Treated like an unsigned char on many systems. See note below.
std::int16_t	2 byte signed	-32,768 to 32,767	
std::uint16_t	2 byte unsigned	0 to 65,535	
std::int32_t	4 byte signed	-2,147,483,648 to 2,147,483,647	
std::uint32_t	4 byte unsigned	0 to 4,294,967,295	
std::int64_t	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
std::uint64_t	8 byte unsigned	0 to 18,446,744,073,709,551,615	

C++ officially adopted these fixed-width integers as part of C++11. They can be accessed by including the *cstdint* header, where they are defined inside the *std* namespace. Here's an example:

```
#include <iostream>
```

```
#include <cstdint>

int main()
{
    std::int16_t i(5); // direct initialization
    std::cout << i;
    return 0;
}
```

The fixed-width integers have two downsides: First, they may not be supported on architectures where those types can't be represented. They may also be less performant than the built-in types on some architectures.

Warning

The above fixed-width integers should be avoided, as they may not be defined on all target architectures.

Fast and least integers

To help address the above downsides, C++11 also defines two alternative sets of integers.

The fast type (std::int_fast#_t) provides the fastest signed integer type with a width of at least # bits (where # = 8, 16, 32, or 64). For example, *std::int_fast32_t* will give you the fastest signed integer type that's at least 32 bits.

The least type (std::int_least#_t) provides the smallest signed integer type with a width of at least # bits (where # = 8, 16, 32, or 64). For example, *std::int_least32_t* will give you the smallest signed integer type that's at least 32 bits.

Here's an example from the author's Visual Studio (32-bit console application):

```
#include <iostream>
#include <cstdint>

int main()
{
    std::cout << "fast 8: " << sizeof(std::int_fast8_t) * 8 << " bits\n";
    std::cout << "fast 16: " << sizeof(std::int_fast16_t) * 8 << " bits\n";
    std::cout << "fast 32: " << sizeof(std::int_fast32_t) * 8 << " bits\n";

    std::cout << "least 8: " << sizeof(std::int_least8_t) * 8 << " bits\n";
    std::cout << "least 16: " << sizeof(std::int_least16_t) * 8 << " bits\n";
    std::cout << "least 32: " << sizeof(std::int_least32_t) * 8 << " bits\n";

    return 0;
}
```

This produced the result:

```
fast 8: 8 bits
fast 16: 32 bits
fast 32: 32 bits
least 8: 8 bits
least 16: 16 bits
least 32: 32 bits
```

You can see that std::int_fast16_t was 32 bits, whereas std::int_least16_t was 16 bits.

There is also an unsigned set of fast and least types (std::uint_fast#_t and std::uint_least#_t).

These fast and least types are guaranteed to be defined, and are safe to use.

Best practice

Favor the std::int_fast#_t and std::int_least#_t integers when you need an integer guaranteed to be at least a certain minimum size.

Warning: std::int8_t and std::uint8_t may behave like chars instead of integers

Note: We talk more about chars in lesson (**4.11 -- Chars**).

Due to an oversight in the C++ specification, most compilers define and treat *std::int8_t* and *std::uint8_t* (and the corresponding fast and least fixed-width types) identically to types *signed char* and *unsigned char* respectively. Consequently, *std::cin* and *std::cout* may work differently than you're expecting. Here's a sample program showing this:

```
#include <cstdint>
#include <iostream>

int main()
{
    std::int8_t myint = 65;
    std::cout << myint;

    return 0;
}
```

On most systems, this program will print 'A' (treating *myint* as a char). However, on some systems, this may print *65* as expected.

For simplicity, it's best to avoid *std::int8_t* and *std::uint8_t* (and the related fast and least types) altogether (use *std::int16_t* or *std::uint16_t* instead). However, if you do use *std::int8_t* or *std::uint8_t*, you should be careful of anything that would interpret *std::int8_t* or *std::uint8_t* as a char instead of an integer (this includes *std::cout* and *std::cin*).

Hopefully this will be clarified by a future draft of C++.

Warning

Avoid the 8-bit fixed-width integer types. If you do use them, note that they are often treated like chars.

Integer best practices

Now that fixed-width integers have been added to C++, the best practice for integers in C++ is as follows:

- *int* should be preferred when the size of the integer doesn't matter (e.g. the number will always fit within the range of a 2 byte signed integer). For example, if you're asking the user to enter their age, or counting from 1 to 10, it doesn't matter whether int is 16 or 32 bits (the numbers will fit either way). This will cover the vast majority of the cases you're likely to run across.
- If you need a variable guaranteed to be a particular size and want to favor performance, use std::int_fast#_t.

- If you need a variable guaranteed to be a particular size and want to favor memory conservation over performance, use std::int_least#_t. This is used most often when allocating lots of variables.

Avoid the following if possible:

- Unsigned types, unless you have a compelling reason.
- The 8-bit fixed-width integer types.
- Any compiler-specific fixed-width integers -- for example, Visual Studio defines __int8, __int16, etc...

What is std::size_t?

Consider the following code:

```
#include <iostream>

int main()
{
    std::cout << sizeof(int) << '\n';

    return 0;
}
```

On the author's machine, this prints:

4

Pretty simple, right? We can infer that operator sizeof returns an integer value -- but what integer type 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 *std::size_t*. **std::size_t** is defined an unsigned integral type, and it is typically used to represent the size or length of objects.

Amusingly, we can use the *sizeof* operator (which returns a value of type *std::size_t*) to ask for the size of *std::size_t* itself:

```
#include <cstddef> // std::size_t
#include <iostream>

int main()
{
    std::cout << sizeof(std::size_t) << '\n';

    return 0;
}
```

Compiled as a 32-bit (4 byte) console app on the author's system, this prints:

4

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 bits, but on most systems will be equivalent to the address-width of the application. That is, 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. *size_t* is defined to be big enough to hold the size of the largest object creatable on your system (in bytes). For example, if *size_t* is 4 bytes, the largest object creatable on your system can't be larger than the largest number representable by a 4-byte unsigned integer (per the table above, *4,294,967,295* bytes).

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 wrapping around.

4.7 -- Introduction to scientific notation

Index

4.5 -- Unsigned integers, and why to avoid them

C++ TUTORIAL | PRINT THIS POST

183 comments to 4.6 — Fixed-width integers and size_t

« Older Comments 1 2 3

salah
February 8, 2020 at 4:02 am · Reply

As mentioned in the lesson: The fast type (std::int_fast#_t) provides the fastest signed integer type with a width of at least # bits (where # = 8, 16, 32, or 64). For example, std::int_fast32_t will give you the fastest signed integer type that's at least 32 bits.

Now let's assume the fastest integer 64 bits.In this case
std::cout << sizeof(std::int_fast8_t) * 8;
std::cout << sizeof(std::int_fast16_t) * 8 ;
std::cout << sizeof(std::int_fast32_t) * 8 ;
the output must be : 64 64 64 right??, since the fastest one is 64 bits, but the result in my computer was 8 , 16 , 32 , so which one is the fastest one ?

nascardriver

February 8, 2020 at 4:08 am · Reply

All the same, according to your standard libraries implementation.

Lord Voldemort
February 6, 2020 at 11:06 am · Reply

What is the address width of the application? Is it like, the 32-bit architecture will have 32 digits in its address?

nascardriver
February 7, 2020 at 8:39 am · Reply

Yes, 32 binary digits, aka 32 bits.

Lord Voldemort
February 7, 2020 at 9:28 am · Reply

You're not getting my point. suppose if we are considering a computer with 8-bit architecture then the address of all memory locations in this computer will be of exactly 8 digits?

nascardriver
February 8, 2020 at 1:48 am · Reply

Yes, 8 binary digits, aka 8 bits.

0000 1010
1111 0010
1010 0011

Those are addresses in an 8 bit architecture. If you're using another number system, you'll have less digits, eg. in hex an 8 bit architecture has 2 digits.

0x0a
0xf2
0xa3

Same as above, just a different way of writing it.

Lord Voldemort
February 8, 2020 at 9:03 am · Reply

Thanks!!

Shanzeh Hussey
February 6, 2020 at 10:59 am · Reply

Is it possible to create a data type using structs with size more than the range of size_t? If no, then why And why it is a compilation error?

nascardriver
February 7, 2020 at 8:37 am · Reply

Types can't be larger than the maximum value representable by `std::size_t`.

salah
February 8, 2020 at 2:03 am · Reply

Hi nascardriver ,,
suppose we create a struct that holds an integer and float, in this case we have a data type(struct)holds a 4bytes integer and 4bytes float totally 8bytes data type.So how this data type is working if we have the maximum value by `std::size_t` 4byte ??

nascardriver
February 8, 2020 at 3:32 am · Reply

The maximum object size is the maximum _value_, not width, of `std::size_t`. The maximum value of `std::size_t` on a 64 bit system is 2^64 (Power, not xor). That's more than 16 million TiB.

salah
February 8, 2020 at 3:44 am · Reply

Thank you a lot

Scarlet Johnson
February 6, 2020 at 8:39 am · Reply

Hey, it is mentioned in this tutorial that "The least type (std::int_least#_t) provides the smallest signed integer type with a width of at least # bits". What is mean by the smallest signed integer here? I've searched a lot on Google but I didn't any information beyond this! And how it is different from int#_t since it also uses at least # bits too.

nascardriver
February 6, 2020 at 9:27 am · Reply

int#_t is exactly # bits wide. int_fast#_t and int_least#_t are at least # bits wide, but can be wider.
int_fast#_t uses the fastest type available.
int_least#_t uses the narrowest type available.

example
int16_t is always a 16 bit integer, but this type might not exist.
int_fast16_t might turn into a 64 bit integer, because that's the fastest on your system.
int_least16_t might turn into a 32 bit integer, because your system doesn't have a 16 bit integer, but 32 is less than 64.

Scarlet Johnson
February 6, 2020 at 11:11 am · Reply

Thanks!
your explanations are helpful a lot.

Bruno
February 1, 2020 at 5:14 pm · Reply

Suuuper small typo BUT i'll point it out anyway :D.

On the first waning you missed a ' - '

"Warning

The above fixed (you missed me Alex) width integers should..."

Why not make such a good tutorial more perfect. :p

nascardriver
February 2, 2020 at 1:15 am · Reply

Lesson updated, thanks!

« Older Comments 1 2 3