

## 4.6 — Fixed-width integers and size\_t

---

 [learncpp.com/cpp-tutorial/fixed-width-integers-and-size-t/](http://learncpp.com/cpp-tutorial/fixed-width-integers-and-size-t/)

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 the early days of 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 implementers 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.

Consider the `int` type. The minimum size for `int` is 2 bytes, but it's often 4 bytes on modern architectures.

If you assume an `int` is 4 bytes because that's most likely, then your program will probably misbehave on architectures where `int` is actually 2 bytes (since you will probably be storing values that require 4 bytes in a 2 byte variable, which will cause overflow or undefined behavior).

If you assume an `int` is only 2 bytes to ensure maximum compatibility, then the range of values you can safely store in an `int` is significantly limited. And on systems where `int` is actually 4 bytes, you're not making use of half of the memory allocated per `int`.

As an aside...

In most cases, we only instantiate a small number of `int` variables at a time, and these are typically destroyed at the end of the function in which they are created. In such cases, wasting 2 bytes of memory per variable isn't a concern (the limited range is a bigger issue). However, in cases where our program allocates thousands or millions of `int` variables, wasting 2 bytes of memory per variable can have a significant impact on the program's overall memory usage.

Fixed-width integers

To address the above issues, C99 defined a set of **fixed-width integers** (in the `stdint.h` header) that are guaranteed to be the same size on any architecture.

These are defined as follows:

Name	Type	Range	Notes
<code>std::int8_t</code>	1 byte signed	-128 to 127	Treated like a signed char on many systems. See note below.
<code>std::uint8_t</code>	1 byte unsigned	0 to 255	Treated like an unsigned char on many systems. See note below.
<code>std::int16_t</code>	2 byte signed	-32,768 to 32,767	
<code>std::uint16_t</code>	2 byte unsigned	0 to 65,535	
<code>std::int32_t</code>	4 byte signed	-2,147,483,648 to 2,147,483,647	
<code>std::uint32_t</code>	4 byte unsigned	0 to 4,294,967,295	
<code>std::int64_t</code>	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
<code>std::uint64_t</code>	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 <cstdint> // for fixed-width integers
#include <iostream>

int main()
{
    std::int16_t i{5};
    std::cout << i << '\n';
    return 0;
}
```

The fixed-width integers have two downsides that are typically raised.

First, the fixed-width integers are not guaranteed to be defined on all architectures. They only exist on systems where there are fundamental types matching their widths and following a certain binary representation. Your program will fail to compile on any such architecture that does not support a fixed-width integer that your program is using. However, given that most modern architectures have standardized around 8/16/32/64-bit variables, this is unlikely to be a problem unless your program needs to be portable to some exotic mainframe or embedded architectures.

Second, if you use a fixed-width integer, it may be slower than a wider type on some architectures. For example, if you need an integer that is guaranteed to be 32-bits, you might decide to use `std::int32_t`, but your CPU might actually be faster at processing 64-bit integers. However, just because your CPU can process a given type faster doesn't mean your program will be faster overall -- modern programs are often constrained by memory usage rather than CPU, and the larger memory footprint may slow your program more than the faster CPU processing accelerates it. It's hard to know without actually measuring.

### Fast and least integers

To help address the above downsides, C++ also defines two alternative sets of integers that are guaranteed to be defined.

The fast types (`std::int_fast#_t` and `std::uint_fast#_t`) provide the fastest signed/unsigned 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. By fastest, we mean the integral type that can be processed most quickly by the CPU.

The least types (`std::int_least#_t` and `std::uint_least#_t`) provide the smallest signed/unsigned integer type with a width of at least # bits (where # = 8, 16, 32, or 64). For example, `std::uint_least32_t` will give you the smallest unsigned integer type that's at least 32 bits.

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

```

#include <cstdint> // for fast and least types
#include <iostream>

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

    return 0;
}

```

This produced the result:

```

least 8:  8 bits
least 16: 16 bits
least 32: 32 bits

fast 8:  8 bits
fast 16: 32 bits
fast 32: 32 bits

```

You can see that `std::int_least16_t` is 16 bits, whereas `std::int_fast16_t` is actually 32 bits. This is because on the author's machine, 32-bit integers are faster to process than 16-bit integers.

However, these fast and least integers have their own downsides: First, not many programmers actually use them, and a lack of familiarity can lead to errors. Second, the fast types can lead to memory wastage, as their actual size may be larger than indicated by their name. More seriously, because the size of the fast/least integers can vary, it's possible that your program may exhibit different behaviors on architectures where they resolve to different sizes. For example:

```

#include <cstdint>
#include <iostream>

int main()
{
    std::uint_fast16_t sometype { 0 };
    sometype = sometype - 1; // intentionally overflow to invoke wraparound behavior

    std::cout << sometype << '\n';

    return 0;
}

```

This code will produce different results depending on whether `std::uint_fast16_t` is 16, 32, or 64 bits.

It's hard to know where your program might not function as expected until you've rigorously tested your program on such architectures.

`std::int8_t` and `std::uint8_t` likely behave like chars instead of integers

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. This means these 8-bit types may (or may not) behave differently than the rest of the fixed-width types, which can lead to errors. This behavior is system-dependent, so a program that behaves correctly on one architecture may not compile or behave correctly on another architecture.

We show an example of this in lesson [4.12 -- Introduction to type conversion and static\\_cast](#).

When storing integral values where a specific size is important, it's generally best to avoid `std::int8_t` and `std::uint8_t` (and the related fast and least types) and use `std::int16_t` or `std::uint16_t` instead.

### Warning

The 8-bit fixed-width integer types are often treated like chars instead of integer values (and this may vary per system). Prefer the 16-bit fixed integral types for most cases.

### Best practices for integral types

Given the various pros and cons of the fundamental integral types, the fixed-width integral types, the fast/least integral types, and signed/unsigned challenges, there is little consensus on integral best practices.

Our stance is that it's better to be correct than fast, and better to fail at compile time than runtime. Therefore, if you need an integral type with a fixed size, we recommend avoiding the fast/least types in favor of the fixed-width types. If you later discover the need to support a platform for which a specific fixed-width type won't compile, then you can decide how to migrate your program (and thoroughly retest) at that point.

### Best practice

- Prefer `int` 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) and the variable is short-lived (e.g. destroyed at the end of the function). 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.
- Prefer `std::int#_t` when storing a quantity that needs a guaranteed range.
- Prefer `std::uint#_t` when doing bit manipulation or where well-defined wrap-around behavior is required.

Avoid the following when possible:

- `short` and `long` integers -- use a fixed-width type instead.
- Unsigned types for holding quantities.
- The 8-bit fixed-width integer types.
- The fast and least fixed-width 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 integral type is that return 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 as an unsigned implementation-defined 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 <cstdint> // for 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

`std::size_t` is defined in a number of different headers. `<cstdint>` is the best one to include, as it contains the least number of other defined identifiers.

Much like an integer can vary in size depending on the system, `std::size_t` also varies in size. `std::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, `std::size_t` will typically be a 32-bit unsigned integer, and for a 64-bit application, `std::size_t` will typically be a 64-bit unsigned integer.

`sizeof` must be able to return the size of an object (in bytes) using a value of type `std::size_t`. Therefore, the largest object creatable on a system has a size (in bytes) equal to the largest value `std::size_t` can hold. If it were possible to create a larger object, `sizeof` would not be able to return its size, as it would overflow the range of `std::size_t`.

Therefore, any object with a size (in bytes) larger than the largest value an object of type `std::size_t` can hold is considered ill-formed (and will cause a compile error).

For example, the range of a 4-byte unsigned integral type is 0 to 4,294,967,295. Assuming `std::size_t` is 4 bytes in size, then an object of type `std::size_t` can hold an integral value from 0 to 4,294,967,295. This means the largest object creatable on such a system is 4,294,967,295 bytes (and calling `sizeof` on that object would return exactly 4,294,967,295).

As an aside...

The size of `std::size_t` imposes a strict mathematical upper limit to an object's size. In practice, the largest creatable object may be smaller than this amount (perhaps significantly so).

Some compilers limit the largest creatable object to half the maximum value of `std::size_t` (an explanation for this can be found [here](#)).

Other factors may also play a role, such as how much contiguous memory your computer has available for allocation.