2.4a — Fixed-width integers and the unsigned controversy

BY ALEX ON NOVEMBER 25TH, 2011 | LAST MODIFIED BY ALEX ON JULY 20TH, 2017

In the previous lesson **2.4 -- Integers** you learned 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, non-technical answer is that this goes back to C, when performance was of 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?

Yes! As a programmer, it's a little ridiculous to have to deal with variables whose size could vary depending on the target architecture.

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
int8_t	1 byte signed	-128 to 127	Treated like a signed char on many systems. See note below.
uint8_t	1 byte unsigned	0 to 255	Treated like an unsigned char on many systems. See note below.
int16_t	2 byte signed	-32,768 to 32,767	
uint16_t	2 byte unsigned	0 to 65,535	
int32_t	4 byte signed	-2,147,483,648 to 2,147,483,647	
uint32_t	4 byte unsigned	0 to 4,294,967,295	
int64_t	8 byte signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
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;
}
```

Even though these weren't adopted in C++ until C++11, because they were part of the C99 standard, some older C++ compilers offer access to these types, typically by including stdint.h. Visual Studio 2005 and 2008 do not include stdint.h, but 2010 does.

If you are using the boost library, boost provides these as part of <boost/cstdint.hpp>.

If your compiler does not include cstdint or stdint.h, the good news is that you can download Paul Hsieh's **pstdint.h cross-platform compatible version of the stdint.h header**. Simply include the pstdint.h file in your project and it will define the fixed width integer types with the proper sizes for your platform.

Warning: int8_t and uint8_t may or may not behave like chars

Due to an oversight in the C++ specification, most compilers define and treat int8_t and uint8_t identically to types signed char and unsigned char respectively, but this is not required. 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()
{
    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 int8_t and uint8_t altogether (use int16_t or uint16_t instead). However, if you do use int8_t or uint8_t, you should be careful of any function that would interpret int8_t or 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++.

Rule: Avoid int8_t and uint8_t. If you do use them, note that they are often treated like chars.

The downsides of fixed-width integers

Fixed-width integers 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.

Fast and least

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

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

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

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. 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 int_fast#_t.
- If you need a variable guaranteed to be a particular size and want to favor memory conservation over performance, use int_least#_t. This is used most often when allocating lots of variables.
- Only use unsigned types if you have a compelling reason.

Some compilers define their own version of fixed width integers -- for example, Visual Studio defines __int8, __int16, etc… You should avoid these like the plague.

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 unexpected behavior can result when you mix signed and unsigned integers.

Consider the following snippet:

```
1  void doSomething(unsigned int x)
2  {
3      // Run some code x times
4  }
5
6  int main()
7  {
8      doSomething(-1);
9  }
```

What happens in this case? -1 gets converted to some large number (probably 4294967295), and your program goes ballistic. But even 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.

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

This doesn't mean you have to avoid unsigned types altogether -- but if you do use them, use them only where they really make sense, and take care not to mix signed and unsigned numbers.

Alex's note: Most of this tutorial was written prior to these fixed-width integers being adopted into C++11, so we may use int in some examples where a fixed-width integer would be more appropriate.


2.5 -- Floating point numbers


Index


2.4 -- Integers

C++ TUTORIAL | PRINT THIS POST

95 comments to 2.4a — Fixed-width integers and the unsigned controversy

« Older Comments 1 2

yugin
July 8, 2018 at 9:30 pm · Reply

What's the difference between a fast type and a least type? Intuitively, I would've thought the smallest possible representation of an integer would also be the fastest since it takes up less memory.

> Alex