4.3 — Object sizes and the sizeof operator

ALEX DECEMBER 16, 2023

Object sizes

As you learned in the lesson 4.1 -- Introduction to fundamental data types (https://www.learncpp.com/cpp-tutorial/introduction-to-fundamental-data-types/), memory on modern machines is typically organized into byte-sized units, with each byte of memory having a unique address. Up to this point, it has been useful to think of memory as a bunch of cubbyholes or mailboxes where we can put and retrieve information, and variables as names for accessing those cubbyholes or mailboxes.

However, this analogy is not quite correct in one regard -- most objects actually take up more than 1 byte of memory. A single object may use 1, 2, 4, 8, or even more consecutive memory addresses. The amount of memory that an object uses is based on its data type.

Because we typically access memory through variable names (and not directly via memory addresses), the compiler is able to hide the details of how many bytes a given object uses from us. When we access some variable x, the compiler knows how many bytes of data to retrieve (based on the type of variable x), and can handle that task for us.

Even so, there are several reasons it is useful to know how much memory an object uses.

First, the more memory an object uses, the more information it can hold.

A single bit can hold 2 possible values, a 0, or a 1:

bit 0
0
1

2 bits can hold 4 possible values:

bit 0	bit 1
0	0
0	1
1	0
1	1

3 bits can hold 8 possible values:

bit 0	bit 1	bit 2
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

To generalize, an object with n bits (where n is an integer) can hold 2^n (2 to the power of n, also commonly written 2^n) unique values. Therefore, with an 8-bit byte, a byte-sized object can hold 2^8 (256) different values. An object that uses 2 bytes can hold 2^16 (65536) different values!

Thus, the size of the object puts a limit on the amount of unique values it can store -- objects that utilize more bytes can store a larger number of unique values. We will explore this further when we talk more about integers.

Second, computers have a finite amount of free memory. Every time we define an object, a small portion of that free memory is used for as long as the object is in existence. Because modern computers have a lot of memory, this impact is usually negligible. However, for programs that need a large amount of objects or data (e.g. a game that is rendering millions of polygons), the difference between using 1 byte and 8 byte objects can be significant.

Key insight

New programmers often focus too much on optimizing their code to use as little memory as possible. In most cases, this makes a negligible difference. Focus on writing maintainable code, and optimize only when and where the benefit will be substantive.

Fundamental data type sizes

The obvious next question is “how much memory do variables of different data types take?”.

Perhaps surprisingly, the C++ standard does not define the exact size (in bits) for any of the fundamental types. A char must be 1 byte, but no assumption is made that a byte is 8 bits. Integral types have a minimum size (in bits), but could be larger.

In this tutorial series, we will take a simplified view, by making some reasonable assumptions that are generally true for modern architectures:

- A byte is 8 bits.
- Memory is byte addressable, so the smallest object is 1 byte.
- Floating point support is IEEE-754 compliant.
- We are on a 32-bit or 64-bit architecture.

Given that, we can state the following:

Category	Type	Minimum Size	Typical Size	Note
Boolean	bool	1 byte	1 byte	
character	char	1 byte	1 byte	always exactly 1 byte
	wchar_t	1 byte	2 or 4 bytes	
	char8_t	1 byte	1 byte	
	char16_t	2 bytes	2 bytes	
	char32_t	4 bytes	4 bytes	
integer	short	2 bytes	2 bytes	
	int	2 bytes	4 bytes	
	long	4 bytes	4 or 8 bytes	
	long long	8 bytes	8 bytes	
floating point	float	4 bytes	4 bytes	
	double	8 bytes	8 bytes	
	long double	8 bytes	8, 12, or 16 bytes	
pointer	std::nullptr_t	4 bytes	4 or 8 bytes	

Tip

For maximum portability, you shouldn't assume that variables are larger than the specified minimum size.

Alternatively, if you want to assume that a type has a certain size (e.g. that an int is at least `4` bytes), you can use `static_assert` to have the compiler fail a build if it is compiled on an architecture where this assumption is not true. We cover how to do this in lesson 9.6 -- Assert and static_assert (https://www.learncpp.com/cpp-tutorial/assert-and-static_assert/#static_assert).

Related content

If you are using C++ on a system that does not comply with our assumptions, or are just curious, you can find more information about what the C++ standard says about the minimum size of various types here (https://en.cppreference.com/w/cpp/language/types).

The sizeof operator

In order to determine the size of data types on a particular machine, C++ provides an operator named sizeof. The **sizeof operator** is a unary operator that takes either a type or a variable, and returns its size in bytes. You can compile and run the following program to find out how large some of your data types are:

```
#include <iomanip> // for std::setw (which sets the width of the subsequent output)
#include <iostream>

int main()
{
    std::cout << std::left; // left justify output
    std::cout << std::setw(16) << "bool:" << sizeof(bool) << " bytes\n";
    std::cout << std::setw(16) << "char:" << sizeof(char) << " bytes\n";
    std::cout << std::setw(16) << "short:" << sizeof(short) << " bytes\n";
    std::cout << std::setw(16) << "int:" << sizeof(int) << " bytes\n";
    std::cout << std::setw(16) << "long:" << sizeof(long) << " bytes\n";
    std::cout << std::setw(16) << "long long:" << sizeof(long long) << " bytes\n";
    std::cout << std::setw(16) << "float:" << sizeof(float) << " bytes\n";
    std::cout << std::setw(16) << "double:" << sizeof(double) << " bytes\n";
    std::cout << std::setw(16) << "long double:" << sizeof(long double) << " bytes\n";

    return 0;
}
```

Here is the output from the author's machine:

```
bool:           1 bytes
char:           1 bytes
short:          2 bytes
int:            4 bytes
long:           4 bytes
long long:      8 bytes
float:          4 bytes
double:         8 bytes
long double:    8 bytes
```

Your results may vary based on compiler, computer architecture, OS, compilation settings (32-bit vs 64-bit), etc...

Trying to use `sizeof` on an incomplete type (such as `void`) will result in a compilation error.

You can also use the sizeof operator on a variable name:

```
#include <iostream>

int main()
{
    int x{};
    std::cout << "x is " << sizeof(x) << " bytes\n";

    return 0;
}
```

```
x is 4 bytes
```

For advanced readers

`sizeof` does not include dynamically allocated memory used by an object. We discuss dynamic memory allocation in a future lesson.

Fundamental data type performance

On modern machines, objects of the fundamental data types are fast, so performance while using or copying these types should generally not be a concern.

As an aside...

You might assume that types that use less memory would be faster than types that use more memory. This is not always true. CPUs are often optimized to process data of a certain size (e.g. 32 bits), and types that match that size may be processed quicker. On such a machine, a 32-bit int could be faster than a 16-bit short or an 8-bit char.

Next lesson
4.4 Signed integers

Back to table of contents

Previous lesson
4.2 Void

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/ are connected to your provided email address.

391 COMMENTS

Newest