

## 4.3 — Object sizes and the sizeof operator

ALEX AUGUST 2, 2021

### Object sizes

As you learned in the lesson [4.1 -- 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 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?”. You may be surprised to find that the size of a given data type is dependent on the compiler and/or the computer architecture!

C++ only guarantees that each fundamental data types will have a minimum size:

Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char	1 byte	Always exactly 1 byte
	wchar_t	1 byte	
	char16_t	2 bytes	
	char32_t	4 bytes	
integer	short	2 bytes	
	int	2 bytes	
	long	4 bytes	
	long long	8 bytes	
floating point	float	4 bytes	
	double	8 bytes	
	long double	8 bytes	

However, the actual size of the variables may be different on your machine (particularly int, which is more often 4 bytes).

### Best practice

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

Objects of fundamental data types are generally extremely fast.

## 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:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
6      std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
7      std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
8      std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes\n";
9      std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes\n";
10     std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
11     std::cout << "int:\t\t\t" << sizeof(int) << " bytes\n";
12     std::cout << "long:\t\t\t" << sizeof(long) << " bytes\n";
13     std::cout << "long long:\t" << sizeof(long long) << " bytes\n";
14     std::cout << "float:\t\t\t" << sizeof(float) << " bytes\n";
15     std::cout << "double:\t\t" << sizeof(double) << " bytes\n";
16     std::cout << "long double:\t" << sizeof(long double) << "
    bytes\n";
17
18     return 0;
19 }
```

Here is the output from the author's x64 machine, using Visual Studio:

bool:	1 bytes
char:	1 bytes
wchar_t:	2 bytes
char16_t:	2 bytes
char32_t:	4 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 if you are using a different type of machine, or a different compiler. Note that you can not use the *sizeof* operator on the *void* type, since it has no size (doing so will cause a compile error).

### For advanced readers

If you're wondering what '\t' is in the above program, it's a special symbol that inserts a tab (in the example, we're using it to align the output columns). We will cover '\t' and other special symbols in [lesson 4.11 -- Chars](#).

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

```
1 #include <iostream>
2
3 int main()
4 {
5     int x{};
6     std::cout << "x is " << sizeof(x) << "
7     bytes\n";
8
9     return 0;
10 }
```

x is 4 bytes

## Fundamental data type performance

On modern machines, objects of the fundamental data types are fast, so performance while using 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... Put C++ code between triple-backticks (markdown style):````Your C++ code`

 Name\*

 Email\*



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

Notify me about replies:



POST COMMENT

DP N N FOUT

Newest ▼

