2.3 — Variable sizes and the sizeof operator

BY ALEX ON JUNE 6TH, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 25TH, 2017

As you learned in the lesson **2.1 -- Fundamental variable definition, initialization, and assignment**, memory on modern machines is typically organized into byte-sized units, with each unit 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 variables actually take up more than 1 byte of memory. Consequently, a single variable may use 2, 4, or even 8 consecutive memory addresses. The amount of memory that a variable uses is based on its data type. Fortunately, because we typically access memory through variable names and not memory addresses, the compiler is largely able to hide the details of working with different sized variables from us.

There are several reasons it is useful to know how much memory a variable takes up.

First, the more memory a variable takes up, the more information it can hold. Because each bit can only hold a 0 or a 1, we say that bit can hold 2 possible values.

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, a variable with n bits can hold 2^n (2 to the power of n, also commonly written 2^n) possible values. With an 8-bit byte, a byte can store 2^8 (256) possible values.

The size of the variable puts a limit on the amount of information it can store -- variables that utilize more bytes can hold a wider range of values. We will address this issue further when we get into the different types of variables.

Second, computers have a finite amount of free memory. Every time we declare a variable, a small portion of that free memory is used for as long as the variable is in existence. Because modern computers have a lot of memory, this often isn’t a problem, especially if only declaring a few variables. However, for programs that need a large amount of variables (eg. 100,000), the difference between using 1 byte and 8 byte variables can be significant.

The size of C++ basic data types

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++ guarantees that the basic data types will have a minimum size:

Category	Type	Minimum Size	Note
boolean	bool	1 byte	
character	char	1 byte	May be signed or unsigned Always exactly 1 byte
	wchar_t	1 byte	
	char16_t	2 bytes	C++11 type
	char32_t	4 bytes	C++11 type
integer	short	2 bytes	
	int	2 bytes	
	long	4 bytes	
	long long	8 bytes	C99/C++11 type
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). 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:

```cpp
#include <iostream>

int main()
{
    std::cout << "bool:\t\t" << sizeof(bool) << " bytes" << std::endl;
    std::cout << "char:\t\t" << sizeof(char) << " bytes" << std::endl;
    std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes" << std::endl;
    std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes" << std::endl; // C++11, may not be supported by your compiler
    std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes" << std::endl; // C++11, may not be supported by your compiler
    std::cout << "short:\t\t" << sizeof(short) << " bytes" << std::endl;
    std::cout << "int:\t\t" << sizeof(int) << " bytes" << std::endl;
    std::cout << "long:\t\t" << sizeof(long) << " bytes" << std::endl;
    std::cout << "long long:\t" << sizeof(long long) << " bytes" << std::endl; // C++11, may not be supported by your compiler
    std::cout << "float:\t\t" << sizeof(float) << " bytes" << std::endl;
    std::cout << "double:\t\t" << sizeof(double) << " bytes" << std::endl;
    std::cout << "long double:\t" << sizeof(long double) << " bytes" << std::endl;
    return 0;
}
```

Here is the output from the author’s x64 machine (in 2015), using Visual Studio 2013:

```
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 take the sizeof the void type, since it has no size (doing so will cause a compile error).

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 when we talk about the char data type.

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

```
    int x;
    std::cout << "x is " << sizeof(x) << " bytes" << std::endl;
```

```
x is 4 bytes
```

We'll discuss the size of different types in the upcoming lessons, as well as a summary table at the end.

2.4 -- Integers

Index

2.2 -- Void

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

155 comments to 2.3 — Variable sizes and the sizeof operator

« Older Comments 1 2

R310
June 28, 2018 at 11:43 pm · Reply

Hello,
Why did you use :\t\t" in that program?

nascardriver
June 29, 2018 at 2:05 am · Reply

Hi R310!

'\t' is the escape character for a tab. Escape characters are covered in lesson 2.7.