4.3 — Object sizes and the sizeof operator

BY ALEX ON JUNE 6TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 8TH, 2020

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	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).

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:

```
#include <iostream>

int main()
{
    std::cout << "bool:\t\t" << sizeof(bool) << " bytes\n";
    std::cout << "char:\t\t" << sizeof(char) << " bytes\n";
    std::cout << "wchar_t:\t" << sizeof(wchar_t) << " bytes\n";
    std::cout << "char16_t:\t" << sizeof(char16_t) << " bytes\n"; // C++11 only
    std::cout << "char32_t:\t" << sizeof(char32_t) << " bytes\n"; // C++11 only
    std::cout << "short:\t\t" << sizeof(short) << " bytes\n";
    std::cout << "int:\t\t" << sizeof(int) << " bytes\n";
    std::cout << "long:\t\t" << sizeof(long) << " bytes\n";
    std::cout << "long long:\t" << sizeof(long long) << " bytes\n"; // C++11 only
    std::cout << "float:\t\t" << sizeof(float) << " bytes\n";
    std::cout << "double:\t\t" << sizeof(double) << " bytes\n";
    std::cout << "long double:\t" << sizeof(long double) << " bytes\n";

    return 0;
}
```

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:

```
#include <iostream>

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

    return 0;
}
```

```
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*.

 **4.4 -- Signed integers**

 **Index**

 **4.2 -- Void**

C++ TUTORIAL | PRINT THIS POST

199 comments to 4.3 — Object sizes and the sizeof operator

« Older Comments 1 2 3

Air Paul
February 2, 2020 at 9:41 pm · Reply

CPUs are often optimized to process data of a certain size. Any reasons???

nascardriver
February 4, 2020 at 8:24 am · Reply

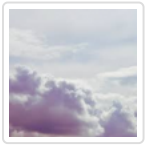
Fast things are expensive. Consumer products have to be affordable.

Air Paul
February 2, 2020 at 9:30 pm · Reply

It is given in this section that "C++ only guarantees that each fundamental data types will have a minimum size", does C++ guarantees about the range of values a particular data type holds.

nascardriver
February 4, 2020 at 8:51 am · Reply

Yep.
"-2^N –1 to 2^(N-1) - 1 (inclusive), where N is called the width of the type"



Chayim
January 6, 2020 at 10:18 pm · Reply

"To generalize, an object with n bits can hold 2n (2 to the power of n"
What is a -n bit-? It was never explained in the lesson

Alex
January 8, 2020 at 12:06 pm · Reply

n is used in an algebraic sense here -- replace n with any integer (e.g. 8, 16, 32, etc...).

Mike
September 26, 2019 at 10:32 am · Reply

Please tell me there is an easier way to rename a .cpp file you're working in. Many times while testing, I end up with code that I would like to keep for reference, but for the life of me I can't seem to find an option in Visual Studio that allows you to do this. And the last time I renamed one from the Solution Explorer, I could never get it to reopen. It kept saying it was missing a number of things. So I've just been copying and pasting the code in a new solution each time, which just seems silly.

Raton

January 22, 2020 at 6:27 am · Reply

Have you tried simply selecting the file by clicking it, and then pressing F2? (Do that on a test project in case it doesn't work properly)

Ganesh Koli
September 21, 2019 at 11:33 pm · Reply

Hello Alex,
Thank you for the great tutorial.
I have question related to sizeof() operator, new & delete.

Q1. How to use sizeof() operator to calculate the correct size of array using pointer?

```
Student *plistStudent = new student[10];  //consider Student is class with size 4 bytes
Student arrayStudent[10];

delete [] plistStudent;
cout<<sizeof(plistStudent)<<endl;   // 4 bytes
cout<<sizeof(*plistStudent)<<endl;  // 4 bytes
cout<<sizeof(&plistStudent)<<endl;  // 4 bytes
cout<<sizeof(arrayStudent)<<endl;   // 40 bytes
```

In above code i have a pointer of 'Student' type which is pointing to the array of Student type.
Suppose in method i received the pointer. How i get to know if this pointer is pointing to the one object or array of object ? and if it is pointing to array then how to calculate the size of array.

To delete the one object we simply use 'delete'.
To delete array of objects we use delete[], but we are not passing the size of array.
Q2. so how can 'delete' calculate the size of array using pointer only ? and how could use the similar technique in first question?

Thanks alot.

nascardriver
September 21, 2019 at 11:54 pm · Reply

You can't obtain the array size from a pointer. `delete[]` uses some internal magic, inaccessible to you. If you need to know the size after the array decayed to a pointer, pass the array's size as an extra argument or use a standard container (covered later).

Ganesh Koli
September 25, 2019 at 10:59 am · Reply

Thank you nascardriver,
As you mentioned the magic 'delete' has to find out the size of array. Does it implementation dependent ? or C++ standard provide the specification of delete to get the size of array ?

My thoughts about why it works for delete. Thanks to correct me.
- Pointer can point to any memory location which can be on heap or stack or data segment(i mean global variable). Pointer treats them same as a memory location.

```
int stackInt = 100;
int *heapInt = new int(10);

int *pIntPointer = &stackInt ; //pointing to local variable
```

```
int *pIntPointer = *heapInt;   // pointing to heap int
int *pIntPointer = &GlobalInt; //consider GlobalInt const is globally defined
```

- 'new' allocates memory in heap. 'delete' release it.
- When we allocate array in heap then its allocate extra byte/bytes to keep the size and other information. Such extra byte is not allocated in stack array.
In delete syntax pointer should be only pointing to the memory allocated in heap. we just need to tell delete() whether we want to delete one object or array. if it is array then delete use the extra byte the get the size information.

```
delete ptr;    // delete one object
delete [] ptr; //delete array
```

I think 'delete' can use this technique because it is sure that the memory allocated in heap; however in normal pointer we can never be sure, if pointer is pointing to the local array or array in heap. So pointer can not calculate the size.

I am confused. Its just my thoughts. Please correct my understanding.

nascardriver
September 26, 2019 at 3:46 am · Reply

Most of the underlying memory model is unspecified in C++. Compiler developers can keep track of the size however they want. A common implementation is to allocate some extra memory when `new[]` is called and store the array's size there.

Ashwathy Anil Alappatt
June 19, 2019 at 11:26 am · Reply

Hi , one query , why is int shown as 2 bytes , when it should be 4 bytes , according to this
4 bytes
For example, in 16-bit machines, sizeof (int) was 2 bytes. 32-bit machines have 4 bytes for int . It has been considered int was the native size of a processor, i.e., the size of register. However, 32-bit computers were so popular, and huge number of software has been written for 32-bit programming model.

Alex
June 19, 2019 at 2:43 pm · Reply

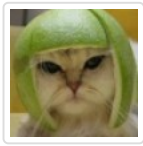
int has a _minimum_ size of 2 bytes. But it can be (and usually is) larger.

(name redacted)
June 5, 2019 at 9:34 am · Reply

Why does bool data type have to take 1 byte?
It can use a bit for true and false

nascardriver
June 6, 2019 at 4:08 am · Reply

Your CPU is good at working with types whose size is a power of 2 (1, 2, 4, 8), not so much with individual bits. Some implementations might even use 4 byte sized `bool`s.



Alex
June 12, 2019 at 12:25 pm · Reply

Modern machines typically have memory that is byte addressable, meaning they can only read/write a minimum of 1 byte. So even though a bool only technically requires 1 bit, we have to read/write 7 other bits of wasted space to access that 1 bit.

alfonso
May 25, 2019 at 12:03 am · Reply

Well, long double is 16 bytes on my system. :)

Code::Blocks and Arch Linux 64 bit.

Vincent C
May 15, 2019 at 5:15 am · Reply

Hi,

I have questions on the maximum value of int type in C++. You mentioned that the minimum byte size of int is 2.

However, the documentation in Microsoft:
https://docs.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=vs-2019

stated the INT_MAX constant refers to 2147483647, which means the integer is 4 bytes.

If the minimum byte size of integer is 2, why would the constant be defined as a 4-byte value? Does it mean that the program using that constant in some computers will crash if the int type in the computer architecture is only 2 bytes?

nascardriver
May 15, 2019 at 5:21 am · Reply

INT_MAX is a define of msvc++, it's not standard, don't use it. As long as that compiler uses 4 bytes for ints, there is no problem. If microsoft decides to use a different sized integer, they need to update the define too.

Louis Cloete
April 26, 2019 at 1:07 pm · Reply

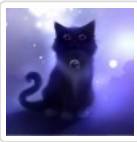
@Alex, in your "Key insight" block, you explain why it isn't necessary on a modern machine to try to use the smallest variable possible. It is, as I understand, not only not necessary, but also less performant in some cases. As far as I know, 4-byte integers are the fastest on modern 32-bit and 64-bit platforms. I was also told not to worry about sizes and just use Integer when I learnt Pascal, but the penny dropped only when I read that it is slower to use other types (and my teacher programmed in C in the '70s, so he would know what he is talking about when he says you can use four-byte integers safely today ;-)).

This long-winded thing is to suggest you drop a mention about the speed benefit of int over short and char into the block.

Alex
April 30, 2019 at 4:54 pm · Reply

While you're not wrong, any optimizations that have a negligible impact on memory usage also likely have a negligible impact on performance.

I added a point about performance to a new section at the end of the lesson.



Em
April 25, 2019 at 2:04 am · Reply

Hi,

You wrote that:
An object that uses 2 bytes can hold 2^16 (65536) different values!

I believe that this was a typo, and you meant to say 16 bytes, not 2 bytes.

nascardriver
April 26, 2019 at 2:50 am · Reply

Bytes is correct. 1 byte is 8 bit, 2 bytes are 16 bit. Each bit can have 2 states (0 or 1). 2^16 = 65536.

Jules
January 20, 2019 at 10:06 am · Reply

I hope someone like nascardriver or Alex sees this,
so I quite don't understand the relationship between number of bytes that a datatype has v/s actual values it can hold.
for eg:
An standard int is of 4 bytes, so the number of values it can hold are 2^4 = 16.
but the value range an integer is -
-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647

Also what is up with there being different datatypes of char?,if a char holds only one character why are there different types of chars like - char,wchar_t,char16_t and char32_t, If i wanted to store a string wont i just declare an array?

i'm kinda lost here.

nascardriver
January 21, 2019 at 4:06 am · Reply

Hi Jules!

> An standard int is of 4 bytes, so the number of values it can hold are 2^4 = 16.
Both nope. An int doesn't have to be 4 bytes and your calculation is off.
Assuming a 4 byte int:
4 bytes are 32 bits. That makes 2^32 possible values. The range is -(2^31) to (2^31)-1

> char
1 byte

> wchar_t
2 bytes (No guarantees, I'm doing this from memory)

> char16_t
2 bytes

> char32_t
4 bytes

The larger char types are required to store non-ascii characters, because those can take up more than 1 byte.
You can store unicode strings in char arrays, but the individual characters will be split into multiple chars.

Jules
January 21, 2019 at 7:06 am · Reply

Hi nascar!

thanks for replying on such a short notice, it seems i mistook bits for bytes here.
as you said assuming that an integer has 4 bytes, the possible values would be 2^32 = 4294967296
now wont the range be: -2147483648 to +2147483647, your definition would have the range as -4294967296 to +4294967296, wont it?

also what do you mean by -
>but the individual characters will be split into multiple chars.
thanks for your time.

~Jules.

nascardriver
January 21, 2019 at 9:24 am · Reply

> your definition would have the range as -4294967296 to +4294967296, wont it?
It won't. I used 2^31 in my range, not 2^32

> also what do you mean by

```
#include <iostream>
#include <cstring>

int main(void)
{
  char szText[16]{};

  // It's usually not a good idea to let the user input into
  // a char array. I'm doing it because I think it's easier
  // to understand here.
  std::cin >> szText;

  std::cout << "input: " << szText << '\n';

  // @static_cast converts one type to another.
  // @std::strlen returns the length of a string in bytes.
  int iLength{ static_cast<int>(std::strlen(szText)) };

  std::cout << "length: " << iLength << '\n';

  for (int i{ 0 }; i < iLength; ++i)
  {
    // print the char at index @i
    std::cout << i << ": " << szText[i];
    // print the integer value of the char at index @i
    std::cout << " (" << static_cast<int>(szText[i]) << ")\n";
  }

```

```
29    return 0;
30  }
```

The lightning symbol has a value of 0x21AF (https://unicode-table.com/en/21AF/). Too much for a single char.

Running the code and giving a lightning as input results in

input: ↯
length: 3
0: ? (-30)
1: ? (-122)
2: ? (-81)

We input a single character, but we need 3 chars to store it. Each char on it's own doesn't make a whole lot of sense. But my terminal (and presumably yours too) knows how to print the 3 successive chars as a single character.

Jules
January 21, 2019 at 9:55 am · Reply

Thanks!, really helped me a lot!

Quyết
January 15, 2019 at 2:12 am · Reply

Hi, i have a question...
look gt1 function and gt2 function...

```
1   int gt1(int n)
2   {
3       int p = 1;
4       for (int i = 1; i < n + 1; i++)
5       p = p * i;
6       return p;
7   }
8
9   int gt2(int n)
10  {
11      if (n == 1) return 1;
12      else return n * gt2(n - 1);
13  }
```

which good ???
Thanks!

nascardriver
January 16, 2019 at 7:03 am · Reply

```
1   int gt1(int n)
2   {
3     // Uniform initialization
4     int p{ 1 };
5
6     // ++prefix
7     for (int i{ 1 }; i <= n; ++i)
8     {
```

```
9       // *=
10      p *= i;
11    }
12
13    return p;
14  }
```

The first solution is better. Avoid recursion (A function calling itself).

Quyết
January 16, 2019 at 4:53 pm · Reply

Thank you!

Edgar J
July 14, 2019 at 10:31 am · Reply

Is there a section of the tutorial on why to avoid recursion in C++? I'd like to jump ahead to it.

nascardriver
July 15, 2019 at 5:46 am · Reply

Lesson 7.11
"
Iterative functions (those using a for-loop or while-loop) are almost always more efficient than their recursive counterparts. This is because every time you call a function there is some amount of overhead that takes place in pushing and popping stack frames. Iterative functions avoid this overhead.
"

nascardriver
September 29, 2018 at 3:08 am · Reply

Hi Alex!
I have doubts about your "C++ guarantees that the basic data types will have a minimum size" table.
The standard only states
"There are five standard signed integer types : “signed char”, “short int”, “int”, “long int”, and “long long int”. In this list, each type provides at least as much storage as those preceding it in the list."
N4762 § 6.7.1 (2)

To me this sounds like a long long int could be a 1 byte sized type.

Alex
October 1, 2018 at 8:28 am · Reply

The C++ standard does only explicitly state as you say. However, the C++ standard apparently references the C standard in this regard, and the C standard implies a minimum range of numbers that each type must be able to hold. Implicitly, that implies a minimum size.

Here's the minimum sizes from the C standard: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (see page 22)

Also see https://stackoverflow.com/questions/50930891/what-is-the-minimum-size-of-an-int-in-c for some discussion about this.

Finally, note that https://en.cppreference.com/w/cpp/language/types corroborates this understanding.

nascardriver
October 2, 2018 at 8:34 am · Reply

Thanks Alex!

« Older Comments 1 2 3