4.12 — Literals

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

In programming, a **constant** is a fixed value that may not be changed. C++ has two kinds of constants: literal constants, and symbolic constants. We'll cover literal constants in this lesson, and symbolic constants in the next lesson.

Literal constants (usually just called **literals**) are values inserted directly into the code. For example:

```
return 5; // 5 is an integer literal
bool myNameIsAlex { true }; // true is a boolean literal
std::cout << 3.4; // 3.4 is a double literal
```

They are constants because their values can not be changed dynamically (you have to change them, and then recompile for the change to take effect).

Just like objects have a type, all literals have a type. The type of a literal is assumed from the value and format of the literal itself.

By default:

Literal value	Examples	Default type
integral value	5, 0, -3	int
boolean value	true, false	bool
floating point value	3.4, -2.2	double (not float)!
char value	'a'	char
C-style string	"Hello, world!"	const char[14]

Literal suffixes

If the default type of a literal is not as desired, you can change the type of a literal by adding a suffix:

Data Type	Suffix	Meaning
int	u or U	unsigned int
int	l or L	long
int	ul, uL, Ul, UL, lu, lU, Lu, or LU	unsigned long
int	ll or LL	long long
int	ull, uLL, Ull, ULL, llu, llU, LLu, or LLU	unsigned long long
double	f or F	float
double	l or L	long double

You generally won't need to use suffixes for integer types, but here are examples:

```
unsigned int value1 { 5u }; // 5 has type unsigned int
long value2 { 6L }; // 6 has type long
```

By default, floating point literal constants have a type of *double*. To make them float literals instead, the f (or F) suffix should be used:

```
float f { 5.0f }; // 5.0 has type float
```

New programmers are often confused about why the following doesn't work as expected:

```
float f { 4.1 }; // warning: 4.1 is a double suffix, not a float suffix
```

Because 4.1 has no suffix, it's treated as a a double literal, not a float literal. When C++ defines the type of a literal, it does not care what you're doing with the literal (e.g. in this case, using it to initialize a float variable). Therefore, the 4.1 must be converted from a double to a float before it can be assigned to variable f, and this could result in a loss of precision.

Literals are fine to use in C++ code so long as their meanings are clear. This is most often the case when used to initialize or assign a value to a variable, do math, or print some text to the screen.

String literals

In lesson **4.11 -- Chars**, we defined a string as a collection of sequential characters. C++ supports string literals:

```
std::cout << "Hello, world!"; // "Hello, world!" is a C-style string literal
std::cout << "Hello," " world!"; // C++ will concatenate sequential string literals
```

String literals are handled very strangely in C++ for historical reasons. For now, it's fine to use string literals to print text with std::cout, but don't try and assign them to variables or pass them to functions -- it either won't work, or won't work like you'd expect. We'll talk more about C-style strings (and how to work around all of those odd issues) in future lessons.

Scientific notation for floating point literals

There are two different ways to declare floating-point literals:

```
double pi { 3.14159 }; // 3.14159 is a double literal in standard notation
double avogadro { 6.02e23 }; // 6.02 x 10^23 is a double literal in scientific notation
```

In the second form, the number after the exponent can be negative:

```
double electron { 1.6e-19 }; // charge on an electron is 1.6 x 10^-19
```

Octal and hexadecimal literals

In everyday life, we count using **decimal** numbers, where each numerical digit can be 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Decimal is also called "base 10", because there are 10 possible digits (0 through 9). In this system, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... By default, numbers in C++ programs are assumed to be decimal.

```
int x { 12 }; // 12 is assumed to be a decimal number
```

In **binary**, there are only 2 digits: 0 and 1, so it is called "base 2". In binary, we count like this: 0, 1, 10, 11, 100, 101, 110, 111, ...

There are two other "bases" that are sometimes used in computing: octal, and hexadecimal.

Octal is base 8 -- that is, the only digits available are: 0, 1, 2, 3, 4, 5, 6, and 7. In Octal, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, ... (note: no 8 and 9, so we skip from 7 to 10).

Decimal	0	1	2	3	4	5	6	7	8	9	10	11
Octal	0	1	2	3	4	5	6	7	10	11	12	13

To use an octal literal, prefix your literal with a 0:

```
#include <iostream>

int main()
{
    int x{ 012 }; // 0 before the number means this is octal
    std::cout << x;
    return 0;
}
```

This program prints:

```
10
```

Why 10 instead of 12? Because numbers are printed in decimal, and 12 octal = 10 decimal.

Octal is hardly ever used, and we recommend you avoid it.

Hexadecimal is base 16. In hexadecimal, we count like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, ...

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11

To use a hexadecimal literal, prefix your literal with 0x.

```
#include <iostream>

int main()
{
    int x{ 0xF }; // 0x before the number means this is hexadecimal
    std::cout << x;
    return 0;
}
```

This program prints:

```
15
```

Because there are 16 different values for a hexadecimal digit, we can say that a single hexadecimal digit encompasses 4 bits. Consequently, a pair of hexadecimal digits can be used to exactly represent a full byte.

Consider a 32-bit integer with value 0011 1010 0111 1111 1001 1000 0010 0110. Because of the length and repetition of digits, that's not easy to read. In hexadecimal, this same value would be: 3A7F 9826. This makes hexadecimal values useful as a concise way to represent a value in memory. For this reason, hexadecimal values are often used to represent memory addresses or raw values in memory.

Prior to C++14, there is no way to assign a binary literal. However, hexadecimal pairs provides us with an useful workaround:

```
#include <iostream>

int main()
{
    int bin{};
    bin = 0x01; // assign binary 0000 0001 to the variable
    bin = 0x02; // assign binary 0000 0010 to the variable
```

```
    bin = 0x04; // assign binary 0000 0100 to the variable
    bin = 0x08; // assign binary 0000 1000 to the variable
    bin = 0x10; // assign binary 0001 0000 to the variable
    bin = 0x20; // assign binary 0010 0000 to the variable
    bin = 0x40; // assign binary 0100 0000 to the variable
    bin = 0x80; // assign binary 1000 0000 to the variable
    bin = 0xFF; // assign binary 1111 1111 to the variable
    bin = 0xB3; // assign binary 1011 0011 to the variable
    bin = 0xF770; // assign binary 1111 0111 0111 0000 to the variable

    return 0;
}
```

C++14 binary literals and digit separators

In C++14, we can assign binary literals by using the 0b prefix:

```
#include <iostream>

int main()
{
    int bin{};
    bin = 0b1;  // assign binary 0000 0001 to the variable
    bin = 0b11; // assign binary 0000 0011 to the variable
    bin = 0b1010; // assign binary 0000 1010 to the variable
    bin = 0b11110000; // assign binary 1111 0000 to the variable

    return 0;
}
```

Because long literals can be hard to read, C++14 also adds the ability to use a quotation mark (‘) as a digit separator.

```
#include <iostream>

int main()
{
    int bin{ 0b1011'0010 };  // assign binary 1011 0010 to the variable
    long value{ 2'132'673'462 }; // much easier to read than 2132673462

    return 0;
}
```

If your compiler isn’t C++14 compatible, your compiler will complain if you try to use either of these.

Printing decimal, octal, hexadecimal, and binary numbers

By default, C++ prints values in decimal. However, you can tell it to print in other formats. Printing in decimal, octal, or hex is easy via use of std::dec, std::oct, and std::hex:

```
#include <iostream>

int main()
{
    int x { 12 };
    std::cout << x << '\n'; // decimal (by default)
    std::cout << std::hex << x << '\n'; // hexadecimal
    std::cout << x << '\n'; // now hexadecimal
    std::cout << std::oct << x << '\n'; // octal
    std::cout << std::dec << x << '\n'; // return to decimal
```

```
    std::cout << x << '\n'; // decimal

    return 0;
}
```

This prints:

```
12
c
c
14
12
12
```

Printing in binary is a little harder, as std::cout doesn't come with this capability built-in. Fortunately, the C++ standard library includes a type called std::bitset that will do this for us (in the <bitset> header). To use std::bitset, we can define a std::bitset variable and tell std::bitset how many bits we want to store. The number of bits must be a compile time constant. std::bitset can be initialized with an unsigned integral value (in any format, including decimal, octal, hex, or binary).

```
#include <iostream>
#include <bitset> // for std::bitset

int main()
{
    // std::bitset<8> means we want to store 8 bits
    std::bitset<8> bin1{ 0b1100'0101 }; // binary literal for binary 1100 0101
    std::bitset<8> bin2{ 0xC5 }; // hexadecimal literal for binary 1100 0101

    std::cout << bin1 << ' ' << bin2 << '\n';
    std::cout << std::bitset<4>{ 0b1010 } << '\n'; // we can also print from std::bitset direc

    return 0;
}
```

This prints:

```
11000101 11000101
1010
```

We can also create a temporary (anonymous) std::bitset to print a single value. In the above code, this line:

```
std::cout << std::bitset<4>{ 0b1010 } << '\n'; // we can also print from std::bitset directly
```

creates a temporary std::bitset object with 4 bits, initializes it with *0b1010*, prints the value in binary, and then discards the temporary std::bitset.

Magic numbers, and why they are bad

Consider the following snippet:

```
int maxStudents{ numClassrooms * 30 };
```

A number such as the 30 in the snippet above is called a magic number. A **magic number** is a literal (usually a number) in the middle of the code that does not have any context. What does 30 mean? Although you can probably guess that in this case it's the maximum number of students per class, it's not absolutely clear. In more

complex programs, it can be very difficult to infer what a hard-coded number represents, unless there's a comment to explain it.

Using magic numbers is generally considered bad practice because, in addition to not providing context as to what they are being used for, they pose problems if the value needs to change. Let's assume that the school buys new desks that allow them to raise the class size from 30 to 35, and our program needs to reflect that. Consider the following program:

```
1 int maxStudents{ numClassrooms * 30 };
2 setMax(30);
```

To update our program to use the new classroom size, we'd have to update the constant 30 to 35. But what about the call to setMax()? Does that 30 have the same meaning as the other 30? If so, it should be updated. If not, it should be left alone, or we might break our program somewhere else. If you do a global search-and-replace, you might inadvertently update the argument of setMax() when it wasn't supposed to change. So you have to look through all the code for every instance of the literal 30, and then determine whether it needs to change or not. That can be seriously time consuming (and error prone).

Fortunately, better options (symbolic constants) exist. We'll talk about those in the next lesson.

Warning

Don't use magic numbers in your code.

 **4.13 -- Const, constexpr, and symbolic constants**

 **Index**

 **4.11 -- Chars**

130 comments to 4.12 — Literals

« Older Comments 1 2

Ryan
February 10, 2020 at 4:40 am · Reply

Hi! Quick typo! In the first paragraph, it says "it's treated as a a double literal", with two 'a's.

I'm unsure if this is an easy fix, but I wanted to point out- I think the apostrophes in some of the code of paragraphs "C++14 binary literals and digit separators" and "Printing decimal, octal, hexadecimal, and binary numbers" mess with the coloring, making the readability a tiny bit harder. The apostrophes used as digit separators seem to be interpreted as quoting.

HolzstockG
December 23, 2019 at 1:47 am · Reply

Should I omit Chapter S and C?

nascardriver
December 23, 2019 at 7:36 am · Reply

Nope, they're regular chapters. They have letters in their names because they were squeezed in between older chapters.

Anderson
November 30, 2019 at 7:51 am · Reply

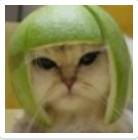
Hey all,

I have a quick question,
For example, in the code snippet below, derived from the content of the current topic

```
1  int bin{};
2      bin = 0x01; // assign binary 0000 0001 to the variable
3      bin = 0x02; // assign binary 0000 0010 to the variable
4      bin = 0x04; // assign binary 0000 0100 to the variable
5      bin = 0x08; // assign binary 0000 1000 to the variable
```

Could someone please, for instance, elaborate how to convert hexadecimal '0x01' to '0000 0001' ?
It would really help me.

Thank you.



Alex
December 2, 2019 at 10:46 pm · Reply

Every hexadecimal digit can be converted to 4 binary digits. So 0x01 = 0000 (the binary for 0) and 0001 (the binary for 1).

Alternatively, hexadecimal 0x01 = decimal 1 = binary 1 = binary 0000 0001 (when expressed as 8 bits)

Anderson
December 6, 2019 at 1:51 pm · Reply

Okay, so each hexadecimal digit has to be 4 binary digits. Ok, got that. thanks

Just another question though if you don't mind. What is the binary equivalent for '0x03' ? I just want to make sure. It seems you missed that out of the examples, I want to piece everything together now. So far, I know how to convert hexadecimal to decimal and then to binary and vice versa.

Thank you.

nascardriver
December 7, 2019 at 4:54 am · Reply

You can convert each binary digit to a decimal number and sum them up.

```
dec 8 4 2 1
bin 0 0 0 0
```

The decimal numbers are powers of 2. Now you just set the bits under the numbers you want to sum up. 3 is 2 + 1, so we set the bits under 2 and 1 to 1.

```
dec 8 4 2 1
bin 0 0 1 1
```

That's your 3.

Say we want an 11, we do 8 + 2 + 1

```
dec 8 4 2 1
bin 1 0 1 1
```

Durgesh Kumar Joshi
November 25, 2019 at 9:26 am · Reply

Hi Alex,
I have a doubt...As you explained above that binary literal is introduced in c++14 but I have used binary literal in C++98/C++03/C+11 and it is compiled and running fine in C++98/C++03/C+11.

```
#include <iostream>
using namespace std;

int main(){

	int a = 0b1111;
	cout<<"a:: "<<a<<endl;

	int var = 4;
	if(var == 0b0100)
		cout<<"True----------\n";
	else
		cout<<"Flase----------\n";

	return 0;
}

```

Output:-
a:: 15
True----------

Kindly please let me know what is the difference point?

Alex
November 25, 2019 at 10:26 pm · Reply

Either you're actually compiling C++14 (or newer) and not realizing it, or your compiler is allowing non-compliant code when using an older language standard.

BooGDaaN
November 12, 2019 at 7:33 am · Reply

Update line 11 at section "Printing decimal, octal, hexadecimal, and binary numbers" using Uniform Initialization

```
1 std::cout << std::bitset<4>(0b1010) << '\n';
```

Mhz93
September 29, 2019 at 2:32 am · Reply

Using a single quotation mark as a separator doesn't change the color of the text in my compiler but here in the examples using aa single quotation mark leads to see the rest of text in red!!
I guess you need to fix this!

Alex
September 30, 2019 at 9:07 am · Reply

Yeah, it's a known issue with the syntax highlighter this site uses. I've tried fixing it to no avail.

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

Thanks to add User defined literals (UDL) C++11 in this or other section.

HD
August 10, 2019 at 1:44 am · Reply

Under the topic "Literal Suffixes", there is an example of:
float f { 4.1 };
Tutorial explained (as I understood) that f is stored as a double type, not float type, because 4.1 has no suffix.

But to check this fact I wrote a code as following:

```
1 #include<iostream>
2 int main()
3 {
4     float f {4.1};
5     std::cout<<"float size: "<<sizeof(float)<<"\ndouble size: "<<sizeof(double)<<'\n';
6     std::cout<<"variable f size: "<<sizeof(f);
7     return 0;
8 }
```

Output was:
float size: 4

double size: 8
variable f size: 4

This proves that the variable f was stored as a float not double.

Am I correct about this ? Please justify.

Btw, Thanks a lot for such a useful tutorial series.

nascardriver
August 10, 2019 at 2:32 am · Reply

`4.1` is a double, but `f` is a float.
`4.1` is converted to a `float` during initialization.
When you declare a variable as a certain type, it will always have that type. It cannot be changed.

If you used `auto` to declare `f`, it'd be a `double`. `auto` automatically uses the type of the value, it's covered later.

```
1 auto f{ 4.1 }; // @f is a double
```

HD
August 12, 2019 at 4:26 am · Reply

Thanks a lot @nascardriver.
I got it. :)
Actually, when I was going through this tutorial, I was in hurry, so misunderstood the concepts.

learning
June 19, 2019 at 12:25 pm · Reply

so that's the same..:

```
1     int value1{ 5u };
```

..as this one?:

```
1     unsigned int value1{ 5 };
```

Alex
June 19, 2019 at 2:44 pm · Reply

The same in that both variables will have the integral value 5, yes.
Not the same in that one variable is of type "int" and the other is of type "unsigned int"

Samira Ferdi
June 12, 2019 at 5:54 pm · Reply

Hi Alex and Nascardriver!

So, because a single hexadecimal digit encompasses 4 bits,

1) 3A7F 9826 is 32 bits?

2) 3A7 is 12 bits?

3) 3A7F is 16 bits?

is this right? So, every hexadecimal digit represent 4 bits?

nascardriver
June 13, 2019 at 3:37 am · Reply

They are integers, and thus use 32 bits each. But they only occupy the last X bits, where X is the number you said respectively. The first 32-X bits are 0.
Assuming little endian and 32 bit (4 byte) integers, this is what the numbers look when they're separated into bytes.

```
// 0x3A7F9826
26 98 7F 3A

// 0x3A7
A7 03 00 00

// 0x3A7F
7F 3A 00 00
```

Louis Cloete
April 28, 2019 at 12:29 pm · Reply

Hi @Alex!

A few places where your updating to use uniform initialization missed something:

Under "C++14 binary literals and digit separators," both of your examples initialize the variable using direct initialization.

Under "Magic numbers, and why they are bad," the second snippet initializes maxStudents via copy initialization.

Alex
May 2, 2019 at 5:03 pm · Reply

Also fixed. Thanks!

Red Lightning
April 12, 2019 at 12:42 pm · Reply

```
std::cout << "Hello, world!" // "Hello, world!" is a C-style string literal
std::cout << "Hello," " world!" // C++ will concatenate sequential string literals
```

Syntax error: missing semicolon
Syntax error: missing semicolon

Alex
April 14, 2019 at 7:57 am · Reply

Fixed, thanks!

Jeroen P. Broks
February 17, 2019 at 3:53 am · Reply

When it comes to 'magic numbers', I'd also like to avoid them in main code, and not only in declarations/initial definitions.
When I was being trained in JavaScript, I had the assignment to make a program that would calculate with several numbers and the teacher wrote those numbers on the board. I was the only one who came up with putting all these numbers in constants prior to writing the formula to calculate it all through. And now it was easy, by commenting out the teachers numbers and make a new set of constants with my own picked numbers, I could check if the outcome was correct without having to change my formula all the time.
Oh, and I did the same with strings I had in some assignments, and boy did it save me a lot of work.

Of course, in some situations magic numbers are quite obvious, like in a little quiz game:

```
1 correct_answers = correct_answers + 1; // 1 is "magic number here"... (although I guess in C
```

Alex
February 17, 2019 at 2:45 pm · Reply

The +1 in an increment or decrement generally isn't considered a magic number, nor is 0 used as an initializer.

Jeremy
January 15, 2019 at 2:10 am · Reply

@Alex

Is it possible to add to the literal suffixes section, nascarsdrivers very helpful answer in the comments that suggests:

```
1 float f{5.0f};    // The right hand side is evaluated first before the compiler looks at wha
```

Which nicely explains the reason for a beginner like me as to why the compiler initially defaults the literal to a double when using the float data type without the f suffix, as the literal is evaluated before the compiler sees the data type.

Just it confused a beginner like me, as well as some others in the comments as i guess we assumed its evaluated from left to right, not vice versa. Hope it might be helpful for someone else.

Sorry if i miss-understood/read something and have made a mistake.

Alex
January 21, 2019 at 11:01 am · Reply

I find that phrasing misleading.

The compiler defaults floating point literals to type double because that's what the standard says it should do. A variable's declared type is independent from the type of the value that it is initialized with. If the two don't match, then a conversion must take place. It's as simple as that.

That said, this does seem to be an area of regular confusion, so I'll flag this for a little more explanation.

Benur21
December 28, 2018 at 9:32 am · Reply

Did you notice the ' breaks color formatting?

http://prntscr.com/m0p4r0

Alex
December 30, 2018 at 10:53 pm · Reply

Yep, the syntax highlighter this site uses doesn't cover this particular case properly. I'll add it to my to-do to see if I can figure out how to fix it manually.

I'm in love with alex's tutor
November 9, 2018 at 1:12 am · Reply

Dear Mr Alex And Mr. Nas!!!
Peace for you!!!!!!

We use Ob prefix to assign or initialize binary literals to the integer variable.How can we do this if the binary number is signed and negative? Do we have to use negative sign(-) B/s we have said that the left most significant Bit of the binary number uses us a sign indicator.
not only the binary , what about Ox(hexa) and O(octal) also?
Waiting for your beloved answer as usual.
God bless

nascardriver
November 9, 2018 at 1:33 am · Reply

When using the prefixes ("0x" or "0b") to initialize a signed integer type you cannot exceed the positive limit to get into negative. You have to use a '-' prefix.

```
1 // Assuming a 32 bit int.
2 int i{ 0b1000'0000'0000'0000'0000'0000'0000'0000 }; // Illegal
3 int j{ 0x80000000 }; // Illegal
4 int k{ -0b0001 }; // Legal (-1)
5 int l{ -0x01 }; // Legal (-1)
```

I'm in love with Alex's tutorial
November 9, 2018 at 1:52 am · Reply

Dear Mr. NAS!!!
MUCH CLEAR!!!
I HAVE NO WORDS TO THANK YOU FOR YOUR PROMPT AND OWSOME ANSWERS.
GOD BLESS ALWAYS!!!!!!

Alireza
January 30, 2019 at 8:08 am · Reply

You answered it clearly, thank you.
I have a question almost out of this topic.
How does a Hardware recognize a negative number ?

If the answer takes a long time, give me a trusty link or subject.

(Recommend for other users: If you're using Windows, you can use the calculator and use ' Alt+3 ' to set it to Programmer calculator and determine Hex, Oct, Bin, Dec numbers with it!)

nascardriver
January 30, 2019 at 9:09 am · Reply

Most number types have a sign bit. If it's 1, the number is negative, if it's 0, the number is positive.
One such representation (two's complement) is covered in lesson 3.7.

Jeff
October 24, 2018 at 10:51 pm · Reply

float f = 4.5; as an example of loss of precision or unexpected rounding makes no sense to me. 4.5 can be exactly represented as 9 * 2^-2 (in non-C notation) in both float and double types

Alex
October 25, 2018 at 12:50 pm · Reply

Changed 4.5 to 4.1 to use a number that's more likely to result in floating point precision errors.

Jeff
July 19, 2018 at 11:07 am · Reply

I'm having a slight issue comprehending the section regarding suffixes defining numeric literal types. for example you put:

```
1 float fValue = 5.0f; // float
2 double d = 6.02e23;  // double (by default)
```

Why would one need to specify "float" but then put in the "f" after the literal definition? Could you not simply put:

```
1 float fValue = 5.0;
```

to achieve the same result as putting the f in? I assumed that the variable "float" would be enough of a signifier that you are using a float variable. The statment "These suffixes are optional, as the compiler can usually tell from context what kind of constant you're intending." might clarify this for me? Could you give an example of a situation where you would NEED to give that suffix? I can't quite wrap my mind around why you would need to define the variable "float" but then put in a suffix after the definition... But perhaps that comes later in the curriculum?

Anywho, thank you in advance!

nascardriver
July 20, 2018 at 4:46 am · Reply

Hi Jeff!

> Could you not simply put [...] to achieve the same result as putting the f in?
You could, it'd work, you might loose precision, because the right side is a double and the left side is a float, but it'd work. Still, if you want a float, they you should go for floats all the way and don't mix types, even if it's not a huge problem with floats and doubles.

> I assumed that the variable "float" would be enough of a signifier that you are using a float variable
It is, but whatever you do on the right hand side will be a double, which will later be converted to a float.

> Could you give an example of a situation where you would NEED to give that suffix?
Your 10/3 example from earlier, it's giving wrong results when you're not using a suffix.
You'll learn about the "auto" keyword later on, it deduces the type from the value

```
auto i = 3; // @i is an int
auto fl = 3.0f; // @fl is a float
auto db = 3.0; // @db is a double
```

> I can't quite wrap my mind around why you would need to define the variable "float" but then put in a suffix after the definition
The right hand side is evaluated before the compiler looks at what type you're intending to use.

Jeff
July 20, 2018 at 8:20 am · Reply

Okay. So that makes sense. I didn't realize that it needed the number to be specified as well because the compiler doesn't look at the type on the left at the same time. I thought it executed from left to right and would know that the number to the right should be a float.

So if I need to use a float then I need to signify with a suffix that the number is a float. Even if the type is defined as a float. Got it.

Thank you very much!

Aaron
July 6, 2018 at 2:40 pm · Reply

I really don't understand why suffixes exist if you're declaring the variable to be a certain type. What's the difference? And why are both necessary?

nascardriver
July 8, 2018 at 4:58 am · Reply

Hi Aaron!

```
double db{ 10 / 3 };
```

@db is 3, because 10 and 3 are integers. You need a suffix to tell the compiler which kind of number to use.

Jeff
July 19, 2018 at 11:01 am · Reply

This answer doesn't make sense to me... it doesn't know to just evaluate the "10 / 3"? I thought it would evaluate mathematical equations within brackets like that when assigning to a variable...?

nascardriver
July 20, 2018 at 3:50 am · Reply

It does evaluate the mathematical equation, but since both, 10 and 3, are integers, the result will be an integer.

Naturally,
10 / 3 = 3.3333,
but integers aren't floating point numbers, so
10 / 3 = 3

The calculation happens before you tell the compiler that you want to have a double, so you have an integer (3) and convert it to a double, which still is 3.

If you want to have floating point values you need to tell the compiler right away by using a suffix.

10.0 / 3.0 = 3.33333

Having one floating point number in the equation is enough, the type with the highest precision determines the result type.

Jeff
July 20, 2018 at 8:15 am · Reply

Makes perfect sense. Thank you!

Johnny
May 9, 2018 at 10:35 pm · Reply

Hi guys.

I don't understand how the hexadecimal pairs workaround works.

Why is 0x8 equivalent to the fourth bit and why is 0x10 equivalent to the fifth bit?

Thanks for your help.

Alex
May 10, 2018 at 10:32 pm · Reply

0x8 = 8 decimal = 0000 1000 binary.
0x10 = 16 decimal = 0001 0000 binary.

If you're having trouble understanding hexadecimal numbers in the first place, see https://learn.sparkfun.com/tutorials/hexadecimal

Prabhat Kumar Roy
April 15, 2018 at 5:22 pm · Reply

Dear Sir!

I am studying and understanding Lesson 2.9 for 12 days(if not more) where also you have used "setMax(MAX_NAME_LENGTH);" in 2nd code-snippet in section 'Symbolic constants'; but the relevancy and way to use of "setMax(MAX_NAME_LENGTH);" is not at all clear to me.

QUOTE

Consider our second example, using #define symbolic constants:

```
1 #define MAX_STUDENTS_PER_CLASS 30
2 #define MAX_NAME_LENGTH 30
3
4 int max_students = numClassrooms * MAX_STUDENTS_PER_CLASS;
5 setMax(MAX_NAME_LENGTH);
```

UNQUOTE

To understand the relevancy and way to use of "setMax(MAX_NAME_LENGTH);", I have revisited the last code-snippet (including the paragraph following it) of the current Lesson(Lesson #2.8).

QUOTE

```
1 int maxStudents = numClassrooms * 30;
2 setMax(30);
```

To update our program to use the new classroom size, we'd have to update the constant 30 to 35. But what about the call to setMax()? Does that 30 have the same meaning as the other 30? If so, it should be updated. If not, it should be left alone, or we might break our program somewhere else. If you do a global search-and-replace, you might inadvertently update the argument of setMax() when it wasn't supposed to change. So you have to look through all the code for every instance of the literal 30, and then determine whether it needs to change or not. That can be seriously time consuming (and error prone).

UNQUOTE

Apart from the 'relevancy and way to use' question, it appeared to me that "setMax(30);" and "setMax(MAX_NAME_LENGTH);(in Lesson #2.9)" are 'variables with direct initialization'; but from the discussion/explanation you have mentioned here, those are 'functions' because variables do not have 'arguments/parameters' and neither those are 'function-prototype'.

Please help me understanding this part where I'm stuck for quite long time.

Thank you.

Best regards.

nascardriver
April 16, 2018 at 2:07 am · Reply

Hi Prabhat!

@setMax is a function. We don't know what it does or what the parameter is used for.
You can tell it's not a variable, because there's no type specification.

```
1 // Assuming @setMax has been declared somewhere before
2
3 setMax(30); // Function call.
4 int setMax2(30); // Variable declaration with direct initialization.
5 setMax3(30); // Error, the compiler doesn't know what @setMax3 is.
```

Prabhat Kumar Roy
April 16, 2018 at 9:24 am · Reply

Thank you Sir.

Prabhat Kumar Roy
April 7, 2018 at 4:46 pm · Reply

Dear Sir!

With reference to your Lesson #2.3, when I checked with "sizeof", I got exactly the same results what you shared except in the last line wherein I got:
long double: 12 bytes
I am using latest version(?) of CodeBlocks 17.12mingw-setup.exe.

Now with reference to the heading "C++14 binary literals and digit separators" of the current Lesson, when I am putting the 2nd example into std::cout:

```
#include <iostream>

int main()
{
    int bin = 0b1011'0010;  // assign binary 1011 0010 to the variable
    long value = 2'532'673'462; // much easier to read than 2532673462

    std::cout << bin << std::endl;
    std::cout << value << std::endl;

    return 0;
}
```

,

I'm getting 178 for 'bin' which is correct; but for 'value' it is showing '-1762293834' which is probably an overflow(I would request your verification and comment); but if I change the 'type' of the variable 'value' from 'long(as you have used)' to 'long long', it is producing the correct figure, but again without any separator.

Thank you.

Best regards.

nascardriver
April 8, 2018 at 12:33 am · Reply

Hi Prabhat Kumar Roy!

> I got exactly the same results what you shared except in the last line
The implementation of 'long double' is up to the compiler. Some use 8, some 12, and some 16 bytes.

> which is probably an overflow
It is. MinGW should've warned you about it. If not, add

```
-Wall -Wextra
```

to your compiler options (Or enable all warnings in your project settings).
The warning is:

```
warning: overflow in implicit constant conversion [-Woverflow]
   long value = 2'532'673'462;
                ^~~~~~~~~~~~
```

> without any separator
The separators are just there to make your code easier to read. They don't affect the number or output.

Prabhat Kumar Roy
April 8, 2018 at 6:58 am · Reply

Thank you.

Tried both the options; still warning message not appearing and output remains as it was before changing the compiler options/project settings.

Anyway, not to worry about at this stage and I'm moving forward to the next lesson. However, I would be writing to SOURCEFORGE.net to convey the same to MinGw.

Best regards.

Prabhat Kumar Roy
April 6, 2018 at 6:56 am · Reply

Dear Sir!

As you have shown above:

"C++ also supports char and string literals:

char c = 'A'; // 'A' is a char literal
std::cout << "Hello, world!" // "Hello, world!" is a C-style string literal
std::cout << "Hello," " world!" // C++ will concatenate sequential string literals
Char literals work just like you'd expect. However, string literals are handled very strangely in C++. For now, it's fine to use string literals to print text with std::cout, but don't try and assign them to variables or pass them to functions -- it either won't work, or won't work like you'd expect. We'll talk more about C-style strings (and how to work around all of those odd issues) in future lessons.
"

I tried here:

```
#include <iostream>

int main()
{
    char c = "A";
    std::cout << c << std::endl;
    return 0;
}
```

;

I got error message: "invalid conversion from 'const char*' to 'char' [-fpermissive]".

However, when I changed from 'char c = "A";' to 'char c = 97;', I got correctly printed a(lowercase a).

Will you please explain it?

Thank you.

Best regards.

nascardriver
April 6, 2018 at 8:09 am · Reply

Hi Prabhat Kumar Roy!

"A" is a string, @c is a char, you can't assign a string to a char.
You need to use single quotation marks (') for characters.

```
char c{ 'A' };
```

97 works too, because a char is an 8bit int.

Prabhat Kumar Roy
April 6, 2018 at 6:01 pm · Reply

Oh Ho! probably I got it.
It is that because it is within double quote(" ").

Thank you Sir!

Best regards.

nascardriver
April 7, 2018 at 1:48 am · Reply

> It is that because it is within double quote(" ")
Yes it is, double quote means string, single quote means char.

Matthew
January 2, 2018 at 9:22 pm · Reply

Hello,

Will I have to worry about the suffixes? As you said, the compiler should be able to tell what type I am using based on what type the variable is.

nascardriver
January 3, 2018 at 12:11 am · Reply

Hi Metthew!
For floats, yes please, because without the 'f' the number will be treated as a double, not a float. Otherwise, you rarely see them, they're sometimes used for (unsigned) long long, but that's about it.

Alex
January 4, 2018 at 3:59 pm · Reply

Yes, you do have to worry about suffixes, and no the compiler won't infer the literal's type from the variable being initialized with the literal.

Sndn
December 13, 2017 at 1:15 am · Reply

Hi Alex,

I want to check out the C++ 14 functionality in this tutorial, in eclipse. How do I do that? I'm using MinGW-w64.

Alex
December 13, 2017 at 10:30 am · Reply

I'm not sure. See if your version of MinGW-w64 has a flag to enable c++14 functionality (something like -std=c++14) and if so, see if you can add that flag to your compiler settings so it gets passed whenever your compiler is called from eclipse.

Jason R.
October 4, 2017 at 3:01 pm · Reply

I'm not sure if you know or not or if it even fits within the scope of this particular lesson but I learned this trick (from a computer networking book) for converting from hexadecimal to binary and vice versa. The only prerequisite is knowing the hexadecimal-binary equivalents for each hexadecimal digit.

Since each digit in hexadecimal is 4 bits of information you can break down a large hexadecimal number one digit at a time and create 4-bit groups independent of each other. It fits in nicely with the programmer's mantra of breaking one large problem into more smaller problems.

As an example: AFED can be broken down one digit at a time. Since A = 1010, F = 1111, E = 1110, and D = 1101, then that means that AFED is equivalent to 1010 1111 1110 1101.

Muharrem
August 17, 2017 at 12:06 am · Reply

Dear Alex, first of all thank you very much for this great tutorial.
I have a question about the hexadecimal example that I quoted below.

```
1    int bin(0);
2    bin = 0x01; // assign binary 0000 0001 to the variable
```

I thought "int bin(0)" will initialize a 2 byte integer (or 4 byte integer on modern architectures) as explained in chapter 2.4. But the assigned binary is 1 byte. Does C++ truncate the integer to 1 byte if the assigned literal can fit into 1 byte? Or any other reason?

Alex
August 17, 2017 at 12:17 pm · Reply

> Does C++ truncate the integer to 1 byte if the assigned literal can fit into 1 byte? Or any other reason?

bin = 0x01 is the equivalent of bin = 1. In either case C++ will simply assign the value of 1 to the integer, regardless of the integer's size.

« Older Comments 1 2