4.1 — Introduction to fundamental data types

BY ALEX ON JUNE 4TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 23RD, 2019

Bits, bytes, and memory addressing

In lesson **1.3 -- Introduction to variables**, we talked about the fact that variables are names for a piece of memory that can be used to store information. To recap briefly, computers have random access memory (RAM) that is available for programs to use. When a variable is defined, a piece of that memory is set aside for that variable.

The smallest unit of memory is a **binary digit** (also called a **bit**), which can hold a value of 0 or 1. You can think of a bit as being like a traditional light switch -- either the light is off (0), or it is on (1). There is no in-between. If you were to look at a random segment of memory, all you would see is ...011010100101010... or some combination thereof.

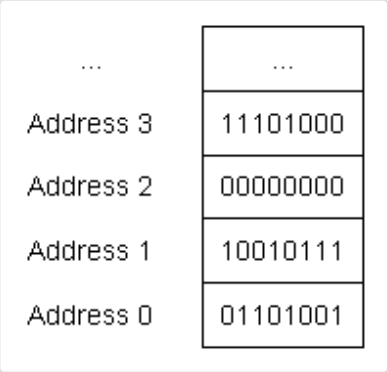
Memory is organized into sequential units called **memory addresses** (or **addresses** for short). Similar to how a street address can be used to find a given house on a street, the memory address allows us to find and access the contents of memory at a particular location.

Perhaps surprisingly, in modern computer architectures, each bit does not get its own unique memory address. This is because the number of memory addresses are limited, and the need to access data bit-by-bit is rare. Instead, each memory address holds 1 byte of data. A **byte** is a group of bits that are operated on as a unit. The modern standard is that a byte is comprised of 8 sequential bits.

Key insight

In C++, we typically work with “byte-sized” chunks of data.

The following picture shows some sequential memory addresses, along with the corresponding byte of data:



As an aside...

Some older or non-standard machines may have bytes of a different size (from 1 to 48 bits) -- however, we generally need not worry about these, as the modern de-facto standard is that a byte is 8 bits. For these tutorials, we’ll assume a byte is 8 bits.

Data types

Because all data on a computer is just a sequence of bits, we use a **data type** (often called a "type" for short) to tell the compiler how to interpret the contents of memory in some meaningful way. You have already seen one example of a data type: the integer. When we declare a variable as an integer, we are telling the compiler "the piece of memory that this variable uses is going to be interpreted as an integer value".

When you give an object a value, the compiler and CPU take care of encoding your value into the appropriate sequence of bits for that data type, which are then stored in memory (remember: memory can only store bits). For example, if you assign an integer object the value *65*, that value is converted to the sequence of bits `0100 0001` and stored in the memory assigned to the object.

Conversely, when the object is evaluated to produce a value, that sequence of bits is reconstituted back into the original value. Meaning that `0100 0001` is converted back into the value *65*.

Fortunately, the compiler and CPU do all the hard work here, so you generally don't need to worry about how values gets converted into bit sequences and back.

All you need to do is pick a data type for your object that best matches your desired use.

Fundamental data types

C++ comes with built-in support for many different data types. These are called **fundamental data types**, but are often informally called **basic types**, **primitive types**, or **built-in types**.

Here is a list of the fundamental data types, some of which you have already seen:

Types	Category	Meaning	Example
float double long double	Floating Point	a number with a fractional part	3.14159
bool	Integral (Boolean)	true or false	true
char wchar_t char8_t (C++20) char16_t (C++11) char32_t (C++11)	Integral (Character)	a single character of text	'c'
short int long long long (C++11)	Integral (Integer)	positive and negative whole numbers, including 0	64
std::nullptr_t (C++11)	Null Pointer	a null pointer	nullptr
void	Void	no type	n/a

This chapter is dedicated to exploring these fundamental data types in detail (except std::nullptr_t, which we'll discuss when we talk about pointers). C++ also supports a number of other more complex types, called *compound types*. We'll explore compound types in a future chapter.

Author's note

The terms "integer" and "integral" are similar, but have different meanings. Integers are a specific data type that hold positive and negative whole numbers, including 0. The term "integral types" (which means "like an integer") includes all of the boolean, characters, and integer types (and thus is a bit broader in definition).

Integral types are named so because they are stored in memory as integers, even though they behave slightly differently.

The _t suffix

Many of the types defined in newer versions of C++ (e.g. std::nullptr_t) use a _t suffix. This suffix means “type”, and it’s a common nomenclature applied to modern types.

If you see something with a _t suffix, it’s probably a type. But many types don’t have a _t suffix, so this isn’t consistently applied.

 **4.2 -- Void**

 **Index**

 **3.x -- Chapter 3 summary and quiz**

C++ TUTORIAL | PRINT THIS POST

226 comments to 4.1 — Introduction to fundamental data types

« Older Comments 1 2 3

Apaulture
January 29, 2020 at 2:04 pm · Reply

"Integral types are named so because they are stored in memory as integers, even though they behave slightly differently."

This confuses me. Are chars, booleans, and ints stored in memory as integers first, then eventually stored in memory as bits? Bits themselves are also integers. So would a bool be stored as 1 and 0 for true and false?

P.S.
Something I found interesting is the addition of 1 to 2147483647, the last index of the int memory address. It returns -2147483648. Does this mean that 0 isn't actually all 0s in computer language, but rather, is a actually 1 then 16 0's? (It is the first index of the positive number space).

-2147483648 to -1: 32 0's to 16 1's and 16 0's
0 to 2147483647: 15 0's then 1 then 16 0's to 32 1's

nascardriver
January 30, 2020 at 2:29 am · Reply

Everything in memory is bits. Integral types are numbers, there's just a nice layer above them that changes how you use them in code (`false`, `true`, 'h', 'i' (Those are integers in disguise)).

```
true + 'k' // Legal, 1 + 107 = 108
```

There are situations in which characters are used for arithmetic, but most of the time it doesn't make sense, just use numbers.

Binary is covered in Chapter O.

I'm using 8 bit integers to keep the examples short

0 is 0000 0000.
1 is 0000 0001.
2 is 0000 0010.
...
127 is 0111 1111

Negative integers are special. To negate a number, you swap all bits and add 1.

1 is 0000 0001
swap all bits
1111 1110
add 1
-1 is 1111 1111
-2 is 1111 1110
...
-128 is 1111 1111

If the first bit is 1, the number is negative.

If we do 127 + 1
0111 1111 + 1
we get
1000 0000
That's -128

Apaulture
January 30, 2020 at 5:24 pm · Reply

That is very fascinating! That actually makes sense logically and semantically since -2 is less than -1 so a bit should be subtracted.

Zero isn't a positive nor negative and yet, the binary system takes into account as if it is a number which can be positive or negative. Otherwise, it would make sense to have a 2-bit encode -1, 0, and 1 but we'd miss out on fitting another value into that space.

If an 8-bit space spans -128 to 127, does that mean a 1-bit space holds -1 and 0 only?
That's interesting since a bit can take on values 0 and 1 but the actual value is -1 and 0.

nascardriver
January 31, 2020 at 12:25 am · Reply

Not all numbers can be negative. `unsigned` integers are only positive. There, 1111 1111 is 256, not -128. Similarly, 1 would be 1, not -1. There are no single-bit types in C++, but your suspicion is correct. 0 would be 0 and 1 would be -1.

Air Paul
January 27, 2020 at 9:04 am · Reply

Hey, you told that the suffix _t is a type but what is its significance. Also, By saying it "type", Are you meaning it is a type of data. And why you said it is a nomenclature.

nascardriver
January 28, 2020 at 2:06 am · Reply

It's a convention used to differentiate between types/variables/functions.

```
function(balloon(3)); // Is balloon a type or a function?
function(balloon_t(3)); // balloon is a type.
```

Paprika
January 13, 2020 at 4:27 am · Reply

In data types statement,an Integer type should be 4 bytes or 32 bits right? And the num 65 in memory is 0000 0000 0000 0000 0000 0000 0100 0001. 0100 0001 should be the type of int8_t or char?

nascardriver
January 13, 2020 at 4:41 am · Reply

An `int` can be 16 or 32 bits wide. If you want to write it out properly, your version is correct. In practice, the leading zeroes are often omitted.

hello
December 14, 2019 at 1:46 pm · Reply

Hello! Is there a way to donate money to the admins of this site? I really don't want to see ads but still want to support the creators.

nascardriver
December 15, 2019 at 1:23 am · Reply

Hello!

You can donate via paypal and bitcoin here https://www.learncpp.com/about/
Donating won't remove ads, feel free to use an ad-blocker :)

test
October 16, 2019 at 6:17 am · Reply

tes

nascardriver
October 16, 2019 at 6:20 am · Reply

t

Mike

September 19, 2019 at 6:35 pm · Reply

I did register it when I installed it prior, but going back to double check, I noticed there was an option to update my license. I clicked it and the message went away, so I guess I'm good now.

Mike
September 17, 2019 at 10:28 am · Reply

Visual Studio Community 2019 is telling me I only have 14 days left, and then I will NOT be able to use the product. My understanding was the community version is the free version.

Alex
September 19, 2019 at 3:55 pm · Reply

It is. You just have to register (which is free) and then it will grant you a license.

A
August 9, 2019 at 11:56 am · Reply

How do you pronounce "char"? Is it ch as in "chance" or ch as in "character"?
On the same topic: how is it with "std::cout"? do you say "STD kout"?

nascardriver
August 10, 2019 at 2:03 am · Reply

> "chance" or ch as in "character"
I've heard both, "chance" is more common.

> do you say "STD kout"?
c-out is the only version I heard.
std comes in
S T D
std (shtd, as if it was an actual word)
standard

A
August 10, 2019 at 11:04 pm · Reply

Thanks!

ethano
March 2, 2019 at 4:49 pm · Reply

In the "dangerous way of defining a variable" part, you write

```
1 int a, b = 5; // wrong (a is uninitialized!)
2
3 int a= 5, b= 5; // correct
```

since it says "correct", people may think you have to have the = next to the variable like this:

```
1 int b= 5;
```

which is fine but ist exactly how people normally write it. Not a bad mistake just slightly weird and may teach someone the non-standard way to write it

Alex
March 4, 2019 at 9:10 pm · Reply

Yup, just a typo. Thanks for pointing that out.

Jeroen P. Broks
February 17, 2019 at 1:27 am · Reply

As a former Pascal coder, I am quite often putting all declarations on the top of a function regardless if the language I use requires me to, as in Pascal you need to declare variables (even locals) outside of the scope unlike C/C++ you can only declare them at the top for code:

```
procedure Hello;
var
   A:String;
Begin
   A:='Hello World';
   WriteLn(A)
end;
```

Begin and End are the Pascal equivalents to { and } and the last statement before end does not require ; in Pascal, although it *is* allowed). As you can see the Var statement comes before 'Begin'. (Oh yeah, Pascal is case insensitive so BEGIN or Begin or BeGiN are all the same in Pascal).
In C that would look like this.

```
void Hello() {
   char A[10] = "Hello World";
   printf("%s\n",A);
}
```

Due to this I am always tempted to place variable declarations at the top, and that gave me one advantage. Organization. I often had a lot of bugs in my Go programs due to the Go's very short declare and define syntax, make the code above look like this:

```
func Hello(){
    A:="Hello World"  // declares A and as "Hello World" is a string, Go declares it as a s
tring and assigns the string immediately.
    fmt.Printf("%s\n",A)
}
```

I forgot all the time, I already declared 'A' before, so the compiler went haywire all the time.

It's not that I am fully against declaring to the closest point before you need it, as I did it it some of my C projects too (giving me issues when I tried to compile my 'Heks' project in MS-DOS, haha), but the advantage of declaring all variables at the top of the code is that you can see in one go what has been declared and what has not, which can particularly in long and complex functions be a blessing. In my time as a Pascal coder I never had issues with duplicate declarations, and forgotten declarations were also rare. This changed when I went to languages less restrictive than Pascal.

Now I do see it as an advantage too to only declare variables as close to the first time needed as possible, but when you want to prevent chaos, putting all declarations at the top can be an option too, I think.
When my functions have multiple tasks, I mostly declare required locals at the start of that task. Like this function in Lua to use the Pythagorean Theorism:

```
function(a,b)
    -- get the power two of both sides of the 90 degree corner
```

```
    local a2
    local b2
    a2 = a * a
    b2 = b * b

    -- get the power 2 of the hypotenuse
    local hypotenuse2
    hypotenuse2 = a2 + b2

    -- square root to get the actual length
    local hypotenuse
    hypotenuse = math.sqrt(hypotenuse2)
end
```

(Now, Lua does support declare and define in one line like C/C++ do, but I deliberately avoided that to show what I mean).
Now I divided this in 3 tasks and used the declaration (as far as you can speak of that in Lua, but locals must be declared as such as undeclared vars are always global in Lua). In declare when you need it first the upper task would look like this:

```
    local a2
    a2 = a * a
    local b2
    b2 = b * b
```

Somehow that does not appeal me, but maybe that's also the Aspie in me :P

Alex
February 17, 2019 at 2:42 pm · Reply

It's a matter of what you're used to. Having done it both ways myself, I find the "define it when you need it" superior, because I find the contextualization provided by where the variable is defined to be of more use than the organization of having all my variables in one place.

Richard
February 20, 2019 at 12:39 pm · Reply

Alex and Jeroen, both of you have valid points. A variable declared just before its first use makes easier editing at later date. But declaring a variable at the beginning of its block emphasizes the variable's scope. My experience finds that the advantages of the latter out way the former.

Modularity is a key aspect of programming style, and statement blocks (historically ALGOL, Pascal, C/C++) are potentially as useful as refactoring using functions. Perhaps both styles are worth mentioning in this regard? In either case, thanks Alex for your excellent tutorial.

Asgar
February 7, 2019 at 8:12 pm · Reply

I remember, in old C++, I was able to initialize an array like this:

```
int intArray[] = {3,9,-1,2};
```

But, that was before C++11, and therefore uniform initialization was not available. Since neither direct- nor copy initialization would be able to initialize a list of values, then what form of initialization should we say this example is?

According to Bjarne Stroustrup in his book, these four ways of initializing a variable current exist, of which the first one was introduced in C++11:

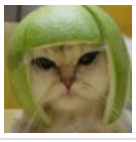
```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

Alex
February 7, 2019 at 9:17 pm · Reply

https://en.cppreference.com/w/cpp/language/aggregate_initialization

Asgar
February 8, 2019 at 6:47 am · Reply

Thanks, Alex. Is aggregate initialization is a form of copy initialization?



Alex
February 9, 2019 at 8:54 am · Reply

Aggregate initialization can be direct or copy. In this case, it's copy.

Alireza
January 24, 2019 at 12:48 pm · Reply

Hi there,
I've noticed that a value initialized with direct initialized takes the value 1, and a value initialized with uniform initialized takes the value 0.

```
    int number(); // number = 1
    int brace{};  //  brace = 0

    std::cout << number << std::endl << brace; // 1   // 0
```

nascardriver
January 25, 2019 at 6:20 am · Reply

Hi!
It doesn't. Line 1 in your code is C++'s most vexing parse. @number is not an int. It's the prototype of a function that returns an int.
@std::cout treats functions as booleans and every function evaluates to true, so it prints 1 (Or "true" if you set @std::boolalpha).
Uniform initialization solves this problem. Use uniform initialization.

Alireza
January 25, 2019 at 9:23 am · Reply

Thank you for answering,
So 'number' is a function declaration that is written in main(), right ?
This is why it returns 1.
Thank you so much and good luck ;D

nascardriver
January 25, 2019 at 9:55 am · Reply

Right.
But the function @number is never called. To call it you'd have to write

```
number()
// But you're writing
number
```

the latter is the address of the function @number.
@std::cout doesn't know how to print function pointers, so it converts it to a bool.

@Alex might be able to explain why @number can be converted at all, as the function it declares doesn't exist and any use should cause a linker error, but conversion to bool doesn't. I am unable to find an explanation for this, I'm uncertain if this behavior is well-defined.

Alireza
January 25, 2019 at 9:58 am · Reply

Thanks bro. you helped me a lot.
Good luck

Alex
January 27, 2019 at 4:37 pm · Reply

I'm not sure why gcc behaves this way. The program doesn't compile on Visual Studio, but it does on Code::Blocks (which uses gcc).

The behavior of implicitly converting a function pointer to a bool is well-defined (though Visual Studio has a compiler extension that will convert it to void* instead). I suspect gcc converting a function prototype to a bool is a bug.

ZISHAN SHEIKH
January 2, 2019 at 8:13 am · Reply

in built in data types table , int , long , long should be a non fractional number or it is a whole number?

nascardriver
January 2, 2019 at 8:40 am · Reply

Neither is right. Whole numbers are natural numbers and 0. Non-fractional numbers include irrational and complex numbers. Correct is integers (Positive and negative whole numbers, including 0).

@Alex

Alex
January 2, 2019 at 3:41 pm · Reply

Updated. Thanks for pointing this out.

Nguyen
September 6, 2018 at 9:11 pm · Reply

Hi,

Please refer to the same picture in this lesson for my example. The picture shows some sequential memory addresses, along with the corresponding byte of data.

Example:

```
int x = 1234;
```

Assuming that piece of memory is set aside for the variable x.

1. Which memory address of variable x will be printed out if I'd like to see?
In the picture, they are address 0, address 1, address 2 & address 3.

2. Converting 1234 to binary. So, the binary value is 010011010010. I'd like to see how the binary value looks like in each byte along with each memory address.

Thanks, Have a great day

nascardriver
September 7, 2018 at 12:59 am · Reply

Hi Nguyen!

1. The address of the first bit. If @x spans across 0x100-0x103, the address of @x is 0x100.

2.

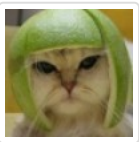
```
// Lets say 1234 is located at 0x100, takes up 32 bits, and is stored in little endi
an

Address  Value
         (binary)       (hexadecimal)  (decimal)
...      ...            ...            ...
0x100    1101 0010      D2             210
0x101    0000 0100      04             4
0x102    0000 0000      00             0
0x103    0000 0000      00             0
...      ...            ...            ...
```

Kio
August 24, 2018 at 3:35 pm · Reply

Hi Alex,
Just to be consistent across the "guide". You are missing

```
int value{5}; //uniform initialization -> this part.
```



Alex
August 25, 2018 at 8:48 am · Reply

Updated. Thanks!

Jon

July 20, 2018 at 8:45 am · Reply

It appears that uniform initialization still does not work in Eclipse (I'm using it on OS X if that matters)? Someone noted the same problem in 2016. I copied

```
1 int value{5};
```

into my code just as they did and received the message "expected ';' at the end of declaration". The place it is pointing to put the semicolon at is at

```
1 int value;{5};
```

which clearly just causes more errors. Any help?

nascardriver
July 20, 2018 at 8:54 am · Reply

Neither your IDE nor OS matter, your compiler is what makes the difference.
Check your settings to figure out which compiler you're using, then look up how to tell the compiler to use C++17 or later and change the settings accordingly. If your compiler doesn't support C++14 or later there's no reason to use it, get a new one.

Jon
July 20, 2018 at 9:09 am · Reply

It says it is using the GCC C++ Compiler. Not sure how to check what version of C++ it is using. I tried changing the compiler to cygwin, but that didn't help.

nascardriver
July 20, 2018 at 9:20 am · Reply

Run
g++ -v
in your command line to see which version of g++ you're using (last line). Without checking I'd say anything after version 6 should support C++14.
Then add -std=c++1z (or -std=c++17 or 14 if you're using a version earlier than gcc 8) to the compiler settings.

Jon
July 20, 2018 at 9:25 am · Reply

It is an old version for some reason: 4.2.1. Been trying to figure out how to update it? Sadly, I'm not knowledgeable about doing unix command line stuff or how to update these types of things.

Jon
July 20, 2018 at 9:28 am · Reply

OK, I am downloading gcc8 now.

Jon
July 20, 2018 at 9:46 am · Reply

So, I think I've updated. Now when I do "g++ -v", I get the following:

Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/opt/local/libexec/gcc/x86_64-apple-darwin17/8.0.0/lto-wrapper
Target: x86_64-apple-darwin17
Configured with: /opt/local/var/macports/build/_opt_bblocal_var_buildworker_ports_build_ports_lang_gcc8/gcc8/work/gcc-8-20170604/configure --prefix=/opt/local --build=x86_64-apple-darwin17 --enable-languages=c,c++,objc,obj-c++,lto,fortran --libdir=/opt/local/lib/gcc8 --includedir=/opt/local/include/gcc8 --infodir=/opt/local/share/info --mandir=/opt/local/share/man --datarootdir=/opt/local/share/gcc-8 --with-local-prefix=/opt/local --with-system-zlib --disable-nls --program-suffix=-mp-8 --with-gxx-include-dir=/opt/local/include/gcc8/c++/ --with-gmp=/opt/local --with-mpfr=/opt/local --with-mpc=/opt/local --with-isl=/opt/local --enable-stage1-checking --disable-multilib --enable-lto --enable-libstdcxx-time --with-build-config=bootstrap-debug --with-as=/opt/local/bin/as --with-ld=/opt/local/bin/ld --with-ar=/opt/local/bin/ar --with-bugurl=https://trac.macports.org/newticket --disable-tls --with-pkgversion='MacPorts gcc8 8-20170604_2'
Thread model: posix
gcc version 8.0.0 20170604 (experimental) (MacPorts gcc8 8-20170604_2)

It still does not compile correctly and I am unsure of how to add what you said to the compiler settings in Eclipse.

nascardriver
July 20, 2018 at 9:55 am · Reply

I don't have eclipse installed, some things might be called different.
Right click on your project
Project properties
C/C++ Build
Settings
Cross G++ Compiler -> Miscellaneous
In the "Other flags" textbox add -std=c++1z

Here's an image of the settings, the contents of the textbox might look different for you, that doesn't matter, just add what I said to the end
http://blog.fellstat.com/wp-content/uploads/2012/04/EclipseScreenSnapz013.png

While you're at it, you can also enable some more warnings to prevent you from writing bad code, add

```
1 -Wall -Wextra -pedantic-errors -std=c++1z
```

Jon
July 20, 2018 at 10:07 am · Reply

It worked! Thanks! I added all of the things you said to.

yugin
July 8, 2018 at 9:37 pm · Reply

Is it merely a coincidence that "int" is used for both initialising integers and functions, or is there some deeper meaning or reason? Also, is it correct to say that when the compiler sees

```
1 int something(<value>)
```

, it knows we're talking about a variable, while when it sees

```
1 int something(<type> <value>)
```

, it instead recognises it as a function?

Alex
July 9, 2018 at 4:44 pm · Reply

> Is it merely a coincidence that "int" is used for both initialising integers and functions, or is there some deeper meaning or reason?

int has nothing to do with initialization. int defines a variable as an integer type, or in the case of a function, that the function either returns or accepts an integer type.

> Also, is it correct to say that when the compiler sees

```
1 int something(<value>)
```

, it knows we're talking about a variable, while when it sees

```
1 int something(<type> <value>)
```

, it instead recognises it as a function?

You have the right idea, though your syntax isn't correct. int something() would definitely be a variable that is being initialized to . something() would be a function call. int something() would be a function declaration.

Ahmed Abbas sd
July 4, 2018 at 5:15 am · Reply

hello everyone
so i wanted to see the difference between the floating points
but once i run the program
i don't get decimals, all i get is integers
only time i got decimal was from the last line of code in main()

these are my results

int = 3
double = 25
long double = 3
input a number
99
input a number again
28
I devided them together, their result is = 3
I devided thier result with long double & the answer is: 1
I devided double by long double & the answer is: 8.33333
Press any key to continue . . .

```
#include "stdafx.h"
#include <iostream>

long double devide(int x, int y)
{
    return x / y;
}

void main()
{

    int x= 10 / 3;
    double y(50 / 2);
    long double z{ 10 / 3 };

    std::cout << "int = " << x << std::endl;
    std::cout << "double = " << y << std::endl;
    std::cout << "long double = " << z << std::endl;

    std::cout << "input a number " << std::endl;
    int input1;
    std::cin >> input1;
    std::cout << "input a number again " << std::endl;
    int input2;
    std::cin >> input2;

    std::cout << "I devided them together, their result is = " << devide(input1, input2) <
< std::endl;
    std::cout << "I devided thier result with long double & the answer is: " << devide(inp
ut1, input2) / z << std::endl;
    std::cout << "I devided double by long double & the answer is: " << y / z << std::endl
;

}
```

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

Hi Ahmed!

@devide divides int by int, which results in an int which is then implicitly casted to a long double when you return it. If you want a floating point result, at least one of the parts of the division has to be a floating point number.
Same thing in line 13 and 14. 50, 2, 10 and 3 are integers. If you want doubles you need to use 50.0, 2.0, 10.0 and 3.0.

roman
June 14, 2018 at 8:38 pm · Reply

Hi there, Thank you for your attention. In the Variables instantiation section, the following statement is confusing to me:

"Note that the equals sign here is just part of the syntax, and is not the same equals sign used to assign a value once the variable has been created."

Are there two different = signs? .....

Alex

June 18, 2018 at 1:30 pm · Reply

No, it's really poor wording on my part.

I'm trying the get the point across that when you're doing an assignment, operator= (the assignment operator) is invoked. When you're doing an initialization, it isn't -- the equals sign is just considered part of the initialization syntax.

This is most likely not that interesting now, but it does come up in programmer interviews, and becomes more relevant later once you learn about classes.

I've reworded the sentence to try and better get the point across. Thanks for the feedback!

Michael Centeno
May 31, 2018 at 7:26 am · Reply

"When we declare a variable as an integer, we are telling the compiler “the piece of memory that this variable addresses is going to be interpreted as a whole number”." A negative integer isn't a whole number, so this should be corrected :)

Alex
June 1, 2018 at 4:56 pm · Reply

Fixed. Thanks!

Brian Gaucher
May 1, 2018 at 12:13 pm · Reply

In your table with the primitive data types. You have a notes section. Nice, but It might be worth adding C++17 to the notes for integer types
long long is C99/C++11 only
to
long long is C99/C++11 and upwards only
unless it doesn't exist in C++17. I'm just guessing you forgot to add it.

Alex
May 6, 2018 at 9:15 pm · Reply

Updated to note that these were introduced in C99/C++11. Thanks for the feedback!

Joe
March 30, 2018 at 10:48 am · Reply

I am at my wits end.

For academic purposes I want to specifically print out the garbage at a location. I have looked up how to do this and on multiple web pages it states the same, declare an uninitialized variable and then cout the result but I keep getting a compile error.

On multiple occasions I have written a function that returned garbage completely by accident, and now for the life of me I can not remember how I did it.

so, my dilemma, how do I make the compiler print the value for an uninitialized variable?

```
1 int main () {
```

```
2
3  int x;
4  std::cout << x;
5
6  return 0;
7
8  }
```

This is what I did, and what I have observed on other web pages.
Namely here:
https://stackoverflow.com/questions/29704710/why-are-memory-locations-assigned-garbage-values

nascardriver
March 30, 2018 at 11:10 am · Reply

Hi Joe!

There are no errors in your code, all you should get is a compile time warning. Which errors are you getting?

Your code will most likely print 0, but then again, it's undefined, so you might get lucky. Make sure to build in release mode as some compilers initialize memory to a certain value automatically in debug mode.

joe
March 30, 2018 at 3:35 pm · Reply

Its just a compile time warning.

There were times though in which it would print out like i said with a garbage value when i goofed on returning a value from a function. I just can not remember how i did it. Now that I'm trying to goof it's not working. Lol.

Ultimately i was trying to use a random garbage value as a seed for an rng to see how that would work but since i can't, I'll guess i try something else.

Alex
March 31, 2018 at 3:45 pm · Reply

Here's one way to do so using pointers:

```
1  int main()
2  {
3      int *ptr = new int;
4      delete ptr;
5      std::cout << *ptr;
6
7      return 0;
8  }
```

When I ran this in release mode a bunch of times, I got a bunch of 0's, and then some random large numbers.

Georges Theodosiou
March 30, 2018 at 6:46 am · Reply

Dear Teacher, please let me this question: In section "Uniform initialization in C++11" you suggest: "Rule: If you're using a C++11 compatible compiler, favor uniform initialization". Do you suggest it for C++17 compatible compiler? Regards.

nascardriver
March 30, 2018 at 7:40 am · Reply

Hi Georges!

A C++17 compiler is C++11 compatible, so yes, use uniform initialization.

Linyuan
February 5, 2018 at 2:38 pm · Reply

Hello!
i am very confused with this rule in the text:

"Rule: Always initialize your fundamental variables, or assign a value to them as soon as possible after defining them."

For a function, declaration is of cause different from definition, Declaration has actually the same meaning with the so called "function prototype". We just give out the return type and "head" of function in XXX.h. Definition of a function means that we write the body of this function in XXX.cpp.

However for a variable, Declaration of variable is some form like

```
1 int value_from_user;
```

in the header file XXX.h.

but when I try to do some definition (initialization) about this value in the XXX.cpp like this:

```
1 int value_from_user = 0;
```

Error occurs. It says "value_from_user" has been defined twice.

so here is my question: why would this happen?
according to the rule, is seems that i have to do the both declaration(definition) and initialization in xxx.h such as:

```
1 int SomeValue = 0;
2 //or in so called uniform initialization:
3 int SomeValue{0};
```

but yeah, that works when i do this in XXX.h

What these three words "declaration", "definition" and "initialization" for a variable exactly mean???

my opinion: for variable, declaration and definition are similar(almost the same in fact):

```
1 int x;
```

and initialization is:

```
1 int x = 4;
```

so should i put this initialization in XXX.h or in a XXX.cpp???

Alex
February 6, 2018 at 11:36 am · Reply

> However for a variable, Declaration of variable is some form like: int value_from_user;

This is a definition (and a declaration). In order for it to be just a declaration, you need to add the keyword extern.

> It says "value_from_user" has been defined twice.

That's because of the definition of the variable in the header (that you thought was a declaration).

Declaration: Tells the compiler about the name and the type of a variable or function, but does not tell the compiler where or how it's implemented.
Definition: Tells the linker about the full specification of a variable or function, so it can be instantiated (for variables) or converted into code (functions)
Initialization: Assigning an initial value to a variable at the point of creation/instantiation.

Your statements under "my opinion" are correct.

Non-const variables should generally be defined in XXX.cpp.
Const variables can be defined in either XXX.cpp or XXX.h.

I talk a lot more about these topics in chapter 4, particularly lessons 4.1 to 4.3.

Sirius
January 21, 2018 at 4:08 pm · Reply

Hi,
Is it good practice to be initialising variables just before they are assigned a value from cin? For example is

```
int x{ 0 };
    std::cin >> x;
```

preferable over

```
int x;
    cin >> x ;
```

Or are they given values so soon afterwards that it's irrelevant? Thanks

nascardriver
January 22, 2018 at 6:50 am · Reply

Hi Sirius!
Yes please, having uninitialized variables can cause undefined behavior.

Alex
January 24, 2018 at 4:26 pm · Reply

Most often, if I'm going to immediately ask the user to enter a value for a variable defined immediately preceding the input, I don't initialize the variable. It seems unnecessary.

Prior to C++11, a failed extraction from std::cin could leave the variable uninitialized. If you then tried to use that variable, you'd be accessing an uninitialized value, and who knows what you'd get. As of C++11, a failed extraction will zero the variable, so initializing beforehand now seems unnecessary. But some people prefer to do so anyway, out of habit.

« Older Comments 1 2 3