1.4 — Variable assignment and initialization

BY ALEX ON FEBRUARY 1ST, 2019 | LAST MODIFIED BY ALEX ON JANUARY 8TH, 2020

In the previous lesson (**1.3 -- Introduction to variables**), we covered how to define a variable that we can use to store values. In this lesson, we'll explore how to actually put values into variables and use those values.

As a reminder, here's a short snippet that first allocates a single integer variable named *x*, then allocates two more integer variables named *y* and *z*:

```
int x; // define an integer variable named x
int y, z; // define two integer variables, named y and z
```

Variable assignment

After a variable has been defined, you can give it a value (in a separate statement) using the = *operator*. This process is called **copy assignment** (or just **assignment**) for short.

```
int width; // define an integer variable named width
width = 5; // copy assignment of value 5 into variable width

// variable width now has value 5
```

Copy assignment is named such because it copies the value on the right-hand side of the = *operator* to the variable on the left-hand side of the operator. The = *operator* is called the **assignment operator**.

Here's an example where we use assignment twice:

```
#include <iostream>

int main()
{
    int width;
    width = 5; // copy assignment of value 5 into variable width

    // variable width now has value 5

    width = 7; // change value stored in variable width to 7

    // variable width now has value 7

    return 0;
}
```

When we assign value 7 to variable *width*, the value 5 that was there previously is overwritten. Normal variables can only hold one value at a time.

Warning

One of the most common mistakes that new programmers make is to confuse the assignment operator (=) with the equality operator (==). Assignment (=) is used to assign a value to a variable. Equality (==) is used to test whether two operands are equal in value.

Copy and direct initialization

One downside of assignment is that it requires at least two statements: one to define the variable, and one to assign the value.

These two steps can be combined. When a variable is defined, you can also provide an initial value for the variable at the same time. This is called **initialization**.

C++ supports three basic ways to initialize a variable. First, we can do **copy initialization** by using an equals sign:

```
int width = 5; // copy initialization of value 5 into variable width
```

Much like copy assignment, this copies the value on the right-hand side of the equals to the variable being created on the left-hand side.

Second, we can do **direct initialization** by using parenthesis.

```
int width( 5 ); // direct initialization of value 5 into variable width
```

For simple data types (like integers), copy and direct initialization are essentially the same. But for some advanced types, direct initialization can perform better than copy initialization. Prior to C++11, direct initialization was recommended over copy initialization in most cases because of the performance boost.

Brace initialization

Unfortunately, direct initialization can't be used for all types of initialization (such as initializing an object with a list of data). In an attempt to provide a more consistent initialization mechanism, C++11 added a new syntax called **brace initialization** (also sometimes called **uniform initialization** or **list initialization**) that uses curly braces.

Brace initialization comes in two forms:

```
int width{ 5 }; // direct brace initialization of value 5 into variable width (preferred)
int height = { 6 }; // copy brace initialization of value 6 into variable height
```

These two forms function almost identically, but the direct form is generally preferred.

Initializing a variable with empty braces indicates zero initialization. **Zero initialization** generally initializes the variable to zero (or empty, if that's more appropriate for a given type).

```
int width{}; // zero initialization to value 0
```

Brace initialization has the added benefit of disallowing "narrowing" conversions. This means that if you try to use brace initialization to initialize a variable with a value it can not safely hold, the compiler will throw an warning or an error. For example:

```
int width{ 4.5 }; // error: not all double values fit into an int
```

In the above snippet, we're trying to assign a number (4.5) that has a fractional part (the .5 part) to an integer variable (which can only hold numbers without fractional parts). Copy and direct initialization would drop the fractional part, resulting in initialization of value 4 into variable *width*. However, with brace initialization, this will cause the compiler to issue an error (which is generally a good thing, because losing data is rarely desired). Conversions that can be done without potential data loss are allowed.

Best practice

Favor direct brace initialization whenever possible.

Author's note

Many examples in this tutorial series were written before brace initialization existed and thus still use copy or direct initialization. We're working on getting these updated. Please forgive our lack of adherence to best practices in this regard.

Q: C++ provides copy, direct, and brace initialization, and copy assignment. Is there a direct or brace assignment?

No, C++ does not support a direct or brace assignment syntax.

In C++11, move assignment was added, but it uses the same syntax as copy assignment. We cover move assignment in the chapter on smart pointers.

Q: When should I initialize with { 0 } vs {}?

Use an explicit initialization value if you're actually using that value.

```
int x { 0 }; // explicit initialization to value 0
std::cout << x; // we're using that zero value
```

Use zero initialization if the value is temporary and will be replaced.

```
int x{}; // zero initialization
std::cin >> x; // we're immediately replacing that value
```

Initialize your variables

Initialize your variables upon creation. You may eventually find cases where you want to ignore this advice for a specific reason (e.g. a performance critical section of code that uses a lot of variables), and that's okay, as long the choice is made deliberately.

For more discussion on this topic, Bjarne Stroustrup (creator of C++) and Herb Sutter (C++ expert) make this recommendation themselves **here**.

We explore what happens if you try to use a variable that doesn't have a well-defined value in lesson **1.6 -- Uninitialized variables and undefined behavior**.

Best practice

Initialize your variables upon creation.

Initializing multiple variables

In the last section, we noted that it is possible to define multiple variables *of the same type* in a single statement by separating the names with a comma:

```
int a, b;
```

We also noted that best practice is to avoid this syntax altogether. However, since you may encounter other code that uses this style, it's still useful to talk a little bit more about it, if for no other reason than to reinforce some of the reasons you should be avoiding it.

You can initialize multiple variables defined on the same line:

```
int a = 5, b = 6; // copy initialization
int c( 7 ), d( 8 ); // direct initialization
int e{ 9 }, f{ 10 }; // brace initialization (preferred)
```

Unfortunately, there's a common pitfall here that can occur when the programmer mistakenly tries to initialize both variables by using one initialization statement:

```
int a, b = 5; // wrong (a is not initialized!)

int a = 5, b = 5; // correct
```

In the top statement, variable "a" will be left uninitialized, and the compiler may or may not complain. If it doesn't, this is a great way to have your program intermittently crash and produce sporadic results. We'll talk more about what happens if you use uninitialized variables shortly.

The best way to remember that this is wrong is to consider the case of direct initialization or brace initialization:

```
int a, b( 5 );
int c, d{ 5 };
```

This makes it seem a little more clear that the value 5 is only being used to initialize variable *b* or *d*, not *a* or *c*.

Quiz time

Question #1

What is the difference between initialization and assignment?

Show Solution

Question #2

What form of initialization should you be using (assume your compiler is C++11 compliant)?

Show Solution

1.5 -- Introduction to iostream: cout, cin, and endl

Index

1.3 -- Introduction to variables

C++ TUTORIAL | PRINT THIS POST

95 comments to 1.4 — Variable assignment and initialization

« Older Comments 1 2

Cuyler Brehaut
January 25, 2020 at 7:22 am · Reply

Hey, you said there wasn't such a thing as "brace assignment", but I wrote:

```
1 int x{};
2 x = { 5 };
```

and it compiled fine and ran fine.
What is this actually doing, if it isn't "brace assignment"?

nascardriver
January 26, 2020 at 1:02 am · Reply

Hi!

What you did works with scalar types and is similar to brace initialization. It disallows narrowing casts like this:

```
1 int i{};
2 i = 3.3; // ok
3 i = { 3.3 }; // no
```

But for class-types, which we'll talk about later, the two assignments do different things.

Marcin Rutkowski
January 21, 2020 at 9:22 pm · Reply

Hi ...impressive course :)
I'm far from a pro but I program c++ for last 6 months as a hobby :)

Question ...what is the consequence is we initialize variables by "not preferred way" ...as a beginner for me is important if "something" is working correctly and my code does not have to be "pretty" and if I get used to

one style I need good reason to change it so when we talking about performance boost what it means and what would be a difference between initializing 10M integers correct way between initializing 10M integers "not preferred way". Is it significant ?? or this is an agreement between pros as a best practice to easier read someone else code ??

nascardriver
January 22, 2020 at 4:53 am · Reply

There's no different in performance if you're using a somewhat recent version of C++. The difference is in versatility and safety. Brace initialization can be used almost everywhere, can zero-initialize, and doesn't allow narrowing casts.

mark
January 21, 2020 at 9:00 pm · Reply

Can you help with this issue?

Using Visual Studio Code on Mac OS Catalina. I tried the final exercise of this chapter.

```
#include <iostream>

int main ()
{
    int x{};
    std::cout << "Enter an integer:  ";
    std::cin >> x;

    int y{};
    std::cout << "Enter a second integer:  ";
    std::cin >> y;

    std::cout << x << " + " << y << " is " << x + y;
    std::cout << x << " - " << y << " is " << x - y;

    return 0;
}
```

VSCode flagged errors upon initialization of variables. I used both clang++ and g++ as compilers. I tried compiling anyway, and this was the result:

```
> Executing task: /usr/bin/g++ -g /Users/markzameda/Documents/Projects/addSubtract/addSubtr

/Users/markzameda/Documents/Projects/addSubtract/addSubtract.cpp:5:10: error: expected ';'
    int x{};
         ^
         ;
/Users/markzameda/Documents/Projects/addSubtract/addSubtract.cpp:9:10: error: expected ';'
    int y{};
         ^
         ;
2 errors generated.
The terminal process terminated with exit code: 1
```

Removal of the squiggly brackets, or introduction of parentheses with a 0 declaration (for example, a "(0)") allows compiling.

Any ideas?

nascardriver
January 22, 2020 at 4:51 am · Reply

You're using an old C++ standard, see lesson 0.12.
For g++ and clang++, add -std=c++17
eg.
clang++ -std=c++17 -your_other_options /path/to/addSubtract.cpp

Mark
January 23, 2020 at 6:29 pm · Reply

Thank you for the help. Loving the course!

« Older Comments 1 2