1.4 — Variable assignment and initialization

learncpp.com/cpp-tutorial/variable-assignment-and-initialization/

In the previous lesson (1.3 -- Introduction to objects and 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 **assignment**, and the = *operator* is called the **assignment operator**.

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

// variable width now has value 5
```

By default, assignment copies the value on the right-hand side of the = *operator* to the variable on the left-hand side of the operator. This is called **copy assignment**.

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

	std::cout << width; // prints 5

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

	std::cout << width; // prints 7

	return 0;
}
```

This prints:

57

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.

Initialization

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

These two steps can be combined. When an object is defined, you can optionally give it an initial value. The process of specifying an initial value for an object is called **initialization**, and the syntax used to initialize an object is called an **initializer**.

```
int width { 5 }; // define variable width and initialize with initial value 5

// variable width now has value 5
```

In the above initialization of variable `width`, `{ 5 }` is the initializer, and `5` is the initial value.

Different forms of initialization

Initialization in C++ is surprisingly complex, so we'll present a simplified view here.

There are 6 basic ways to initialize variables in C++:

```
int a;         // no initializer (default initialization)
int b = 5;     // initial value after equals sign (copy initialization)
int c( 6 );    // initial value in parenthesis (direct initialization)

// List initialization methods (C++11) (preferred)
int d { 7 };   // initial value in braces (direct list initialization)
int e = { 8 }; // initial value in braces after equals sign (copy list
initialization)
int f {};      // initializer is empty braces (value initialization)
```

You may see the above forms written with different spacing (e.g. `int d{7};`). Whether you use extra spaces for readability or not is a matter of personal preference.

Default initialization

When no initializer is provided (such as for variable `a` above), this is called **default initialization**. In most cases, default initialization performs no initialization, and leaves a variable with an indeterminate value.

We'll discuss this case further in lesson (1.6 -- Uninitialized variables and undefined behavior).

Copy initialization

When an initial value is provided after an equals sign, this is called **copy initialization**. This form of initialization was inherited from C.

```
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 into the variable being created on the left-hand side. In the above snippet, variable `width` will be initialized with value `5`.

Copy initialization had fallen out of favor in modern C++ due to being less efficient than other forms of initialization for some complex types. However, C++17 remedied the bulk of these issues, and copy initialization is now finding new advocates. You will also find it used in older code (especially code ported from C), or by developers who simply think it looks more natural and is easier to read.

For advanced readers

Copy initialization is also used whenever values are implicitly copied or converted, such as when passing arguments to a function by value, returning from a function by value, or catching exceptions by value.

Direct initialization

When an initial value is provided inside parenthesis, this is called **direct initialization**.

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

Direct initialization was initially introduced to allow for more efficient initialization of complex objects (those with class types, which we'll cover in a future chapter). Just like copy initialization, direct initialization had fallen out of favor in modern C++, largely due to being superseded by list initialization. However, we now know that list initialization has a few quirks of its own, and so direct initialization is once again finding use in certain cases.

For advanced readers

Direct initialization is also used when values are explicitly cast to another type.

One of the reasons direct initialization had fallen out of favor is because it makes it hard to differentiate variables from functions. For example:

```
int x();  // forward declaration of function x
int x(0); // definition of variable x with initializer 0
```

List initialization

The modern way to initialize objects in C++ is to use a form of initialization that makes use of curly braces. This is called **list initialization** (or **uniform initialization** or **brace initialization**).

List initialization comes in three forms:

```
int width { 5 };    // direct list initialization of initial value 5 into variable width
int height = { 6 }; // copy list initialization of initial value 6 into variable height
int depth {};       // value initialization (see next section)
```

As an aside…

Prior to the introduction of list initialization, some types of initialization required using copy initialization, and other types of initialization required using direct initialization. List initialization was introduced to provide a more consistent initialization syntax (which is why it is sometimes called “uniform initialization”) that works in most cases.

Additionally, list initialization provides a way to initialize objects with a list of values (which is why it is called “list initialization”). We show an example of this in lesson 16.2 -- Introduction to std::vector and list constructors.

List initialization has an added benefit: “narrowing conversions” in list initialization are ill-formed. This means that if you try to brace initialize a variable using a value that the variable can not safely hold, the compiler is required to produce a diagnostic (usually an error). For example:

```
int width { 4.5 }; // error: a number with a fractional value can't 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 simply drop the fractional part, resulting in the initialization of value 4 into variable *width*. Your compiler may optionally warn you about this, since losing data is rarely desired. However, with list initialization, your compiler is required to generate a diagnostic in such cases.

Conversions that can be done without potential data loss are allowed.

To summarize, list initialization is generally preferred over the other initialization forms because it works in most cases (and is therefore most consistent), it disallows narrowing conversions, and it supports initialization with lists of values (something we'll cover in a future lesson). While you are learning, we recommend sticking with list initialization (or value initialization).

Best practice

Prefer direct list initialization (or value initialization) for initializing your variables.

Author's note

Bjarne Stroustrup (creator of C++) and Herb Sutter (C++ expert) also recommend using list initialization to initialize your variables.

In modern C++, there are some cases where list initialization does not work as expected. We cover one such case in lesson 16.2 -- Introduction to std::vector and list constructors.

Because of such quirks, some experienced developers now advocate for using a mix of copy, direct, and list initialization, depending on the circumstance. Once you are familiar enough with the language to understand the nuances of each initialization type and the reasoning behind such recommendations, you can evaluate on your own whether you find these arguments persuasive.

Value initialization and zero initialization

When a variable is initialized using empty braces, **value initialization** takes place. In most cases, **value initialization** will initialize the variable to zero (or empty, if that's more appropriate for a given type). In such cases where zeroing occurs, this is called **zero initialization**.

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

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

```
int x {};      // value 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 as the choice is made deliberately.

Related content

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 };     // direct brace initialization
int g = { 9 }, h = { 10 }; // copy brace initialization
int i {}, j {};            // value initialization
```

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 or 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 };
```

Because the parenthesis or braces are typically placed right next to the variable name, this makes it seem a little more clear that the value 5 is only being used to initialize variable *b* and *d*, not *a* or *c*.

Unused initialized variables warnings

Modern compilers will typically generate warnings if a variable is initialized but not used (since this is rarely desirable). And if “treat warnings as errors” is enabled, these warnings will be promoted to errors and cause the compilation to fail.

Consider the following innocent looking program:

```
int main()
{
    int x { 5 }; // variable defined

    // but not used anywhere

    return 0;
}
```

When compiling this with the g++ compiler, the following error is generated:

```
prog.cc: In function 'int main()':
prog.cc:3:9: error: unused variable 'x' [-Werror=unused-variable]
```

and the program fails to compile.

There are a few easy ways to fix this.

1. If the variable really is unused, then the easiest option is to remove the defintion of `x` (or comment it out). After all, if it’s not used, then removing it won’t affect anything.
2. Another option is to simply use the variable somewhere:

```
#include <iostream>

int main()
{
    int x { 5 };

    std::cout << x; // variable now used somewhere

    return 0;
}
```

But this requires some effort to write code that uses it, and has the downside of potentially changing your program's behavior.

The `[[maybe_unused]]` attribute C++17

In some cases, neither of the above options are desirable. Consider the case where we have a bunch of math/physics values that we use in many different programs:

```
int main()
{
    double pi { 3.14159 };
    double gravity { 9.8 };
    double phi { 1.61803 };

    // assume some of the above are used here, some are not

    return 0;
}
```

If we use these a lot, we probably have these saved somewhere and copy/paste/import them all together.

However, in any program where we don't use *all* of these values, the compiler will complain about each variable that isn't actually used. While we could go through and remove/comment out the unused ones for each program, this takes time and energy. And later if we need one that we've previously removed, we'll have to go back and re-add it.

To address such cases, C++17 introduced the `[[maybe_unused]]` attribute, which allows us to tell the compiler that we're okay with a variable being unused. The compiler will not generate unused variable warnings for such variables.

The following program should generate no warnings/errors:

```
int main()
{
    [[maybe_unused]] double pi { 3.14159 };
    [[maybe_unused]] double gravity { 9.8 };
    [[maybe_unused]] double phi { 1.61803 };

    // the above variables will not generate unused variable warnings

    return 0;
}
```

Additionally, the compiler will likely optimize these variables out of the program, so they have no performance impact.

Author's note

In future lessons, we'll often define variables we don't use again, in order to demonstrate certain concepts. Making use of `[[maybe_unused]]` allows us to do so without compilation warnings/errors.

Quiz time

Question #1

What is the difference between initialization and assignment?

Show Solution

Question #2

What form of initialization should you prefer when you want to initialize a variable with a specific value?

Show Solution

Question #3

What are default initialization and value initialization? What is the behavior of each? Which should you prefer?

Show Solution