1.4 — Variable assignment and initialization

ALEX NOVEMBER 29, 2022

In the previous lesson (1.3 -- Introduction to objects and variables (https://www.learncpp.com/cpp-tutorial/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 **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.

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**. The value used to initialize a variable is called an **initializer**.

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

There are 4 basic ways to initialize variables in C++:

```
int a; // no initializer
int b = 5; // initializer after equals
sign
int c( 6 ); // initializer in parenthesis
int d { 7 }; // initializer in braces
```

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 initialization value is provided (such as for variable a above), this is called **default initialization**. In most cases, default initialization leaves a variable with an indeterminate value. We'll cover this case further in lesson (1.6 -- Uninitialized variables and undefined behavior (https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/)).

Copy initialization

When an initializer is provided after an equals sign, this is called **copy initialization**. Copy initialization was inherited from the C language.

```
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. In the above snippet, variable `width` will be initialized with value `5`.

Copy initialization is not used much in modern C++. However, you may still see it in older code, or in code written by developers who learned C first.

Direct initialization

When an initializer 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). However, like copy initialization, direct initialization is not used much in modern C++ (except for one specific case that we'll cover when we get to it).

Brace initialization

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

Brace initialization comes in three 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
int depth {}; // value initialization (see next section)
```

As an aside...

Prior to the introduction of brace initialization, some types of initialization required using copy initialization, and other types of initialization required using direct initialization. Brace 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, brace initialization provides a way to initialize objects with a list of values (which is why it is sometimes called "list initialization").

Brace initialization has an added benefit: it disallows "narrowing conversions". This means that if you try to brace initialize a variable using a value that the variable can not safely hold, the compiler will produce 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 produce a warning about this, since losing data is rarely desired). However, with brace initialization, the compiler will generate an error instead, forcing you to remedy this issue before proceeding.

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

Best practice

Favor initialization using braces whenever possible.

Value initialization and zero initialization

When a variable is initialized with 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 {}; // 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 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 (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#es20-always-initialize-an-object).

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 (https://www.learncpp.com/cpp-tutorial/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 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 and [[maybe_unused]]

Some compilers will generate warnings if a variable is initialized but not used. 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 gcc or clang, both compilers generate warnings that are promoted to errors:

```
prog.cc:3:9: error: unused variable 'x' [-Werror,-Wunused-variable]
```

and the program fails to compile.

There are a few easy ways to fix this.

The first option is to turn off "treat warning as errors" temporarily (just don't forget to turn it back on).

The second option is to simply use the variable somewhere:

```
#include <iostream>

int main()
{
    int x { 5 };

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

    return 0;
}
```

In C++17, the best solution is to use the `[[maybe_unused]]` attribute. This attribute tells the compiler to expect that the variable may not be used, so it will not generate unused variable warnings.

The following program should generate no warnings/errors, even though x is unused:

```
int main()
{
    [[maybe_unused]] int x { 5 };

    // since x is [[maybe_unused]], no warning
generated

    return 0;
}
```

Quiz time

Question #1

What is the difference between initialization and assignment?

Show Solution (javascript:void(0))

Question #2

What form of initialization should you be using?

Show Solution (javascript:void(0))

Next lesson
1.5 Introduction to iostream: cout, cin, and endl

Back to table of contents

Previous lesson
1.3 Introduction to objects and variables