1.4 — Variable assignment and initialization

ALEX JUNE 28, 2021

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*:

```
1 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.

```
1 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:

```cpp
#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
```

We'll cover the case of what happens when no initializer is provided in the next lesson (4.6 -- Uninitialized variables and undefined behavior).

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.

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.

For simple types like int, copy initialization is efficient. However, when types get more complex, copy initialization can be inefficient.

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
```

For simple data types (like int), copy and direct initialization are essentially the same. For more complicated types, direct initialization tends to be more efficient than copy initialization.

List initialization

Unfortunately, direct initialization can't be used for all types of initialization (such as initializing an object with a list of data). To provide a more consistent initialization mechanism, there's list initialization (also sometimes called uniform initialization or brace initialization) that uses curly braces.

List initialization comes in three forms:

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

Direct and copy list initialization function almost identically, but the direct form is generally preferred.

List initialization has the added benefit of disallowing "narrowing" conversions. This means that if you try to use list initialization to initialize a variable with a value it can not safely hold, the compiler will throw a 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 list 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 initialization using braces whenever possible.

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

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

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.

We explore what happens if you try to use a variable that doesn't have a well-defined value in lesson1.6 -- 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 }; // list 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 list 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?

Show Solution

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

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

206 COMMENTS

Newest