1.3 — A first look at variables, initialization, and assignment

BY ALEX ON MAY 30TH, 2007 | LAST MODIFIED BY ALEX ON MAY 19TH, 2018

Objects

C++ programs create, access, manipulate, and destroy objects. An **object** is a piece of memory that can be used to store values. You can think of an object as a mailbox, or a cubbyhole, where we can store and retrieve information. All computers have memory, called RAM (random access memory), that is available for programs to use. When an object is defined, a piece of that memory is set aside for the object.

Most of the objects that we use in C++ come in the form of variables.

Variables

A statement such as `x = 5;` seems obvious enough. As you might guess, we are assigning the value of 5 to x. But what exactly is x? x is a variable.

A **variable** in C++ is simply an object that has a name.

In this section, we are only going to consider integer variables. An **integer** is a number that can be written without a fractional component, such as -12, -1, 0, 4, or 27. An integer variable is a variable that holds an integer value.

In order to create a variable, we generally use a special kind of declaration statement called a **definition** (we'll explain the precise difference between a declaration and a definition later). Here's an example of defining variable x as an integer variable (one that can hold integer values):

```
1 int x;
```

When this statement is executed by the CPU, a piece of memory from RAM will be set aside (called **instantiation**). For the sake of example, let's say that the variable x is assigned memory location 140. Whenever the program sees the variable x in an expression or statement, it knows that it should look in memory location 140 to get the value.

One of the most common operations done with variables is assignment. To do this, we use the assignment operator, more commonly known as the = symbol. For example:

```
1 x = 5;
```

When the CPU executes this statement, it translates this to "put the value of 5 in memory location 140".

Later in our program, we could print that value to the screen using std::cout:

```
1 std::cout << x;  // prints the value of x (memory location 140) to the console
```

l-values and r-values

In C++, variables are a type of l-value (pronounced ell-value). An **l-value** is a value that has a persistent address (in memory). Since all variables have addresses, all variables are l-values. The name l-value came about because l-values are the only values that can be on the left side of an assignment statement. When we do an assignment, the left hand side of the assignment operator must be an l-value. Consequently, a statement like `5 = 6;` will cause a compile error, because 5 is not an l-value. The value of 5 has no memory, and thus nothing can be assigned to it. 5 means 5, and its value can not be reassigned. When an l-value has a value assigned to it, the current value at that memory address is overwritten.

The opposite of l-values are r-values (pronounced arr-values). An **r-value** refers to values that are not associated with a persistent memory address. Examples of r-values are single numbers (such as 5, which evaluates to 5) and expressions (such as 2 + x, which evaluates to the value of variable x plus 2). r-values are generally temporary in nature and are discarded at the end of the statement in which they occur.

Here is an example of some assignment statements, showing how the r-values evaluate:

```
1 int y;      // define y as an integer variable
2 y = 4;      // r-value 4 evaluates to 4, which is then assigned to l-value y
3 y = 2 + 5;  // r-value 2 + r-value 5 evaluates to r-value 7, which is then assigned to l-value y
```

```
int x;      // define x as an integer variable
x = y;      // l-value y evaluates to 7 (from before), which is then assigned to l-value x.
x = x;      // l-value x evaluates to 7, which is then assigned to l-value x (useless!)
x = x + 1;  // l-value x + r-value 1 evaluate to r-value 8, which is then assigned to l-value x.
```

Let's take a closer look at the last assignment statement above, since it causes the most confusion.

```
x = x + 1;
```

In this statement, the variable x is being used in two different contexts. On the left side of the assignment operator, "x" is being used as an l-value (variable with an address) in which to store a value. On the right side of the assignment operator, x is evaluated to produce a value (in this case, 7). When C++ evaluates the above statement, it evaluates as:

```
x = 7 + 1;
```

Which makes it obvious that C++ will assign the value 8 back into variable x.

For the time being, you don't need to worry about l-values or r-values much, but we'll return to them later when we start discussing some more advanced topics.

The key takeaway here is that on the left side of the assignment, you must have something that represents a memory address (such as a variable). Everything on the right side of the assignment will be evaluated to produce a value.

Initialization vs. assignment

C++ supports two related concepts that new programmers often get mixed up: assignment and initialization.

After a variable is defined, a value may be **assigned** to it via the assignment operator (the = symbol):

```
int x; // this is a variable definition
x = 5; // assign the value 5 to variable x
```

C++ will let you both define a variable AND give it an initial value in the same step. This is called **initialization**.

```
int x = 5; // initialize variable x with the value 5
```

A variable can only be initialized when it is defined.

Although these two concepts are similar in nature, and can often be used to achieve similar ends, we'll see cases in future lessons where some types of variables require an initialization value, or disallow assignment. For these reasons, it's useful to make the distinction now.

Rule: When giving variables an initial value, favor initialization over assignment.

Uninitialized variables

Unlike some programming languages, C/C++ does not initialize most variables to a given value (such as zero) automatically. Thus when a variable is assigned a memory location by the compiler, the default value of that variable is whatever (garbage) value happens to already be in that memory location! A variable that has not been given a known value (through initialization or assignment) is called an **uninitialized variable**.

Note: Some compilers, such as Visual Studio, *will* initialize the contents of memory when you're using a debug build configuration. This will not happen when using a release build configuration.

Using the values of uninitialized variables can lead to unexpected results. Consider the following short program:

```
// #include "stdafx.h" // Uncomment if Visual Studio user
#include <iostream>

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized

    // print the value of x to the screen (dangerous, because x is uninitialized)
    std::cout << x;
```

```
    return 0;
}
```

In this case, the computer will assign some unused memory to x. It will then send the value residing in that memory location to std::cout, which will print the value. But what value will it print? The answer is “who knows!”, and the answer may change every time you run the program. When the author ran this program with the Visual Studio 2013 compiler, std::cout printed the value `7177728` one time, and `5277592` the next.

If you want to run this program yourself, make sure you’re using a release build configuration (see section **0.6a -- Build configurations** for information on how to do that). Otherwise the above program may print whatever value your compiler is initializing memory with (Visual Studio uses -858993460).

If your compiler won’t let you run this program because it flags variable x as an uninitialized variable, here is a possible solution to get around this issue:

```
// #include "stdafx.h" // Uncomment if Visual Studio user
#include <iostream>

void doNothing(const int &x)
{
}

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized

    doNothing(x); // make compiler think we're using this variable

    // print the value of x to the screen (dangerous, because x is uninitialized)
    std::cout << x;

    return 0;
}
```

Using uninitialized variables is one of the most common mistakes that novice programmers make, and unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized value happened to get assigned to a spot of memory that had a reasonable value in it, like 0).

Fortunately, most modern compilers will print warnings at compile-time if they can detect a variable that is used without being initialized. For example, compiling the above program on Visual Studio 2005 express produced the following warning:

```
c:vc2005projectstesttesttest.cpp(11) : warning C4700: uninitialized local variable 'x' used
```

A good rule of thumb is to initialize your variables. This ensures that your variable will always have a consistent value, making it easier to debug if something goes wrong somewhere else.

Rule: Make sure all of your variables have known values (either through initialization or assignment).

Undefined behavior

Using the value from an uninitialized variable is our first example of undefined behavior. **Undefined behavior** is the result of executing code whose behavior is not well defined by the language. In this case, the C++ language doesn’t have any rules determining what happens if you use value of a variable that has not been given a known value. Consequently, if you actually do this, undefined behavior will result.

Code implementing undefined behavior may exhibit any of the following symptoms:

- Your program produces an consistently incorrect result.
- Your program produces different results every time it is executed.
- Your program behaves inconsistently.
- Your program seems like its working but produces incorrect results later in the program.
- Your program crashes, either immediately or later.
- Your program works on some compilers but not others.

- Your program works until you change some other unrelated code.

Or, your code may actually produce the correct behavior anyway. The nature of undefined behavior is that you never quite know what you're going to get, whether you'll get it every time, and whether it'll change when you make other changes.

C++ contains many cases that can result in undefined behavior if you're not careful. We'll ensure we point these out in future lessons. Take note of where these cases are and make sure you avoid them.

Rule: Take care to avoid situations that result in undefined behavior.

Quiz
What values does this program print?

```
1  int x = 5; // this is an initialization
2  x = x - 2; // this is an assignment
3  std::cout << x << std::endl; // #1
4
5  int y = x;
6  std::cout << y << std::endl; // #2
7
8  // x + y is an r-value in this context, so evaluate their values
9  std::cout << x + y << std::endl; // #3
10
11 std::cout << x << std::endl; // #4
12
13 int z;
14 std::cout << z << std::endl; // #5
```

6) What is undefined behavior?

Quiz Answers

To see these answers, select the area below with your mouse.

1) **Hide Solution**

The program outputs 3. x - 2 evaluates to 3, which was assigned to x.

2) **Hide Solution**

The program outputs 3. y was assigned the value of x, which evaluated to 3.

3) **Hide Solution**

The program outputs 6. x + y evaluates to 6. There was no assignment here.

4) **Hide Solution**

The program outputs 3. The value of x is still 3 because it was never reassigned.

5) **Hide Solution**

The output is indeterminate. z is an uninitialized variable.

6) **Hide Solution**

Undefined behavior occurs when the program implements code whose behavior is not well defined by the language. When code with undefined behavior is implemented, the program may exhibit unexpected results.

.

1.3a -- A first look at cout, cin, and endl