6.11 — Reference variables

BY ALEX ON JULY 16TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 31ST, 2018

So far, we've discussed two basic variable types:

- Normal variables, which hold values directly.
- Pointers, which hold the address of another value (or null) and can be dereferenced to retrieve the value at the address they point to.

References are the third basic type of variable that C++ supports. A **reference** is a type of C++ variable that acts as an alias to another object or value.

C++ supports three kinds of references:

1. References to non-const values (typically just called "references", or "non-const references"), which we'll discuss in this lesson.
2. References to const values (often called "const references"), which we'll discuss in the next lesson.
3. C++11 added r-value references, which we cover in detail in the chapter on move semantics.

References to non-const values

A reference (to a non-const value) is declared by using an ampersand (&) between the reference type and the variable name:

```
int value = 5; // normal integer
int &ref = value; // reference to variable value
```

In this context, the ampersand does not mean "address of", it means "reference to".

References to non-const values are often just called "references" for short.

References as aliases

References generally act identically to the values they're referencing. In this sense, a reference acts as an alias for the object being referenced.

Let's take a look at references in use:

```
#include <iostream>

int main()
{
    int value = 5; // normal integer
    int &ref = value; // reference to variable value

    value = 6; // value is now 6
    ref = 7; // value is now 7

    std::cout << value; // prints 7
    ++ref;
    std::cout << value; // prints 8

    return 0;
}
```

This code prints:

7
8

In the above example, ref and value are treated synonymously.

Using the address-of operator on a reference returns the address of the value being referenced:

```
cout << &value; // prints 0012FF7C
cout << &ref; // prints 0012FF7C
```

Just as you would expect if ref is acting as an alias for the value.

A brief review of l-values and r-values

Way back in lesson **1.3 -- A first look at variables, initialization, and assignment**, we talked about l-values and r-values, and told you not to worry too much about them. We're finally at a point where it becomes useful to use those terms again.

To recap, l-values are objects that have a defined memory address (such as variables), and persist beyond a single expression. r-values are temporary values that do not have a defined memory address, and only have expression scope. R-values include both the results of expressions (e.g. 2 + 3) and literals.

References must be initialized

References must be initialized when created:

```
int value = 5;
int &ref = value; // valid reference, initialized to variable value

int &invalidRef; // invalid, needs to reference something
```

Unlike pointers, which can hold a null value, there is no such thing as a null reference.

References to non-const values can only be initialized with non-const l-values. They can not be initialized with const l-values or r-values.

```
int x = 5;
int &ref1 = x; // okay, x is an non-const l-value

const int y = 7;
int &ref2 = y; // not okay, y is a const l-value

int &ref3 = 6; // not okay, 6 is an r-value
```

Note that in the middle case, you can't initialize a non-const reference with a const object -- otherwise you'd be able to change the value of the const object through the reference, which would violate the const-ness of the object.

References can not be reassigned

Once initialized, a reference can not be changed to reference another variable. Consider the following snippet:

```
int value1 = 5;
int value2 = 6;

int &ref = value1; // okay, ref is now an alias for value1
ref = value2; // assigns 6 (the value of value2) to value1 -- does NOT change the reference!
```

Note that the second statement may not do what you might expect! Instead of reassigning ref to reference variable value2, it instead assigns the value from value2 to value1 (which ref is a reference of).

References as function parameters

References are most often used as function parameters. In this context, the reference parameter acts as an alias for the argument, and no copy of the argument is made into the parameter. This can lead to better performance if the argument is large or expensive to copy.

In lesson **6.8 -- Pointers and arrays** we talked about how passing a pointer argument to a function allows the function to dereference the pointer to modify the argument's value directly.

References work similarly in this regard. Because the reference parameter acts as an alias for the argument, a function that uses a reference parameter is able to modify the argument passed in:

```
#include <iostream>

// ref is a reference to the argument passed in, not a copy
```

```
void changeN(int &ref)
{
    ref = 6;
}

int main()
{
    int n = 5;

    std::cout << n << '\n';

    changeN(n); // note that this argument does not need to be a reference

    std::cout << n << '\n';
    return 0;
}
```

This program prints:

```
5
6
```

When argument n is passed to the function, the function parameter ref is set as a reference to argument n. This allows the function to change the value of n through ref! Note that n does not need to be a reference itself.

Best practice: Pass arguments by non-const reference when the argument needs to be modified by the function.

The primary downside of using non-const references as function parameters is that the argument must be a non-const l-value. This can be restrictive. We'll talk more about this (and how to get around it) in the next lesson.

Using references to pass C-style arrays to functions

One of the most annoying issues with C-style arrays is that in most cases they decay to pointers when evaluated. However, if a C-style array is passed by reference, this decaying does not happen.

Here's an example (h/t to reader nascardriver):

```
#include <iostream>

// Note: You need to specify the array size in the function declaration
void printElements(int (&arr)[4])
{
  int length{ sizeof(arr) / sizeof(arr[0]) }; // we can now do this since the array won't decay

  for (int i{ 0 }; i < length; ++i)
  {
    std::cout << arr[i] << std::endl;
  }
}

int main()
{
    int arr[]{ 99, 20, 14, 80 };

    printElements(arr);

    return 0;
}
```

Note that in order for this to work, you explicitly need to define the array size in the parameter.

References as shortcuts

A secondary (much less used) use of references is to provide easier access to nested data. Consider the following struct:

```
struct Something
```

```
{
    int value1;
    float value2;
};

struct Other
{
    Something something;
    int otherValue;
};

Other other;
```

Let's say we needed to work with the value1 field of the Something struct of other. Normally, we'd access that member as `other.something.value1`. If there are many separate accesses to this member, the code can become messy. References allow you to more easily access the member:

```
int &ref = other.something.value1;
// ref can now be used in place of other.something.value1
```

The following two statements are thus identical:

```
other.something.value1 = 5;
ref = 5;
```

This can help keep your code cleaner and more readable.

References vs pointers

References and pointers have an interesting relationship -- a reference acts like a pointer that is implicitly dereferenced when accessed (references are usually implemented internally by the compiler using pointers). Thus given the following:

```
int value = 5;
int *const ptr = &value;
int &ref = value;
```

*ptr and ref evaluate identically. As a result, the following two statements produce the same effect:

```
*ptr = 5;
ref = 5;
```

Because references must be initialized to valid objects (cannot be null) and can not be changed once set, references are generally much safer to use than pointers (since there's no risk of dereferencing a null pointer). However, they are also a bit more limited in functionality accordingly.

If a given task can be solved with either a reference or a pointer, the reference should generally be preferred. Pointers should only be used in situations where references are not sufficient (such as dynamically allocating memory).

Summary

References allow us to define aliases to other objects or values. References to non-const values can only be initialized with non-const l-values. References can not be reassigned once initialized.

References are most often used as function parameters when we either want to modify the value of the argument, or when we want to avoid making an expensive copy of the argument.

 **6.11a -- References and const**

 **Index**