12.5 — Pass by lvalue reference

learncpp.com/cpp-tutorial/pass-by-lvalue-reference/

In the previous lessons, we introduced lvalue references (12.3 -- Lvalue references) and lvalue references to const (12.4 -- Lvalue references to const). In isolation, these may not have seemed very useful -- why create an alias to a variable when you can just use the variable itself?

In this lesson, we'll finally provide some insight into what makes references useful. And then starting later in this chapter, you'll see references used regularly.

First, some context. Back in lesson 2.4 -- Introduction to function parameters and arguments we discussed `pass by value`, where an argument passed to a function is copied into the function's parameter:

```
#include <iostream>

void printValue(int y)
{
    std::cout << y << '\n';
} // y is destroyed here

int main()
{
    int x { 2 };

    printValue(x); // x is passed by value (copied) into parameter y (inexpensive)

    return 0;
}
```

In the above program, when `printValue(x)` is called, the value of `x` (`2`) is *copied* into parameter `y`. Then, at the end of the function, object `y` is destroyed.

This means that when we called the function, we made a copy of our argument's value, only to use it briefly and then destroy it! Fortunately, because fundamental types are cheap to copy, this isn't a problem.

Some objects are expensive to copy

Most of the types provided by the standard library (such as `std::string`) are `class types`. Class types are usually expensive to copy. Whenever possible, we want to avoid making unnecessary copies of objects that are expensive to copy, especially when we will destroy those copies almost immediately.

Consider the following program illustrating this point:

```
#include <iostream>
#include <string>

void printValue(std::string y)
{
    std::cout << y << '\n';
} // y is destroyed here

int main()
{
    std::string x { "Hello, world!" }; // x is a std::string

    printValue(x); // x is passed by value (copied) into parameter y (expensive)

    return 0;
}
```

This prints

```
Hello, world!
```

While this program behaves like we expect, it's also inefficient. Identically to the prior example, when `printValue()` is called, argument `x` is copied into `printValue()` parameter `y`. However, in this example, the argument is a `std::string` instead of an `int`, and `std::string` is a class type that is expensive to copy. And this expensive copy is made every time `printValue()` is called!

We can do better.

Pass by reference

One way to avoid making an expensive copy of an argument when calling a function is to use `pass by reference` instead of `pass by value`. When using **pass by reference**, we declare a function parameter as a reference type (or const reference type) rather than as a normal type. When the function is called, each reference parameter is bound to the appropriate argument. Because the reference acts as an alias for the argument, no copy of the argument is made.

Here's the same example as above, using pass by reference instead of pass by value:

```
#include <iostream>
#include <string>

void printValue(std::string& y) // type changed to std::string&
{
    std::cout << y << '\n';
} // y is destroyed here

int main()
{
    std::string x { "Hello, world!" };

    printValue(x); // x is now passed by reference into reference parameter y
(inexpensive)

    return 0;
}
```

This program is identical to the prior one, except the type of parameter `y` has been changed from `std::string` to `std::string&` (an lvalue reference). Now, when `printValue(x)` is called, lvalue reference parameter `y` is bound to argument `x`. Binding a reference is always inexpensive, and no copy of `x` needs to be made. Because a reference acts as an alias for the object being referenced, when `printValue()` uses reference `y`, it's accessing the actual argument `x` (rather than a copy of `x`).

Key insight

Pass by reference allows us to pass arguments to a function without making copies of those arguments each time the function is called.

Pass by reference allows us to change the value of an argument

When an object is passed by value, the function parameter receives a copy of the argument. This means that any changes to the value of the parameter are made to the copy of the argument, not the argument itself:

```
#include <iostream>

void addOne(int y) // y is a copy of x
{
    ++y; // this modifies the copy of x, not the actual object x
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x has not been modified

    return 0;
}
```

In the above program, because value parameter `y` is a copy of `x`, when we increment `y`, this only affects `y`. This program outputs:

```
value = 5
value = 5
```

However, since a reference acts identically to the object being referenced, when using pass by reference, any changes made to the reference parameter *will* affect the argument:

```
#include <iostream>

void addOne(int& y) // y is bound to the actual object x
{
    ++y; // this modifies the actual object x
}

int main()
{
    int x { 5 };

    std::cout << "value = " << x << '\n';

    addOne(x);

    std::cout << "value = " << x << '\n'; // x has been modified

    return 0;
}
```

This program outputs:

```
value = 5
value = 6
```

In the above example, `x` initially has value `5`. When `addOne(x)` is called, reference parameter `y` is bound to argument `x`. When the `addOne()` function increments reference `y`, it's actually incrementing argument `x` from `5` to `6` (not a copy of `x`). This changed value persists even after `addOne()` has finished executing.

Key insight

Passing values by reference to non-const allows us to write functions that modify the value of arguments passed in.

The ability for functions to modify the value of arguments passed in can be useful. Imagine you've written a function that determines whether a monster has successfully attacked the player. If so, the monster should do some amount of damage to the player's health. If you pass your player object by reference, the function can directly modify the health of the actual player object that was passed in. If you pass the player object by value, you could only modify the health of a copy of the player object, which isn't as useful.

Pass by reference can only accept modifiable lvalue arguments

Because a reference to a non-const value can only bind to a modifiable lvalue (essentially a non-const variable), this means that pass by reference only works with arguments that are modifiable lvalues. In practical terms, this significantly limits the usefulness of pass by reference to non-const, as it means we can not pass const variables or literals. For example:

```
#include <iostream>

void printValue(int& y) // y only accepts modifiable lvalues
{
    std::cout << y << '\n';
}

int main()
{
    int x { 5 };
    printValue(x); // ok: x is a modifiable lvalue

    const int z { 5 };
    printValue(z); // error: z is a non-modifiable lvalue

    printValue(5); // error: 5 is an rvalue

    return 0;
}
```

Fortunately, there’s an easy way around this, which we will discuss next lesson. We’ll also take a look at when to pass by value vs. pass by reference.