9.5 — Pass by lvalue reference

ALEX JANUARY 26, 2022

In the previous lessons, we introduced lvalue references (9.3 -- Lvalue references) and lvalue references to const (9.4 -- 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 2.3 -- 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:

```cpp
#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:

```cpp
#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:

```cpp
#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:

```cpp
#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:

```
5
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:

```cpp
#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:

```
5
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 to non-const 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:

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

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.

Pass by const reference

Unlike a reference to non-const (which can only bind to modifiable lvalues), a reference to const can bind to modifiable lvalues, non-modifiable lvalues, and rvalues. Therefore, if we make our reference parameter const, then it will be able to bind to any type of argument:

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

void printValue(const int& y) // y is now a const reference
{
    std::cout << y << '\n';
}

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

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

    printValue(5); // ok: 5 is a literal rvalue

    return 0;
}
```

Passing by const reference offers the same primary benefit as pass by reference (avoiding making a copy of the argument), while also guaranteeing that the function can *not* change the value being referenced.

For example, the following is disallowed, because `ref` is const:

```cpp
void addOne(const int& ref)
{
    ++ref; // not allowed: ref is const
}
```

In most cases, we don't want our functions modifying the value of arguments.

Best practice

Favor passing by const reference over passing by non-const reference unless you have a specific reason to do otherwise (e.g. the function needs to change the value of an argument).

Now we can understand the motivation for allowing const lvalue references to bind to rvalues: without that capability, there would be no way to pass literals (or other rvalues) to functions that used pass by reference!

Mixing pass by value and pass by reference

A function with multiple parameters can determine whether each parameter is passed by value or passed by reference individually.

For example:

```cpp
#include <string>

void foo(int a, int& b, const std::string& c)
{
}

int main()
{
    int x { 5 };
    const std::string s { "Hello, world!" };

    foo(5, x, s);

    return 0;
}
```

In the above example, the first argument is passed by value, the second by reference, and the third by const reference.

When to pass by reference

Because class types can be expensive to copy (sometimes significantly so), class types are usually passed by const reference instead of by value to avoid making an expensive copy of the argument. Fundamental types are cheap to copy, so they are typically passed by value.

Best practice

Pass fundamental types by value, and class (or struct) types by const reference.

The cost of pass by value vs pass by reference (advanced)

Not all class types need to be passed by reference. And you may be wondering why we don't just pass everything by reference. In this section (which is optional reading), we discuss the cost of pass by value vs pass by reference, and refine our best practice as to when we should use each.

There are two key points that will help us understand when we should pass by value vs pass by reference:

First, the cost of copying an object is generally proportional to two things:

- The size of the object. Object that use more memory take more time to copy.
- Any additional setup costs. Some class types do additional setup when they are instantiated (e.g. such as opening a file or database, or allocating a certain amount of dynamic memory to hold an object of a variable size). These setup costs must be paid each time an object is copied.

On the other hand, binding a reference to an object is always fast (about the same speed as copying a fundamental type).

Second, accessing an object through a reference is slightly more expensive than accessing an object through a normal variable identifier. With a variable identifier, the compiler can just go to the memory address assigned to that variable and access the value. With a reference, there usually is an extra step: the compiler must first determine which object is being referenced, and only then can it go to that memory address for that object and access the value. The compiler can also sometimes optimize code using objects passed by value more highly than code using objects passed by reference. This means code generated for objects passed by reference is typically slower than the code generated for objects passed by value.

We can now answer the question of why we don't pass everything by reference:

- For objects that are cheap to copy, the cost of copying is similar to the cost of binding, so we favor pass by value so the code generated will be faster.
- For objects that are expensive to copy, the cost of the copy dominates, so we favor pass by (const) reference to avoid making a copy.

Best practice

Prefer pass by value for objects that are cheap to copy, and pass by const reference for objects that are expensive to copy. If you're not sure whether an object is cheap or expensive to copy, favor pass by const reference.

The last question then is, how do we define "cheap to copy"? There is no absolute answer here, as this varies by compiler, use case, and architecture. However, we can formulate a good rule of thumb: An object is cheap to copy if uses 2 or fewer "words" of memory (where a "word" is approximated by the size of a memory address) and it has no setup costs.

The following program defines a macro that can be used to determine if a type (or object) uses 2 or fewer memory addresses worth of memory:

```cpp
#include <iostream>

// Evaluates to true if the type (or object) uses 2 or fewer memory addresses worth of memory
#define isSmall(T) (sizeof(T) <= 2 * sizeof(void*))

struct S
{
    double a, b, c;
};

int main()
{
    std::cout << std::boolalpha; // print true or false rather than 1 or 0
    std::cout << isSmall(int) << '\n'; // true
    std::cout << isSmall(double) << '\n'; // true
    std::cout << isSmall(S) << '\n'; // false

    return 0;
}
```

As an aside…

We use a preprocessor macro here so that we can substitute in a type (normal functions disallow this).

However, it can be hard to know whether a class type object has setup costs or not. It's best to assume that most standard library classes have setup costs, unless you know otherwise that they don't.

Tip

An object of type T is cheap to copy if `sizeof(T) <= 2 * sizeof(void*)` and has no additional setup costs.

Common types that are cheap to copy include all of the fundamental types, enumerated types, and std::string_view.
Common types that are expensive to copy include std::array, std::string, std::vector, and std::ostream.

Next lesson
9.6 Introduction to pointers

Back to table of contents

Previous lesson
9.4 Lvalue references to const