12.5 — Pass by lvalue reference

ALEX[1] DECEMBER 6, 2024

In the previous lessons, we introduced lvalue references (12.3 -- Lvalue references (https://www.learncpp.com/cpp-tutorial/lvalue-references/)[2]) and lvalue references to const (12.4 -- Lvalue references to const (https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/)[3]). 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 (https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/)[4] 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:

```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.

The following program demonstrates that a value parameter is a separate object from the argument, while a reference parameter is treated as if it were the argument:

```
#include <iostream>

void printAddresses(int val, int& ref)
{
    std::cout << "The address of the value parameter is: " << &val << '\n';
    std::cout << "The address of the reference parameter is: " << &ref << '\n';
}

int main()
{
    int x { 5 };
    std::cout << "The address of x is: " << &x << '\n';
    printAddresses(x, x);

    return 0;
}
```

One run of this program produced the following output:

```
The address of x is: 0x7ffd16574de0
The address of the value parameter is: 0x7ffd16574de4
The address of the reference parameter is: 0x7ffd16574de0
```

We can see that the argument has a different address than the value parameter, meaning the value parameter is a different object. Since they have separate memory addresses, in order for the value parameter to have the same value as the argument, the argument's value must be copied into memory held by the value parameter.

On the other hand, we can see that taking the address of the reference parameter yields an address identical to that of the argument. This means that the reference parameter is being treated as if it were the same object as the argument.

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.

Next lesson
12.6 Pass by const lvalue reference

5

Back to table of contents

6

Previous lesson
12.4 Lvalue references to const

3

7

352 COMMENTS Newest

Binary Rambo

Binary Rambo
May 30, 2025 1:13 am PDT

```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;
}
```

In this example, if
void printValue(std::string& y)
is defined usin a string_view as
void printValue(std::string_view y)

it will be equivalent right ?,
but the difference will raise if we want printValue function to modify the parameter ?

0 Reply

vb101

Reply to Binary Rambo [11] May 30, 2025 1:25 pm PDT

yea !, that's right...

1 Reply

Leni

March 2, 2025 8:38 pm PST

Passing by lvalue reference avoids unnecessary copies, improving efficiency when handling expensive-to-copy objects like std::string; how does this approach impact function design in large-scale applications?

3 Reply

Gabe

Reply to Leni [12] May 19, 2025 5:12 am PDT

It doesn't really impact it, by default you should always be passing references or pointers if it's a big object. This applies to every other language (particularly the interpreted ones like Python) where every object by default is passed as a reference.

Large-scale applications could not exist if data structures and objects are being copied around every function call; the responsiveness would be so bad, that there would be no point in using them.

0 Reply

ice-shroom
February 11, 2025 2:47 am PST

Thank you, this is one of the most interesting lessons.

6 Reply

Sumit kumar (remo)
February 8, 2025 10:50 am PST

try this code:

```
#include<iostream>
using namespace std;

void printInt(const int& a){

cout<<"value: "<< a<<endl;
};

int main(){

short b {10};

const int& c = b;

--b;

cout<< "b value: "<<b<<endl;
printInt(c);

return 0;
};
```

0 Reply

AJAY.CHALLA
Reply to Sumit kumar (remo) [13] May 14, 2025 8:26 pm PDT

It displays ----------
b value = 9
value = 10
---

0 Reply

Ree
February 4, 2025 6:38 am PST

But isn't 'expensive copy' quite a reach considering advances in processing speed and power. Would pass by reference really be necessary in this case?

1 Reply

Alex Author

Reply to Ree [14] February 7, 2025 6:24 pm PST

The larger the object being passed is, the more it matters. We more precisely detail when to use each in lesson https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/

0 Reply

cpp learner

December 22, 2024 8:07 pm PST

the website is truly garbage without an adblocker. i tried using firefox - none of the ads even load correctly and messes up the layout of everything.

i would like to support you by not using an adblocker but you should really fix it.

0 Reply

AJAY.CHALLA

Reply to cpp learner [15] May 14, 2025 8:27 pm PDT

Use brave its completeley free and a very good adblocker

1 Reply

AJAY.CHALLA

Reply to AJAY.CHALLA [16] May 14, 2025 8:28 pm PDT

click on this link to download

brave (https://brave.com/download/)[17]

0 Reply

Liu Ronggui

Reply to cpp learner [15] January 5, 2025 12:45 am PST

I am using Google Chrome, and I feel that the reading environment is quite clean. There are almost no pop-up ads interrupting my reading process. I am very grateful to the author for providing this valuable free learning tutorial.

5 Reply

Lucas Smith

Reply to cpp learner [15] December 25, 2024 3:43 pm PST

All of this material is given to us 100% free. Most courses (I've found at least) don't go into the depth this website does.

Thanks Alex and nascardriver for your hard work making this course!

10 Reply

Cpp Learner

December 5, 2024 1:02 pm PST

I think in this lesson, you should print x and passed by copy variable's memory address to show how they different while, passed by references' address are same. It will be easier to visualize student's mind that both variable belongs to same memory address easier

2 Reply

Alex Author

Reply to Cpp Learner [18] December 6, 2024 9:52 am PST

Good idea. Done.

3 Reply

ImWorking

December 2, 2024 1:20 am PST

It is regularly used in games especially functional programming as a parameter.
Thanks Alex!

2 Reply

Colin

September 1, 2024 10:14 pm PDT

if we're using strings for this, would there be a difference between string view and pass by reference?

0 Reply

Alex Author

Reply to Colin [19] September 4, 2024 12:59 pm PDT

What is "for this" in your sentence?

0 Reply

Colin

Reply to Alex [20] September 4, 2024 4:07 pm PDT

I was asking about passing strings as a reference, but I've already read the part in a later chapter about string views. Thank you!

0 Reply

Joyboy
August 22, 2024 3:35 am PDT

just wanted to ask you alex, in your opinion how long do u think this whole course would take to complete for someone who is reletively serious about this

0 Reply

Mart
Reply to Joyboy [21] August 26, 2024 8:14 am PDT

For me it took 2 weeks to get here and there is a lot yet...

0 Reply

Alex Author
Reply to Joyboy [21] August 22, 2024 8:07 pm PDT

I really don't know. Most readers don't indicate how long it took to complete the course, and for the ones that have, they usually don't indicate their background or how many hours a week they were devoting.

4 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/lvalue-references/
3. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/
5. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/
6. https://www.learncpp.com/
7. https://www.learncpp.com/pass-by-lvalue-reference/
8. https://www.learncpp.com/cpp-tutorial/passing-arguments-by-value/
9. https://www.learncpp.com/cpp-tutorial/pass-by-address/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-610559
12. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-608201
13. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-607513
14. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-607372
15. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-605622
16. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-610070
17. https://brave.com/download/
18. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-604921

19. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-601503
20. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-601606
21. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-601146
22. https://g.ezoic.net/privacy/learncpp.com