12.6 — Pass by const lvalue reference

ALEX[1] NOVEMBER 21, 2024

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 a reference parameter const, then it will be able to bind to any type of argument:

```
#include <iostream>

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

int main()
{
    int x { 5 };
    printRef(x);   // ok: x is a modifiable lvalue, y binds to x

    const int z { 5 };
    printRef(z);   // ok: z is a non-modifiable lvalue, y binds to z

    printRef(5);   // ok: 5 is rvalue literal, y binds to temporary int object

    return 0;
}
```

Passing by const reference offers the same primary benefit as pass by non-const 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:

```
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!

Passing arguments of a different type to a const lvalue reference parameter

In lesson 12.4 -- Lvalue references to const (https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/)[2], we noted that a const lvalue reference can bind to a value of a different type, as long as that value is convertible to the type of the reference. The conversion creates a temporary object that the reference parameter can then bind to.

The primary motivation for allowing this is so that we can pass a value as an argument to either a value parameter or a const reference parameter in exactly the same way:

```
#include <iostream>

void printVal(double d)
{
    std::cout << d << '\n';
}

void printRef(const double& d)
{
    std::cout << d << '\n';
}

int main()
{
    printVal(5); // 5 converted to temporary double, copied to parameter d
    printRef(5); // 5 converted to temporary double, bound to parameter d

    return 0;
}
```

With pass-by-value, we expect a copy to be made, so if a conversion occurs first (resulting in an additional copy) it's rarely an issue (and the compiler will likely optimize one of the two copies away).

However, we often use pass by reference when we *don't* want a copy to be made. If a conversion occurs first, this will typically result in a (possibly expensive) copy being made, which can be suboptimal.

Warning

With pass by reference, ensure the type of the argument matches the type of the reference, or it will result in an unexpected (and possibly expensive) conversion.

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 use pass by value vs pass by reference

For most C++ beginners, the choice of whether to use pass by value or pass by reference isn't very obvious. Fortunately, there's a straightforward rule of thumb that will serve you well in the majority cases.

- Fundamental types and enumerated types are cheap to copy, so they are typically passed by value.
- Class types can be expensive to copy (sometimes significantly so), so they are typically passed by const reference.

Best practice

As a rule of thumb, pass fundamental types by value and class types by const reference.

If you aren't sure what to do, pass by const reference, as you're less likely to encounter unexpected behavior.

Tip

Here's a partial list of other interesting cases:

The following are often passed by value (because it is more efficient):

- Enumerated types (unscoped and scoped enumerations).
- Views and spans (e.g. `std::string_view`, `std::span`).
- Types that mimic references or (non-owning) pointers (e.g. iterators, `std::reference_wrapper`).
- Cheap-to-copy class types that have value semantics (e.g. `std::pair` with elements of fundamental types, `std::optional`, `std::expected`).

Pass by reference should be used for the following:

- Arguments that need to be modified by the function.
- Types that aren't copyable (such as `std::ostream`).
- Types where copying has ownership implications that we want to avoid (e.g. `std::unique_ptr`, `std::shared_ptr`).

- Types that have virtual functions or are likely to be inherited from (due to object slicing concerns, covered in lesson 25.9 -- Object slicing[3]).

The cost of pass by value vs pass by reference Advanced

Not all class types need to be passed by reference (such as `std::string_view`, which is normally passed by value). 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.

First, we need to consider the cost of initializing the function parameter. With pass by value, initialization means making a copy. The cost of copying an object is generally proportional to two things:

- The size of the object. Objects 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, we need to consider the cost of using the function parameter. When setting up a function call, the compiler may be able to optimize by placing a reference or copy of a passed-by-value argument (if it is small in size) into a CPU register (which is fast to access) rather than into RAM (which is slower to access).

Each time a value parameter is used, the running program can directly access the storage location (CPU register or RAM) of the copied argument. However, when a reference parameter is used, there is usually an extra step. The running program must first directly access the storage location (CPU register or RAM) allocated to the reference, in order to determine which object is being referenced. Only then can it access the storage location of the referenced object (in RAM).

Therefore, each use of a value parameter is a single CPU register or RAM access, whereas each use of a reference parameter is a single CPU register or RAM access plus a second RAM access.

Third, the compiler can sometimes optimize code that uses pass by value more effectively than code that uses pass by reference. In particular, optimizers have to be conservative when there is any chance of **aliasing** (when two or more pointers or references can access the same object). Because pass by value results in the copy of argument values, there is no chance for aliasing to occur, allowing optimizers to be more aggressive.

We can now answer these 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, but accessing the objects is faster and the compiler is likely to be able to optimize better.
- For objects that are expensive to copy, the cost of the copy dominates other performance considerations.

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 it 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 function-like macro that can be used to determine if a type (or object) is cheap to copy accordingly:

```cpp
#include <iostream>

// Function-like macro that evaluates to true if the type (or object) is equal to or smaller than
// the size of two memory addresses
#define isSmall(T) (sizeof(T) <= 2 * sizeof(void*))

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

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

    double d {};
    std::cout << isSmall(d) << '\n'; // true
    std::cout << isSmall(S) << '\n'; // false

    return 0;
}
```

As an aside...

We use a preprocessor function-like macro here so that we can provide either an object OR a type name as a parameter (as C++ functions disallow passing types as a parameter).

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.

()For function parameters, prefer `std::string_view` over `const std::string&` in most cases (#stringparameter)[4]

One question that comes up often in modern C++: when writing a function that has a string parameter, should the type of the parameter be `const std::string&` or `std::string_view`?

In most cases, `std::string_view` is the better choice, as it can handle a wider range of argument types efficiently. A `std::string_view` parameter also allows the caller to pass in a substring without having to copy that substring into its own string first.

```cpp
void doSomething(const std::string&);
void doSomething(std::string_view);   // prefer this in most cases
```

There are a few cases where using a `const std::string&` parameter may be more appropriate:

- If you're using C++14 or older, `std::string_view` isn't available.

- If your function needs to call some other function that takes a C-style string or `std::string` parameter, then `const std::string&` may be a better choice, as `std::string_view` is not guaranteed to be null-terminated (something that C-style string functions expect) and does not efficiently convert back to a `std::string`.

Best practice

Prefer passing strings using `std::string_view` (by value) instead of `const std::string&`, unless your function calls other functions that require C-style strings or `std::string` parameters.

Why `std::string_view` parameters are more efficient than `const std::string&` Advanced

In C++, a string argument will typically be a `std::string`, a `std::string_view`, or a C-style string/string literal.

As reminders:

- If the type of an argument does not match the type of the corresponding parameter, the compiler will try to implicitly convert the argument to match the type of the parameter.
- Converting a value creates a temporary object of the converted type.
- Creating (or copying) a `std::string_view` is inexpensive, as `std::string_view` does not make a copy of the string it is viewing.
- Creating (or copying) a `std::string` can be expensive, as each `std::string` object makes a copy of the string.

Here's a table showing what happens when we try to pass each type:

Argument Type	std::string_view parameter	const std::string& parameter
std::string	Inexpensive conversion	Inexpensive reference binding
std::string_view	Inexpensive copy	Expensive explicit conversion to `std::string`
C-style string / literal	Inexpensive conversion	Expensive conversion

With a `std::string_view` value parameter:

- If we pass in a `std::string` argument, the compiler will convert the `std::string` to a `std::string_view`, which is inexpensive, so this is fine.
- If we pass in a `std::string_view` argument, the compiler will copy the argument into the parameter, which is inexpensive, so this is fine.
- If we pass in a C-style string or string literal, the compiler will convert these to a `std::string_view`, which is inexpensive, so this is fine.

As you can see, `std::string_view` handles all three cases inexpensively.

With a `const std::string&` reference parameter:

- If we pass in a `std::string` argument, the parameter will reference bind to the argument, which is inexpensive, so this is fine.
- If we pass in a `std::string_view` argument, the compiler will refuse to do an implicit conversion, and produce a compilation error. We can use `static_cast` to do an explicit conversion (to `std::string`), but this conversion is expensive (since `std::string` will make a copy of the string being viewed). Once the conversion is done, the parameter will reference bind to the result, which is inexpensive. But we've made an expensive copy to do the conversion, so this isn't great.
- If we pass in a C-style string or string literal, the compiler will implicitly convert this to a `std::string`, which is expensive. So this isn't great either.

Thus, a `const std::string&` parameter only handles `std::string` arguments inexpensively.

The same, in code form:

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

void printSV(std::string_view sv)
{
    std::cout << sv << '\n';
}

void printS(const std::string& s)
{
    std::cout << s << '\n';
}

int main()
{
    std::string s{ "Hello, world" };
    std::string_view sv { s };

    // Pass to `std::string_view` parameter
    printSV(s);              // ok: inexpensive conversion from std::string to std::string_view
    printSV(sv);             // ok: inexpensive copy of std::string_view
    printSV("Hello, world"); // ok: inexpensive conversion of C-style string literal to std::string_view

    // pass to `const std::string&` parameter
    printS(s);              // ok: inexpensive bind to std::string argument
    printS(sv);             // compile error: cannot implicit convert std::string_view to std::string
    printS(static_cast<std::string>(sv)); // bad: expensive creation of std::string temporary
    printS("Hello, world"); // bad: expensive creation of std::string temporary

    return 0;
}
```

Additionally, we need to consider the cost of accessing the parameter inside the function. Because a `std::string_view` parameter is a normal object, the string being viewed can be accessed directly. Accessing a `std::string&` parameter requires an additional step to get to the referenced object before the string can be accessed.

Finally, if we want to pass in a substring of an existing string (of any type), it is comparatively cheap to create a `std::string_view` substring, which can then be cheaply passed to a `std::string_view` parameter. In

comparison, passing a substring to a `const std::string&` is more expensive, as the substring must at some point be copied into the `std::string` that the reference parameter binds to.

Next lesson
12.7 Introduction to pointers

5

Back to table of contents

6

Previous lesson
12.5 Pass by lvalue reference

7

8

44 COMMENTS

Newest

Binary Rambo

Binary Rambo
June 20, 2025 12:34 am PDT

Hi Alex,

However, when a reference parameter is used, there is usually an extra step. **The running program must first directly access the storage location (CPU register or RAM) allocated to the reference**, in order to determine which object is being referenced. Only then can it access the storage location of the referenced object (in RAM).

I would like to clarify some understanding, the bolded line, states that there is a need to "access the storage location allocated to the reference",

somewhere in other lessons, i believe its mentioned that "references are not objects", so how does references will have a storage location ?

somewhere in other lessons, i believe its mentioned that "under the hood, references are pointers".

Please help to clarify, whether references are objects ? does have storage location ?

0 Reply

Kappa
June 3, 2025 6:43 am PDT

If const references as parameters are used to avoid creating another expensive object, and passing an rvalue to it still creates a temporary object, then how is it different from just passing an rvalue by value? Is the only reason for this feature to let the same parameter be used for everything at the cost of secretly being unoptimized for particular arguments? Or would the temporary object be created regardless of whether it's passed by value or reference, and the pass by reference takes advantage of that?

0 Reply

Felipe
May 22, 2025 2:47 pm PDT

Can someone help me out with the following? I'm testing overload resolution with the following code:

```
#include <iostream>

void func(const int& x) { std::cout << 1; } // func 1

void func(int& x) { std::cout << 2; } // func 2

void func(int x) { std::cout << 3; } // func 3

int main(){
int x { 5 };
func(x); // call
}
```

Case 1: As is - won't compile due to ambiguity, makes sense.

Case 2: Only func1 & func2 - will compile and run. The call will resolve to func2, not sure why, but noted.
I learned that func2 has priority over func1 when resolving int lvalues.

Case 3: Only func2 & func3 - won't compile due to ambiguity.
I learned that func2 & func3 take equal priority when resolving int lvalues.

Case 4: Only func1 & func3 - won't compile due to ambiguity.
I learned that func1 & func3 take equal priority when resolving int lvalues.

Combining everything: func2 > func1 = func3 = func2
func2 > func2 ???

What does this mean?

0 Reply

Kappa

Reply to Felipe [12] June 3, 2025 6:58 am PDT

Judging from those cases, it seems that the ambiguity is caused when there are functions that only differ by the kind of passing. (case 1, case 3, case 4 have both pass-by-value and pass-by-reference, while case 2 only has pass-by-reference). You shouldn't have assumed that there's a linear order of priority.

As for the reason for that priority within case 2, I checked, and it's because it chooses that version of the function, where reference type would match the referent type most closely. "int x" gets the "int &x" function, but "const int x" would get the "const int &x" function.

0 Reply

Ali M.

March 27, 2025 4:20 am PDT

In the following example:

```
1 | std::cout << isSmall(S) << '\n'; // false
```

Variable S has not been defined.

0 Reply

Dietskittles

Reply to Ali M. [13] March 27, 2025 11:37 am PDT

S is the identifier for the struct above main, not a variable to be defined within main.

0 Reply

Ali M.

Reply to Dietskittles [14] March 27, 2025 11:43 am PDT

Oh, I see it now. Thanks.

1 Reply

Badger Patcher

March 2, 2025 1:45 pm PST

Hands down one of the best lessons if not the best! (for now:)

1 Reply

Nidhi Gupta
February 28, 2025 3:01 pm PST

This subject addresses const lvalue references in C++ with the emphasis on their ability to bind both modifiable lvalues, non-modifiable lvalues, and rvalues, unlike non-const lvalue references. This is what makes them extremely useful for passing arguments into functions without copying them yet ensuring immutability. Through const references, functions can now accept literals and temporary objects, which would otherwise not be available using regular lvalue references. The lesson also defines how type conversions affect reference binding and cautions against automatic temporary object creation because of implicit conversions. It further explains when to pass-by-value and pass-by-reference, recommending passing the basic types by value and class types by const reference in the interest of efficiency. The lesson then discusses the effectiveness of passing strings, which the author suggests is best done by using std::string_view over const std::string& in general use due to its potential to pass different types of strings with minimal overhead. Finally, it explains why references cannot be reseated and how compiler optimizations affect performance considerations while determining whether to pass-by-value or pass-by-reference.

0 Reply

Tim
February 5, 2025 3:33 pm PST

Just to make sure it is clear, you should only be passing `std::string&` when you want to edit the argument for whatever reason. Any other time, passing a `std::string_view` is what you want?

2 Reply

Alex Author
Reply to Tim [15] February 8, 2025 8:47 pm PST

There's a list of guidelines at the bottom of https://www.learncpp.com/cpp-tutorial/stdstring_view-part-2/

1 Reply

tok
November 19, 2024 9:15 am PST

'Pass by (const) reference should be used for the following:

Arguments that need to be modified by the function.'

I may be reading this wrong, or haven't learned the previous subjects correctly. If we needed to pass an argument that needed to be modified by the function, shouldn't it be passed by non-const reference? and not const reference? To quote one of your previous lessons:

"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 this was the case, and we wanted to damage the player we'd have to pass it by non-const reference no? If not, could you please explain? Thank you for the great work you do on this website!

1 Reply

Alex Author

Reply to tok [16] November 21, 2024 1:51 pm PST

Yes, you should use pass by non-const reference when the function needs to modify the argument passed in.

I've modified the wording used in the article a bit so that "pass by reference" means either by non-const or const reference, and the terms "non-const" or "const" are applying to denote a specific case.

2 Reply

trtr

November 10, 2024 11:43 am PST

Hello,
The lesson reads: "(...) a const lvalue reference can bind to a value of a different type, as long as that value is convertible to the type of the reference. The conversion creates a temporary object that the reference parameter can then bind to."
If I understood this comment https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/#comment-573036 correctly, a temporary object is created even if arg <-> param types are trivially convertible.
Am I right to assume that in the case where types mismatch, e.g. double <-> int, there would be 2 temporaries created in order to pass the argument? One would be of type int, which would get initialized with a value of the argument, and then the second one of type const int&, which would bind to the first temporary, and then be itself passed to the function being called.

Btw. found a typo, in "The running program must first directly access the storage location (CPU register or RAM) allocated to the reference, in order to determine which object is being reference." missing 'd' as the last letter.

0 Reply

Alex Author

Reply to trtr [17] November 11, 2024 1:50 pm PST

Upon further research and review, I revised that comment, as it was technically incorrect. When a reference is being bound to an object of the same type, the mismatch in the type of the reference (e.g. `int&` or `const int&`) vs the type of the object being referenced (e.g. `int`) is handled as part of the reference initialization process (not as a trivial conversion). As such, no temporary reference needs to

be created. A conversion is only involved when the object being bound is an rvalue or a different type altogether.

A note about this has been added to the initial lesson on lvalue references.

Although the question is no longer relevant in this case, I would like to mention that a single conversion can apply both a trivial conversion and a type conversion.

2 Reply

trtr

Reply to Alex [18] November 14, 2024 3:31 am PST

Thank you very much, it clears things up.

0 Reply

Tomáš Nadrchal

October 29, 2024 10:58 am PDT

Relating to the advanced part of pass by value vs. const reference, would be better from C++20 use consteval function?

```
consteval bool isSmall(auto value)
{
    return sizeof(decltype(value)) <= 2 * sizeof(void*);
}
```

0 Reply

Alex Author

Reply to Tomáš Nadrchal [19] November 2, 2024 3:14 pm PDT

We use a function-like macro so we can pass either an object or a type using the same syntax. If we use a consteval function instead, we have to use overloaded functions and a different syntax when passing an object or a type.

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/
3. https://www.learncpp.com/cpp-tutorial/object-slicing/
4. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#stringparameter

5. https://www.learncpp.com/cpp-tutorial/introduction-to-pointers/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/
8. https://www.learncpp.com/pass-by-const-lvalue-reference/
9. https://www.learncpp.com/cpp-tutorial/struct-miscellany/
10. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-610317
13. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-608830
14. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-608840
15. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-607413
16. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-604319
17. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-604036
18. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-604086
19. https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/#comment-603648
20. https://g.ezoic.net/privacy/learncpp.com