

12.3 — Lvalue references

👤 ALEX¹ ⌚ NOVEMBER 11, 2024

In C++, a **reference** is an alias for an existing object. Once a reference has been defined, any operation on the reference is applied to the object being referenced. This means we can use a reference to read or modify the object being referenced.

Although references might seem silly, useless, or redundant at first, references are used everywhere in C++ (we'll see examples of this in a few lessons).

Key insight

A reference is essentially identical to the object being referenced.

You can also create references to functions, though this is done less often.

Modern C++ contains two types of references: lvalue references, and rvalue references. In this chapter, we'll discuss lvalue references.

Related content

Because we'll be talking about lvalues and rvalues in this lesson, please review [12.2 -- Value categories \(lvalues and rvalues\)](https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/)² if you need a refresher on these terms before proceeding. Rvalue references are covered in the chapter on move semantics ([chapter 22](https://www.learncpp.com/#Chapter22))³.

Lvalue references types

An **lvalue reference** (commonly just called a “reference” since prior to C++11 there was only one type of reference) acts as an alias for an existing lvalue (such as a variable).

Just like the type of an object determines what kind of value it can hold, the type of a reference determines what type of object it can reference. Lvalue reference types can be identified by use of a single ampersand (&) in the type specifier:

```
1 // regular types
2 int      // a normal int type (not an reference)
3 int&     // an lvalue reference to an int object
4 double&  // an lvalue reference to a double object
5 const int& // an lvalue reference to a const int object
```

For example, `int&` is the type of an lvalue reference to an `int` object, and `const int&` is the type of an lvalue reference to a `const int` object.

A type that specifies a reference (e.g. `int&`) is called a **reference type**. The type that can be referenced (e.g. `int`) is called the **referenced type**.

Nomenclature

There are two kinds of lvalue references:

- An lvalue reference to a non-const is commonly just called an “lvalue reference”, but may also be referred to as an **lvalue reference to non-const** or a **non-const lvalue reference** (since it isn’t defined using the `const` keyword).
- An lvalue reference to a const is commonly called either an **lvalue reference to const** or a **const lvalue reference**.

We’ll focus on non-const lvalue references in this lesson, and on const lvalue references in the next lesson ([12.4 -- Lvalue references to const](https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/) ⁴).

Lvalue reference variables

One of the things we can do with an lvalue reference type is create an lvalue reference variable. An **lvalue reference variable** is a variable that acts as a reference to an lvalue (usually another variable).

To create an lvalue reference variable, we simply define a variable with an lvalue reference type:

```
1 #include <iostream>
2
3 int main()
4 {
5     int x { 5 };    // x is a normal integer variable
6     int& ref { x }; // ref is an lvalue reference variable that can now be used as an
    alias for variable x
7
8     std::cout << x << '\n'; // print the value of x (5)
9     std::cout << ref << '\n'; // print the value of x via ref (5)
10
11     return 0;
12 }
```

In the above example, the type `int&` defines `ref` as an lvalue reference to an `int`, which we then initialize with lvalue expression `x`. Thereafter, `ref` and `x` can be used synonymously. This program thus prints:

```
5
5
```

From the compiler’s perspective, it doesn’t matter whether the ampersand is “attached” to the type name (`int& ref`) or the variable’s name (`int &ref`), and which you choose is a matter of style. Modern C++ programmers tend to prefer attaching the ampersand to the type, as it makes clearer that the reference is part of the type information, not the identifier.

Best practice

When defining a reference, place the ampersand next to the type (not the reference variable’s name).

For advanced readers

For those of you already familiar with pointers, the ampersand in this context does not mean “address of”, it means “lvalue reference to”.

Modifying values through a non-const lvalue reference

In the above example, we showed that we can use a reference to read the value of the object being referenced. We can also use a non-const reference to modify the value of the object being referenced:

```
1  #include <iostream>
2
3  int main()
4  {
5      int x { 5 }; // normal integer variable
6      int& ref { x }; // ref is now an alias for variable x
7
8      std::cout << x << ref << '\n'; // print 55
9
10     x = 6; // x now has value 6
11
12     std::cout << x << ref << '\n'; // prints 66
13
14     ref = 7; // the object being referenced (x) now has value 7
15
16     std::cout << x << ref << '\n'; // prints 77
17
18     return 0;
19 }
```

This code prints:

```
55
66
77
```

In the above example, `ref` is an alias for `x`, so we are able to change the value of `x` through either `x` or `ref`.

Reference initialization

Much like constants, all references must be initialized. References are initialized using a form of initialization called **reference initialization**.

```
1  int main()
2  {
3      int& invalidRef; // error: references must be initialized
4
5      int x { 5 };
6      int& ref { x }; // okay: reference to int is bound to int variable
7
8      return 0;
9  }
```

When a reference is initialized with an object (or function), we say it is **bound** to that object (or function). The process by which such a reference is bound is called **reference binding**. The object (or function) being referenced is sometimes called the **referent**.

Non-const lvalue references can only be bound to a *modifiable* lvalue.

```
1 | int main()
2 | {
3 |     int x { 5 };
4 |     int& ref { x };           // okay: non-const lvalue reference bound to a modifiable
5 |     lvalue
6 |
7 |     const int y { 5 };
8 |     int& invalidRef { y };    // invalid: non-const lvalue reference can't bind to a
9 |     non-modifiable lvalue
10 |    int& invalidRef2 { 0 };    // invalid: non-const lvalue reference can't bind to an
11 |    rvalue
12 |
13 |    return 0;
14 | }
```

Key insight

If non-const lvalue references could be bound to non-modifiable (const) lvalues or rvalues, then you would be able to change those values through the reference, which would be a violation of their constness.

Lvalue references to `void` are disallowed (what would be the point?).

Even though the type of the reference (e.g. `int&`) doesn't exactly match the type of the object being bound (e.g. `int`), no conversion is applied here (not even a trivial conversion) -- the difference in types is handled as part of the reference initialization process.

A reference will (usually) only bind to an object matching its referenced type

In most cases, a reference will only bind to an object whose type matches the referenced type, (there are some exceptions to this rule that we'll discuss when we get into inheritance).

If you try to bind a reference to an object that does not match its referenced type, the compiler will try to implicitly convert the object to the referenced type and then bind the reference to that.

Key insight

Since the result of a conversion is an rvalue, and a non-const lvalue reference can't bind to an rvalue, trying to bind a non-const lvalue reference to an object that does not match its referenced type will result in a compilation error.

```

1 | int main()
2 | {
3 |     int x { 5 };
4 |     int& ref { x };           // okay: referenced type (int) matches type of
5 |     initializer
6 |
7 |     double d { 6.0 };
8 |     int& invalidRef { d };    // invalid: conversion of double to int is narrowing
9 |     double& invalidRef2 { x }; // invalid: non-const lvalue reference can't bind to
10 |    rvalue (result of converting x to double)
11 |
12 |     return 0;
13 | }

```

References can't be reseated (changed to refer to another object)

Once initialized, a reference in C++ cannot be **reseated**, meaning it cannot be changed to reference another object.

New C++ programmers often try to reseat a reference by using assignment to provide the reference with another variable to reference. This will compile and run -- but not function as expected. Consider the following program:

```

1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int x { 5 };
6 |     int y { 6 };
7 |
8 |     int& ref { x }; // ref is now an alias for x
9 |
10 |    ref = y; // assigns 6 (the value of y) to x (the object being referenced by ref)
11 |    // The above line does NOT change ref into a reference to variable y!
12 |
13 |    std::cout << x << '\n'; // user is expecting this to print 5
14 |
15 |    return 0;
16 | }

```

Perhaps surprisingly, this prints:

6

When a reference is evaluated in an expression, it resolves to the object it's referencing. So `ref = y` doesn't change `ref` to now reference `y`. Rather, because `ref` is an alias for `x`, the expression evaluates as if it was written `x = y` -- and since `y` evaluates to value `6`, `x` is assigned the value `6`.

Reference scope and duration

Reference variables follow the same scoping and duration rules that normal variables do:

```

1  #include <iostream>
2
3  int main()
4  {
5      int x { 5 }; // normal integer
6      int& ref { x }; // reference to variable value
7
8      return 0;
9  } // x and ref die here

```

References and referents have independent lifetimes

With one exception (that we'll cover next lesson), the lifetime of a reference and the lifetime of its referent are independent. In other words, **both of the following are true:**

- **A reference can be destroyed before the object it is referencing.**
- **The object being referenced can be destroyed before the reference.**

When a reference is destroyed before the referent, the referent is not impacted. The following program demonstrates this:

```

1  #include <iostream>
2
3  int main()
4  {
5      int x { 5 };
6
7      {
8          int& ref { x }; // ref is a reference to x
9          std::cout << ref << '\n'; // prints value of ref (5)
10         } // ref is destroyed here -- x is unaware of this
11
12         std::cout << x << '\n'; // prints value of x (5)
13
14         return 0;
15     } // x destroyed here

```

The above prints:

```

5
5

```

When `ref` dies, variable `x` carries on as normal, blissfully unaware that a reference to it has been destroyed.

Dangling references

When an object being referenced is destroyed before a reference to it, the reference is left referencing an object that no longer exists. Such a reference is called a **dangling reference**. **Accessing a dangling reference leads to undefined behavior.**

Dangling references are fairly easy to avoid, but we'll show a case where this can happen in practice in lesson [12.12 -- Return by reference and return by address](https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/) (<https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/>)⁵.

References aren't objects

Perhaps surprisingly, references are not objects in C++. A reference is not required to exist or occupy storage. If possible, the compiler will optimize references away by replacing all occurrences of a reference with the referent. However, this isn't always possible, and in such cases, references may require storage.

This also means that the term "reference variable" is a bit of a misnomer, as variables are objects with a name, and references aren't objects.

Because references aren't objects, they can't be used anywhere an object is required (e.g. you can't have a reference to a reference, since an lvalue reference must reference an identifiable object). In cases where you need a reference that is an object or a reference that can be reseated, `std::reference_wrapper` (which we cover in lesson [23.3 -- Aggregation](https://www.learncpp.com/cpp-tutorial/aggregation/) (<https://www.learncpp.com/cpp-tutorial/aggregation/>)⁶) provides a solution.

As an aside...

Consider the following variables:

```
1 | int var{};
2 | int& ref1{ var }; // an lvalue reference bound to var
3 | int& ref2{ ref1 }; // an lvalue reference bound to var
```

Because `ref2` (a reference) is initialized with `ref1` (a reference), you might be tempted to conclude that `ref2` is a reference to a reference. It is not. Because `ref1` is a reference to `var`, when used in an expression (such as an initializer), `ref1` evaluates to `var`. So `ref2` is just a normal lvalue reference (as indicated by its type `int&`), bound to `var`.

A reference to a reference (to an `int`) would have syntax `int&&` -- but since C++ doesn't support references to references, this syntax was repurposed in C++11 to indicate an rvalue reference (which we cover in lesson [22.2 -- R-value references](https://www.learncpp.com/cpp-tutorial/rvalue-references/) (<https://www.learncpp.com/cpp-tutorial/rvalue-references/>)⁷).

Author's note

If references seem a bit useless at this point, don't worry. References are used a lot, and we'll cover one of the primary reasons why shortly, in lessons [12.5 -- Pass by lvalue reference](https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/) (<https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/>)⁸ and [12.6 -- Pass by const lvalue reference](https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/) (<https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/>)⁹.

Quiz time

Question #1

Determine what values the following program prints by yourself (do not compile the program).

```

1  #include <iostream>
2
3  int main()
4  {
5      int x{ 1 };
6      int& ref{ x };
7
8      std::cout << x << ref << '\n';
9
10     int y{ 2 };
11     ref = y;
12     y = 3;
13
14     std::cout << x << ref << '\n';
15
16     x = 4;
17
18     std::cout << x << ref << '\n';
19
20     return 0;
21 }

```

[Show Solution](#) (javascript:void(0))¹⁰



[Next lesson](#)

12.4 [Lvalue references to const](#)

4



[Back to table of contents](#)

11



[Previous lesson](#)

12.2 [Value categories \(lvalues and rvalues\)](#)

2

12



B

U

URL

INLINE CODE

C++ CODE BLOCK

HELP!

Leave a comment...



Name*



Email*



Notify me about replies:



POST COMMENT

🚩 Find a mistake? Leave a comment above!?

Avatars from <https://gravatar.com/>¹⁵ are connected to your provided email address.

351 COMMENTS

Newest ▼



Shantanu Singh

🕒 June 9, 2025 1:26 am PDT

There is a mistake in this line:

In the above example, the type `int&` defines `ref` as an lvalue reference to an `int`, which we then initialize with lvalue expression `x`. Thereafter, `ref` and `x` can be used synonymously. This program thus prints:

I think it should be:

In the above example, the type `int&` defines `ref` as an lvalue reference to an `int`, which we then initialize with rvalue expression `x` (lvalue implicitly converted to rvalue here).

👍 0 ➡ Reply



mrme

🗨 Reply to [Shantanu Singh](#)¹⁶ 🕒 June 23, 2025 12:42 am PDT

We initialize with the rvalue expression `x` but the compiler implicitly converts the lvalue into an rvalue when performing numeric promotion to match the required reference type (`double`).

👍 0 ➡ Reply



Copernicus

🕒 May 17, 2025 8:36 pm PDT

Question #1

11

22

44

👍 1 ➡ Reply



RSH

🕒 May 11, 2025 10:24 pm PDT

Ffdsafasdguhepufqwerh I love this course

👍 2 ➡ Reply



Leni

🕒 March 2, 2025 8:34 pm PST

This explanation does a great job of covering the fundamentals of lvalue references in C++, emphasizing their role as aliases for existing objects. I wonder why references were introduced if pointers already

existed?

👍 4 ➡ Reply

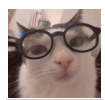


Nidhi Gupta

🕒 February 28, 2025 2:59 pm PST

This lesson is about lvalue references in C++, explaining that a reference is an alias for an existing object and anything done on the reference is done on the original object. There are two types of references in modern C++: lvalue references and rvalue references, and this chapter discusses the first one. Lvalue references are aliases for lvalues that can be modified, allowing reading and modification of the referred-to object. They have to be initialized on declaration and can't be reseated later to refer to another object. The reference type has to be typically the same as the referenced object's type, and there are no implicit conversions performed in binding. And references follow the same scope and lifetime rules as regular variables but have their lifetimes decoupled from the objects they are referencing. The lvalue reference concept is crucial to good C++ programming since lvalue references are used extensively in parameter passing and optimization.

👍 0 ➡ Reply



NordicPeace

🕒 December 1, 2024 9:19 pm PST

Why I don't get a notification when someone replied to my comment?? Notify me about replies is enabled even then i don't get any notification.

👍 1 ➡ Reply

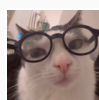


Alex Author

🔗 Reply to [NordicPeace](#) ¹⁷ 🕒 December 3, 2024 2:57 pm PST

Either you mistyped your email address or it's getting blocked somewhere for some reason.

👍 4 ➡ Reply



NordicPeace

🔗 Reply to [Alex](#) ¹⁸ 🕒 December 4, 2024 10:25 am PST

Oh!! I opened my email after an eternity. Sorry! It's my mistake, I assumed that you would get a notification like in general but it were emails.

👍 1 ➡ Reply



Yassin

🕒 December 1, 2024 1:48 am PST

Why after the conversion the object becomes rvalue as in:
`Int x {5};`


```
double& invalidRef2 { x };
```

 Last edited 7 months ago by Yassin

 1  Reply



Alex Author

 Reply to [Yassin](#)¹⁹  December 2, 2024 8:24 pm PST

A conversion produces a temporary object as its result. That temporary object is an rvalue.

In order for a `double&` to bind to an `int`, the `int` must be converted to a `double`, yielding a rvalue. Since a non-const lvalue reference can't bind to an rvalue, the binding is illegal.

 2  Reply



Amitrisu

 November 15, 2024 2:26 pm PST

"A type that specifies a reference (e.g. `int&`) is called a **reference type**. The type that can be referenced (e.g. `int`) is called the **referenced type**."

"If you try to bind a reference to an object that does not match its referenced type, the compiler will try to implicitly convert the object to the referenced type and then bind the reference to that."

A bit confused by terms, maybe in the latter fragment you meant **reference types** (not **referenced types**)?

 0  Reply



Alex Author

 Reply to [Amitrisu](#)²⁰  November 17, 2024 9:18 pm PST

I think it's written correctly. Let's say we have a `const double&` parameter. If we pass it an `int` object, the type of the object (`int`) does not match the referenced type (`double`). Therefore, the object will be converted to the referenced type (`double`), so the reference can bind to it.

 1  Reply



Amitrisu

 Reply to [Alex](#)²¹  November 22, 2024 6:34 am PST

I was just too tired probably and thought that a referenced type is the type of the value that we use for a reference initialization, since it is "referenced" and is not copied. :D Thought you mixed them up accidentally.

Your wording was absolutely fine! I just for some reason believed that in a context of a reference initialization the used value has some special meaning.

Thanks again, I think I'm the only one who got confused by this

 0  Reply



Ben Dover

🕒 October 25, 2024 2:05 am PDT

Me learning about references and pointers in C++: Okay this is stupid why we need this?
Me joining a C++ project for the first time at work seeing stars and ampersands everywhere: Holy crap!

👍 2 ➡ Reply



Alex

Author

🗨 Reply to [Ben Dover](#) ²² 🕒 October 25, 2024 5:19 pm PDT

^^^ this

(Or maybe *this is more appropriate)

👍 6 ➡ Reply



Slim

🗨 Reply to [Alex](#) ²³ 🕒 November 9, 2024 2:34 am PST

We are still too young to appreciate this joke Alex :D

👍 1 ➡ Reply



Nahman

🕒 October 15, 2024 4:05 am PDT

```
1 | int var{};
2 | int& ref1{ var }; // an lvalue reference bound to var
3 | int& ref2{ ref1 }; // an lvalue reference bound to var
```

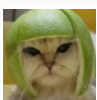
In the above code how does compiler covert a reference to its lvalue object ?

As conversions are mostly done with constructors and typecasts, are these used for the above conversion ?

If yes, they mostly return by value doesn't it make all the above kind of conversions expensive !. ig mostly no but please explain the conversion.

📝 Last edited 8 months ago by Nahman

👍 0 ➡ Reply



Alex

Author

🗨 Reply to [Nahman](#) ²⁴ 🕒 October 18, 2024 11:17 am PDT

The compiler is responsible for treating the reference as if it were the referenced object. This is not considered to be a conversion. If the reference needs to exist in memory, this is usually just a pointer dereference operation.

👍 0 ➡ Reply

Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/>
3. <https://www.learncpp.com/#Chapter22>
4. <https://www.learncpp.com/cpp-tutorial/lvalue-references-to-const/>
5. <https://www.learncpp.com/cpp-tutorial/return-by-reference-and-return-by-address/>
6. <https://www.learncpp.com/cpp-tutorial/aggregation/>
7. <https://www.learncpp.com/cpp-tutorial/rvalue-references/>
8. <https://www.learncpp.com/cpp-tutorial/pass-by-lvalue-reference/>
9. <https://www.learncpp.com/cpp-tutorial/pass-by-const-lvalue-reference/>
10. `javascript:void(0)`
11. <https://www.learncpp.com/>
12. <https://www.learncpp.com/lvalue-references/>
13. <https://www.learncpp.com/cpp-tutorial/pointers-and-const/>
14. <https://www.learncpp.com/cpp-tutorial/member-selection-with-pointers-and-references/>
15. <https://gravatar.com/>
16. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-610781>
17. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-604755>
18. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-604829>
19. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-604737>
20. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-604204>
21. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-604248>
22. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-603508>
23. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-603545>
24. <https://www.learncpp.com/cpp-tutorial/lvalue-references/#comment-603142>
25. <https://g.ezoic.net/privacy/learncpp.com>