12.3 — Lvalue references

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.

Key insight

A reference is essentially identical 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).

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) if you need a refresher on these terms before proceeding.

Rvalue references are covered in the chapter on `move semantics` (chapter 22).

Lvalue reference 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).

To declare an lvalue reference type, we use an ampersand (&) in the type declaration:

```
int      // a normal int type
int&     // an lvalue reference to an int object
double&  // an lvalue reference to a double object
```

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:

```
#include <iostream>

int main()
{
    int x { 5 };    // x is a normal integer variable
    int& ref { x }; // ref is an lvalue reference variable that can now be used as an
alias for variable x

    std::cout << x << '\n';  // print the value of x (5)
    std::cout << ref << '\n'; // print the value of x via ref (5)

    return 0;
}
```

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 an 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 reference to modify the value of the object being referenced:

```
#include <iostream>

int main()
{
    int x { 5 }; // normal integer variable
    int& ref { x }; // ref is now an alias for variable x

    std::cout << x << ref << '\n'; // print 55

    x = 6; // x now has value 6

    std::cout << x << ref << '\n'; // prints 66

    ref = 7; // the object being referenced (x) now has value 7

    std::cout << x << ref << '\n'; // prints 77

    return 0;
}
```

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`.

Initialization of lvalue references

Much like constants, all references must be initialized.

```
int main()
{
    int& invalidRef;   // error: references must be initialized

    int x { 5 };
    int& ref { x }; // okay: reference to int is bound to int variable

    return 0;
}
```

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**.

Lvalue references must be bound to a *modifiable* lvalue.

```
int main()
{
    int x { 5 };
    int& ref { x }; // valid: lvalue reference bound to a modifiable lvalue

    const int y { 5 };
    int& invalidRef { y };  // invalid: can't bind to a non-modifiable lvalue
    int& invalidRef2 { 0 }; // invalid: can't bind to an rvalue

    return 0;
}
```

Lvalue references can't be bound to non-modifiable lvalues or rvalues (otherwise you'd be able to change those values through the reference, which would be a violation of their const-ness). For this reason, lvalue references are occasionally called **lvalue references to non-const** (sometimes shortened to **non-const reference**).

In most cases, the type of the reference must match the type of the referent (there are some exceptions to this rule that we'll discuss when we get into inheritance):

```
int main()
{
    int x { 5 };
    int& ref { x }; // okay: reference to int is bound to int variable

    double y { 6.0 };
    int& invalidRef { y }; // invalid; reference to int cannot bind to double
variable
    double& invalidRef2 { x }; // invalid: reference to double cannot bind to int
variable

    return 0;
}
```

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

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:

```
#include <iostream>

int main()
{
    int x { 5 };
    int y { 6 };

    int& ref { x }; // ref is now an alias for x

    ref = y; // assigns 6 (the value of y) to x (the object being referenced by ref)
    // The above line does NOT change ref into a reference to variable y!

    std::cout << x << '\n'; // user is expecting this to print 5

    return 0;
}
```

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`.

Lvalue reference scope and duration

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

```
#include <iostream>

int main()
{
    int x { 5 }; // normal integer
    int& ref { x }; // reference to variable value

     return 0;
} // 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:

```
#include <iostream>

int main()
{
    int x { 5 };

    {
        int& ref { x };   // ref is a reference to x
        std::cout << ref << '\n'; // prints value of ref (5)
    } // ref is destroyed here -- x is unaware of this

    std::cout << x << '\n'; // prints value of x (5)

    return 0;
} // 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.

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) provides a solution.

As an aside…

Consider the following variables:

```
int var{};
int& ref1{ var };  // an lvalue reference bound to var
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).

Quiz time

Question #1

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

```
#include <iostream>

int main()
{
    int x{ 1 };
    int& ref{ x };

    std::cout << x << ref << '\n';

    int y{ 2 };
    ref = y;
    y = 3;

    std::cout << x << ref << '\n';

    x = 4;

    std::cout << x << ref << '\n';

    return 0;
}
```

Show Solution