12.2 — Value categories (lvalues and rvalues)

ALEX[1] JANUARY 3, 2025

Before we talk about our first compound type (lvalue references), we're going to take a little detour and talk about what an `lvalue` is.

In lesson 1.10 -- Introduction to expressions (https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/)[2], we defined an expression as "a combination of literals, variables, operators, and function calls that can be executed to produce a singular value".

For example:

```
#include <iostream>

int main()
{
    std::cout << 2 + 3 << '\n'; // The expression 2 + 3 produces the value 5

    return 0;
}
```

In the above program, the expression `2 + 3` is evaluated to produce the value 5, which is then printed to the console.

In lesson 6.4 -- Increment/decrement operators, and side effects (https://www.learncpp.com/cpp-tutorial/increment-decrement-operators-and-side-effects/)[3], we also noted that expressions can produce side effects that outlive the expression:

```
#include <iostream>

int main()
{
    int x { 5 };
    ++x; // This expression statement has the side-effect of incrementing x
    std::cout << x << '\n'; // prints 6

    return 0;
}
```

In the above program, the expression `++x` increments the value of `x`, and that value remains changed even after the expression has finished evaluating.

Besides producing values and side effects, expressions can do one more thing: they can evaluate to objects or functions. We'll explore this point further in just a moment.

The properties of an expression

To help determine how expressions should evaluate and where they can be used, all expressions in C++ have two properties: a type and a value category.

The type of an expression

The type of an expression is equivalent to the type of the value, object, or function that results from the evaluated expression. For example:

```
int main()
{
    auto v1 { 12 / 4 }; // int / int => int
    auto v2 { 12.0 / 4 }; // double / int => double

    return 0;
}
```

For `v1`, the compiler will determine (at compile time) that a division with two `int` operands will produce an `int` result, so `int` is the type of this expression. Via type inference, `int` will then be used as the type of `v1`.

For `v2`, the compiler will determine (at compile time) that a division with a `double` operand and an `int` operand will produce a `double` result. Remember that arithmetic operators must have operands of matching types, so in this case, the `int` operand gets converted to a `double`, and a floating point division is performed. So `double` is the type of this expression.

The compiler can use the type of an expression to determine whether an expression is valid in a given context. For example:

```
#include <iostream>

void print(int x)
{
    std::cout << x << '\n';
}

int main()
{
    print("foo"); // error: print() was expecting an int argument, we tried to pass in
a string literal

    return 0;
}
```

In the above program, the `print(int)` function is expecting an `int` parameter. However, the type of the expression we're passing in (the string literal `"foo"`) does not match, and no conversion can be found. So a compile error results.

Note that the type of an expression must be determinable at compile time (otherwise type checking and type deduction wouldn't work) -- however, the value of an expression may be determined at either compile time (if the expression is constexpr) or runtime (if the expression is not constexpr).

The value category of an expression

Now consider the following program:

```
int main()
{
    int x{};

    x = 5; // valid: we can assign 5 to x
    5 = x; // error: can not assign value of x to literal value 5

    return 0;
}
```

One of these assignment statements is valid (assigning value `5` to variable `x`) and one is not (what would it mean to assign the value of `x` to the literal value `5`?). So how does the compiler know which expressions can legally appear on either side of an assignment statement?

The answer lies in the second property of expressions: the `value category`. The **value category** of an expression (or subexpression) indicates whether an expression resolves to a value, a function, or an object of some kind.

Prior to C++11, there were only two possible value categories: `lvalue` and `rvalue`.

In C++11, three additional value categories (`glvalue`, `prvalue`, and `xvalue`) were added to support a new feature called `move semantics`.

Author's note

In this lesson, we'll stick to the pre-C++11 view of value categories, as this makes for a gentler introduction to value categories (and is all that we need for the moment). We'll cover move semantics (and the additional three value categories) in a future chapter.

Lvalue and rvalue expressions

An **lvalue** (pronounced "ell-value", short for "left value" or "locator value", and sometimes written as "l-value") is an expression that evaluates to an identifiable object or function (or bit-field).

The term "identity" is used by the C++ standard, but is not well-defined. An entity (such as an object or function) that has an identity can be differentiated from other similar entities (typically by comparing the addresses of the entity).

Entities with identities can be accessed via an identifier, reference, or pointer, and typically have a lifetime longer than a single expression or statement.

```
int main()
{
    int x { 5 };
    int y { x }; // x is an lvalue expression

    return 0;
}
```

In the above program, the expression `x` is an lvalue expression as it evaluates to variable `x` (which has an identifier).

Since the introduction of constants into the language, lvalues come in two subtypes: a **modifiable lvalue** is an lvalue whose value can be modified. A **non-modifiable lvalue** is an lvalue whose value can't be modified

(because the lvalue is const or constexpr).

```
int main()
{
    int x{};
    const double d{};

    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression

    return 0;
}
```

An **rvalue** (pronounced "arr-value", short for "right value", and sometimes written as `r-value`) is an expression that is not an lvalue. Rvalue expressions evaluate to a value. Commonly seen rvalues include literals (except C-style string literals, which are lvalues) and the return value of functions and operators that return by value. Rvalues aren't identifiable (meaning they have to be used immediately), and only exist within the scope of the expression in which they are used.

```
int return5()
{
    return 5;
}

int main()
{
    int x{ 5 }; // 5 is an rvalue expression
    const double d{ 1.2 }; // 1.2 is an rvalue expression

    int y { x }; // x is a modifiable lvalue expression
    const double e { d }; // d is a non-modifiable lvalue expression
    int z { return5() }; // return5() is an rvalue expression (since the result is returned by value)

    int w { x + 1 }; // x + 1 is an rvalue expression
    int q { static_cast<int>(d) }; // the result of static casting d to an int is an rvalue expression

    return 0;
}
```

You may be wondering why `return5()`, `x + 1`, and `static_cast<int>(d)` are rvalues: the answer is because these expressions produce temporary values that are not identifiable objects.

Key insight

Lvalue expressions evaluate to an identifiable object.
Rvalue expressions evaluate to a value.

Value categories and operators

Unless otherwise specified, operators expect their operands to be rvalues. For example, binary `operator+` expects its operands to be rvalues:

```cpp
#include <iostream>

int main()
{
    std::cout << 1 + 2; // 1 and 2 are rvalues, operator+ returns an rvalue

    return 0;
}
```

The literals `1` and `2` are both rvalue expressions. `operator+` will happily use these to return the rvalue expression `3`.

Now we can answer the question about why `x = 5` is valid but `5 = x` is not: an assignment operation requires its left operand to be a modifiable lvalue expression. The latter assignment (`5 = x`) fails because the left operand expression `5` is an rvalue, not a modifiable lvalue.

```cpp
int main()
{
    int x{};

    // Assignment requires the left operand to be a modifiable lvalue expression and the right operand to be an rvalue expression
    x = 5; // valid: x is a modifiable lvalue expression and 5 is an rvalue expression
    5 = x; // error: 5 is an rvalue expression and x is a modifiable lvalue expression

    return 0;
}
```

Lvalue-to-rvalue conversion

Since assignment operations expect the right operand to be an rvalue expression, you might be wondering why the following works:

```cpp
int main()
{
    int x{ 1 };
    int y{ 2 };

    x = y; // y is not an rvalue, but this is legal

    return 0;
}
```

In cases where an rvalue is expected but an lvalue is provided, the lvalue will undergo an lvalue-to-rvalue conversion so that it can be used in such contexts. This basically means the lvalue is evaluated to produce its value, which is an rvalue.

In the above example, the lvalue expression `y` undergoes an lvalue-to-rvalue conversion, which evaluates `y` to produce an rvalue (`2`), which is then assigned to `x`.

Key insight

An lvalue will implicitly convert to an rvalue. This means an lvalue can be used anywhere an rvalue is expected.
An rvalue, on the other hand, will not implicitly convert to an lvalue.

Now consider this example:

```
int main()
{
    int x { 2 };

    x = x + 1;

    return 0;
}
```

In this statement, the variable `x` is being used in two different contexts. On the left side of the assignment operator (where an lvalue expression is required), `x` is an lvalue expression that evaluates to variable `x`. On the right side of the assignment operator, `x` undergoes an lvalue-to-rvalue conversion and is then evaluated so that its (`2`) can be used as the left operand of `operator+`. `operator+` returns the rvalue expression `3`, which is then used as the right operand for the assignment.

How to differentiate lvalues and rvalues

You may still be confused about what kind of expressions qualify as an lvalue vs an rvalue. For example, is the result of `operator++` an lvalue or an rvalue? We'll cover various methods you can use to determine which is which here.

Tip

A rule of thumb to identify lvalue and rvalue expressions:

- Lvalue expressions are those that evaluate to functions or identifiable objects (including variables) that persist beyond the end of the expression.
- Rvalue expressions are those that evaluate to values, including literals and temporary objects that do not persist beyond the end of the expression.

For a more complete list of lvalue and rvalue expressions, you can consult technical documentation.

Tip

A full list of lvalue and rvalue expressions can be found here (https://en.cppreference.com/w/cpp/language/value_category)[4]. In C++11, rvalues are broken into two subtypes: prvalues and xvalues, so the rvalues we're talking about here are the sum of both of those categories.

Finally, we can write a program and have the compiler tell us what kind of expression something is. The following code demonstrates a method that determines whether an expression is an lvalue or an rvalue:

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

// T& is an lvalue reference, so this overload will be preferred for lvalues
template <typename T>
constexpr bool is_lvalue(T&)
{
    return true;
}

// T&& is an rvalue reference, so this overload will be preferred for rvalues
template <typename T>
constexpr bool is_lvalue(T&&)
{
    return false;
}

// A helper macro (#expr prints whatever is passed in for expr as text)
#define PRINTVCAT(expr) { std::cout << #expr << " is an " << (is_lvalue(expr) ? "lvalue\n" : "rvalue\n"); }

int getint() { return 5; }

int main()
{
    PRINTVCAT(5);         // rvalue
    PRINTVCAT(getint()); // rvalue
    int x { 5 };
    PRINTVCAT(x);         // lvalue
    PRINTVCAT(std::string {"Hello"}); // rvalue
    PRINTVCAT("Hello");  // lvalue
    PRINTVCAT(++x);       // lvalue
    PRINTVCAT(x++);       // rvalue
}
```

This prints:

```
5 is an rvalue
getint() is an rvalue
x is an lvalue
std::string {"Hello"} is an rvalue
"Hello" is an lvalue
++x is an lvalue
x++ is an rvalue
```

This method relies on two overloaded functions: one with an lvalue refrence parameter and one with an rvalue reference parameter. The lvalue reference version will be preferred for lvalue arguments, and the rvalue reference version will be preferred for rvalue arguments. Thus we can determine whether the argument is an lvalue or rvalue based on which function gets selected.

So as you can see, whether `operator++` results in an lvalue or an rvalue depends on whether it is used as a prefix operator (which returns an lvalue) or a postfix operator (which returns an rvalue)!

For advanced readers

Unlike the other literals (which are rvalues), a C-style string literal is an lvalue because C-style strings (which are C-style arrays) decay to a pointer. The decay process only works if the array is an lvalue (and thus has an address that can be stored in the pointer). C++ inherited this for backwards compatibility.

We cover array decay in lesson 17.8 -- C-style array decay (https://www.learncpp.com/cpp-tutorial/c-style-array-decay/)[5].

Now that we've covered lvalues, we can get to our first compound type: the `lvalue reference`.

Next lesson
12.3 Lvalue references

6

Back to table of contents

7

Previous lesson
12.1 Introduction to compound data types

8

9

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/[11] are connected to your provided email address.

137 COMMENTS Newest

AJAY.CHALLA
May 13, 2025 10:47 pm PDT

I just cant understand what happened in this program can anyone explain??

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

// T& is an lvalue reference, so this overload will be preferred for lvalues
template <typename T>
constexpr bool is_lvalue(T&)
{
    return true;
}

// T&& is an rvalue reference, so this overload will be preferred for rvalues
template <typename T>
constexpr bool is_lvalue(T&&)
{
    return false;
}

// A helper macro (#expr prints whatever is passed in for expr as text)
#define PRINTVCAT(expr) { std::cout << #expr << " is an " << (is_lvalue(expr) ? "lvalue\n" : "rvalue\n"); }

int getint() { return 5; }

int main()
{
    PRINTVCAT(5);        // rvalue
    PRINTVCAT(getint()); // rvalue
    int x { 5 };
    PRINTVCAT(x);        // lvalue
    PRINTVCAT(std::string {"Hello"}); // rvalue
    PRINTVCAT("Hello");  // lvalue
    PRINTVCAT(++x);      // lvalue
    PRINTVCAT(x++);      // rvalue
}
```

0 Reply

Robert

Reply to AJAY.CHALLA [12] May 23, 2025 12:35 pm PDT

macro function call PRINTVCAT() will be replaced in preprocessing time by the expresion `std::cout << #expr << " is an " << (is_lvalue(expr) ? "lvalue\n" : "rvalue\n")` so you can imagine each line calling PRINTVCAT like that. So for example in the first line `PRINTVCAT(5);` will be `std::cout << 5 << " is an " << (is_lvalue(5) ? "lvalue\n" : "rvalue\n")` , now is_lvalue() is defined above, but there is overloading template function, what will happen ? if 5 if an L-VALUE then will call the first function is_lvalue(T&) because of & (parameter to receive an l-value reference), that function is returning true. for r-values overloading resolution will call function with T&&, that function is returning false, so you can see the output depends on which function is called

0 Reply

LatinoSunset

April 14, 2025 8:13 pm PDT

neat

0 Reply

andy
April 11, 2025 2:47 am PDT

cool

Last edited 2 months ago by andy

0 Reply

Chayim
April 5, 2025 11:22 am PDT

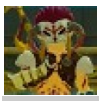
Your definition of what is a lvaue and what is a rvalue is very confusing and not fundamental, in simply put: a lvalue that stands for locate value is a variable that holds a value in memory and a rvalue is a variable itself and that it has no memory allocated.

0 Reply

RSH
Reply to Chayim [13] May 11, 2025 2:58 am PDT

I couldn't agree with this anymore, "an lvalue is an expression that assigns to expression"??? Thats all I understood reading the definitions

0 Reply



Teh Yong Lip
March 18, 2025 6:27 am PDT

There is one great way to demonstrate that C-style strings are rvalue, they have memory addresses, try these codes, it will run!

```
1 std::cout << &"Hello World" << '\n';
2 std::cout << *&"Hello World" << '\n';
```

1 Reply

RURU
Reply to Teh Yong Lip [14] June 19, 2025 10:17 am PDT

i guess u meant l value not r value since strings are having an adress and can be identified with references ...

0 Reply

Andreas Krug
March 12, 2025 1:46 am PDT

Fix typo:

... one with an lvalue refrence parameter -> ... one with an lvalue reference parameter

0 Reply

Nidhi Gupta
February 28, 2025 2:57 pm PST

This chapter deals with lvalue references by first explaining what an lvalue is. A C++ expression is a combination of literals, variables, operators, and function calls that evaluates to a single value. Expressions can return values, incur side effects (such as modifying variables), or evaluate to objects or functions. All expressions have a type and a value category. The type defines the kind of value it returns, and the value category defines if it is an lvalue or an rvalue. Lvalues are named functions or objects that exist outside a single expression, and rvalues are temporary values with no permanent identity. Modifiable lvalues can be assigned new values, but non-modifiable lvalues (e.g., constants) cannot. Rvalues include literals, function return values, and calculated expressions. Operators usually take rvalues as operands, except for assignments, where they require a modifiable lvalue on the left. Lvalues and rvalues are concepts that need to be grasped for more advanced features like move semantics and lvalue references.

0 Reply

Antonia
February 25, 2025 8:24 am PST

I just wanted to say thank you for the tutorials. I tried them before and got very stuck, but now I am writing notes as I go along and it is much clearer. I didn't understand why "Hello" is an lvalue but now I understand that it is a "C-style string literal". No need to respond to this - thank you so much

0 Reply

StepanKo
February 16, 2025 4:34 am PST

I stare at this lesson since Friday. For some reason it doesn't get through. I see the words, but they quickly lose their meaning. All this is simple, but my mind declines to comprehend. I feel like whenever I try to get the whole picture a couple of pieces fall off.

I'll manage. Thank you for your titanic work.

2 Reply

BiscuitRE-L
Reply to StepanKo [15] February 21, 2025 2:33 am PST

Lvalue

```cpp
int x = 10; //stored: has a memory address and can be modified
x = 20; //also lvalue
```

Rvalue

```cpp
int y = x + 5; // 'x + 5' is a temp value(sum if x + 5 is the value that is stored) hence rvalue

10 = x; //rvalues cannot be on the left

x + 5 = 20; //cannot assign temp value to a temp value
```

i also didn't got it at first then i asked gpt to explain..

1 Reply

Sina

February 7, 2025 5:26 am PST

I'm getting a database error accessing lesson 12.3

5 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-expressions/
3. https://www.learncpp.com/cpp-tutorial/increment-decrement-operators-and-side-effects/
4. https://en.cppreference.com/w/cpp/language/value_category
5. https://www.learncpp.com/cpp-tutorial/c-style-array-decay/
6. https://www.learncpp.com/cpp-tutorial/lvalue-references/
7. https://www.learncpp.com/
8. https://www.learncpp.com/cpp-tutorial/introduction-to-compound-data-types/
9. https://www.learncpp.com/value-categories-lvalues-and-rvalues/
10. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/
11. https://gravatar.com/
12. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/#comment-610016
13. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/#comment-609075
14. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/#comment-608642
15. https://www.learncpp.com/cpp-tutorial/value-categories-lvalues-and-rvalues/#comment-607787
16. https://g.ezoic.net/privacy/learncpp.com