Understanding Lvalue & Rvalue in C++

# Introduction

The terms *lvalue* and *rvalue* are not something one runs into often in C/C++ programming, but when one does, it's usually not immediately clear what they mean. The most common place to run into these terms are in compiler error & warning messages. For example, compiling the following with gcc:

int foo() {return 2;}

int main()

{

foo() = 2;

return 0;

}

You get:

test.c: In function 'main':

test.c:8:5: error: *lvalue* required as left operand of assignment

## So Why Do We Care about *lvalue* & *rvalue*?

* *lvalue* and *rvalue* help us to understand C++ construct and explain some similar behaviour of C++. It also helps us to decipher compiler errors or warnings.
* C++11 introduced *rvalue reference*. without clear understanding of *lvalue* & *rvalue*, it's difficult to understand *rvalue reference*.

# Simplified Definition

Every expression in C++ is either an *rvalue* or an *lvalue*. These names are inherited from C and originally had a simple mnemonic purpose: ***lvalues* could stand on the left-hand side of an assignment whereas rvalues could not**.

It's hard to give rigorous definition of *lvalue* & *rvalue*. So, I provide you a simplified definition of them.

*lvalue* - An object that occupies some ***identifiable location in memory***.

*rvalue* - Any object that is not *lvalue*.

*lvalue* and *rvalue* are applied to expressions. Expression can be an object but implicitly object is an *lvalue* or *rvalue*. Following are some examples of expression:

int x;

double \*ptr;

int arr[5] = 43;

int z = x + y;

Now, we can think about *lvalue* & *rvalue* as **name vs temporary**. *lvalue* has **name** and they are persistent. You can reference to that name over and over in your code. And *rvalue* doesn't.

For example:

int x; // 'x' is a lvalue

int \*ptr; // pointer (dereference \*) is a lvalue

double arr[3]; // array is an lvalue.

int y = x; // both 'y' and 'x' are lvalue

All the above example has **name** which we can **refer(use)** over and over in the code.

In other hand, *rvalue* is **temporary**.

For example:

int x = 1243 // 1243 is a rvalue

int z = x + y; // (x + y) is rvalue as it’s a temporary

// object. Its life will end after ';'.

Clear way of distinguish between those two is, ask question to yourself that **can it has address**. If we get address of an expression that is a *lvalue* otherwise it’s a *rvalue*.

In the above example of *lvalue*, we can say:

&(x);

&(\*ptr);

&(arr[3]);

&(y);

So, we can get address of all the above expressions. But we can't get address of

&(1243);

&(x + y);

And they are *rvalue*.

# Basic examples

*lvalue* Examples:

int i; // i is a *lvalue*

int \*p = &i; // i's address is identifiable.

i = 2; // memory content is modified.

void func(int pl); // pl is a ***lvalue***

void funcL(int& pl); // pl is a ***lvalue***

void funcL(const int& pl); // pl is a ***lvalue***

int funcR(int&& pl); // pl is a ***lvalue***

*rvalue* Examples:

int x = 2; // 2 is a rvalue

int y = x + 2; // (x + 2) is a rvalue

int \*p = &( x + 2); // Error

x + 2 = 4; // Error

2 = x; // Error

animal a;

a = animal(); // animal() is rvalue of user

// defined type (class)

int sum(int x, int y) // x & y are lvalue

{ return (x + y); }

int i = sum(3, 4); // sum(3, 4) is rvalue

Operators differ as to whether they require *lvalue* or *rvalue* operands and as to whether they return *lvalues* or *rvalues*. The important point is that **we can use an *lvalue* when an *rvalue* is required, but we cannot use an *rvalue* when an *lvalue* (i.e., a location) is required**. When we use an *lvalue* in place of an *rvalue*, the object’s contents (its value) are used. We have already used several operators that involve *lvalues*.

• Assignment (=) requires a (non *const*) *lvalue* as its left-hand operand and yields its

left-hand operand as an *lvalue*.

• The address-of operator (&) requires an *lvalue* operand and returns

a pointer to its operand as an *rvalue*.

• The built-in dereference (\*) and subscript operators ([]) and the iterator dereference and string and vector subscript operators all yield *lvalues*.

• The built-in and iterator increment (++) and decrement (--) operators require *lvalue* operands and the prefix versions (which are the ones we have used so far) also yield *lvalues*.

As we present the operators, we will note whether an operand must be an *lvalue* and whether the operator returns an *lvalue*.

*lvalues* and *rvalues* also differ when used with decltype. When we apply decltype to an expression (other than a variable), the result is a reference type if the expression yields an *lvalue*. As an example, assume p is an int\*. Because dereference yields an *lvalue*, decltype(\*p) is int&. On the other hand, because the address-of operator yields an *rvalue*, decltype(&p) is int\*\*, that is, a pointer to a pointer to type int.

Note: *lvalue* & *rvalue* are independent of their **type**.

int x = 3; // int type

int& ri=x; // int type

void func(int&& x); // int rvalue reference type

template<typename T>

void funcT(T&& x); // T rvalue reference type

auto&& y = std::move(x); // rvalue reference type

all the above examples are *lvalue* and it’s not dependent of its data type.

Here type of x is int and type of ri is *lvalue reference*. But both are *lvalue*.

For *void func (int&& x),* x is an ***lvalue*** although **type** of x is ***rvalue reference***. Similar for *void funcT (T&& x)* and *auto&& y = std::move(x).*

# Modifiable *lvalue*

Initially when *lvalues* were defined for C, it literally meant "values suitable for ***left-hand-side of assignment***". Later, however, when ISO C added the ***const*** keyword, this definition had to be refined. After all:

const int a = 10; // 'a' is an lvalue

a = 10; // but it can't be assigned!

So a further refinement had to be added. Not all *lvalues* can be assigned to. Those that can are called *modifiable lvalues*.

# Conversions between *lvalues* and *rvalues*

Generally speaking, language constructs operating on object values require *rvalues* as arguments. For example, the binary addition operator (+) takes two *rvalues* as arguments and returns an *rvalue*:

int a = 1; // a is an lvalue

int b = 2; // b is an lvalue

int c = a + b; // + needs rvalues, so a and b are converted

// to rvalues and an rvalue is returned

As we've seen earlier, ***a*** and ***b*** are both *lvalues*. Therefore, in the third line, they undergo an implicit ***lvalue-to-rvalue*** conversion. All *lvalues* that aren't *arrays, functions or of incomplete types* can be converted thus to *rvalues*.

What about the other direction? Can *rvalues* be converted to *lvalues*? Of course not! This would violate the very nature of an *lvalue* according to its definition.

This doesn't mean that *lvalues* can't be produced from *rvalues* by more explicit means. For example, the unary dereference operator (\*) takes an *rvalue* argument but produces an *lvalue* as a result. Consider this valid code:

int arr[] = {1, 2};

int\* p = &arr[0];

\*(p + 1) = 10; // OK: p + 1 is an rvalue, but \*(p + 1)

// is an lvalue

Conversely, the unary address-of operator (&) takes an *lvalue* argument and produces an *rvalue*:

int var = 10;

int\* bad\_addr = &(var + 1); // ERROR: lvalue required as

// unary '&' operand

int\* addr = &var; // OK: var is an lvalue

&var = 40; // ERROR: lvalue required as left

// operand of assignment

The ampersand (&) plays another role in C++ - it allows to define reference types. These are called "*lvalue references*". Non-const *lvalue references* cannot be assigned *rvalues*, since that would require an invalid *rvalue-to-lvalue* conversion:

std::string& sref = std::string(); // ERROR: invalid initialization

// of non-const reference of

// type 'std::string&' from an

// rvalue of type 'std::string'

Constant *lvalue references* can be assigned *rvalues*. Since they're constant, the value can't be modified through the reference and hence there's no problem of modifying an *rvalue*. This makes possible the very common C++ idiom of accepting values by constant references into functions, which avoids unnecessary copying and construction of temporary objects.

## CV-qualified rvalues

If we read carefully the portion of the C++ standard discussing lvalue-to-rvalue conversions [2], we notice it says:

An lvalue (3.10) of a non-function, non-array type T can be converted to an rvalue. [...] If T is a non-class type, the type of the rvalue is the **cv-unqualified** version of T. Otherwise, the type of the rvalue is T.

What is this "*cv-unqualified*" thing? CV-qualifier is a term used to describe const and volatile type qualifiers.

From section 3.9.3:

Each type which is a cv-unqualified complete or incomplete object type or is void (3.9) has three corresponding cv-qualified versions of its type: a const-qualified version, a volatile-qualified version, and a const-volatile-qualified version. [...] The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.9)

But what has this got to do with *rvalues*? Well, in C, *rvalues* never have *cv-qualified* types. Only *lvalues* do. In C++, on the other hand, class *rvalues* can have *cv-qualified* types, but built-in types (like int) can't. Consider this example:

#include <iostream>

class A {

public:

void foo() const { std::cout << "A::foo() const\n"; }

void foo() { std::cout << "A::foo()\n"; }

};

A bar() { return A(); }

const A cbar() { return A(); }

int main()

{

bar().foo(); // calls foo

cbar().foo(); // calls foo const

}

The second call in main actually calls the foo() const method of A, because the type returned by cbar is const A, which is distinct from A. This is exactly what's meant by the last sentence in the quote mentioned earlier. Note also that the return value from cbar is an *rvalue*. So this is an example of a *cv-qualified rvalue* in action.

# Misconception

Instead of discussing what *lvalue* & *rvalue* are, let’s discuss about what they are not. Let’s discuss about some misconception of *lvalue* and *rvalue* that needs to be clear up.

## Misconception 1: function or operator always yields rvalues.

This conclusion is easily drawn from previous examples:

int x = i + 3;

int y = sum(3, 4);

above cases (i + 3) & sum(3, 4) are *rvalues*. But that is not true for all the cases.

int global;

int& func() { return global; }

func() = 50;

Above example func() return a *lvalue* and we can assign value to it. You can say that I’ll never write code like this. Ok, they what about assign value in array using [] operator?

Array[3] = 50; // operator [] almost always generates *lvalue*

## Misconception 2: lvalues are modifiable.

By the definition of *lvalue* in C, *lvalue* means “*value suitable for left-hand-side of assignment*”. But this is not true for C++, because C++ has const type expression. And we can’t modify const type expression (except using constant-ness mechanism). So, *lvalues* are not always modifiable.

const int c = 1; // c is a lvalue

c = 3 // Error, c is not modifiable

## Misconception 3: rvalues are not modifiable

This conception is true for ***build in type expression***.

i + 3 = 6; // Error

sum(3, 4) = 7; // Error

But it’s not true for ***user defined type*** (class)

class animal;

animal().bark();

Above example is syntactically correct. In animal().bark() statement, animal() create an object which is a *rvalue* and bark() may change the state of that animal object.

So, it’s not always true that *rvalues* are not modifiable.

# Reference

Reference for an expression is an another **name** or **alias** of that expression. When we declare a reference to an expression (object), it’s bind with that object. Means if we get the address of the reference it’s same as the address of that object and we can also modify that object through reference.

int x = 5; // declared x

int& xr = x; // bind x with xr

xr = 15; // x value also be 15, as it’s modified via

// it’s reference xr.

Before C++11, we have only one reference and that is *lvalue reference* and we often called it as ***reference***. Now we have only two references: *lvalue reference* and *rvalue reference*.

***lvalue reference*** - Reference that can bind to an *lvalue*.

***rvalue reference*** - Reference that can bind to an *rvalue*.

*lvalue reference* bind with an *lvalue*. we denote *lvalue reference* by (&).

int i;

int &r = i;

int &r = 7; // error: lvalue cannot be bound to rvalue 7

However, we can bind an ***rvalue*** to a ***const lvalue reference*** (const reference):

const int& r = 7; // OK

It's legal since when a compiler sees const, it converts 7 to an *lvalue*, and then assign it to the *reference*.

The rule *"lvalue references can only be bound to lvalues but not rvalues"* equally applies to an argument to a function:

int square(int& x) { return x\*x; }

int i = 7;

square(i); // OK

square(7); // error, 7 is an rvalue and cannot be

// assigned to a reference

*Is there a way to pass 7 to the square function?* Seems I've seen it working.

Yes, we can. The const is here for us!

int square(const int& x) { return x\*x; }

square(i); // OK

square(7); // OK

**So, if we want to increase the life time of a temporary expression**, we have to bind it to a const *lvalue reference* (const &) and it’s prolong as long as the life time of that const *lvalue reference*.

We can’t bind a temporary expression with a modifiable reference (&) and it’s a bad idea as if we pass modifiable reference of a temporary expression, we finish with an object which already deleted.

String& r = s + t; // Error, can’t bind a

// temporary/rvalue

const string& cr = s + t; // Ok, now life time of (s + t)

// increases to life time of cr

Void Func(string& r );

Void CFunc(const string& r);

String s;

Func(s); // Ok

Func( s + t); // Error

CFunc(s); // Ok

CFunc( s + t); // Ok

To support *move operations*, the new standard introduced a new kind of reference, an *rvalue reference*. **An *rvalue reference* is a reference that must be bound to an rvalue**. An *rvalue reference* is obtained by using && rather than &. As we’ll see, *rvalue references* have the **important property that they may be bound only to an object that is about to be destroyed**. As a result, we are free to “move” resources from an *rvalue reference* to another object.

Recall that *lvalue* and *rvalue* are properties of an expression. Some expressions yield or require *lvalues*; others yield or require *rvalues*. Generally speaking, **an *lvalue* expression refers to an object’s identity** whereas **an *rvalue* expression refers to an object’s value**.

Like any reference, an *rvalue reference* is just another name for an object. As we know, we cannot bind regular references—which we’ll refer to as *lvalue references* when we need to distinguish them from *rvalue references*—to expressions that require a conversion, to literals, or to expressions that return an *rvalue*. *rvalue references* have the opposite binding properties: We can bind an *rvalue reference* to these kinds of expressions, but we cannot directly bind an *rvalue reference* to an *lvalue*:

int i = 42;

int &r = i; // ok: r refers to i

int &&rr = i; // error: cannot bind an rvalue reference

// to an lvalue

int &r2 = i \* 42; // error: i \* 42 is an rvalue

const int &r3 = i \* 42; // ok: we can bind a reference to const

// to an rvalue

int &&rr2 = i \* 42; // ok: bind rr2 to the result of the

// multiplication

Functions that return *lvalue references*, along with the assignment, subscript, dereference, and prefix increment/decrement operators, are all examples of expressions that return *lvalues*. We can bind an *lvalue reference* to the result of any of these expressions.

Functions that return a non-reference type, along with the arithmetic, relational, bitwise, and postfix increment/decrement operators, all yield *rvalues*. We cannot bind an *lvalue reference* to these expressions, but we can bind either an *lvalue reference* to const or an *rvalue reference* to such expressions.

# Bibliography

**Video:**

*Advanced C++: Understanding rvalue and lvalue* by Bo Qian

<https://www.youtube.com/watch?annotation_id=annotation_1694685217&feature=iv&src_vid=IOkgBrXCtfo&v=UTUdhjzws5g>

*C9 Lectures:* Stephan T. Lavavej *- Standard Template Library (STL), 9 of n* by Stephan T. Lavavej

<https://channel9.msdn.com/Series/C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-/C9-Lectures-Stephan-T-Lavavej-Standard-Template-Library-STL-9-of-n>

**Article:**

*Understanding lvalues and rvalues in C and C++* by Eli Bendersky

<https://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c>

*C++11/C++14 4. rvalue and lvalue – 2017* by K Hong

<http://www.bogotobogo.com/cplusplus/C11/4_C11_Rvalue_Lvalue.php>

*Lvalues and Rvalues* by Mikael Kilpeläinen

<https://accu.org/index.php/journals/227>

*C++ Rvalue References Explained* by Thomas Becker

<http://thbecker.net/articles/rvalue_references/section_01.html>

**Book:**

*Effective modern C++* by Scott Meyers

<https://www.aristeia.com/EMC++.html>

*C++ Primer (5th edition)* by B. Lippman Barbara E. Moo, Josée Lajoie, Stanley

<https://www.safaribooksonline.com/library/view/c-primer-fifth/9780133053043/>