Understanding the meaning of lvalues and rvalues in C++

Monocasual Laboratories

— Written by Triangles on September 15, 2016 • updated on February 26, 2020 • ID 42 —

A lightweight introduction to a couple of basic C++ features that act as a foundation for bigger structures.

I have been struggling with the concepts of *lvalue* and *rvalue* in C++ since forever. I think that now is the right time to understand them for good, as they are getting more and more important with the evolution of the language.

Once the meaning of lvalues and rvalues is grasped, you can dive deeper into advanced C++ features like *move semantics* and *rvalue references* (more on that in future articles).

Lvalues and rvalues: a friendly definition

First of all, let's keep our heads away from any formal definition. In C++ an *lvalue* is something that points to a specific memory location. On the other hand, a *rvalue* is something that doesn't point anywhere. In general, rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables. It's also fun to think of lvalues as *containers* and rvalues as *things contained in the containers*. Without a container, they would expire.

Let me show you some examples right away.

```
int x = 666;
```

Here `666` is an rvalue; a number (technically a *literal constant*) has no specific memory address, except for some temporary register while the program is running. That number is assigned to `x`, which is a variable. A variable has a specific memory location, so its an lvalue. C++ states that an assignment requires an lvalue as its left operand: this is perfectly legal.

Then with `x`, which is an lvalue, you can do stuff like that:

```
int* y = &x;
```

Here I'm grabbing the the memory address of `x` and putting it into `y`, through the address-of operator `&`. It takes an lvalue argument and produces an rvalue. This is another perfectly legal operation: on the left side of the assignment we have an lvalue (a variable), on the right side an rvalue produced by the address-of operator.

However, I can't do the following:

```
int y;
666 = y;
```

Yeah, that's obvious. But the technical reason is that `666`, being a literal constant — so an rvalue, doesn't have a specific memory location. I am assigning `y` to nowhere.

This is what GCC tells me if I run the program above:

```
error: lvalue required as left operand of assignment
```

He is damn right; the left operand of an assigment always require an lvalue, and in my program I'm using an rvalue (`666`).

I can't do that either:

```
int* y = &666;
```

GCC says:

```
error: lvalue required as unary '&' operand`
```

He is right again. The `&` operator wants an lvalue in input, because only an lvalue has an address that `&` can process.

Functions returning lvalues and rvalues

We know that the left operand of an assigment must be an lvalue. Hence a function like the following one will surely throw the `lvalue required as left operand of assignment` error:

```
int setValue()
{
    return 6;
}



setValue() = 3;
```

Crystal clear: `setValue()` returns an rvalue (the temporary number `6`), which cannot be a left operand of assignment. Now, what happens if a function returns an lvalue instead? Look closely at the following snippet:

```
int global = 100;

int& setGlobal()
{
    return global;
}



setGlobal() = 400;
```

It works because here `setGlobal` returns a reference, unlike `setValue()` above. A reference is something that points to an existing memory location (the `global` variable) thus is an lvalue, so it can be assigned to. Watch out for `&` here: it's not the address-of operator, it defines the type of what's returned (a reference).

The ability to return lvalues from functions looks pretty obscure, yet it is useful when you are doing advanced stuff like implementing some overloaded operators. More on that in future chapters.

Lvalue to rvalue conversion

An lvalue may get converted to an rvalue: that's something perfectly legit and it happens quite often. Let's think of the addition `+` operator for example. According to the C++ specifications, it takes two rvalues as arguments and returns an rvalue.

Let's look at the following snippet:

```
int x = 1;
int y = 3;
int z = x + y;
```

Wait a minute: `x` and `y` are lvalues, but the addition operator wants rvalues: how come? The answer is quite simple: `x` and `y` have undergone an implicit **lvalue-to-rvalue conversion**. Many other operators perform such conversion — subtraction, addition and division to name a few.

Lvalue references

What about the opposite? Can an rvalue be converted to lvalue? Nope. It's not a technical limitation, though: it's the programming language that has been designed that way.

In C++, when you do stuff like

```
int y = 10;
int& yref = y;
yref++;
```

you are declarying `yref` as of type `int&`: a reference to `y`. It's called an **lvalue reference**. Now you can happily change the value of `y` through its reference `yref`.

We know that a reference must point to an existing object in a specific memory location, i.e. an lvalue. Here `y` indeed exists, so the code runs flawlessly.

Now, what if I shortcut the whole thing and try to assign `10` directly to my reference, without the object that holds it?

```
int& yref = 10;
```

On the right side we have a temporary thing, an rvalue that needs to be stored somewhere in an lvalue.

On the left side we have the reference (an lvalue) that *should* point to an existing object. But being `10` a numeric constant, i.e. without a specific memory address, i.e. an rvalue, the expression clashes with the very spirit of the reference.

If you think about it, that's the forbidden conversion from rvalue to lvalue. A volatile numeric constant (rvalue) should become an

lvalue in order to be referenced to. If that would be allowed, you could alter the value of the numeric constant through its reference. Pretty meaningless, isn't it? Most importantly, what would the reference point to once the numeric value is gone?

The following snippet will fail for the very same reason:

```
void fnc(int& x)
{
}

int main()
{
    fnc(10);



}
```

I'm passing a temporary rvalue (`10`) to a function that takes a reference as argument. Invalid rvalue to lvalue conversion. There's a workaround: create a temporary variable where to store the rvalue and then pass it to the function (as in the commented out code). Quite inconvenient when you just want to pass a number to a function, isn't it?

Const lvalue reference to the rescue

That's what GCC would say about the last two code snippets:

```
error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
```

GCC complains about the reference not being **const**, namely a **constant**. According to the language specifications, *you are allowed to bind a const lvalue to an rvalue*. So the following snippet works like a charm:

```
const int& ref = 10;
```

And of course also the following one:

```
void fnc(const int& x)
{
```

```
}

int main()
{
    fnc(10);
}
```

The idea behind is quite straightforward. The literal constant `10` is volatile and would expire in no time, so a reference to it is just meaningless. Let's make the reference itself a constant instead, so that the value it points to can't be modified. Now the problem of modifying an rvalue is solved for good. Again, that's not a technical limitation but a choice made by the C++ folks to avoid silly troubles.

This makes possible the very common C++ idiom of accepting values by constant references into functions, as I did in the previous snipped above, which avoids unnecessary copying and construction of temporary objects.

Under the hood the compiler creates an hidden variable for you (i.e. an lvalue) where to store the original literal constant, and then bounds that hidden variable to your reference. That's basically the same thing I did manually in a couple of snippets above. For example:

```
const int& ref = 10;
```

```
int __internal_unique_name = 10;
const int& ref = __internal_unique_name;
```

Now your reference points to something that exists for real (until it goes out of scope) and you can use it as usual, except for modifying the value it points to:

```
const int& ref = 10;
std::cout << ref << "\n";
std::cout << ++ref << "\n";
```

Conclusion

Understanding the meaning of lvalues and rvalues has given me the chance to figure out several of the C++'s inner workings. C++11 pushes the limits of rvalues even further, by introducing the concept of *rvalue references* and *move semantics*, where — surprise! — rvalues too are modifiable. I will restlessly dive into that minefield in one of my next articles.

Sources

Thomas Becker's Homepage - *C++ Rvalue References Explained* (link)
Eli Bendersky's website - *Understanding lvalues and rvalues in C and C++* (link)
StackOverflow - *Rvalue Reference is Treated as an Lvalue?* (link)
StackOverflow - *Const reference and lvalue* (link)
CppReference.com - *Reference declaration* (link)