# C++ rvalue references and move semantics for beginners

*Monocasual Laboratories*

A collection of personal notes and thoughts on rvalue references, their role in move semantics and how they can significantly increase the performance of your applications.

In my previous article Understanding the meaning of lvalues and rvalues in C++ I had the chance to explain to myself the logic behind *rvalues*. The core idea is that in C++ you will find such temporary, short-lived values that you cannot alter in any way.

Surprisingly, modern C++ (C++0x and greater) has introduced **rvalue references**: a new type that can bind to temporary objects, giving you the ability to modify them. Why?

Let's begin this journey with a little brush up of temporary values:

```
int x = 666;
int y = x + 5;

std::string s1 = "hello ";
std::string s2 = "world";
std::string s3 = s1 + s2;

std::string getString() {
  return "hello world";
}
std::string s4 = getString();
```

On line (1) the literal constant `666` is an rvalue: it has no specific memory address, except for some temporary register while the program is running. It needs to be stored in a lvalue (`x`) to be useful. Line (4) is similar, but here the rvalue is not hard-coded, rather it is being returned by the function `getString()`. However, as in line (1), the temporary object must be stored in an lvalue (`s4`) to be meaningful.

Lines (2) and (3) seem more subtle: the compiler has to create a temporary object to hold the result of the `+` operator. Being a temporary one, the output is of course an rvalue that must be stored somewhere. And that's what I did by putting the results in `y` and `s3` respectively.

## Introducing the magic of rvalue references

The traditional C++ rules say that you are allowed to take the address of an rvalue only if you store it in a **const** (immutable) variable. More technically, *you are allowed to bind a const lvalue to an rvalue*. Consider the following example:

```
int& x = 666;
const int& x = 666;
```

The first operation is wrong: it's an invalid initialization of non-const reference of type `int&` from an rvalue of type `int`. The second line is the way to go. Of course, being `x` a constant, you can't alter it.

C++0x has introduced a new type called **rvalue reference**, denoted by placing a double ampersand `&&` after some type. Such rvalue reference lets you modify the value of a temporary object: it's like removing the `const` attribute in the second line above!

Let's play a bit with this new toy:

```
  std::string   s1     = "Hello ";
  std::string   s2     = "world";
  std::string&& s_rref = s1 + s2;
  s_rref += ", my friend";
  std::cout << s_rref << '\n';
```

Here I create two simple strings `s1` and `s2`. I join them and I put the result (a temporary string, i.e. an rvalue) into `std::string&& s_rref`. Now `s_rref` is a reference to a temporary object, or an rvalue reference. There are no `const` around it, so I'm free to modify the temporary string to my needs. This wouldn't be possible without rvalue references and its double ampersand notation. To better distinguish it, we refer to traditional C++ references (the single-ampersand one) as **lvalue references**.

This might seem useless at a first glance. However rvalue references pave the way for the implementation of **move semantics**, a technique which can significantly increase the performance of your applications.

## Move semantics, the scenic route

Move semantics is a new way of moving resources around in an optimal way by avoiding unnecessary copies of temporary objects, based on rvalue references. In my opinion, the best way to understand what move semantics is about is to build a wrapper class around a *dynamic resource* (i.e. a dynamically allocated pointer) and keep track of it as it moves in and out functions. Keep in mind however that move semantics does not apply only to classes!

That said, let's take a look at the following example:

```
class Holder
{
  public:

    Holder(int size)
    {
      m_data = new int[size];
      m_size = size;
    }

    ~Holder()
    {
      delete[] m_data;
    }

  private:

    int*   m_data;
    size_t m_size;
};
```

It is a naive class that handles a dynamic chunk of memory: nothing fancy so far, except for the allocation part. When you choose to manage the memory yourself you should follow the so-called Rule of Three. This rule states that if your class defines one or more of the following methods it should probably explicitly define all three:

- destructor;
- copy constructor;

- copy assignment operator.

A C++ compiler will generate them by default if needed, in addition to the constructor and other functions we don't care about right now. Unfortunately the default versions are just "not enough" when your class deals with dynamic resources. Indeed, the compiler couldn't generate a constructor like the one in the example above: it doesn't know anything about the logic of our class.

## Implementing the copy constructor

Let's stick to the Rule of Three and implement the copy constructor first. As you may know, the copy constructor is used to create a *new* object from another *existing* object. For example:

```
Holder h1(10000);
Holder h2 = h1;
Holder h3(h1);
```

How a copy constructor would look like:

```
Holder(const Holder& other)
{
  m_data = new int[other.m_size];
  std::copy(other.m_data, other.m_data + other.m_size, m_data);
  m_size = other.m_size;
}
```

Here I'm initializing a new `Holder` object out of the existing one passed in as `other`: I create a new array of the same size (1) and then I copy the actual data from `other.m_data` to `m_data` (i.e. `this.m_data`) (2).

## Implementing the assignment operator

It's now time for the assignment operator, used to replace an *existing* object with another *existing* object. For example:

```
Holder h1(10000);
Holder h2(60000);
h1 = h2;
```

How an assigment operator would look like:

```
Holder& operator=(const Holder& other)
{
  if(this == &other) return *this;
  delete[] m_data;
  m_data = new int[other.m_size];
  std::copy(other.m_data, other.m_data + other.m_size, m_data);
  m_size = other.m_size;
  return *this;
}
```

First of all a little protection against self-assignment (1). Then, since we are replacing the content of this class with another one, let's wipe out the current data (2). What's left is just the same code we wrote in the copy constructor. By convention a reference to this class is returned (3).

The key point of the copy constructor and the assignment operator is that they both receive a `const` reference to an object in input and make *a copy* out of it for the class they belong to. The object in input, being a constant reference, is of course left untouched.

## The limitations of our current class design

Our class is good to go, but it lacks of some serious optimization. Consider the following function:

```
Holder createHolder(int size)
{
  return Holder(size);
}
```

It returns a `Holder` object *by value*. We know that when a function returns an object by value, the compiler has to create a temporary — yet fully-fledged — object (rvalue). Now, our `Holder` is a heavy-weight object due to its internal memory allocation, which is a very expensive task: returning such things by value with our current class design would trigger multiple expensive memory allocations, which is rarely a great idea. How come? Consider this:

```
int main()
{
```

```
  Holder h = createHolder(1000);
}
```

A temporary object coming out from `createHolder()` is passed to the copy constructor. According to our current design, the copy constructor allocates its own `m_data` pointer by *copying* the data from the temporary object. Two expensive memory allocations: a) during the creation of the temporary, b) during the actual object copy-construct operation.

The same *copy* procedure occurs within the assignment operator:

```
int main()
{
  Holder h = createHolder(1000);
  h = createHolder(500);
}
```

The code inside our assignment operator wipes the memory out and then reallocates it from scratch by *copying* the data from the temporary object. Yet another two expensive memory allocations: a) during the creation of the temporary, b) in the actual object assignment operator.

Too many expensive copies! We already have a fully-fledged object, the temporary and short-lived one returning from `createHolder()`, built for us by the compiler: it's an rvalue that will fade away with no use at the next instruction: why, during the construction/assignment stages, don't we *steal* — or *move* the allocated data inside the temporary object instead of making an expensive copy out of it?

In the old days of C++ there was no way to optimize this out: returning heavy-weight objects by value was simply a no-go. Fortunately in C++11 and greater we are allowed (and encouraged) to do this, by improving our current `Holder` class with move semantics. In a nutshell, we will steal existing data from temporary objects instead of making useless clones. Don't copy, just *move*, because moving is always cheaper.

## Implementing move semantics with rvalue references

Let's spice up our class with move semantics: the idea is to add new versions of the copy constructor and assignment operator so that they can take a temporary object in input to *steal* data from. To steal data means to modify the object the data belongs to: how can we modify a temporary object? By using rvalue references!

At this point we naturally follow another C++ pattern called the Rule of Five. It's an extension to the Rule of Three seen before and it states that any class for which move semantics are desirable, has to declare two additional member functions:

- the **move constructor** — to construct *new* objects by stealing data from temporaries;
- the **move assignment operator** — to replace *existing* objects by stealing data from temporaries.

## Implementing the move constructor

A typical move constructor:

```
Holder(Holder&& other)
{
  m_data = other.m_data;
  m_size = other.m_size;
  other.m_data = nullptr;
  other.m_size = 0;
}
```

It takes in input an rvalue reference to another `Holder` object. This is the key part: being an rvalue reference, we can modify it. So let's steal its data first (1), then set it to null (2). No deep copies here, we have just moved resources around! It's important to set the rvalue reference data to some valid state (2) to prevent it from being accidentally deleted when the temporary object dies: our `Holder` destructor calls `delete[] m_data`, remember? In general, for reasons that will become more clear in a few paragraphs, it's a good idea to always leave the objects being stolen from in some well-defined state.

## Implementing the move assignment operator

The move assignment operator follows the same logic:

```
Holder& operator=(Holder&& other)
{
  if (this == &other) return *this;

  delete[] m_data;

  m_data = other.m_data;
```

```
  m_size = other.m_size;

  other.m_data = nullptr;
  other.m_size = 0;

  return *this;
}
```

We steal data (2) from the other object coming in as an rvalue reference, after a cleanup of the existing resources (1). Let's not forget to put the temporary object to some valid state (3) as we did in the move constructor. Everything else is just regular assignment operator duty.

Now that we have our new methods in place, the compiler is smart enough to detect whether you are creating an object with a temporary value (rvalue) or a regular one (lvalue) and trigger the proper constructor/operator accordingly. For example:

```
int main()
{
  Holder h1(1000);
  Holder h2(h1);
  Holder h3 = createHolder(2000);

  h2 = h3;
  h2 = createHolder(500);
}
```

# Where and when move semantics apply

Move semantics provide a smarter way of passing heavy-weight things around. You create your heavy-weight resource only once and then you move it where needed in a natural way. As I said before, move semantics is not only about classes. You can make use of it whenever you need to change the ownership of a resource across multiple areas of your application. However keep in mind that, unlike a pointer, you are not sharing anything: if object A steals data from object B, data in object B no longer exists, thus is no longer valid. As we know this is not a problem when dealing with temporary objects, but you can also steal from regular ones. We will see how shortly.

## I tried your code: the move constructor never gets called!

That's right. If you run the last snippet above you will notice how the move constructor does not get called during (1). The regular constructor is called instead: this is due to a trick called **Return Value Optimization (RVO)**. Modern compilers are able to detect that you are returning an object by value, and they apply a sort of return shortcut to avoid useless copies.

You can tell the compiler to bypass such optimization: for example, GCC supports the `-fno-elide-constructors` flag. Compile the program with such flag enabled and run it again: the amount of constructor/destructor calls will increase noticeably.

### Why should I care implementing move semantics if the RVO does its optimization job by default?

RVO is only about return values (output), not function parameters (input). There are many places where you may pass movable objects as input parameters, which would make the move constructor and the move assignment operator come into play, if implemented. The most important one: the Standard Library. During the upgrade to C++11 all the algorithms and containers in there were extended to support move semantics. So if you use the Standard Library with classes that follow the Rule of Five you will gain an important optimization boost.

## Can I move lvalues?

Yes you can, with the utility function `std::move` from the Standard Library. It is used to convert an lvalue into an rvalue. Say we want to steal from an lvalue:

```
int main()
{
  Holder h1(1000);
  Holder h2(h1);
}
```

This will not work: since `h2` receives an lvalue in input, the copy constructor is being triggered. We need to force the move constructor on `h2` in order to make it steal from `h1`, so:

```
int main()
{
  Holder h1(1000);
  Holder h2(std::move(h1));
}
```

Here `std::move` has converted the lvalue `h1` into an rvalue: the compiler sees such rvalue in input and then triggers the move constructor on `h2`. The object `h2` will steal data from `h1` during its construction stage.

Mind that at this point `h1` is a *hollow* object. However, we did a good thing when in our move constructor we set the stolen object's data to a valid state (`other.m_data = nullptr`, remember?). Now you may want to reuse `h1`, test it in some way or let it go out of scope without causing nasty crashes.

# Final notes and possible improvements

This article is way too long and I've only scratched the surface of move semantics. What follows is a quick list of additional concepts I will further investigate in the future.

## We did RAII in our basic `Holder` example

**Resource Acquisition Is Initialization (RAII)** is a C++ technique where you wrap a class around a *resource* (file, socket, database connection, allocated memory, ...). The resource is initialized in the class constructor and cleaned up in the class destructor. This way you are sure to avoid resource leaks. More information: here.

## Mark you move constructors and move assignment operators with `noexcept`

The C++11 keyword `noexcept` means "this function will never throw exceptions". It is used to optimize things out. Some people say that move constructors and move assignment operators should never throw. Rationale: you should not allocate memory or call other code in there. You should only copy data and set the other object to null, i.e. non-throwing operations. More information: here, here.

## Further optimizations and stronger exception safety with copy-and-swap idiom

All the constructors/assignment operators in the `Holder` class are full of duplicate code, which is not so great. Moreover, if the allocation throws an exception in the copy assignment operator the source object might be left in a bad state. The **copy-and-swap idiom** fixes both issues, at the cost of adding a new method to the class. More information: here, here.

## Perfect forwarding

This technique allows you to move your data across multiple template and non-template functions without wrong type conversions (i.e. perfectly). More information: here, here.

## Sources

Stack Overflow - *When is an rvalue evaluated?* (link)
Mikw's C++11 blog - *Lesso #5: Move Semantics* (link)
Artima - *A Brief Introduction to Rvalue References* (link)
Stack Overflow - *C++11 rvalues and move semantics confusion (return statement)* (link)
Cpp-patterns - *The rule of five* (link)
open-std.org - *A Brief Introduction to Rvalue References* (link)
Microsoft - *Rvalue Reference Declarator: &&* (link)
Wikipedia - *Rule of three (C++ programming)* (link)
Stack Overflow - *What are all the member-functions created by compiler for a class? Does that happen all the time?* (link)
cplusplus.com - *Copy constructors, assignment operators, and exception safe assignment* (link)
Stack Overflow - *What is the copy-and-swap idiom?* (link)
Wikipedia - *Assignment operator (C++)* (link)
Stack Overflow - *When the move constructor is actually called if we have (N)RVO?* (link)
cppreference.com - *The rule of three/five/zero* (link)
cprogramming.com - *Move semantics and rvalue references in C++11* (link)
Stack Overflow - *What is std::move(), and when should it be used?* (link)