M. D. Wilson, "Move Constructors"

Copy Semantics

All types that support value-semantics [1] provide copy constructors and copy assignment operators, or have them provided by the compiler [1]. Consider the following class:

```
class Simple
{
public:
    explicit Simple(int val)
        : m_val(val)
    {}

    Simple(Simple const &rhs)
        : m_val(rhs.m_val)
    {}

protected:
    int m_val;
};
```

The author of the class has specified the copy constructor, but left it to the compiler to provide the copy assignment-operator [2]. It will be equivalent to the hand written form as follows.

```
class Simple
{
public:
    ...

    Simple &operator =(Simple const &rhs)
    {
        m_val = rhs.m_val;

        return *this;
    }

    ...
};
```

Move Semantics

In a copy constructor, the instance being copied from – the rvalue (`rhs` in the examples above) – is unaltered. The same is true for copy assignment-operators. However, there are occasionally good reasons [3] for specifying and using constructors and/or assignment-operators where the rvalue is altered. Commonly, the values or resources are removed from the rvalue and into the copied-to instance – the lvalue (the instance on which the methods are called) – hence the terms "move constructor" and "move assignment-operator" to describe these operations.

Adapting the previous class with move semantics in the constructor and assignment-operators requires the author to explicitly provide both functions, since the compiler can provide only the copy variants. We might imagine the implementation as follows:

```
class Simple
{
public:
    explicit Simple(int val)
        : m_val(val)
    {}

    Simple(Simple &rhs)
        : m_val(rhs.m_val)
    {
        rhs.m_val = 0; // "Release" the resource
    }

    Simple &operator =(Simple &rhs)
    {
        m_val = rhs.m_val;

        rhs.m_val = 0; // "Release" the resource

        return *this;
    }

protected:
    int m_val;
};
```

Note that both functions now take a non-const argument, so that they can modify the state of the rvalue. In this example, the resource being held is an integer value, and so the "release" of this resource is to set it to 0. Of course, there are much more sophisticated examples of the use of move constructors and move assignment-operators, where the managed resource may be a file, a window, a container iteration state, or an object allocated on the heap. These two functions perform move semantics correctly in the following two scenarios:

```
Simple          s1(1);
Simple          s2(s1); // Move constructor called.
Simple          s3(2);

s1 = s3;                // Move assignment operator called
```

Move constructors and assignment operators are required in other circumstances to these, however, and that is where some of the challenges can be witnessed.

Moving Problems

Alas, only two compilers – Digital Mars C/C++ [4] and Watcom C/C++ 12.0 [WATCOM12] – fully support move semantics [5], and only two others do partially. Intel C/C++ and Visual C++ (except version 7.1) support move semantics when language extensions are enabled [6]. (Table 1 lists the support for many popular compilers.)

Table 1

Compiler	Move Semantics	Move + Copy Constructors*
Borland C/C++ 5.5 – 5.6	No	Yes
Comeau C++ 4.30b, 4.3.1	No	Yes
Digital Mars C/C++ 8.26 – 8.34	Yes	Yes
GCC 2.95, 3.2	No	Yes
Intel C/C++ 6	Yes, but via RVO [9] (see text)	Yes
(with language extensions)	Yes	Yes
Metrowerks CodeWarrior 7 (2.4), 8 (3.0)	No	Yes
Microsoft Visual C++ 5, 6, .7.0 and 7.1	No	Only 7.0
(with language extensions)	Yes	Only 7.0
Microsoft Visual C++ 7.1	No	Yes
Watcom C/C++ 12.0	Yes	Yes

*Note that none of the tested compilers selected the move-constructor in a situation where either would suffice, so this support is all but useless.

The general lack of support is arguably reasonable [7], according to the rules of the language, which (for good reason) stipulates that temporary objects cannot be bound to non-const reference parameters [8]. For example, consider the following function that returns an instance of `Simple`, and an extract of some client code:

```
Simple MakeSimple(int i)
{
    return Simple(i); // "move-ctor" called on explicit temporary as return value
};


...

    Simple  s1(MakeSimple(10)); // "move-ctor" called on return into s1

...
```

There are two points at which the (move) constructor would be called, as noted (although some compilers may be able to apply the Return Value Optimisation [9] to get rid of the second one). Actually, the compilers exhibit a variety of behaviours in this case (as shown in Tables 2 & 3).

Interestingly, without language extensions [6] enabled Intel C++ warns that it has elided the inaccessible copy-constructor function whilst performing the RVO – "warning #734: "Simple::Simple(const Simple &)", required for copy that was eliminated, is inaccessible" – which seems a little scary. The code is only rendered legal by optimising out the illegal part!

Digital Mars and Watcom performs an excellent job of optimisation, rendering the function call, the temporary and the two (move) constructors down to a single inline construction (`Simple(int)`) call. To be fair, with language extensions [6] enabled, Visual C++ (5.0 – 7.0) performs the same optimisation, though version 7.1 does not do so.

Const Casting

So the question, then, is how can move semantics be achieved, without resorting to compiler-specific extensions, and for all compilers? There are two ways, the simple (read quick and dirty) and the sophisticated. The simple version involves accepting the compiler's dictates, and then casting away the constness of the rvalue to release its resource. In other words, copy syntax with move semantics. For example

```
class Simple
{
public:
    ...

    Simple &operator =(Simple const &rhs) // A promise not to change ...
    {
        m_val = rhs.m_val;

        const_cast<Simple&>(rhs).m_val = 0; // ... broken here!

        return *this;
    }

    ...
};
```

Needless to say, this is a very unpleasant way to achieve move semantics. It can cause many problems, not least of which is that it switches off the compiler's ability to warn you about inappropriate use. For example, if we were to write the following innocent looking function

```
template <typename T>
void get_type_size(T const &t_)
{
    T   t(t_);

    printf("sizeof(t): %d\n", sizeof(t));
}
```

it would successfully compile with our revised class, due to its copy syntax. Internally to the function, the implementation uses move semantics, so client code of your classes would experience surprising results, and your libraries would swiftly fall (from misuse) into disuse.

Despite this serious criticism, there are a limited few valid uses of this technique, such as in the implementation of supporting classes for other components. In such cases it is a straightforward matter to implement functions and macros that resolve to using casts on compilers that do not support move-semantics, and to straight code on those that do. The STLSoft [10] libraries do just this in the form of the `stlsoft_define_move_rhs_type()` macro and the `move_lhs_from_rhs()` function.

Moving By Proxy (Reference Transfer)

As described above, the const-cast technique is of limited utility when it comes to implementing robust and usable classes with move semantics (and without nasty surprises). A better technique is to use proxy (also called reference) objects that act as intermediaries in the construction/assignment, taking responsibility for the (reference(s) to the) resources during the transfer – hence Reference Transfer. Because the proxy objects are passed by value, there are no issues with the constness, or lack thereof, of the arguments and return values of functions in which they are involved. To change the `Simple` class to use this technique, we would see something like the following:

```
class Simple
{
private:
    struct Simple_proxy
    {
        int     m_val;
    };

public:
    explicit Simple(int val)
        : m_val(val)
    {}

    Simple(Simple &rhs)      // Move constructor
        : m_val(rhs.m_val)
    {
        rhs.m_val = 0;
    }

    Simple(Simple_proxy sp)   // Proxy move constructor
        : m_val(sp.m_val)
    {
        sp.m_val = 0;
    }

    Simple &operator =(Simple &rhs) // Move assignment operator
    {
        m_val = rhs.m_val;

        rhs.m_val = 0;

        return *this;
    }

    Simple operator =(Simple_proxy sp) // Proxy move assignment operator
    {
        m_val = sp.m_val;

        sp.m_val = 0;

        return *this;
    }

    operator Simple_proxy () // "to-proxy" implicit conversion operator
    {
        Simple_proxy    sp;

        sp.m_val = m_val;
```

```
        return sp;
    }

protected:
    int m_val;
};
```

The compiler does not generate the copy-constructor, because we have provided a move-constructor (which will be used where possible). Where the compiler would normally be compelled to use the copy-constructor (by dint of the temporaries being unacceptable for the move constructors), it will select instead the `operator Simple_proxy()` on the rvalue matched with the `Simple(Simple_proxy sp)` constructor.

This technique is to be found in the latest incarnation of the standard library's `auto_ptr` [1] smart-pointer template; earlier implementations of this class used const-casting. (See [11, 12] for an interesting view on the history, and [13] for the modern version of the template.)

Since the `Simple_proxy` class is a simple `struct`, and does not have value semantics [1] – or, rather, does not in the general case, e.g. when used for pointer types - its use could result in loss of the managed resource if the `operator Simple_proxy()` method was invoked in isolation. This is prevented by declaring `Simple_proxy` as a member class, whose access is private, making inadvertent call of the conversion nigh-on impossible.

Return Value (and other) Optimisations

We have seen with Digital Mars, Intel and Watcom (and language-extended Visual C++ 5.0-7.0) that compiler optimisations are having an effect on the way the code is generated. It would be worthwhile, then, to see exactly what optimisations are being performed. The program shown below exercises the semantics of the `Simple` class in a number of construction and assignment scenarios.

```
Simple MakeSimple2(int i);

inline Simple MakeSimple(int i)
{
    return Simple(i);
}

void UseSimple(Simple &s)
{
    Simple  simple(s);

    simple = s;
}

int main(int, char**)
{
1   Simple          s1(1);
2   Simple          s2(s1);
3   Simple          s3(2);
4   Simple const    s4(3);

5   s2 = s3;
```

```
6   UseSimple(s3);

7   Simple          s5(MakeSimple(4));

8   s5 = MakeSimple(5);

9   Simple          s6(MakeSimple2(6));

10  s6 = MakeSimple2(7);

    return 0;
}

Simple MakeSimple2(int i)
{
    return Simple(i);
}
```

In the program there are ten statements in which `Simple` instances are either created or assigned. All the compilers tested perform the same operations for the first five statements, as shown in Table 2.

Table 2

Compiler	**1, 3, 4**	**2**
All compilers	Simple::Simple(int)	Simple::Simple(Simple&)
	5	**6**
	Simple::operator =(Simple &)	UseSimple(Simple &) Simple::Simple(Simple&) Simple::operator =(Simple &)

As soon as temporaries and function calls are involved, however, the differences are quite significant, as shown in Table 3.

The joint "winners" are Digital Mars, Visual C++ (5.0-7.0 with language extensions - **/Ze**) and Watcom, followed closely by Intel and CodeWarrior, and that Visual C++ (5.0-7.0 without language extensions - **/Za** - , and 7.1) is joint last with GCC. (Please remember that this ordering reflects only the function-call optimisations: it does not in any way imply an ordering to the efficiencies of other optimisations.)

Table 3

Compiler	7, 9	8, 10
Borland C/C++ 5.51, 5.6	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy() Simple::Simple(Simple_proxy)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::operator =(Simple_proxy) Simple::Simple(Simple &)
Digital Mars 8.30	MakeSimple/MakeSimple2() Simple::Simple(int)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator =(Simple&)
GCC 2.95, 3.2	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy) Simple::operator Simple_proxy () Simple::operator =(Simple_proxy) Simple::Simple(Simple &)
Intel C/C++ 7.0 **(with or without –Za)**	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::Simple(Simple &)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator =(Simple&)
CodeWarrior 8 (3.0)	MakeSimple/MakeSimple2() Simple::Simple(int)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::operator =(Simple_proxy) Simple::Simple(Simple &)
Visual C++ 5.0/6.0/7.0 (/Ze)	MakeSimple/MakeSimple2() Simple::Simple(int)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator =(Simple&)
Visual C++ 5.0/6.0/7.0 (/Za)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy) Simple::operator Simple_proxy () Simple::operator =(Simple_proxy) Simple::Simple(Simple &)
Visual C++ 7.1 **(with or without –Za)**	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator Simple_proxy () Simple::Simple(Simple_proxy) Simple::operator Simple_proxy () Simple::operator =(Simple_proxy) Simple::Simple(Simple &)
Watcom C/C++ 12.0	MakeSimple/MakeSimple2() Simple::Simple(int)	MakeSimple/MakeSimple2() Simple::Simple(int) Simple::operator =(Simple&)

Placing Visual C++ at the front and the back of the results presents an interesting decision for the developer wishing to use move semantics in his/her code. However, it is possible to write discriminating code, conditional on the presence (/Ze) or absence (/Za) of the symbol `_MSC_EXTENSIONS` (`__STDC__` is used for C compilation only), that will use "pure" move semantics, a la Digital Mars/Watcom, when language extensions are on, and that will use the proxy technique when they are off.

Conclusion

Move semantics can be a very useful part of the functionality of classes, but clash against some important language rules and are, consequently, difficult to implement correctly. This article has presented two alternatives for implementing move semantics.

Moving By Proxy (Reference Transfer) is a relatively complex solution, which must be implemented on a per-class basis, but that works with all compilers, and it adheres to both the rules of the language and common idiom.

Const Casting, by contrast, is simpler, can be effected generally (by the use of templates and macros), but has invalid and misleading semantics. It's use is acceptable where suitable, but *only* where suitable; one should opt for the more verbose and time-consuming Moving By Proxy everywhere else.

In any case, the extra effort involved in Moving By Proxy is not really too onerous, because move semantics are not very common, and are, at least in my experience, entirely restricted to library/utility classes, which (should) get more detailed and considered design, testing and documentation.

One last caveat: you have to choose whether your type will be supporting copy semantics or move semantics, since you cannot have both. Whilst it is true that *some* compilers will let you define both (see Table 1), you'll have all kinds of trouble getting the compiler to select the one you want in all circumstances. All the compilers I tried this out on always selected the copy constructor. Short answer: don't do it!

Notes and References

[1] Bjarne Stroustrup, "The C++ Programming Language", Third Edition, Addison-Wesley, 1997

[2] This is so as long as the definition of the class's members allows it. If `m_val` was declared as `const int`, then the copy assignment-operator could not be automatically generated (though the copy-constructor could).

[3] An example is where one wishes to return allocated values from a function by wrapping them inside instances of "managing" classes, in order to ensure that the resources are not leaked even when the function return is ignored as a result of sloppy programming. The following is the canonical auto_ptr<> example:

```
// in Resource.h
class Resource
{
   virtual ~Resource() = 0
   {}
   virtual void Execute() = 0;
};
typedef std::auto_ptr<Resource>      Resource_ptr;

Resource_ptr CreateResource(int cArgs, char const **args);


// in ResourceFactory.cpp
Resource_ptr CreateResource(int cArgs, char const **args)
{
   // Perform some processing to "switch" on the input args, and create
   // an instance of one of a set of concrete sub-classes of Resource

  if( ... )
```

```
  {
    return Resource_ptr(new SimplexResource(args[0]));
  }
  else if( ... )
  {
    return Resource_ptr(new ComplexResource(args[0], args[1]));
  }
  else
    ...
}

// in main.cpp
int main(int argc, char **argv)
{
   Resource_ptr rptr = CreateResource(argc, reinterpret_cast<char const **>(argv));

   rptr->Execute();

   return 0;
}
```

[4] Digital Mars C/C++, Intel C/C++ and Metrowerks CodeWarrior are, in my personal opinion, the most impressive compilers currently available for the Windows platform. They do not come with anything like the exceedingly good Visual C++ IDE (though Intel can plug into it with near 100% compatibility), but what's the good of having the best IDE when one has a compiler that is so poor? I think it's worth noting our two "winners" – Digital Mars and Watcom – are free. Watcom is open-source (http://openwatcom.org/); though Digital Mars (http://www.digitalmars.com) is not, it is written by one man, Walter Bright, who has been known to fix bugs and add new requested features in turnaround times that'd make a large vendor's support engineer's head spin. It doesn't get any better than that! **[Note that this article was written prior to the release of Visual C++ 7.1, which represents a *considerable* improvement in the compiler, and it is now generally of a very high quality; see the article "Comparing C++ Compilers" in the October 2003 issue of Dr. Dobb's Journal. However, its improved language conformance has made it less competent at move semantics, so in respect of the issues examined here we remain to be wowed by Visual C++.]**

[5] It is conceivable that Digital Mars C/C++ doesn't actually support move semantics at all, but is simply so optimised that it appears that way. I intend to consult Walter Bright on the matter.

[6] The **/Ze** option is on by default in Visual C++. Alas, so much of the Win32 API is dependent on the Visual C++ extensions, that very few non-trivial programs can be compiled with the option off (/**Za**).

[7] I have consulted with Walter Bright [4], and he assures me that the optimisations are valid.

[8] Scott Meyers, "More Effective C++", Addison-Wesley, 1995. Item #16

[9] http://www.semantics.org/gotchas/gotcha36.pdf

[10] STLSoft is an open-source organisation whose focus is the development of robust, lightweight, cross-platform STL software, and is located at http://stlsoft.org.

[11] Scott Meyers, "Effective STL", Addison-Wesley, 2001. Item #8

[12] http://www.awprofessional.com/content/images/020163371X/autoptrupdate%5Cauto_ptr_update.html

[13] http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/1997/N1128.pdf