Move Semantics and Perfect Forwarding in C++11

Mikhail Semenov

5 May 2015 CPOL

Essential features in Visual C++ 11 and GCC 4.7.0: move, rvalue references, prvalues, xvalues, perfect forwarding.

Download move_semantics.zip

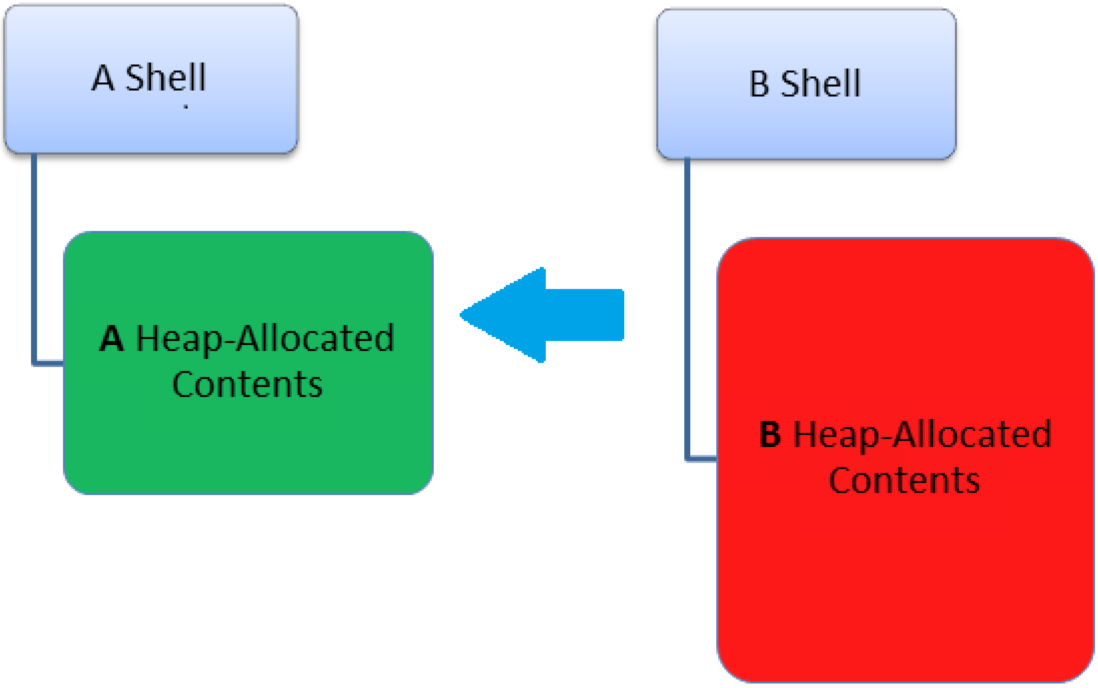
Download perfect_forwarding_extra.zip

Introduction

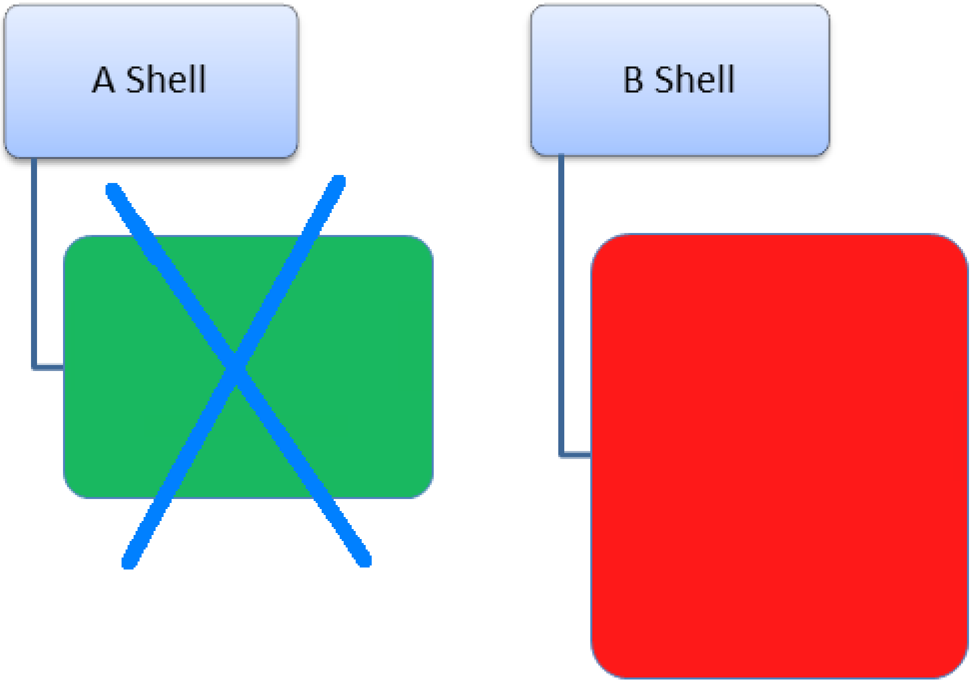
In this article, I will discuss the move functionality in the C++11, particular emphasis will be given to writing move constructors and move assignment operators, to issues with perfect forwarding. Some of the features are well presented in [1,2], and I recommend readers to view the presentation or read the notes. The document, which is close to C++11 Standard, can be found here [3].

Background

The move functionality has been designed to optimize data transferring between the objects, in situations when objects have large contents allocated on the heap. Such an object can be represented as comprising of two parts: a small shell (the class itself) and its heap-allocated contents (HAC), which is pointed to from the shell. Say, we want to copy the contents of object B to A. In most cases sizes of their contents will be different. The objects are shown here:



The usual operation will be to delete the contents of A:



Then, it will be necessary to allocate the new space, which has the same size as the contents of B:

And then it is required to copy the contents of B to A:

Normally this is how it should be done if we need to use both objects A and B in the future. But if, for some reason, the object B is not needed (such reasons we will be discussing shortly), it is possible to swap the contents of A and B:

There are two major advantages of the last approach:

(1) we do not have to allocate new memory from the heap, which reduces the time and memory space;

(2) we do not have to copy the contents of B to A, which saves the processor time on copying.

Obviously, the larger the contents the bigger the gain.

The RValue References

In order to identify the objects that are good for moving, a new feature is introduced in C++11, which is called the **rvalue reference**. A rvalue reference is usually used to specify a reference to an object that is going to be destroyed. In C++11, the symbol && (two ampersand characters) is put after the name of a type to identify that it is an rvalue reference. For instance, the rvalue reference to class T will be written as: T&&.

The main use of rvalue references is to specify formal parameters of functions. If, in the past, for parameters (of the class T), we used two main specifications:

(1) **T&**

(2) **const T&**

The first has been used to specify constants and values which are going to be changed, the second to specify constants and values, which are not going to be changed.

Now, in C++11, there are three options:

(1) **const T&**

(2) **T&**

(3) **T&&**

First of all we can still use only two options. But to make your code more efficient, it is good to take advantage of the third option. Obviously, if the third option is used, the area covering the first two is reduced.

The third option refers to rvalue refrences, which correspond to the actual parameters that are temporary and going to be destroyed. Let us see, what actual parameters correspond to rvalue references. These are functions and expressions, which return standard values of a particular type, not a reference to it. This resolves the issue of the difference between **const T&** and

T&:

- **const T&** refers to variables and constants;
- **T&&** refers to expressions (including function calls), but not those returning **T&**.

There is a possibility to state that the contents of a particular variable (say, **x**) is not needed and can be treated as an rvalue reference, in this case it can be wrapped by using `std::move(x)`. We will discuss it below.

Since two options are available for parameters , the constructors and assignments have two options as well: in addition to the traditional copy constructor and a `copy` assignment operators, there are now the `move` constructor **T(T&& t)** and the `move` assignment operator `T operator=(T&& t)`.

If, for a particular class, the move constructor and the move assignment operator are defined, the program can take advantage of the function values and instead of copying them, swap them with their contents, as I have shown in the picture.

One issue is particular important when writing a move constructor and a move assignment operator: the contents of the parameter, which is being moved, should remain valid for its further destruction. There are two approaches, which can be used here:

(1) swapping its contents with the target object (as in case of assignment, as I showed in pictures);

(2) setting it contents to a valid empty value, by assigning `nullptr` to the relevant pointers inside the shell class (often used for a move constructor).

In case of a move constructor, swap can also be considered, if the target is set to an empty value first, before the swap.

A Sample Class Definition with Move Functionality

Let us look at a simple example: a class, which defines an array of double values. The size is fixed, but it can be changed during assignment. You have probably seen many examples of it. But it's good to use it for an illustration: it's functionality is well known. The Array class definition without move semantics may look like this:

```
class Array
{
    int m_size;
    double *m_array;
public:
    Array():m_size(0),m_array(nullptr) {} // empty constructor

    Array(int n):m_size(n),m_array(new double[n]) {}

    Array(const Array& x):m_size(x.m_size),m_array(new double[m_size]) // copy constructor
    {
        std::copy(x.m_array, x.m_array+x.m_size, m_array);
    }

    virtual ~Array() // destructor
    {
        delete [] m_array;
    }

    auto Swap(Array& y) -> void
    {
        int n = m_size;
        double* v = m_array;
        m_size = y.m_size;
        m_array = y.m_array;
        y.m_size = n;
        y.m_array = v;
    }

    auto operator=(const Array& x) -> Array& // copy assignment
    {
        if (x.m_size == m_size)
        {
            std::copy(x.m_array, x.m_array+x.m_size, m_array);
        }
        else
        {
            Array y(x);
            Swap(y);
        }
        return *this;
    }

    auto operator[](int i) -> double&
    {
        return m_array[i];
    }

    auto operator[](int i) const -> double
    {
        return m_array[i];
    }

    auto size() const ->int { return m_size;}

    friend auto operator+(const Array& x, const Array& y) -> Array // adding two vectors
    {
        int n = x.m_size;
        Array z(n);
        for (int i = 0; i < n; ++i)
        {
            z.m_array[i] = x.m_array[i]+y.m_array[i];
        }
        return z;
    }
};
```

It is convenient to use swap in assignment, and very safe as well. Here is a small program, which uses this class:

```
int main()
{
    Array v(3);
    v[0] = 2.5;
    v[1] = 3.1;
    v[2] = 4.2;
    const Array v2(v);
    Array v3 = v+(v2+v);
    std::cout << "v3:";
    for (int i = 0; i < 3; ++i)
    {
        std::cout << " " << v3[i];
    };
    std::cout << std::endl;
    Array v4(3);
    v4[0] = 100.1;
    v4[1] = 1000.2;
    v4[2] = 10000.3;
    v4 = v4 + v3;
    std::cout << "v4:";
    for (int i = 0; i < 3; ++i)
    {
        std::cout << " " << v4[i];
    };
    std::cout << std::endl;
    return 0;
}
```

The program will print:

```
v3: 7.5 9.3 12.6
v4: 107.6 1009.5 10012.9
```

Let's add some move functionality. First of all, we shall define the move constructor. You may guess how we shall start. The parameter will be the rvalue reference to the class:

```
Array(Array&& x) ...
```

As I mentioned before, the contents of the parameter will be moved to the contents of the object:

```
Array(Array&& x):m_size(x.m_size),m_array(x.m_array)  ...
```

But it is important that the parameter gets the correct value in return: its contents cannot stay the same, it will be destroyed soon. So, in the body of the constructor, we assign empty values to the contents of the parameter. Here is the full definition of the move constructor:

```
Array(Array&& x):m_size(x.m_size),m_array(x.m_array)
{
    x.m_size = 0;
    x.m_array = nullptr;
}
```

Now, we shall write the move assignment operator. This is easy. The parameter is the rvalue reference and the body should just swap the contents of the current object with that of the parameter, as we discussed before:

```
auto operator=(Array&& x) -> Array&
{
    Swap(x);
    return *this;
}
```

That's all to it.

Now, there is an issue with the addition operator. It would be good to take advantage of the rvalue parameters as well. Three are two parameters, two options for either of them: there should be four definitions of the addition operator. We have written one, we have to write another three. Let's start with the following one:

```
friend auto operator+(Array&& x, const Array& y) -> Array
```

It is pretty straightforward. We just modify the contents of the parameter. We do not need it any more. But we will have to move it at the last moment. So, we write:

```
friend auto operator+(Array&& x, const Array& y) -> Array
{
    int n = x.m_size;
    for (int i = 0; i < n; ++i)
    {
        x.m_array[i] += y.m_array[i];
    }
```

The main thing is to return the right contents in the end. If we right `return x;` the contents of the parameter will be returned as it is. Although the type of the parameter is rvalue, the variable `x` is lvalue: we haven't done anything to move the value out of the `x`. If we just return the value of `x`, there will be no move operation. To do it correctly we have to use `return std:move(x);`

which will ensure that the contents of the parameter will be swapped with the target object. Here is the full definition of this operator:

```
friend auto operator+(Array&& x, const Array& y) -> Array
{
    int n = x.m_size;
    for (int i = 0; i < n; ++i)
    {
        x.m_array[i] += y.m_array[i];
    }
    return std::move(x);
}
```

Now, we have to write the other two operators. Here we take advantage of the commutativity of addition. We swap the parameters so that the previously defined operator will be called.

```
friend auto operator+(Array&& x, Array&& y) -> Array
{
    return std::move(y)+x;
}

friend auto operator+(const Array& x, Array&& y) -> Array
{
    return std::move(y)+x;
}
```

But, as you have noticed we have written quite a lot of code: four definitions of the addition operator. Is it possible to write less? In order to do this we must look at perfect forwarding.

Reducing the Number of Member Functions: Perfect Forwarding

Perfect forwarding was specially designed to reduc the amount of code the programmer has to write, although it has some other usage as well.

Let's consider the last two definitions of the addition operator. They have very much in common. the only difference is that the first parameter is defined differently: `Array&& x` and `const Array&& x`. These two definitions can be replaced by one using the following approach

(1) defining the template `template<class T>`;

(2) replacing the corresponding parameter definition with **T&& x**;

(3) in places where `std::move(x)` is used, put `std::forward(x)`.

In this particular code `std::move(x)` is not used, so we do not have to use `std::forward(x)`. Here is the result:

```
template<class T>
friend auto operator+(T&& x, Array&& y) -> Array
{
    return std::move(y)+x;
}
```

In order to combine the other two operator definitions into one we have to change the code slightly, so that the bodies of the two definitions look similar.

But first of all, let's consider some additional information about local parameters. If you look at the first addition operator, you will see that it has a local variable, called **z**. According to the rules of C++11, when this parameter is returned it will be moved to the new contents (not copied!). You do not need to use `std::move(z)`, to take advantage of move, in fact it will be moved as long as the relevant move constructor and move assignment operator are defined.

In order to make the two addition operators similar, we can create a local variable in both, where we can transfer (copy or move) the contents of the first parameter, and then add the value of the second parameter to this local variable. Here is the modified versions of the two operator definitions:

```
friend auto operator+(const Array& x, const Array& y) -> Array
{
    int n = x.m_size;
    Array z(x);
    for (int i = 0; i < n; ++i)
    {
        z.m_array[i] += y.m_array[i];
    }
    return z;
}


friend auto operator+(Array&& x, const Array& y) -> Array
{
    int n = x.m_size;
    Array z(std::move(x));
    for (int i = 0; i < n; ++i)
    {
        z.m_array[i] += y.m_array[i];
    }
    return z;
}
```

This is exactly what we need for perfect forwarding. If we use the three rules that we discussed before, we can rewrite these two operators as follows:

```
template<class T>
friend auto operator+(T&& x, const Array& y) -> Array
{
    int n = x.m_size;
    Array z(std::forward<T>(x));
    for (int i = 0; i < n; ++i)
    {
        z.m_array[i] += y.m_array[i];
    }
    return z;
}
```

You may consider that some of our modifications have slowed the code. It is possible that for the sake of efficiency, it would be better to leave the first two operator definitions without modification.

Putting This All Together: Let's Run Some Tests

In order to see what the advantage is of the move functionality, I have to put extra printing into the code in order to see what member functions are called. It will help to measure the efficiency.

```
//MOVE SEMANTICS REVISED

#include <iostream>
#include <string>
#include <algorithm>
#include <cmath>


#define MOVE_FUNCTIONALITY

int count_copies = 0;
int count_allocations = 0;
int elem_access = 0;

class Array
{
    int m_size;
    double *m_array;
public:
    Array():m_size(0),m_array(nullptr) {}
    Array(int n):m_size(n),m_array(new double[n])
    {
        count_allocations += n;
    }

    Array(const Array& x):m_size(x.m_size),m_array(new double[m_size])
    {
        count_allocations += m_size;
        count_copies += m_size;
        std::copy(x.m_array, x.m_array+x.m_size, m_array);
    }

#ifdef MOVE_FUNCTIONALITY
    Array(Array&& x):m_size(x.m_size),m_array(x.m_array)
    {
        x.m_size = 0;       // clearing the contents of x
        x.m_array = nullptr;
    }
#endif

    virtual ~Array()
    {
        delete [] m_array;
    }


    auto Swap(Array& y) -> void
    {
        int n = m_size;
        double* v = m_array;
        m_size = y.m_size;
        m_array = y.m_array;
        y.m_size = n;
        y.m_array = v;
    }

#ifdef MOVE_FUNCTIONALITY
    auto operator=(Array&& x) -> Array&
    {
        Swap(x);
        return *this;
    }
#endif

    auto operator=(const Array& x) -> Array&
    {
        if (x.m_size == m_size)
        {
            count_copies += m_size;
            std::copy(x.m_array, x.m_array+x.m_size, m_array);
        }
        else
        {
            Array y(x);
            Swap(y);
        }
        return *this;
    }


    auto operator[](int i) -> double&
    {
        elem_access++;
        return m_array[i];
    }
    auto operator[](int i) const -> double
    {
        elem_access++;
        return m_array[i];
    }

    auto size() const ->int { return m_size;}

#ifdef MOVE_FUNCTIONALITY

    template<class T>
    friend auto operator+(T&& x, const Array& y) -> Array
    {
        int n = x.m_size;
        Array z(std::forward<T>(x));
        for (int i = 0; i < n; ++i)
        {
            elem_access+=2;
            z.m_array[i] += y.m_array[i];
        }
        return z;
    }

    template<class T>
    friend auto operator+(T&& x, Array&& y) -> Array
    {
        return std::move(y)+x;
    }
#else
    friend auto operator+(const Array& x, const Array& y) -> Array
    {
        int n = x.m_size;
        Array z(n);
        for (int i = 0; i < n; ++i)
        {
            elem_access += 3;
            z.m_array[i] = x.m_array[i] + y.m_array[i];
        }
        return z;
    }
#endif

    void print(const std::string& title)
    {
        std::cout << title;
        for (int i = 0; i < m_size; ++i)
        {
            elem_access++;
            std::cout << " " << m_array[i];
        };
        std::cout << std::endl;
    }
};


int main()
{
    const int m = 100;
    const int n = 50;
    {
        Array v(m);
        for (int i = 0; i < m; ++i) v[i] = sin((double)i);
        const Array v2(v);
        Array v3 = v+(v2+v);
        v3.print("v3:");
        Array v4(m);
        for (int i = 0; i < m; ++i) v4[i] = cos((double)i);
        v4 = v4 + v3;
        v4.print("v4:");
        Array v5(n);
        for (int i = 0; i < n; ++i) v5[i] = cos((double)i);
        Array v6(n);
        for (int i = 0; i < n; ++i) v6[i] = tan((double)i);
        Array v7(n);
        for (int i = 0; i < n; ++i) v7[i] = exp(0.001*(double)i);
        Array v8(n);
        for (int i = 0; i < n; ++i) v8[i] = 1/(1+0.001*(double)i);
        v4 = (v5+v6)+(v7+v8);
        v4.print("v4 new:");
    }
    std::cout << "total allocations (elements):" << count_allocations << std::endl;
    int total_elem_access = count_copies*2 + elem_access;
    std::cout << "total elem access (elements):" << total_elem_access << std::endl;
    return 0;
}
```

After I had done some tests with the previous version of the code I realised that it is necessary to count the total element access instances, instead of only copy operations (one copy counts as two instances of element access: 1 source+1 target). Here is the table of the results (for both GCC 4.7.0 and Visual C++ 2011 in release mode):

Counter	Copy	Move
Element Allocations	1000	800
Element Access	2500	2350

Copy Elision

In certain cases, the C++11 Standard permits omission of copy or move construction, even if the constructor and/or destructor have side effects. This feature is called **copy elision**. Copy elision is allowed in the following circumstances:

- in a return statement in a function when the return value is a local (non-volatile) variable and its type is the same as the function return type (here, in type comparison, `const` and other qualifiers are ignored); this case I mentioned before;<o:p>
- in a throw-expression, when the operand is a local, non-volatile variable whose scope does not extend beyond the end of the innermost enclosing try-block.<o:p>
- when a temporary object (not bound to a reference) would be copied or moved to a class object with the same type (type qualifiers are ignored, as in the previous case);<o:p>
- in the exception-declaration (in the try-catch block), where the parameter in the catch has the same type as that object in the throw statement (type qualifiers are ignored, as in the previous case).<o:p>

In all these cases, the object may be directly constructed or moved to the target. In all these cases, not only copy, but even move can be elided. In all such cases, the rvalue reference is considered first (move), and only then the lvalue reference (copy) before the copy elision takes place.

Here is a sample program, which uses the array class that we discussed before:

```
const Array f()
{
    Array z(2);
    z[0] = 2.1;
    z[1] = 33.2;
    return z;
}

Array f1()
{
    Array z(2);
    z[0] = 2.1;
    z[1] = 33.2;
    return z;
}

void g(Array&& a)
```

```
{
    a.print("g(Array&&)");
}

void g(const Array& a)
{
    a.print("g(const Array&)");
}

void pf(Array a)
{
    a.print("pf(Array)");
}

int main()
{
    {
        Array p(f());
        p.print("p");
        g(f());
        g(f1());
        pf(f());
    }
    std::cout << "total allocations (elements):" << count_allocations << std::endl;
    int total_elem_access = count_copies*2 + elem_access;
    std::cout << "total elem access (elements):" << total_elem_access << std::endl;
    return 0;
}
```

In debug mode, using Visual C++ 11, when the move constructor is defined, the program prints the following:

```
p 2.1 33.2
g(const Array&) 2.1 33.2
g(Array&&) 2.1 33.2
pf(Array) 2.1 33.2
total allocations (elements):8
total elem access (elements):16
```

First of all, if you look at the `g` function calls, depending on whether `f()` or `f1()` is used as the parameter, the right overloaded function is selected. But this does not prevent the compiler from using the move operation (or to elide it altogether) in all these calls.

You may look at the instrumented version of the class, where extra printing is done, in order to trace which constructor is performed. In the debug mode (Visual C++ 11), without move functionality, the figures are:

total allocations (elements):16

total elem access (elements):32

In the release mode, the figures are always 8 and 16, and the compiler elides move or copy, neither of them is performed.

GCC 4.7.0 always elides move and copy, even in the debug mode.<o:p>

What does it mean from the programming point of view. First of all, if you do not write the overloaded function versions with rvalue parameters, you may still take advantage of the move operation: elision is like optimized move. The general approach would be: <o:p>

- to define both copy and move constructors;<o:p>
- to define functions or operators that can take advantage of move (like the operator+ for the Array class).

In all other cases, when the optimization is not visible, write your functions the usual, the old way, by using lvalue references (T&& or const T&&) for parameters.

Value Categories

Every expression in C++11 belongs exactly to one of the categories: `lvalue`, `xvalue` and `prvalue`. They all can be const and non-const, although some of the values do not have constness (like literals and enum enumerators). Let us look at all these categories.

- an lvalue designates a function or a non-temporary object, which can be referenced and is usually either named or pointed at (for example, variables, function formal parameters, user-defined constants (using a const declaration) and objects pointed at by pointers; a value of type T& (where T itself does not contain a reference) returned by a function is also an lvalue (for example, the function `Array& f1()` will return an l-value);<o:p>
- a prvalue designates either an literal (for example 27, 2.5, 'a', true), an enum enumerator,

a value of a function or an operator, which is not a reference (for example, a class object returned by a function; the function `Array f2()` returns a prvalue);

- an xvalue designates a value of some type T&& (where T itself does not contain a reference), which is the result of a user-defined function, `std::move`, or cast operator, converting to a T&& (for example, `static_cast<Array&&>(x)` and the function `Array&& f3()` returns an xvalue).

A `glvalue` is an lvalue or an xvalue. An `rvalue` is an xvalue or an prvalue.

Any function formal parameter, a variable or a user-defined constant (not a enumerator) is an lvalue.

As for prvalue constants, they can only be class objects, which are returned by functions or operators. The following function return a const prvalue:

```
const Array fc()
{
   Array p(2);
   p[0] = 3.2;
   p[1] = 2.2;
   return p;
}
```

But the problem with this function is that you cannot assign its result to a variable or a parameter defined as `Array&&`. It's better not to use `const` here.

Prvalues, which are literals and enum enumerators are all considered non-const and can be assigned to variables and parameters of rvalue reference type. Although in practice we rarely use rvalue references to simple types.

As for `xvalues`, `std::move(x)` or `static_cast<Array&&>(x)` are good examples. It's better not define functions that return xvalues at all, just ordinary class (value **T)** is sufficient.

Here is the table, which explains the rules setting values to variables, constants (initialization) and parameters, depending of the categories of these values (abbreviations mean: C – copying, M – moving, R – passing the reference, E –possible copy elision, t – type conversion is allowed, X – illegal):

Variable, Formal Parameter or Constant		**Value**			
Declaration	**Category**	**non-const prvalue**	**const lvalue**	**non-const lvalue**	**xvalue**
T or **const T**	lvalue or const lvalue	E/M/t	**C/t**	**C/t**	M/t
T&	lvalue	X	X	[R]	X
T&&	lvalue	[R]/ t	X	X	[R]/t
const T&	const lvalue	R/t	[R]/t	R/t	R/t

In square brackets, the preferable choice of the overloaded function is shown, in case the three overloads `f(T&)`,`f(T&&)` and `f(const T&)` are defined. Unfortunately, if `f(T)` and `f(T&)` are defined together then calling such functions with an lvalue actual parameter can lead to an ambiguity, which is not allowed.

Just to remind you that all these variables or parameters or constants belongs to the lvalue category, and as such cannot be passed as rvalues; `std::move` (or conversion) should be used if you want to do so.

When type conversion takes place (because the type of the value is different the type of the variable), then usually a new, temporary object is created, which is treated as a prvalue. We are mainly interested in values when T is a class object. The table shows that when prvalues and xvalues are passed to parameters or assigned to variables, the operation is very efficient. When lvalues are passed to parameters which are declared of type **T** or **const T** then copy takes place.

The Microsoft compiler is more flexible in passing prvalues to T& parameters: it is allowed by the compiler, which is an extension to the C++ standard. It is not permitted by the standard.

In terms of preferences, if **T** is a class, avoid using **T** or **const T** for parameters, unless you really need a copy of the actual parameters. Use **T, T&& or const T&**. In order to achieve efficiency in passing rvalues, it's a good thing to define move constructors and move assignment operators in your classes. Usually, as I mentioned before, **T&&** can be used to make some functions more efficient (like an extra code for operator+); otherwise it is efficient to use **const T&** to pass parameters by values.

In terms of the functions you write (but not move constructors or move assignment operators), it often does not matter, whether you write **T&&** or **const T&** for prvalues, because copy elision will make parameter passing very efficient anyway. But **const T&** allows you to pass lvalues by reference, which is very efficient.

In terms of function return values: use **T, T&** or **const T&.**

If you look at the first line of the table again, this is where the choice between elision, move and copy is important. The general rule for the assignment is that if the source is a temporary object (prvalue) and the target is an variable, a move assignment will take place. And for the initialization or parameter passing, if the source is a temporary object then a copy elision takes place. But there can be a case where the source is a "hidden" lvalue. Consider the following function:

```
Array g(Array& y)
{
    return std::move(y);
}
```

Here is a code fragment that uses this function:

```
Array x(1);
v[0] = 2.5;
Array z(g(x));
```

In this case, **x** is still a valid lvalue, not a temporary, which is referenced by the **y** formal parameter. As a result, when **z** is initialized move construction will take place.

Here are some conditions for an actual parameter when an overloaded function with an rvalue reference is chosen:

1. a call to a function or an operator, which does not return an lvalue reference (**T&**);
2. invoking a constructor, which creates a temporary object;
3. a conversion has to be applied to the const lvalue reference parameter, which creates a new, temporary object;`std::move(x)` or `static_cast<T&&>(x)` is used;
4. a literal is used (for example: 2.0, true, 'A'), in which case a temporary object is created to contain it.

Here is a program that illustrates these rules:

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

class A
{
    int i;
public:
    A():i(3){}
    A(int j):i(j) {}
    int getInt() const { return i;}
    void setInt(int j) { i = j;}
};

auto f(A&& a) -> void
{
    a.setInt(a.getInt()+1);
    std::cout << "f(A&&):" << a.getInt() << std::endl;
}
```

```
auto f(const A& a) -> void
{
    std::cout << "f(const A&):" << a.getInt() << std::endl;
}

auto g() -> A
{
    return A(27);
}


auto f(const std::string& s) -> void
{
    std::cout << "f(const string&):" << s << std::endl;
}

auto f(std::string&& s) -> void
{
    std::cout << "f(string&&):" << s << std::endl;
}

auto f(const int& i) -> void
{
    std::cout << "f(const int&):" << i << std::endl;
}

auto f(int&& i) -> void
{
    i++;
    std::cout << "f(int&&):" << i << std::endl;
}

int main()
{
    f(A()); // a temporary object
    f(g()); // a temporary value returned
    A x(21);
    f(std::move(x)); // an explicit std::move is used
    std::cout << "x: " << x.getInt() << std::endl;
    f("abc");        // creates a temporary string first
    f(7);
    int z = 100;
    f(std::move(z)); // an explicit std::move is used
    std::cout << "z: " << z << std::endl;
    return 0;
}
```

This program will print:

```
f(A&&):4
f(A&&):28
f(A&&):22
x: 22
f(string&&):abc
f(int&&):8
f(int&&):101
z: 101
```

Look at the side effects. The rvalue references may be used to change values of function parameters. In some sense they are similar to ordinary references.

Accessing Members of Temporary Objects. Ref-qualifiers for Member Functions

Let's us look at the following code:

```
struct Container
{
    Array m_v;
public:

    Container():m_v(2)
    {
        m_v[0] = 177.1;
        m_v[1] = 12.7;
    }

    Container(const Array& x):m_v(x) {}
    Array moveArray() { return std::move(m_v);}
    Array getArray() const { return m_v;}
};

int main()
{
    {
        Array zz(Container().moveArray()); // uses move
        Array k(Container().getArray());   // uses copy
        zz.print("zz");
    }
    return 0;
}
```

Here the `Container` class is used for a temporary object. We have created two access functions for the Array object: `getArray` and `moveArray`. The first one is more efficient: it uses the move operation. But we explicitly gave them different names.

It would be nice to be able to define overloaded access functions, which perform differently depending on whether the object is temporary or not. In C++11 Standard it is possible and can be done using ref-qualifiers (**&** or **&&**):<o:p>

```
Array getArray() && { return std::move(m_v);}
Array getArray() const& { return m_v;}
```

The rule is that if you use a ref-qualifier for one of the overloaded functions you have to use ref-qualifiers for all the others. Unfortunately, this feature is not available in Visual C++ 11 or in GCC 4.7.0.

Other Uses of Perfect Forwarding

In addition to the cases discussed before, there are following cases, where it is convenient to use perfect forwarding:

<o:p>

- to define a function template which will create an object of a class and possibly provide extra manipulations with it;
- to create a wrapper for a function call.

The first case is often used in by `emplace(...)` methods in the new C++11 STL, which both create and store objects in containers.

For example, for `v` of class vector<A> , instead of using `v.push_back(A(p1, p2, …,pn))`, we can use `v.emplace_back(p1,p2, …, pn)`. In order to provide this feature for an arbitrary number of parameters in the constructor, the variadic templates are needed. They are not available in Visual C++ 11, but they exist in GNU C++ 4.7.0.

Another example, here is the definition of `make_unique`, which is a function template that returns a unique pointer to the given class:

```
template<typename T, typename... U>
auto make_unique(U&&... p) -> std::unique_ptr<T>
{
    return std::unique_ptr<T>(new T(std::forward<U>(p)...));
}
```

In Visual C++ 11, you may define a similar template for a class with a single-parameter constructor.

Creating a wrapper can be used, for instance, in case there is a function **t**, with one parameter, but several overloading definitions. You may create a wrapper as follows:

```
template<class U>
void f(U&& x)
{
    std::cout << "Calling function t... " << x << std::endl;
    t(std::forward<U>(x));
}
```

There is a similar case of a wrapper, where the function is provided as a parameter. This case is not the exact perfect forwarding: the parameter type is derived not from the value `x`, but from the function. In this case you may use either `static_cast<T>(x)` or `std:forward<T>(x)`:

```
template<class R, class T, class U>
auto call(auto (*g)(T) -> R, U&& x) -> R
{
    std::cout << "call(g, x)" << x << std::endl;
    return g(static_cast<T>(x)); //return g(std::forward<T>(x));
```

We still have to use U&&: otherwise some of the reference parameters will not work correctly.

Here is the program, which illustrates all the those features:

```
#include <iostream>
#include <memory>
#include <utility>

struct A {
    A(const int& n) { std::cout << "A(const int&), n=" << n << "\n"; }
    A(int&& n) { std::cout << "A(int&&), n=" << n << "\n"; }
    A(int& n)  { n++; std::cout << "A(int&), n=" << n << "\n"; }
};

template<class T, class U>
auto make_unique(U&& u) -> std::unique_ptr<T>
{
    return std::unique_ptr<T>(new T(std::forward<U>(u)));
}

auto t(int& x) -> void
{
    x++;
    std::cout << "t(int& x): " << x << std::endl;
}

auto t(int&& x) -> void
{
    std::cout << "t(int&& x): " << x << std::endl;
}

auto t(const int& x) -> void
{
    std::cout << "t(const int& x): " << x << std::endl;
}

template<class U>
auto f(U&& x) -> void
{
    std::cout << "Calling function t... " << x << std::endl;
    t(std::forward<U>(x));
}

auto ten_times(int i) -> int
{
    return 10*i;
}

auto g1(const int& n) -> void { std::cout << "g1(const int&), n=" << n << "\n"; }
auto g2(int&& n) -> void { std::cout << "g2(int&&), n=" << n << "\n"; }
auto g3(int& n) -> int&  { n++; std::cout << "g3(int&), n=" << n << "\n"; return n; }
auto g4(int n)  -> int   { std::cout << "g4(int), n=" << n << "\n"; return 4*n; }

template<class R, class T, class U>
auto call(auto (*g)(T) -> R, U&& x) -> R
{
    std::cout << "call(g, x)" << x << std::endl;
    return g(static_cast<T>(x)); //return g(std::forward<T>(x));
}

int main()
{
    const int j = 5;
    std::unique_ptr<A> pj = make_unique<A>(j);
    std::unique_ptr<A> p1 = make_unique<A>(2);
    int i = 10;
    std::unique_ptr<A> p2 = make_unique<A>(i);
    std::cout << "i: " << i << std::endl;
    f(i);
    std::cout << "after call f(i). i: " << i << std::endl;
    f(j);
    f(ten_times(8));
    std::cout << "i: " << i << std::endl;
    A a(i);
    call(g1,j);
    call(g2,2);
    int& k = call(g3, i);
    k *= 100;
    std::cout << "i: " << i << " k: " << k << std::endl;
    int k2 = call(g4,7);
    std::cout << "k2: " << k2 << std::endl;
    return 0;
}
```

The program will print:

```
A(const int&), n=5
A(int&&), n=2
A(int&), n=11
i: 11
Calling function t... 11
t(int& x): 12
after call f(i). i: 12
Calling function t... 5
t(const int& x): 5
Calling function t... 80
t(int&& x): 80
i: 12
A(int&), n=13
call(g, x)5
g1(const int&), n=5
call(g, x)2
g2(int&&), n=2
call(g, x)13
g3(int&), n=14
i: 1400 k: 1400
call(g, x)7
g4(int), n=7
k2: 28
```

The U&& is important: in this program if you change **U&& x** for **U x**, you won't obtain the right value 1400 for i or k.

Classes, which can only be moved, never copied

There are classes, which can only be moved (mentioned in [4]), for example: `fstream` and `unique_ptr`. Here is a program with `ifstream`, which demonstrates this feature:

```
#include <iostream>
#include <string>
#include <fstream>

std::ifstream OpenMyFile(const std::string& filename)
{
    std::ifstream file(filename);
    if (!file.good())
    {
        file.close();
        return file;
    }
    std::string s;
    std::getline(file, s);
    if (s.substr(0,7) != "My File")
    {
        file.close();
        file.setstate(std::ios_base::failbit);
        return file;
    }
    return file;
}

int main(int count, char *args[])
{
    if (count < 2)
    {
        std::cout << "Incorrect number of parameters" << std::endl;
        return -1;
    }
    std::ifstream myFile = OpenMyFile(args[1]);
    if (myFile.good())
    {
        while (!myFile.eof())
        {
            std::string s;
            std::getline(myFile,s);
            std::cout << s << std::endl;
        }
    }
    else
    {
        std::cout << "*** FILE ERROR ***" << std::endl;
    }
    return 0;
}
```

If we use, as this program's parameter, the file *myfile.txt* with the following contents:

```
My File
Some types are not amenable to copy semantics but can still be made movable. For example:
(1)    fstream
(2)    unique_ptr (non-shared, non-copyable ownership)
(3)    A type representing a thread of execution
```

the program will print the lines of the file (except for the first one). But if the first line is not "My File", the program will report an error.

The main feature of this program is the function OpenMyFile, which returns the value of the local variable `file`. As you may remember, the return value is such cases can be moved. This value will be moved to the variable `myFile` in the main program.

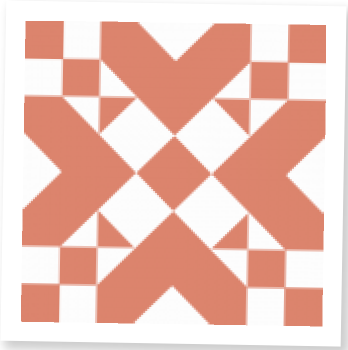
Unfortunately, the last program works only in Visual C++ 11. In GCC 4.7.0, the move semantics is not implemented for fstreams.

References

1. Scott Meyers. "Move Semantics, Rvalue References, and Perfect Forwarding". Notes in PDF. http://www.aristeia.com/TalkNotes/ACCU2011_MoveSemantics.pdf
2. Scott Meyers. "Move Semantics, Rvalue References, and Perfect Forwarding". Presentation. http://skillsmatter.com/podcast/home/move-semanticsperfect-forwarding-and-rvalue-references
3. C++ Working Draft, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3376.pdf
4. Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki. "A Brief Introduction to Rvalue References". http://www.artima.com/cppsource/rvalue.html

License

About the Author



Mikhail Semenov
Software Developer (Senior)
United Kingdom

No Biography provided

Comments and Discussions

20 messages have been posted for this article Visit **https://www.codeproject.com/Articles/397492/Move-Semantics-and-Perfect-Forwarding-in-Cplusplus** to post and view comments on this article, or click **here** to get a print view with messages.