Understanding Move Semantics and Perfect Forwarding: Part 3

Drew Campbell Follow
Sep 15, 2018 · 9 min read

Perfect Forwarding

Perfect Forwarding and Everything Else

In the previous article I introduced rvalue references and discussed how they can be used to improve performance of C++ applications using move semantics. In this final

article we will look at how rvalue references improve the flexibility of C++ code when writing template programs using perfect forwarding.

What is Perfect Forwarding

Perfect forwarding allows a template function that accepts a set of arguments to forward these arguments to another function whilst retaining the lvalue or rvalue nature of the original function arguments.

Perfect forwarding reduces excessive copying and simplifies code by reducing the need to write overloads to handle lvalues and rvalues separately.

Note: The function the arguments are forwarded to can be a normal function, another template function, or a constructor.

```
template<typename T>
void OuterFunction(T& param)
{
    InnerFunction(param);
}
```

Without having written template code in C++ before this might not make a whole lot of sense so above is an example of a template function that forwards its argument param to another inner function.

Notice that the outer function accepts an lvalue reference, therefore we can only pass into it lvalues, the following code will not compile.

```
OuterFunction(5); // Wont work trying to pass an rvalue to lvalue
reference
```

To fix this we could make the outer function accept a const lvalue reference to allow us to pass rvalues but then the inner function would not be allowed to modify its argument. Instead we would have to write an overload of the outer function that handles rvalues.

```
template<typename T>
void OuterFunction(T&& param)
{
   InnerFunction(param);
}
```

More importantly what if our inner function must accept an rvalue reference as an argument. From the previous article we know that an rvalue reference is in fact treated as an lvalue so even if our outer function accepts an rvalue reference when it comes to passing that argument to the inner function it will be seen by the compiler to be an lvalue and is therefore not allowed.

The problems mentioned above plus many other issues that can crop up when writing template code is what perfect forwarding solves. However, to truly understand perfect forwarding several other concepts must first be understood.

Template Type Deduction

```
void OuterFunction(T&& t)
{
}
…
OuterFunction(x);
OuterFunction(X());
```

Template type deduction relates to how the compiler deduces the type T passed into the template function when the functions parameter is of type T&&. Two rules govern how T is resolved that depends on whether the argument passed is an lvalue or rvalue. The rules are as followed:

1. If the argument passed to T is a lvalue, then T is deduced to a lvalue reference, X&
2. If the argument passed to T is a rvalue, then T is deduced to a lvalue X.

Basically, we can call a template function without having to specify the type T being passed and have the compiler determine the type itself with the rules above governing

what the type T value becomes in respect to its expression value. The type of T is deduced from the argument passed into the function call.

```
OuterFunction(6);
```

If the above call was made the compiler would determine the type T as an int and given that an rvalue is being passed the expression value of T would be an lvalue, T = int.

```
OuterFunction(helloWorldString);
```

If the above call was made the compiler would determine the type T as a string and given that an lvalue is being passed the expression value of T would be an lvalue reference. T = string&.

Note: type deduction occurs when the type is not known and therefore must be deduced, so in the case of OuterFunction(“Hello”); would not compile as the function excepts an lvalue reference of type string which an rvalue “Hello” cannot be bound to.

Reference Collapsing

Reference collapsing is a set of rules in C++ 11 to determine the value of T of a template function argument when trying to take the reference of a reference which is something that is illegal in C++. Taking the address of an address doesn’t make any sense but it can sometimes occur when writing templates.

```
template<typename T>
void func(T t)
{
   T& k = t;
}
...
string hello = “Hello”;
func(hello);
```

In the above example the type T is deduced to int& and then we are trying to store a reference to t in the form of k which means we are trying to do (int&)& k = t. Trying to take a reference to our reference.

Therefore, reference collapsing rules states what the value of T is dependent on the type of referencing that is occurring:

- Taking the reference of an lvalue reference results in an lvalue reference X& & becomes X&
- Taking the rvalue reference of an lvalue reference is an lvalue reference X& && becomes X&
- Taking the lvalue reference of an rvalue reference is an lvalue reference X&& & becomes X&
- Taking the rvalue reference of an rvalue reference is an rvalue reference X&& && becomes X&&

In any situation where an lvalue reference is involved the compiler will always collapse the type to an lvalue reference, if an rvalue references is involved then the type deduced is an rvalue reference.

Forwarding with forward

The function std::forward is required for solving the perfect forwarding problem with the functions purpose to resolve that awkward rule in which rvalue references are treated as lvalues (see the last article for more information).

The problem this causes is that even when passing an rvalue to a function that accepts an rvalue reference given that the rvalue reference parameter is treated as an lvalue reference we cannot then forward this to a function that accepts an rvalue. We have lost all the benefits move semantics gives us.

This is where std::forward comes in as it does two things dependent upon the value that is passed to the function:

1. If passed an argument that isn't an lvalue reference e.g. int& val, then it will return an rvalue reference.

2. If an argument passed in an lvalue reference the function returns an lvalue reference, it does nothing to the argument.

This means that passing in an rvalue to a template function that accepts an rvalue reference will be able to forward that argument as an rvalue to any inner functions and if an lvalue is passed and we have an overload for our inner function that accepts lvalues then that function is called instead.

Universal References

The differences between a universal reference and an rvalue reference is that although it is denoted using '&&' it doesn't have to be an rvalue reference, it can sometimes mean '&', be an lvalue reference.

As will be shown later this is how we can pass lvalues and rvalues to a template with a parameter of type T&& without the compiler complaining. The previous article referred to how a function with an rvalue reference can only accept rvalue references and that is true because in the context of a normal function '&&' means rvalue reference. With template functions T&& can mean T&& or it can mean T&.

In general, '&&' means a universal reference only when template type deduction is involved which is not the case if we are using a non-template function. With the advent of a new reference type there is unfortunately more rules that need to be learned:

- If the expression initializing a universal reference is an lvalue, the universal reference becomes an lvalue reference.
- If the expression initializing the universal reference is an rvalue, the universal reference becomes an rvalue reference.

Bringing it all Together, Perfect Forwarding

Bringing all the above rules together brings us what is referred to as perfect forwarding. A set of rules and functions within C++ that provide us with the ability to pass a value regardless of whether it is an lvalue or an rvalue and have that value preserved when being passed to a function called within a template function.

Below is a simple example of perfect forwarding where an object of type A has its constructor called within a template function, with the move constructor, or copy

constructor being called dependent on the value passed to template function.

```
class A
{
public:
A(std::string b) : b(b) {}

// Copy Constructor
A(const A& other) : b(b)
{
b = other.b;
std::cout << "Copy Constructor" << std::endl;
}

// Move Constructor
A(A&& other)
{
b = std::move(other.b); std::cout << "Move Constructor" << std::endl;
}
private:
std::string b;
};
//And a template function
…
template<typename T>
void OuterFunction(T&& param)
{
A a(std::forward(param));
}
…
// Passing an lvalue
A a = A("Hello");
OuterFunction(a);

// Passing an rvalue
OuterFunction(A("World"));
```

After researching around on the internet I couldn't find an answer that definitely explained how std::forward works but this post explains std::forward in the context of perfect forwarding. Basically, std::forward returns a static_cast(t) when T is explicitly defined and t is our passed parameter.

```
template<typename T>
void func(T&& t)
{
```

```
        std::forward(t);
    }
```

If we let the template parameter be defined as a universal reference T&& then std::forward is either going to be passed an lvalue reference or an rvalue reference. Given std::forward is basically a static_cast(param) where T is explicitly defined not deduced then reference collapsing rules determine the returning value where static cast looks either like static_cast<A& &&>(param) or static_cast<A&& &&>(param).

Passing an Lvalue

Looking at the above example we can look at how everything we discussed comes into practice. Passing the variable 'a' to OuterFunction which is an lvalue means that the type T is deduced to A&.

Next, given that the param now looks like A& & as universal reference rule states that if an lvalue is passed T&& is treated as an lvalue reference. For param we can look at the reference collapsing rules to determine that param A&, is an lvalue reference.

Finally, param is passed to std::forward and the static cast of std::forward returns an lvalue reference due to the reference collapsing rules.

In this instance an lvalue reference is passed to 'a' and thus the copy constructor is invoked.

Passing an Rvalue

Passing the object A("World") an rvalue expression to OuterFunction results in T being deduced to A.

Next, given that param now looks like A && where the universal reference is treated as a rvalue reference, so we just get A&&.

Finally, param is passed to std::forward and the static cast of std::forward returns an rvalue reference due to the reference collapsing rules.

In this instance an rvalue reference is passed to ‘a’ and thus the move constructor is invoked and we can take advantage of move semantics avoiding an unnecessary copy of member variable ‘b’.

Step by Step Guide

To summarise, for perfect forwarding to work it requires that a template function accept a parameter of type T&& and that template type deduction is used. As well as this std::forward must have its type explicitly defined.

1. The argument passed to a template function first has its type deduced, which results in an lvalue or an lvalue reference. T becomes T& or T
2. Because a universal reference is being used if the expression passed to the function (the argument) is an lvalue then T&& param becomes T& param, if its an rvalue it becomes T&& param referring to an rvalue reference.
3. At this point we have either (T& && param) or T && param and reference collapsing rules are applied so the parameter to the function results in either an lvalue reference or an rvalue reference.
4. Finally, given that an rvalue reference is in fact treated like an lvalue reference std::forward is required to cast the template parameter to the correct value. Again, reference collapsing rules are required that determines if we have an lvalue reference then an lvalue reference should be forwarded and if we have an rvalue reference our value should be cast to an rvalue reference.

Summary

Admittedly when I first started writing this article I didn’t quite realise how complicated perfect forwarding was and how many subtle behaviours of C++ was required to fully comprehend what was going on. After all I still don’t fully understand std::forward and I can’t guarantee that what I have wrote is 100% correct but I think it makes sense. Either way I hope this article and the previous two have provided enough information to emphasise the need and practical uses of rvalue references, move semantics and perfect forwarding within C++ 11.

Like always if you have any questions then feel free to message me @gamedevunboxed :)

Further Resources

- https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
- https://eli.thegreenplace.net/2014/perfect-forwarding-and-universal-references-in-c
- https://www.justsoftwaresolutions.co.uk/cplusplus/rvalue_references_and_perfect_forwarding.html

Programming Software Development Coding Code Software Engineering