Perfect Forwarding

25 December 2016

Contents [Show]

Today, we solve " ... a herefore unsolved problem in C++" (Bjarne Stroustrup). To make the long story short, I will write about perfect forwarding.

But, what is **perfect forwarding?**

> If a function templates forward its arguments without changing its lvalue or rvalue characteristics, we call it perfect forwarding.

Great. But what are lvalues and rvalues? Now, I have to make a little detour.

Lvalues and rvalues

I will not talk about the details about lvalues and rvalues and introduce therefore *glvalues, xvalues* ,and *prvalues.* That's not necessary. In case, you are curious, read the post from Anthony Williams: Core C++ - lvalues and rvalues. (https://www.justsoftwaresolutions.co.uk/cplusplus/core-c++-lvalues-and-rvalues.html)I will provide in my post a sustainable intuition.

Rvalues are

- temporary objects.
- objects without names.
- objects which have no address.

If one of the characteristics holds for an object, it will be an rvalue. In reverse, that means that lvalues have a name and an address. A few examples for rvalues:

```
int five= 5;
std::string a= std::string("Rvalue");
std::string b= std::string("R") +  std::string("value");
std::string c= a + b;
std::string d= std::move(b);
```

Rvalues are on the right side of an assignment. The value 5 and the constructor call are `std::string("Rvalue")` rvalues because you can neither determine the address of the value 5 nor has the created string object a name. The same holds for the addition of the rvalues in the expression `std::string("R") + std::string("value")` .

The addition of the two strings `a + b` is interesting. Both strings are lvalues, but the addition creates a temporary object. A special use case is `std::move(b)` . The new C++11 function converts the lvalue b into an rvalue reference.

Rvalues are on the right side of an assignment; lvalues can be on the left side of an assignment. But that is not always true:

```
const int five= 5;
five= 6;
```

Although, variable five is an lvalue. But five is constant and you can not use it on the left side of an assignment.

But now to the challenge of this post: Perfect forwarding. To get an intuition for the unsolved problem, I will create a few *perfect* factory methods.

A perfect factory method

At first, a short disclaimer. The expression a perfect factory method is no formal term.

A **perfect factory method** is for me a totally generic factory method. In particular, that means that the function should have the following characteristics:

- Can take an arbitrary number of arguments
- Can accept lvalues and rvalues as an argument
- Forwards it arguments identical to the underlying constructor

I want to say it less formal. A perfect factory method should be able to create each arbitrary object.

Let's start with the first iteration.

First iteration

For efficiency reasons, the function template should take its arguments by reference. To say it exactly. As a non-constant lvalue reference. Here is the function template `create` in my first iteration.

```
1  // perfectForwarding1.cpp
2
3  #include <iostream>
4
5  template <typename T,typename Arg>
6  T create(Arg& a){
7    return T(a);
8  }
9
10
11 int main(){
12
13   std::cout << std::endl;
14
15   // Lvalues
16   int five=5;
17   int myFive= create<int>(five);
18   std::cout << "myFive: "  << myFive << std::endl;
19
20   // Rvalues
21   int myFive2= create<int>(5);
22   std::cout << "myFive2: " << myFive2 << std::endl;
23
24   std::cout << std::endl;
25
26 }
```

If I compile the program, I will get a compiler error. The reason is that the rvalue (line 21) can not be bound to a non-constant lvalue reference.

Now, I have two ways to solve the issue.

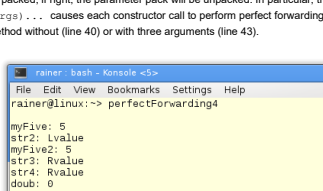
1. Change the **non-constant lvalue reference** (line 6) in a **constant lvalue reference.** You can bind an rvalue to a constant lvalue reference. But that is not perfect, because the function argument is constant and I can therefore not change it.
2. Overload the function template for a **constant lvalue reference** and a **non-const lvalue reference.** That is easy. That is the right way to go.

Second iteration

Here is the factory method `create` overloaded for a constant lvalue reference and a non-constant lvalue reference.

```
1  // perfectForwarding2.cpp
2
3  #include <iostream>
4
5  template <typename T,typename Arg>
6  T create(Arg& a){
7    return T(a);
8  }
9
10 template <typename T,typename Arg>
11 T create(const Arg& a){
12   return T(a);
13 }
14
15 int main(){
16
17   std::cout << std::endl;
18
19   // Lvalues
20   int five=5;
21   int myFive= create<int>(five);
22   std::cout << "myFive: "  << myFive << std::endl;
23
24   // Rvalues
25   int myFive2= create<int>(5);
26   std::cout << "myFive2: " << myFive2 << std::endl;
27
28   std::cout << std::endl;
29
30 }
```

The program produces the expected result.



That was easy. Too easy. The solution has two conceptional issues.

1. To support n different arguments, I have to overload 2^n +1 variations of the function template `create`. 2^n +1 because the function `create` without an argument is part of the perfect factory method.
2. The function argument mutates in the function body of creating to an lvalue, because it has a name. Does this matter? Of course, yes. a is not movable anymore. Therefore, I have to perform an expensive copy instead of a cheap move. But what is even worse. If the constructor of T (line 12) needs an rvalue, it will not work anymore.

Now, I have the solution in the shape of the C++ function `std::forward`.

Third iteration

With `std::forward`, the solution looks promising.

```
1  // perfectForwarding3.cpp
2
3  #include <iostream>
4
5  template <typename T,typename Arg>
6  T create(Arg&& a){
7    return T(std::forward<Arg>(a));
8  }
9
10 int main(){
11
12   std::cout << std::endl;
13
14   // Lvalues
15   int five=5;
16   int myFive= create<int>(five);
17   std::cout << "myFive: "  << myFive << std::endl;
18
19   // Rvalues
20   int myFive2= create<int>(5);
21   std::cout << "myFive2: " << myFive2 << std::endl;
22
23   std::cout << std::endl;
24
25 }
```

Before I present the recipe from cppreference.com (http://en.cppreference.com/w/cpp/utility/forward)to get perfect forwarding, I will introduce the name universal reference.

The name **universal reference** is coined by Scott Meyers.

> The universal reference (`Arg&& a`) in line 7 is a powerful reference that can bind lvalues or rvalues. You have it at your disposal if you declare a variable `Arg&& a` for a derived type A.

To achieve perfect forwarding you have to combine a universal reference with `std::forward`. `std::forward<Arg>(a)` returns the underlying type because a is a universal reference. Therefore, an rvalue remains an rvalue.

Now to the pattern

```
template<class T>
void wrapper(T&& a){
    func(std::forward<T>(a));
}
```

I used the color red to emphasize the key parts of the pattern. I used exactly this pattern in the function template `create`. Only the name of the type changed from T to `Arg`.

Is the function template `create` perfect? Sorry to say, but now. `create` needs exactly one argument which is perfectly forwarded to the constructor of the object (line 7). The last step is now to make a variadic template out of the function template.

Fourth iteration - the perfect factory method

Variadic Templates (http://en.cppreference.com/w/cpp/language/parameter_pack) are templates that can get an arbitrary number of arguments. That is exactly the missing feature of the perfect factory method.

```
1  // perfectForwarding4.cpp
2
3  #include <iostream>
4  #include <string>
5  #include <utility>
6
7  template <typename T, typename ... Args>
8  T create(Args&& ... args){
9    return T(std::forward<Args>(args)...);
10 }
11
12 struct MyStruct{
13   MyStruct(int i,double d,std::string s){}
14 };
15
16 int main(){
17
18     std::cout << std::endl;
19
20   // Lvalues
21   int five=5;
22   int myFive= create<int>(five);
23   std::cout << "myFive: "  << myFive << std::endl;
24
25   std::string str{"Lvalue"};
26   std::string str2= create<std::string>(str);
27   std::cout << "str2: " << str2 << std::endl;
28
29   // Rvalues
30   int myFive2= create<int>(5);
31   std::cout << "myFive2: " << myFive2 << std::endl;
32
33   std::string str3= create<std::string>(std::string("Rvalue"));
34   std::cout << "str3: " << str3 << std::endl;
35
36   std::string str4= create<std::string>(std::move(str3));
37   std::cout << "str4: " << str4 << std::endl;
38
39   // Arbitrary number of arguments
40   double doub= create<double>();
41   std::cout << "doub: " << doub << std::endl;
42
43   MyStruct myStr= create<MyStruct>(2011,3.14,str4);
44
45
46   std::cout << std::endl;
47
48 }
```

The three dots in line 7 -9 are the so-called parameter pack. If the three dots (also called ellipse) are left of `Args`, the parameter pack will be packed; if right, the parameter pack will be unpacked. In particular, the three dots in line 9 `std std::forward<Args>(args)...` causes each constructor call to perform perfect forwarding. The result is impressive. Now, I can invoke the perfect factory method without (line 40) or with three arguments (line 43).

What's next?

RAII, short for Resource Acquisition Is Initialization is a very important idiom in C++. Why? Read in the next post. (https://www.modernescpp.com/index.php/garbage-collectio-no-thanks)

My Mentoring Program "Fundamentals for C++ Professionals" is Open for Registration

My new mentoring program "Fundamentals for C++ Professionals" starts on April 22nd. Registration is open until April 17nd. Here is more information:

Launch page: https://www.modernescpp.org/ (https://www.modernescpp.org/)

Introduction: https://www.modernescpp.org/my-mentoring-program-fundamentals-for-c-professionals/ (https://www.modernescpp.org/my-mentoring-program-fundamentals-for-c-professionals/)

Still open questions? Please, call me: info@modernescpp.de (mailto:info@modernescpp.de)

Modernes C++,

Rainer Grimm