Perfect Forwarding

This lesson explains the principle of perfect forwarding.

WE'LL COVER THE FOLLOWING

A classic problem: A function wants to get its arguments by reference.

```
struct BigData{
  BigData(vector<int>& d): data(d){} // Lvalue-Ref
  BigData(const vector<int>& d): data(d){} ... // const Lvalue-Ref
};

struct BigData2{
  template<typename T>
  BigData2(T&& d): data(std::forward<T>(d)){} // Lvalue- and Rvalue-Ref
};
```

For **n** parameters, 2^n function overloads are necessary.

Definition

> If a function template forwards its arguments without changing their lvalue or rvalue characteristics, we call it **perfect forwarding**.

A perfect factory method

First, a short disclaimer; the expression, *perfect factory method*, is not a formal term.

A perfect factory method is a totally generic factory method. In particular, it implies that the function should have the following characteristics:

- takes an arbitrary number of arguments.
- accepts lvalues and rvalues as arguments.
- forwards its arguments in the same way as the underlying constructor.

Let's describe this in a more formal way. A perfect factory method should be able to create each arbitrary object.

Perfect forwarding enables us to write a function that can identically forward its arguments. The lvalue and rvalue properties are respected.

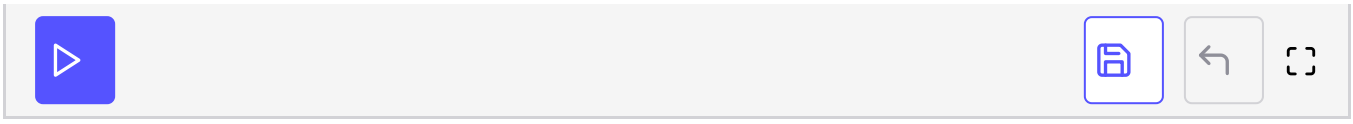
Let's start with the first iteration and move towards implementing the perfect factory method.

First iteration

For efficiency reasons, the function template should take its arguments by reference. Specifically, it should take an argument as a non-constant lvalue reference. Here is the function template `create` in the first iteration.

```
#include <iostream>

template <typename T,typename Arg>
T create(Arg& a){
  return T(a);
}


int main(){

  std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;
```

```
  std::cout << std::endl;

}
```



If we compile the program, we will get a compiler error. The reason is that the rvalue in line 19 cannot be bound to a non-constant lvalue reference.

We have two ways to solve the issue.

1. Change the **non-constant lvalue reference** in line 4 to a **constant lvalue reference**. We can bind an rvalue to a constant lvalue reference but that is not ideal because the function argument is constant and therefore, cannot be changed.

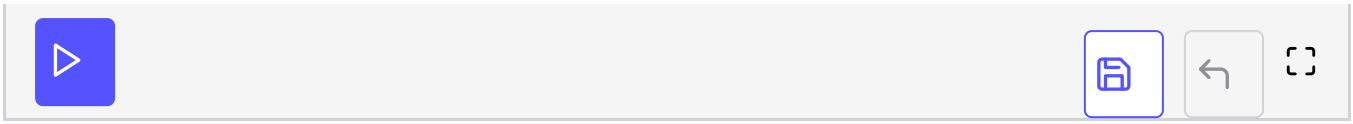
2. Overload the function template for a **constant lvalue reference** and a **non-const lvalue reference**. That is an easier and better approach.

Second iteration

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

```
#include <iostream>

template <typename T,typename Arg>
T create(Arg& a){
  return T(a);
}

template <typename T,typename Arg>
T create(const Arg& a){
  return T(a);
}

int main(){

  std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;

  std::cout << std::endl;
```

```
}
```



That was easy, but the solution has two conceptual issues.

1. To support n different arguments, we have to overload $2^n + 1$ variations of the `create` function template. This is because, without an argument, the `create` function is part of the perfect factory method.

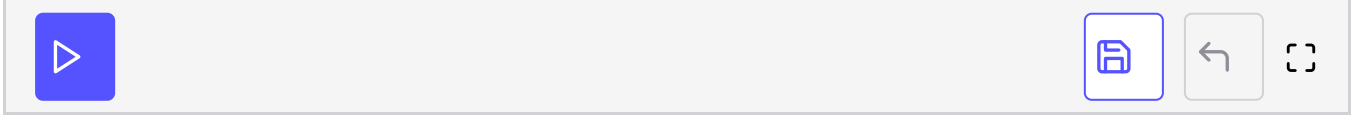
2. The function argument materializes in the function body of `create` to an lvalue because it has a name. Does this matter? Of course. `a` is not movable anymore. Therefore, we have to perform an expensive *copy* instead of a cheap *move*. But what is even worse, is that if the constructor of `T` (line 10) needs an rvalue, it will not work anymore.

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

Third iteration - `std::forward`

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

```
#include <iostream>

template <typename T,typename Arg>
T create(Arg&& a){
  return T(std::forward<Arg>(a));
}

int main(){

  std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;

  std::cout << std::endl;

}
```



Before we present the recipe for std::forward to achieve perfect forwarding, we will introduce the name, **universal reference**.

> 💡 The `(Arg&& a)` **universal reference** in line 4 is a powerful reference that can bind lvalues or rvalues. Sometimes the term perfect forwarding reference is used for this special reference.

To achieve perfect forwarding, we have to combine a universal reference with `std::forward`. `std::forward<Arg>(a)` returns the underlying type of `a`, because `a` is a universal reference. We can think of `std::forward` as a conditional move operation.

When the argument `a` is an rvalue, `std::forward` moves its argument. When the argument `a` is an lvalue, it copies the argument. Therefore, an rvalue remains an rvalue.

Now to the pattern:

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

We used exactly this pattern in the function template, `create`. Only the name of the type changed from `T` to `Arg` in the example above.

Fourth iteration - The perfect factory method

Variadic Templates are templates that can get an arbitrary number of arguments. That is exactly the feature missing from our current perfect factory method.

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

template <typename T, typename ... Args>
T create(Args&& ... args){
  return T(std::forward<Args>(args)...);
}
```

```
}

struct MyStruct{
  MyStruct(int i,double d,std::string s){}
};

int main(){

    std::cout << std::endl;

  // Lvalues
  int five=5;
  int myFive= create<int>(five);
  std::cout << "myFive: "  << myFive << std::endl;

  std::string str{"Lvalue"};
  std::string str2= create<std::string>(str);
  std::cout << "str2: " << str2 << std::endl;

  // Rvalues
  int myFive2= create<int>(5);
  std::cout << "myFive2: " << myFive2 << std::endl;

  std::string str3= create<std::string>(std::string("Rvalue"));
  std::cout << "str3: " << str3 << std::endl;

  std::string str4= create<std::string>(std::move(str3));
  std::cout << "str4: " << str4 << std::endl;

  // Arbitrary number of arguments
  double doub= create<double>();
  std::cout << "doub: " << doub << std::endl;

  MyStruct myStr= create<MyStruct>(2011,3.14,str4);

  std::cout << std::endl;

}
```

The three dots, in lines 5 – 7, are the so-called **parameter pack**. If the three dots (also called ellipsis) are on the left of `Args`, the parameter pack will be **packed**; if they are on the right of `arg`, the parameter pack will be **unpacked**.

In particular, the three dots in line 7 (`std::forward<Args>(args)...`) initiate each constructor call that performs perfect forwarding, and the result is impressive. Now, we can invoke the perfect factory method without any arguments (line 38) or with three arguments (line 41).

Further information

- lvalue or rvalue characteristics

- std::forward

- Variadic Templates

The example in the next lesson will build on our understanding of this topic.