Perfect Forwarding

We will learn about perfect forwarding in this lesson.

WE'LL COVER THE FOLLOWING

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

A Perfect Factory Method

Firstly, a short disclaimer: the expression a *perfect factory method* is not a formal term.

A perfect factory method is actually a generic factory method, meaning that the function should have the following characteristics:

- It can take an arbitrary number of arguments
- It can accept lvalues and rvalues as an argument
- It forwards it arguments identical to the underlying constructor

In other words, a perfect factory method should be able to create each arbitrary object.

Perfect forwarding enables it to write a function that can identically forward its arguments. In doing so, 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. To say it exactly, it should take it as a non-constant lvalue reference. The following is the function template `create` in our first iteration.

```
// perfectForwarding1.cpp

#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, due to the fact that the rvalue (line 21) cannot be bound to a non-constant lvalue reference.

Now, we have two ways to solve the issue.

1. Change the **non-constant lvalue reference** (line 6) in a **constant lvalue reference**. We can bind an rvalue to a constant lvalue reference. But that is not perfect since the function argument is constant and we can therefore not change it.
2. Overload the function template for a **constant lvalue reference** and a **non-const lvalue reference**. This is the easiest and better technique to

implement.

Second Iteration

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

```
// perfectForwarding2.cpp

#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;

}
```

The solution has two conceptional issues.

1. To support n different arguments, we must 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 materializes in the function body of `create` to an lvalue since it has a name. `a` is not movable anymore. Therefore, we have to perform an expensive *copy* rather than a cheap *move*. If the constructor of `T` (line 12) needs an rvalue, it will not work anymore

constructor of `T` (line 12) needs an lvalue, 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:

```
// perfectForwarding3.cpp

#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 from std::forward to get perfect forwarding, we will introduce the name universal reference.

> 💡 The **universal reference** `(Arg&& a)`, in line 6, 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, it moves its argument. When the argument `a` is an lvalue, it copies its argument. Therefore, an rvalue remains an rvalue.

Now to the pattern:

Now to the pattern:

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

We used this exact pattern in the function template `create`, but the name of the type has been changed from `T` to `Arg` in the example above.

Fourth Iteration - The Perfect Factory Method

Variadic Templates are templates that get an arbitrary number of arguments, which is exactly the feature missing from the perfect factory method.

```
// perfectForwarding4.cpp

#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));
```

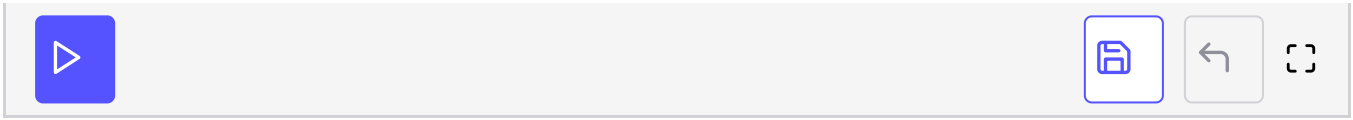
```
  std::cout << "str4: " << str4 << std::endl;

  // 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 7 - 9, are the so-called parameter pack. If the three dots (also called ellipse) are on the left of `Args`, the parameter pack will be **packed**. If they are at the right of `arg`, the parameter pack will be **unpacked**. In case of the dots in line 3, the expression to the left is expanded, and a ‘,’ is put in between.

In particular, the three dots in line 9 `std::forward<Args>(args)...` causes each constructor call that performs perfect forwarding and the result is impressive. Now, we can invoke the perfect factory method without any arguments (line 40) or with three arguments (line 43).

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