Forward

Now, we'll learn about the way std::forward helps us create generic function templates.

The function `std::forward`, defined in the header `<utility>`, empowers you to write function templates, which can identically forward their arguments. Typical use cases for `std::forward` are factory functions or constructors. **Factory functions** are functions which create an object and must therefore identically pass the arguments. Constructors often use their arguments to initialize their base class with the identical arguments. So `std::forward` is the perfect tool for authors of generic libraries:

```
// forward.cpp
...
#include <utility>
...
using std::initialiser_list;

struct MyData{
  MyData(int, double, char){};
};

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

...

int a= createT<int>();
int b= createT<int>(1);

std::string s= createT<std::string>("Only for testing.");
MyData myData2= createT<MyData>(1, 3.19, 'a');

typedef std::vector<int> IntVec;
IntVec intVec= createT<IntVec>(initialiser_list<int>({1, 2, 3}));
```

The function template `createT` has to take their arguments as a universal reference: Args&&... args`. A universal reference or also called forwarding reference is an rvalue reference in a type deduction context.

> 🔑 **std::forward in combination with variadic templates allows completely generic functions**
> If you use `std::forward` together with variadic templates, you can define completely generic function templates. Your function template can accept an arbitrary number of arguments and forward them unchanged.

In the next lesson, I will discuss the swap function of the C++ Standard Library.