std::bind and std::function

Programmers can use this pair of utilities to create and bind functions to variables.

The two functions `std::bind` and `std::function` fit very well together. While `std::bind` enables us to create new function objects on the fly, `std::function` takes these temporary function objects and binds them to a variable. Both functions are powerful tools from functional programming and need the header `<functional>`.

Let's consider the example here:

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <numeric>
#include <vector>

int main(){

  std::cout << std::endl;

  std::vector<int> myVec(20);
  std::iota(myVec.begin(), myVec.end(), 0);

  std::cout << "myVec: ";
  for (auto i: myVec) std::cout << i << " ";
  std::cout << std::endl;

  std::function< bool(int)> myBindPred= std::bind( std::logical_and<bool>(),
                                        std::bind( std::greater <int>(), std::placeholders::_

  myVec.erase(std::remove_if(myVec.begin(), myVec.end(), myBindPred), myVec.end());

  std::cout << "myVec: ";
  for (auto i: myVec) std::cout << i << " ";
  std::cout << std::endl;


}
```

Creating and binding function objects

> 🔑 **std::bind and std::function are mostly superfluous**
> `std::bind` and `std::function`, which were part of TR1, are mostly unnecessary with C++11. Instead, we can use lambda functions instead of `std::bind` and most often can use the automatic type deduction instead of `std::function`.

Now, let's discuss the behavior of `std::bind` and `std::function` in detail.

std::bind

Because of `std::bind`, we can create function objects in a variety of ways:

- bind the arguments to an arbitrary position.
- change the order of the arguments.
- introduce placeholders for arguments.
- partially evaluate functions.
- invoke the newly created function objects, use them in the algorithm of the STL or store them in `std::function`.

std::function

`std::function` can store arbitrary callables in variables. It's a kind of polymorphic function wrapper. A callable may be a lambda function, a function object, or a function. `std::function` is always necessary and can't be replaced by `auto` if the type of the callable must be specified explicitly.

To understand this more clearly, let's look at the example below:

```
#include <algorithm>
#include <functional>
```

```
#include <iostream>
#include <iterator>
#include <vector>


double divMe(double a, double b){
  return double(a/b);
}

using namespace std::placeholders;

int main(){

  std::cout << std::endl;

  // invoking the function object directly
  std::cout << "1/2.0= " << std::bind(divMe, 1, 2.0)() << std::endl;

  // placeholders for both arguments
  std::function<double(double, double)> myDivBindPlaceholder = std::bind(divMe, _1, _2);
  std::cout << "1/2.0= " << myDivBindPlaceholder(1, 2.0) << std::endl;

  // placeholders for both arguments, swap the arguments
  std::function<double(double, double)> myDivBindPlaceholderSwap = std::bind(divMe, _2, _1);
  std::cout << "1/2.0= " << myDivBindPlaceholderSwap(2.0, 1) << std::endl;

  // placeholder for the first argument
  std::function<double(double)> myDivBind1St = std::bind(divMe, _1, 2.0);
  std::cout<< "1/2.0= " << myDivBind1St(1) << std::endl;

  // placeholder for the second argument
  std::function<double(double)> myDivBind2Nd = std::bind(divMe, 1.0, _1);
  std::cout << "1/2.0= " << myDivBind2Nd(2.0) << std::endl;

  std::cout << std::endl;

}
```

Variation of Using Arguments

Let’s take a look at another example:

```
#include <cmath>
#include <functional>
#include <iostream>
#include <map>

int main(){

  std::cout << std::endl;

  // dispatch table
  std::map< const char , std::function<double(double, double)> > dispTable;
  dispTable.insert( std::make_pair('+', [](double a, double b){ return a + b;}));
  dispTable.insert( std::make_pair('-', [](double a, double b){ return a - b;}));
  dispTable.insert( std::make_pair('*', [](double a, double b){ return a * b;}));
  dispTable.insert( std::make_pair('/', [](double a, double b){ return a / b;}));

  // do the math
  std::cout << "3.5+4.5= " << dispTable['+'](3.5, 4.5) << std::endl;
  std::cout << "3.5-4.5= " << dispTable['-'](3.5, 4.5) << std::endl;
  std::cout << "3.5*4.5= " << dispTable['*'](3.5, 4.5) << std::endl;
  std::cout << "3.5/4.5= " << dispTable['/'](3.5, 4.5) << std::endl;

  // add a new operation
  dispTable.insert( std::make_pair('^', [](double a, double b){ return std::pow(a, b);}));
  std::cout << "3.5^4.5= " << dispTable['^'](3.5, 4.5) << std::endl;

  std::cout << std::endl;

};
```

A dispatch table with `std::function`

How does the magic work? The dispatch table in our case is an `std::map` that contains pairs of `const char` and `std::function<double(double,double)`. Of course, we can use an `std::unordered_map` instead of an `std::map`. `std::function` is a polymorphic function wrapper. Thanks to `std::function`, it can take anything that behaves like a function. This can be a function, a function object, or a lambda-function (line 12 -15). The only requirement of `std::function<double(double,double)>` is that its entities must have two double arguments and return a double argument. This requirement is fulfilled by the lambda-functions.

We use the function object in the lines 18 - 21. Therefore, the call of `dispTable['^']` in line 25 returns the function object which was initialized by the lambda-function `[](double a, double b){ return std::pow(a, b);}`. To execute the function object, two arguments are needed. We use them in the expression `dispTable['^'](3.5, 4.5)`.

An `std::map` is a dynamic data structure. Therefore, we can add and use the `'^'` operation (line 25) at runtime.

The type parameter of `std::function` defines the type of callables `std::function` will accept.

Function type	Return type	Type of the

		arguments
`double(double, double)`	`double`	`double`
`int()`	`int`	
`double(int, double)`	`double`	`int`, `double`
`void()`		

Return type and type of the arguments

Further information

- TR1

There will be an exercise for us in the next lesson for better understanding of this concept.