Functors (Function Objects)

Note: some of the code may be using C++ 11 features.

What is a **functor**?

Let's start with a very simple usage of functor:

#include <iostream>

#include <string>

class A

{

public:

void operator()(std::string str) {

std::cout << "functor A " << str << std::endl;

}

};

int main()

{

A aObj;

aObj("Hello");

}

We have a class **A** which has an operator() defined. In main(), we create an instance and passing in a string argument to that object. Note that we're using an instance as if it's a function. This is the core idea of **functors**.

A functor’s claim:   
"Anything behaves like a function is a function. In this case, **A** behaves like a function. So, **A** is a function even though it's a class."

We counters with a question:  
Why do we need you, functor? We can just use a regular function. We do not need you.

Functor explains the benefits of using it:

"We're not simple plain functions. We're smart. We feature more than **operator()**. We can have states. For example, the class A can have its member for the state. Also, we can have types as well. We can differentiate function by their signature. We have more than just the signature."

What is a functor?

Functor or function object is a C++ class which defines the operator **( )**. Functor let’s you create objects which “looks like” functions.

Consider below code to add two numbers:

#include<cassert>

struct MyAddFunctor {

// Constructor

MyAddFunctor(int inp){

x = inp;

}

// Defining operator()

int operator()(int y){

return x+y;

}

int x; // State

};

int main(){

MyAddFunctor func(5);

int ret = func(10);

//ret would be 15.

assert(ret == 15);

int ret2 = func(25);

// ret would br 30

assert(ret2 == 30);

}

Now **MyAddFunctor** is a functor. It is a functor because **MyAddFunctor** is C++ class which implemented operator().

How are functors called?

We call functors just like regular C++ class. Unlike functions we first need to create object of functor like MyAddFunctor func(5). We are passing 5 as argument only becase we have a constructor defined in **MyAddFunctor** class which takes int as an argument.

Now when object of functor class is created we invoke the function using () operator just like func(10).

We could also combine instantiation and invocation in single statement as MyAddFunctor(5)(10).

Why are they called as Function Objects?

The reason why they are called as function objects is because we can call class **MyAddFunctor** as if it is a function. Example : func(10).

Why are functors used?

The main advantage of using functor is that they store state. Consider our previous example of **MyAddFunctor**. In that example we stored value of x in the class and so we didn’t need to pass x in each call.

One can argue that the work done by **MyAddFunctor** can simply be done by writing C++ function as below:

int addFunction(int x){

return 5+x;

}

But in **addFunction** we are hardcoding the value 5 and in functor we are not doing so. So **MyAddFunctor** is more customizable.

Functor is a parameterized function

A functor is a **parameterized** function.

#include <iostream>

#include <string>

class A

{

public:

A(int i) : id(i) {}

void operator()(std::string str) {

std::cout << "functor A " << str << std::endl;

}

private:

int id;

};

int main()

{

A(2014)("Hello");

}

Now, the class **A** is taking two parameters.

Why do we want like that. Can we just use a regular function that takes the two parameters?

Let's look at the following example:

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void add10(int n)

{

cout << n+10 << " ";

}

int main()

{

std::vector<int> v = { 1, 2, 3, 4, 5 };

for\_each(v.begin(), v.end(), add10); // {11 12 13 14 15}

}

In main(), the function **add10** will be invoked for each element of the vector **v**, and prints out one by one after adding 10.

But note that we hard coded **10**. We can use a global variable for that. But we do not want to use global variable.

We may use template like this:

template<int number>

void addNumber(int i)

{

std::cout << i + number << std::endl;

}

int main()

{

std::vector<int> v = { 1, 2, 3, 4, 5 };

for\_each(v.begin(), v.end(), addNumber<10>); // 11 12 13 14 15

}

But there is a caveat: we cannot easily change the value to addNumber because a template should be resolved at compile time and thus it requires const. Therefore, we cannot do this in the main():

int main()

{

std::vector<int> v = { 1, 2, 3, 4, 5 };

int val = 10;

for\_each(v.begin(), v.end(), addNumber<val>); // Not OK

}

As you guessed it, we need to use functor as shown in the code below:

#include <iostream>

#include <vector>

#include <algorithm>

class A

{

public:

A(int k) : val(k) {}

void operator()(int i) {

std::cout << i + val << std::endl;

}

private:

int val;

};

int main()

{

std::vector<int> v = { 1, 2, 3, 4, 5 };

int val = 10;

for\_each(v.begin(), v.end(), A(val)); // 11 12 13 14 15

}

Note that we set the **A::val** via constructor. If we want another value for the addition, we may use another instance with a different value for the constructor.

In this way, we can change the value to add using the state of the function object. We get this kind of flexibility by using functors.

This is the fundamental advantage of functors - they can easily preserve a state between calls. In other words, they can support multiple independent states, one for each functor instance while functions only support a single state.

Another good thing is that we do not have to write all functors. STL provides quite a few functors for us.

STL Functors

1. Arithmetic binary functors  
   plus, minus, multiplies, divides, modulus
2. Relational binary functors  
   equal\_to, not\_equal\_to, greater, greater\_equal, less, less\_equal
3. Logical binary functors  
   logical\_and, logical\_or
4. Built-in functors of unary type (available from <functional>)  
   negate, logical\_not

Simple usage of them could be:

int n = std::multiplies<int>()(4, 5); // n = 4\*5

if (std::not\_equal\_to<int>()(n, 10))

std::cout << n << std::endl; // 20

Parameter binding

In the code below, **transform()** requires a functor that takes one parameter, however, the **multiplies()** takes two. That's why we need a **bind()**. The **bind()** take its first parameter from the set and the second parameter 10 10.

#include <iostream>

#include <vector>

#include <set>

#include <algorithm>

#include <iterator> // std::back\_inserter

#include <functional> // std:: bind

int main()

{

std::set<int> s = { 1, 2, 3, 4, 5 };

std::vector<int> v;

// multiply set's elements by 100 and store it into v

std::transform(s.begin(), s.end(), // source

std::back\_inserter(v), // destination

std::bind(std::multiplies<int>(), std::placeholders::\_1, 100) // C++11

);

}

The **transform()** takes elements from the set as it's first parameter (**std::placeholders::\_1**, and multiplies the second parameter (**100**). Then, it puts each of them into the vector via **std::back\_inserter()**.

One of the issues in our earlier example can be resolved by using the **bind()** function:

#include <iostream>

#include <vector>

#include <algorithm>

#include <functional> // std:: bind

void addNumber(int i, int number)

{

std::cout << i + number << std::endl;

}

int main()

{

std::vector v = { 1, 2, 3, 4, 5 };

for\_each(v.begin(), v.end(),

std::bind(addNumber,

std::placeholders::\_1, 10)); // 11 12 13 14 15

}