

Functors (Function Objects)

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 functors' 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."

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
}

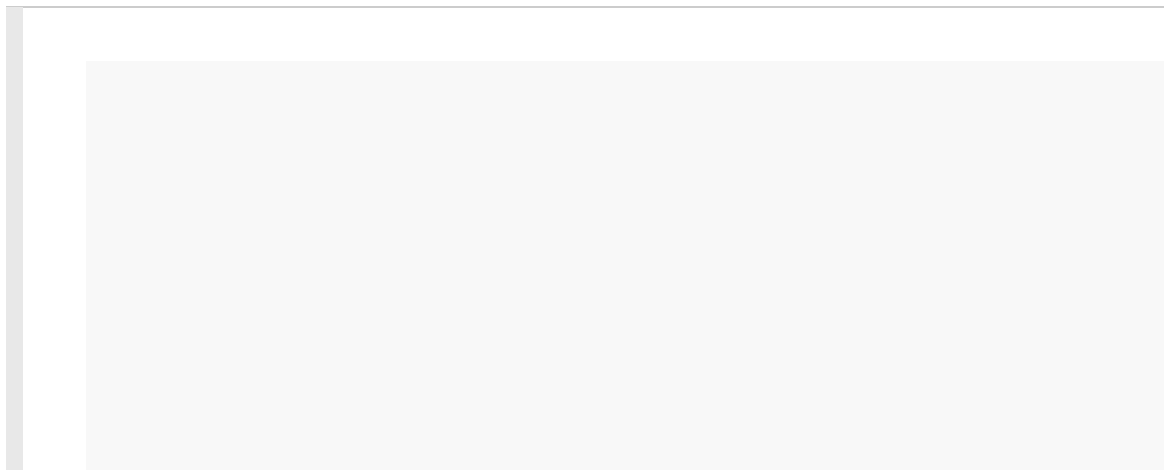
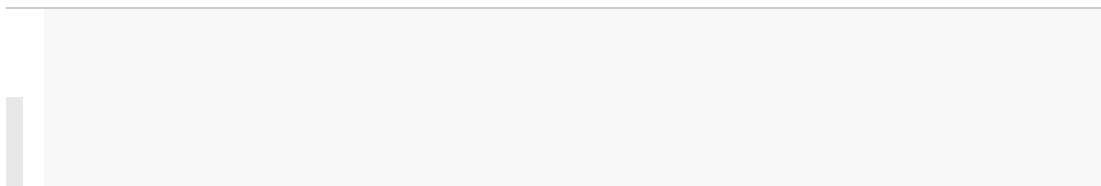
```

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.



C++ TUTORIAL OPERATOR OVERLOADING

Function Overloading

Function overloading lets us use multiple functions sharing the same name. We usually utilize the function overloading to design a family of functions that do the same thing while using different argument lists.

The key to function overloading is a function's argument list (**function signature**). If two functions use the same number and types of arguments in the same order, they have the same signature. C++ allows us to define two multiple functions by the same name, provided that the functions have different signatures. The signature can differ:

1. in the number of arguments
2. or in the type of arguments
3. or both

For example, we defines several versions of **f()** with the following prototypes:

```
void f(const char *s, int n); // (a)
void f(double d, int n);      // (b)
void f(long l, int n);        // (c)
void f(int m, int n);         // (d)
void f(const char *s);        // (e)
```

Some signatures which appears to be different but actually both have the same signature:

```
double square(double d);
double square(double &d);
```

But we need to look with compiler's perspective. To call them, we use:

```
square(z);
```

and the **z** argument matches both **double d** and the **double &d**, thus the compiler has no way of knowing which function to use. So, to avoid such ambiguity, when it checks function signatures, the compiler considers a reference to a type and the type itself to be the same signature.

Note that in the function-matching process, the compiler discriminates between **const** and **non-const** variables:

```
int f(char *s);           // overloaded
int f(const char *s);     // overloaded
```

Also note that the signature, not the return type, enables function overloading. For instance, the following two have the same signature, and can't be overloaded.

```
float f(int m, int *n)    // Not overloaded
double f(int m, int *n)  // Not overloaded
```

Operator Overloading

Here is a very simple code that shows the essence of operator overloads: '+', '++ (post)', and '++ (pre)'. If you do not understand what's going on in the code, please do not worry, at the end of this chapter, you will know how it works.

```
class A
{
public:
    A() {}
    explicit A(int n) :data(n) {}
    int data;
    A& operator+(A&);
    A operator++(int);
    A& operator++();

    int Get()
    {
        return data;
    }
};

// + overloading
A& A::operator+(A& obj)
{
    A tmp = *this;
    tmp.data = this->data + obj.data;
    return tmp;
}

// post increment (x++) overloading
// returns original value, and then increment the value
// copy needed
// return a locally created object.
// Note that it's not returning a reference since it's a temporary obj.
A A::operator++(int)
```

```

{
    A tmp = *this;
    this->data = (this->data)++;
    return tmp;
}

// pre increment (++x) overloading
// returns incremented tvalue
// no copy necessary
A& A::operator++()
{
    this->data = (this->data)++;
    return *this;
}

int main()
{
    A obj1(10);
    A obj2(20);
    A obj3 = obj1 + obj2; // obj3.data = 10 + 20 = 30

    cout << obj3.Get() << endl;

    A obj4 = obj1++;      // obj4.data = 10, obj1.data = 11
    A obj5 = ++obj2;      // obj5.data = 21, obj2.data = 21
    system("pause");
    return 0;
}

```

Operator overloading extends the overloading concept to operators so that we can assign new meanings to C++ operators. It lets us extend operator overloading to user-defined types. That is by allowing us to use the "+" to add two objects. The compiler determines which definition of addition to use depending on the number and type of operands. Overloaded operators can often make code look more natural. In other words, operator overloading can be very useful to make our class look and behave more like built-in types.

To overload an operator, we use a special function, **operator function**. For example, when we overload "+":

```
operator+(argument_list)
```

Suppose, for example, that we have a **MyComplex** class for which we define an **operator+()** member function to overload the + operator so that it adds one complex number to another complex number. Then, if **c1**, **c2**, **c3** are all objects of **MyComplex** class, we can write this:

```
c3 = c1 + c2;
```

The compiler, recognizing the operands as belonging to the **MyComplex** class, replaces the operator with the corresponding operator function:

```
c3 = c1.operator+(c2);
```

The function then use the **c1** object which invokes the method, and the **c2** object is passed as an argument to calculate the sum, and returns it. Note that we use assignment operator = which is also need to be overload.

Overloading '='

In this section we'll learn how to overload the assignment (=) operator between two objects of the Complex class.

Let's look at the following code:

```
class MyComplex
{
private:
    double real, imag;
public:
    MyComplex() {
        real = 0;
        imag = 0;
    }

    MyComplex(double r, double i) {
        real = r;
        imag = i;
    }

    double getReal() const {
        return real;
    }

    double getImag() const {
        return imag;
    }

    MyComplex & operator=(const MyComplex &); //Overloading '='
    MyComplex & operator+(const MyComplex&); //Overloading '+'
};

//Overloading '='
MyComplex & MyComplex::operator=(const MyComplex& c) {
    real = c.real;
    imag = c.imag;
    return *this;
}
```



```

//Overloading '+'
MyComplex & MyComplex::operator+(const MyComplex& c) {
    real += c.real;
    imag += c.imag;
    return *this;
}

int main()
{
    //Overloading '='
    MyComplex c1(5, 10);
    MyComplex c2(50, 100);
    cout << "c1= " << c1.getReal() << "+" << c1.getImag() << "i" << endl;
    cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;

    //c2.operator=(c1);
    c2 = c1;

    cout << "assign c1 to c2:" << endl;
    cout << "c2= " << c2.getReal() << "+" << c2.getImag() << "i" << endl;

    //-----
-
    // Overloading '+'
    MyComplex c3(10, 100);
    MyComplex c4(20, 200);
    cout << "c3= " << c3.getReal() << "+" << c3.getImag() << "i" << endl;
    cout << "c4= " << c4.getReal() << "+" << c4.getImag() << "i" << endl;

    //c5 = c3.operator+(c4)
    MyComplex c5 = c3 + c4;
    cout << "adding c3 and c4" << endl;
    cout << "c3= " << c3.getReal() << "+" << c3.getImag() << "i" << endl;
    cout << "c4= " << c4.getReal() << "+" << c4.getImag() << "i" << endl;
    cout << "c5= " << c5.getReal() << "+" << c5.getImag() << "i" << endl;

    system("pause");
    return 0;
}

```

Note that when we're using '+' for the object of MyComplex type,

```
c5 = c3 + c4;
```

actually, we are calling a function something like this.

```
c5 = c3.operator+(c4)
```

Output of the code above is:

```

c1= 5+10i
c2= 50+100i

```

```
assign c1 to c2:
c2= 5+10i
c3= 10+100i
c4= 20+200i
adding c3 and c4
c3= 30+300i
c4= 20+200i
c5= 30+300i
```

We got the right result at least for **c5**. But the value of **c3** has been changed.

Let look at the code overloading '+'.

```
MyComplex & MyComplex::operator+(const MyComplex& c) {
    real += c.real;

    imag += c.imag;

    return *this;
}
```

As it turned out, the operation inside the function returning the reference to **c3** object which has been changed.

So, let's rewrite the overloading function.

```
const MyComplex operator+(const MyComplex & );  
const MyComplex MyComplex::operator+(const MyComplex& c)  
{  
    MyComplex temp;  
    temp.real = this->real + c.real;  
    temp.imag = this->imag + c.imag;  
    return temp;  
}
```

Note that this doesn't return **Complex &**, but instead returns a **const Complex** class variable. As you can see, the implementation is a little bit different from the previous example. Here, we're not returning ***this**. Instead, we're creating a temporary variable and assigning the results of the addition to **temp**. This explains why the function returns **const**

Complex and not Complex &. In other words, the function creates a new **MyComplex** object **temp** that represents the sum of the other two **MyComplex** objects. Returning the object creates a copy of the object that the calling function can use. If the return type were **MyComplex &**, however, the reference would be the **temp** object. But the **temp** object is a **local variable** and is destroyed when the function terminates, so the reference would be a reference to a non-existing object. Using a **MyComplex** return type, however, means the program constructs a **copy** of **MyComplex** object before destroying it, and the calling function gets the copy.

Then, we'll get the right answer.

```
c1= 5+10i
c2= 50+100i
assign c1 to c2:
c2= 5+10i
c3= 10+100i
c4= 20+200i
adding c3 and c4
c3= 10+100i
c4= 20+200i
c5= 30+300i
```

Function Call Operator () Overloading

The function call operator can be overloaded for objects of class type. The overloaded **operator()** should be declared as a member function. It is invoked by applying an argument list to an object of the class type.

In the example below, we call the algorithm **transform()** to apply the operation defined by **absValue** to every element of the **vec**.

```
class absValue
{
public:
    int operator()(int val) {
        return val < 0 ? -val : val;
    }
};

int main()
{
    int a[] = { -3,-2,-1, 0, 1, 2, 3 };
    int size = sizeof(a) / sizeof(a[0]);
    vector<int> vec(a, a + size);

    for (int i = 0; i < size; i++) cout << vec[i] << " ";

    transform(vec.begin(), vec.end(),
              vec.begin(),
              absValue());

    cout << "\nafter transform()\n";
    for (int i = 0; i < size; i++) cout << vec[i] << " ";

    system("pause");
    return 0;
}
```