Function-Call Operator

A simple example:

```
struct absInt{
    int operator()(int val)const{
        return val<0?-val:val;
    }
};
```

Call the function-call operator:

```
int i=-42;
absInt absobj;
int ui=absobj(i); // pass i to absobj.operator(int val)
```

Even though `absobj` is an object, not a function, we can "call" this object. Calling an object runs its overloaded call operator. In this case, that operator takes an `int` value and returns its absolute value.

Another example that can be executed.

```
#include<iostream>
class Foo {
private:
    std::string str;
public:
    Foo(const std::string  &s=std::string()):str(s){}
    ~Foo() = default;
    void operator()(void)const {
        std::cout << "here";
    }
};
int main() {
    Foo f;
    f();
}
```

Classes that overload the call operator allow objects of its type to be used as if they were a function. Because such classes can also store state, they can be more flexible than ordinary functions.

> The function-call operator must be a member function. A class may define multiple versions of the call operator, each of which must differ as to the number or types of their parameter.

```
class Foo;
return_type operator()();
return_type operator()(T);
```

Classes that define the function-call operator, will be referred to as **function object**. We can call such objects, and thus these objects act like the functions.

Function-Object Classes with State

```
class PrintStr{
    public:
        PrintStr(ostream &o=std::cout, char c=' '):os(o),sep(c){}
        void operator()(const string &s)const {
            os<<s<<sep;
        }
    private:
        ostream &os;
        char sep;
};

PrintStr printer;
printer(s); // s is a string
PrintStr errors(std::cerr,'\n');
errors(s);
```

Function objects are most often used as arguments to the generic (ж   ež   ) algorithm. For example, we can use the library `for_each` algorithm.

```
for_each(vs.begin(),ve.end(),PrintStr(std::cout,'\n'));
```

Lambdas Are Function Objects

When we write a lambda, the compiler translates that expression into an unnamed object of an unnamed class. The classes generated from a lambda contain an overload function-call operator.

For example, the lambda that we passed as the last argument to `stable_sort:`

```
stable_sort(vc.begin(),vc.end(),[](const string &a,const string &b)->bool{return
a.size()<b.size();})

// other version
auto func=[](const string &a,const strnig &b){
    return a.size()<b.size();
}
stable_sort(vc.begin(),vc.end(),func);
```

The unnamed class yielded from the lambda function would look like

```
class ShorterStr{
    public:
        bool operator()(const string &s1,const string &s2)const {
            return s1.size()<s2.size();
        }
};
```

The generated class has a single member, which is a function-call operator that takes two `string`s. The parameter list and function body are the same as the lambda. Thus, the code can be rewriten as,

```
stable_sort(ve.begin(),ve.end(),ShorterStr());
```

The third argument is a newly constructed `ShorterStr` object. The code in `stable_sort` will "call" this object each time it compares two `string`s, and pass these two `string`s into function-call operator's parameters.

Classes Representing Lambdas with Captures

```
auto wc=find_fi(vc.begin(),vc.end(),[sz](const string &a)->bool{return
a.size()>=sz;};
```

This lambda function would generate a class that looks somethinig like

```
class Foo{
    public:
        Foo(size_t n):sz(n){}// parameter for each captured varable

        //
        bool operator()(const string &s)const{return s.size()>=sz;}
    private:
        size_t sz; // a data member for each variable captured by value
};
```

Variable that are captured by value are copied into the lambda. As a result, classes generated from lambdas that capture variables by value have data members corresponding to each such variable. These classes also have a constructor to initializer these data members from the value of the captured variables.

In contrast, a lambda captures a variable by reference, the program would ensure that the variable to which the reference refers exists when the lambda is executed. Therefore, the compiler is permitted to use the reference directly without storing that references as a data member in the generated class.

The generated class `Foo` has a data member (`size_t sz`) and a constructor to initialize that member. This synthesized class does not have a default constructor; to use this class, we must pass an argument.

```
auto wc=find_if(vc.begin(),vc.end(),Foo(sz));
auto wc=find_if(vc.begin(),vc.end(),Foo());// error
```

> Classes generated from a lambda expression have a deleted default constructor, deleted assignment operators, and a default destructor. Whether the class has a defaulted or deleted copy/move constructor depends in the usual ways on the types of the captured data members.

Library-Defined Function Objects

The standard libray defines a set of classes that represent the arithmetic, relational, and logical operators.

`plus` class has a function-call operator that applies `+` to a pair of operands; `modulus` `%`; `equal_to` `==`; and so on.

These classes are templates to which we supply a single type. That type specifies the parameter type for the call operator. For example, `plus<string>` applies the `string` addition operator to `string` objects;

```
plus<int> intadd;
negate<int> intnegate;

int sum=intadd(10,20); // equivalent to sum=30;
sum=intnegate(intadd(10,20));// sum=-30
```

These types, list in the following table, are defined in the `functional` header.

arithmetic	*relational*	*logical*
`plus<T>`	`equal_to<T>`	`logical_and<T>`
`minus<T>`	`greater<T>`	`logical_or<T>`
...	...	...

Using a Library Function Object with the Algorithm

The function-object classes that represent operators are often used to override the default operator used by an algorithm.

```
sort(vec.begin(),vec.end(),[](const int &a,const int &b)->bool{return a<b;});
```

One important rule is we cannot compare two unrelated pointers, of which the behaviour is undefined. We can make it works using library function objects:

```
vector<string *> nametable;// vector of pointers
sort(nametable.begin(),nametable.end(),[](string *a,string *b){return a<b;});//
error, the pointers in nametable are unrelated.
```

```
sort(nametable.begin(),nametable.end(),less<string *>());
// ok, library guarantees that less on pointer types is well defined.
```

Callable Objects and `function`

C++ has several kinds of callable objects:

1. functions
2. pointers to functions
3. lambdas
4. objects created by `bind`
5. classes that overload the function-call operator (function objects)

Like any other object, a callable object has a type. For example, each lambda has its own unique (unnamed) class type. Function and function-pointer types vary by their return type and argument types, and so on.

However, two callable objects with different types may share the same **call signature**. The call signature specifies the type returned by a call to the object and the argument type that must be passed in the call. A call signature corresponds to a function type. E.g.,

```
int(int,int)
```

is a function types that takes two `int`s and returns and `int`.

Different Types Can Have the Same Call Signature

Sometimes we want to treat several callable objects that share a call signature as if they had the same type. For example, consider the following different types of callable objects.

```
// ordinary function
int add(int i,int j){return i+j;}

// lambda, which generates an unnamed function-object class
auto mod=[](int a,int b){return a%b;}

// function-object class
class div{
    public:
        int operator()(int a,int b){
            return a/b;
        }
};
```

The call signature is

```
int(int,int)
```

We could define a **function table** to store "pointers" to these callables. When the program needs to execute a particular operation, it will look in the table to find which function to call.

```
// maps an operator to a pointer to a function taking two ints and returning an int
std::map<std::string,int(*) (int,int)> binops;

binops.insert({"+",add});// {"+",add} is a pair
binops.insert({"%",mod});// error, mod is not a pointer to function
```

We can't store `mod` or `div` in `binops`. The problem is that `mod` is a lambda, and each lambda has its own class type. That type does not match the type of the values stored in `binops`.

The Library `function` type

We can solve this problem using a new library type named `function` that is defined in the `functional` header;

`function` is a template. We specify the type inside angle brackets:

```
// f is a null function object that can store callable
// objects with a call signature that is equivalent to the function
// type int(int,int)
function<int(int,int)> f;

function<int(int,int)> f1=add; // function pointer
function<int(int,int)> f2=div();// object of a function-object class
function<int(int,int)> f3=[](int a,int b){return a*b;} // lambda

f1(4,2); // 4+2=6;
f2(4,2);// 4/2=2;
```

Refine the function table using `map`, which is defined via `function` type.

```
map<string,function<int(int,int)> >binops={
    {"+",add},// function pointer
    {"-",std::minus<int>()},// library function object
    {"/",div()},// user-defined function object
    {"*",[](int i,int j){return i*j;}},// unnamed lambda
    {"%",mod}
};
```

When we index `binops`, we get a reference to an object of type `function`. The `function` type overloads the call operator. That call operator takes its own arguments and pass them along to its stored callable object:

```
binops["+"](10,5); // call add(10,5);
binops["-"](10,5); // uses the call operator of the minus<int> object

binops["/"](10,5); // uses the call operator of the div object
binops["*"](10,5); // calls the lambda function object
```