【C++】Day68

Class	C++
Date	@March 6, 2022
Material	
Series Number	
Summary	

【Ch14】Overloaded Operations and Conversions

14.8.2 Library-Defined Function Objects

The standard library defines a set of classes that represent the arithmetic, relational, and logical operators. Each class defines a call operator that applies the named operations.

For example, the `plus` class has a function-call operator that applies `+` to a pair of operands; the `modulus` class defines a call operator that applies the binary `%` operator; the `equal_to` class applies `==`.

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; // function object that can add two int values
negate<int> intNegate; // function object that can negate an int value
int sum = intAdd(10, 20); // sum = 30
sum = intNegate(intAdd(10, 20)); // sum = -30
sum = intAdd(10, intNegate(10)); // sum = 0
```

These types are defined in the `functional` header.

Table 14.2. Library Function Objects

Arithmetic	*Relational*	*Logical*
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

Using a Library Function Object with the Algorithms

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

For example, if `svec` is a `vector<string>`

```
sort(svec.begin(), svec.end(), greater<string>());
```

This code sorts the vector in descending order.

One important aspect of these library function object is that the library guarantees that they will work for pointers. Recall that comparing two unrelated pointers is undefined. However, we might want to sort a vector of pointers based on their address in memory.

Although it would be undefined for us to do so directly, we can do so through one of the library function objects:

```
vector<string *> nameTable; // vector of pointers
sort(nameTable.begin(), nameTable.end(), less<string *>());
```

14.8.3 Callable Objects and function

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

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 types that must be passed in the call. A call signature corresponds to a function type. For example:

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

is a function type that takes two ints and returns an int.

We might want to use these callables to build a simple desk calculator. To do so, we'd want to define a function table to store "pointers" to these callables.

We could define our map as:

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

We could put a pointer to add into binops as follows:

```
// ok: add is a pointer to function of the appropriate type
binops.insert({"+", add});
```

Different Types can Have Same Type Signature

```
// ordinary function
int add(int i, int j) { return i + j; }
auto mod = [](int i, int j) { return i % j; }
struct div {
  int operator()(int i, int j) {
    return i / j;
  }
};
```

The Library function Type

We can solve this problem using a new library type named `function` that is defined in the `functional` header. The following table lists the operations defined by `function`.

Table 14.3. Operations on function

function<T> f;	f is a null function object that can store callable objects with a call signature that is equivalent to the function type T (i.e., T is *retType* (*args*)).
function<T> f(nullptr);	Explicitly construct a null function.
function<T> f(obj);	Stores a copy of the callable object obj in f.
f	Use f as a condition; true if f holds a callable object; false otherwise.
f(*args*)	Calls the object in f passing *args*.
Types defined as members of function<T>	
result_type	The type returned by this function type's callable object.
argument_type first_argument_type second_argument_type	Types defined when T has exactly one or two arguments. If T has one argument, argument_type is a synonym for that type. If T has two arguments, first_argument_type and second_argument_type are synonyms for those argument types.

function is a template. As with other templates we've used, we must specify

`function` is a template. We must specify additional information when we create a function type. In this case, that information is the call signature of the object that this particular function type can represent.

```
function<int (int, int)>
```

Here we've declared a `function` type that can represent callable objects that return an `int` result and have two `int` parameters.

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

We can now redefine our map using this function type:

```
// table of callable objects corresponding to each binary operator
// all the callables must take two ints and return an int
```

```
// an element can be a function pointer, function object, or lambda
map<string, function<int (int, int)>> binops;
```

Exercise

Exercises Section 14.8.3

Exercise 14.44: Write your own version of a simple desk calculator that can handle binary operations.

See 14_44.cpp for code