MODULE 38
--THE STL--
FUNCTION OBJECT & MISC

My Training Period: hours

Note: Compiled using Microsoft Visual C++ .Net, win32 empty console mode application.

Abilities

- Able to understand and use the function objects.

38.1 Function Objects

- Functional arguments for algorithms don't have to be functions. They can be **objects** that behave as functions. Such an object is called a **function object**, or **functor**.
- Sometimes you can use a function object when an ordinary function won't work. The STL often uses function objects and provides several function objects that are very helpful.
- Function objects are another example of the generic programming capabilities and the concept of pure abstraction. You could say that anything that **behaves** like a function is a function. So, if you define an object that behaves as a function, it can be used as a function.
- A functional behavior is something that you can call by using parentheses and passing arguments. For example:

```
function(arg1, arg2); //a function call
```

- So, if you want objects to behave this way you have to make it possible to call them by using parentheses and passing arguments.
- All you have to do is define `operator()` with the appropriate parameter **,** for example:

```
class XYZ
{
public:
//define "function call" operator
return-value operator() (arguments) const;
...
};
```

- Now you can use objects of this class to behave as a function that you can call:

```
XYZ foo;
...
//call operator() for function object foo
foo(arg1, arg2);
```

- The call is equivalent to:

```
//call operator() for function object foo
foo.operator() (arg1, arg2);
```

- The following is a complete example.

```
//function object example
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//simple function object that prints the passed argument
class PrintSomething
{
public:
void operator() (int elem) const
{
cout<<elem<<' ';
}
};

int main()
{
```

```
vector<int> vec;
//insert elements from 1 to 10
for(int i=1; i<=10; ++i)
vec.push_back(i);

//print all elements
for_each (vec.begin(), vec.end(), //range
PrintSomething()); //operation
cout<<endl;
}
```

Output:

- The class `PrintSomething()` defines objects for which you can call `operator()` with an int argument.
- The expression:

```
PrintSomething()
```

- In the statement

```
for_each(vec.begin(), vec.end(),
PrintSomething());
```

- Creates a temporary object of this class, which is passed to the `for_each()` algorithm as an argument. The `for_each()` algorithm is written like this:

```
namespace std
{
        template <class Iterator, class Operation>
        Operation for_each(Iterator act, Iterator end, Operation op)
        {
                while(act != end)
                { //as long as not reached the end
                        op(*act); //call op() for actual element
                        act++; //move iterator to the next element
                }
        return op;
        }
}
```

- `for_each()` uses the temporary function object op to call `op(*act)` for each element act. If the third parameter is an ordinary function, it simply calls it with `*act` as an argument.
- If the third parameter is a function object, it calls operator () for the function object op with `*act` as an argument. Thus, in this example program `for_each()` calls:

```
PrintSomething::operator()(*act)
```

- Function objects are more than functions, and they have some advantages:

38.2 Function objects are smart functions

- Objects that behave like pointers are smart pointers. This is similarly true for objects that behave like functions: They can be smart functions because they may have abilities beyond `operator()`. Function objects may have other member functions and attributes.
- This means that function objects have a state. In fact, the same function, represented by a function object, may have different states at the same time. This is not possible for ordinary functions.
- Another advantage of function objects is that you can initialize them at runtime before you call them.

38.3 Each function object has its own type.

- Ordinary functions have different types only when their signatures differ. However, function objects can have different types even when their signatures are the same.
- In fact, each functional behavior defined by a function object has its own type. This is a significant improvement for generic programming using templates because you can pass functional behavior as a template parameter.
- It enables containers of different types to use the same kind of function object as a sorting criterion. This ensures that you don't assign, combine, or compare collections that have different sorting criteria.
- You can even design hierarchies of function objects so that you can, for example, have different, special kinds of one general criterion.

38.4 Function objects are usually faster than ordinary functions.

- The concept of templates usually allows better optimization because more details are defined at compile time. Thus, passing function objects instead of ordinary functions often results in better performance.
- For example, suppose you want to add a certain value to all elements of a collection. If you know the value you want to add at compile time, you could use an ordinary function:

```
void add10 (int& elem)
{
        elem += 10;
}

void funct()
{
        vector<int> vec;
        ...
        for_each(vec.begin(), vec.end(), //range
        add10); //operation
}
```

- If you need different values that are known at compile time, you could use a template instead:

```
template <int theValue>
void add(int& elem)
{
        elem += theValue;
}
void funct()
{
        vector<int> vec;
        ...
        for_each (vec.begin(), vec.end(), //range
        add<10>); //operation
}
```

- If you process the value to add at runtime, things get complicated. You must pass the value to the function before the function is called.
- This normally results in some global variable that is used both by the function that calls the algorithm and by the function that is called by the algorithm to add that value. Look like a messy style.
- If you need such a function twice, with two different values to add, and both values are processed at runtime, you can't achieve this with one ordinary function. You must either pass a tag or you must write two different functions.
- With function objects, you can write a smarter function that behaves in the desired way because the object may have a state; it can be initialized by the correct value.
- Here is a full example:

```
//function object example
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

//function object that adds the value with which it is initialized
class AddValue
{
private:
//the value to add
int theValue;
public:
//constructor initializes the value to add
AddValue(int val) : theValue(val) {}
```

```
//the function call for the element adds the value
void operator() (int& elem) const
{
       elem += theValue;
}
};

int main()
{
       list<int> lst1;

       //The first call of for_each() adds 10 to each value:
       for_each(lst1.begin(), lst1.end(), //range
       AddValue(10)); //operation
}
```

- Here, the expression `AddValue(10)` creates an object of type `AddValue` that is initialized with the value `10`. The constructor of `AddValue` stores this value as the member `theValue`.
- Inside `for_each()`,`"()"` is called for each element of lst1. Again, this is a call of `operator()` for the passed temporary function object of type `AddValue`. The actual element is passed as an argument. The function object adds its value `10` to each element. The elements then have the following values: after adding 10: 11 12 13 14 15 16 17 18 19
- The second call of `for_each()` uses the same functionality to add the value of the first element to each element. It initializes a temporary function object of type `AddValue` with the first element of the collection:

```
AddValue (*lst1.begin())
```

- The output is then as follows:

```
after adding first element: 22 23 24 25 26 27 28 29 30
```

- By using this technique, two different function objects can solve the problem of having a function with two states at the same time.
- For example, you could simply declare two function objects and use them independently:

```
AddValue addx (x); //function object that adds value x
AddValue addy (y); //function object that adds value y
for_each (vec.begin(), vec.end(), //add value x to each element
addx);
...
for_each (vec.begin(), vec.end(), //add value y to each element
addy);
...
for_each (vec.begin(), vec.end(), //add value x to each element
addx);
```

38.5 Predefined Function Objects

- The C++ standard library contains several predefined function objects that cover fundamental operations. By using them, you don't have to write your own function objects in several cases.
- A typical example is a function object used as a sorting criterion.
- The default sorting criterion for `operator<` is the predefined sorting criterion `less<>`. Thus, if you declare:

```
set<int> st;
```

- It is expanded to:

```
set<int, less<int> > st; //sort elements with <
```

- From there, it is easy to sort elements in the opposite order:

```
set<int, greater<int> > st; //sort elements with >
```

- Similarly, many function objects are provided to specify numeric processing. For example, the following statement **negates** all elements of a collection:

```
transform(vec.begin(), vec.end(), //source
vec.begin(), //destination
negate<int>()); //operation
```

- The expression:

```
negate<int>()
```

- Creates a function object of the predefined template class negate that simply returns the negated element of type `int` for which it is called.
- The `transform()` algorithm uses that operation to transform all elements of the first collection into the second collection. If source and destination are equal (as in this case), the returned negated elements overwrite themselves. Thus, the statement negates each element in the collection.
- Similarly, you can process the square of all elements in a collection:

```
//process the square of all elements
transform(vec.begin(), vec.end(), //first source
vec.begin(), //second source
vec.begin(), //destination
multiplies<int>()); //operation
```

- Here, another form of the `transform()` algorithm combines elements of two collections by using the specified operation, and writes the resulting elements into the third collection.
- Again, all collections are the same, so each element gets multiplied by itself, and the result overwrites the old value.

----------**The rEaL End for STL**----------
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.

38.6 Miscellaneous

The following items may not be covered in details in this part of tutorial but you may have encountered them somewhere in the various program examples.

Class `pair`

Is provided to treat two values as a single unit. It is used in several places within the C++ standard library. In particular, the container classes' `map` and `multimap` use pairs to manage their elements, which are key/value pairs.

`make_pair()` template function

Enables you to create a value pair without writing the types explicitly.

The `auto_ptr` type.

The `auto_ptr` type is provided by the C++ standard library as a kind of a smart pointer that helps to avoid resource leaks when exceptions are thrown.

Numeric types

In general have platform-dependent limits. The C++ standard library provides these limits in the template `numeric_limits`. These numeric limits replace and supplement the ordinary preprocessor constants of C.
These constants are still available for integer types in <`climits`> and <`limits.h`>, and for floating-point types in <`cfloat`> and <`float.h`>.
The new concept of numeric limits has two advantages: First, it offers more type safety. Second, it enables a programmer to write templates that evaluate these limits.
Note, that it is always better to write platform-independent code by using the minimum guaranteed precision of the types.

Others

The algorithm library, header file <algorithm>, includes three **auxiliary functions**, one each for the selection of the **minimum** and **maximum** of two values and one for the **swapping** of two values. These auxiliary functions have been used in various program examples in part of tutorial.
Four template functions define the comparison operators !=, >, <=, and >= by calling the operators == and <.
These functions are defined in <utility>.
Don't forget also other standard C++ header files such as <ios>, <locale>, <valarray>, <new>, <memory>, <complex> etc.

----------Anymore? No more----------
---www.tenouk.com---

Further reading and digging:

1. Check the best selling C / C++ and STL books at Amazon.com.