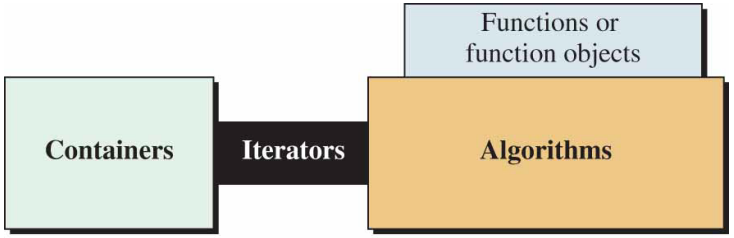
Object-oriented Programming
Standard Template Library (STL) Part 2

YoungWoon Cha

Computer Science and Engineering

Components



Four components of STL

Containers

Containers are used to store and manipulate collections of objects. STL provides various container classes like vector, list, deque, set, map, and more.

Iterators

Iterators are used to traverse and access elements within containers one by one.

Algorithms

Algorithms are operations that we need to apply to the container elements. These algorithms include sorting, searching, manipulating, and performing other operations on the elements within the containers.

Functions and Function Objects (a.k.a. functors)

To apply algorithms on container, the STL provides a set of predefined function objects, such as predicates, comparators, and arithmetic operations. These function objects are used in algorithms to define specific behaviors or criteria.

KU KONKUK UNIVERSITY

Function Objects

USING FUNCTIONS

We can use library algorithms or define our own algorithms.

In each case, an algorithm applies an operation to a number of elements in the container.

The question is how can this operation be defined as a parameter in the algorithm.

It can be done in two ways: a pointer to function or a function object (*functor*). We discuss each approach next.

Using Pointer to Function

```
// Definition of print function
void print (int value)
{
      cout << value << endl;
}
// Definition of fun function
void fun (int x, void(*f)(int))
{
      f(x);
}

int main()
{
      fun(24, print); // Calling function fun
      fun(88, print); // Calling function fun
      return 0;
}
```

Using Function Object

```
class Print
{
      public:
      void operator()(int value) {cout << value;}
};

int main()
{
      Print print; // Instantiation of an object of type Print
      print(45); // calling operator()
      return 0;
}
```

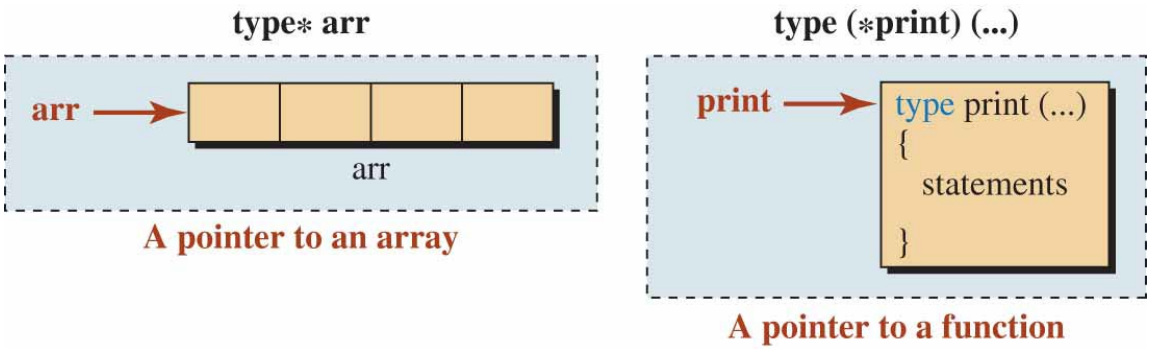
Pointer to Function

We know that the definition of a function is stored in memory. Every entity that is stored in memory has an address.

In fact, the name of a function is a pointer to the first byte of memory where the function is stored just as the name of an array is a pointer to the first element of an array.

The figure shows how the name of an array and the name of a function are pointers.

The figure also shows how a pointer to an array and a pointer to a function are declared.



Using Pointer to Functions

```
#include <iostream>
using namespace std;

// Definition of print function
void print (int value)
{
      cout << value << endl;
}
// Definition of fun function
void fun (int x, void(*f)(int))
{
      f(x);
}

int main()
{
      fun(24, print); // Calling function fun
      fun(88, print); // Calling function fun
      return 0;
}
```

```
Run:
24
88
```

We want to use a function, *fun*, that uses another function, *print*, as an argument.

In *main*, we call *fun* two times, which calls the print function in each call.

Note that the definition of *fun* does not define which function is to be called as the second parameter.

The calling statement passes a pointer to the *print* function (*print* is a pointer to the beginning of the *print* function).

The declaration of a pointer to function, void (*f)(int), may look unusual, but the good news is that most algorithms in STL have already defined functions that use pointer to functions (such as *print*) in the above example.

We do not have to declare them. In other words, if an STL algorithm uses a pointer to function, we only need to give the name of the function in the call.

Using for_each With a User-defined Function

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// Definition of the print function
void print(int value)
{
      cout << value << " ";
}

int main()
{
// Instantiation of a vector object and
storing three values
      vector <int> vec;
      vec.push_back(24);
      vec.push_back(42);
      vec.push_back(73);
// Using a print function to print the value
of each element
      for_each(vec.begin(), vec.end(), print);
      return 0;
}
```

```
Run:
24   42   73
```

We use the STL generic algorithm, *for_each*, that applies a function to a range of items in a container.

The algorithm applies the function defined as the third parameter to the range [first, last).

The algorithm defines that the third algorithm must be a pointer to a function.

We pass the name *print* and then we define a function named *print* with one argument.

The print function takes the value of its argument from the iterator (*iterator).

Function Objects (Functors)

We can overload the *function operator*, a pair of parentheses, to allows us to create a *function object* (sometimes called a *functor*).

The Figure shows how we write a regular function and a function object.

Regular function

```
void sample (...)
{
    statements
}
```

Function object

```
class sample
{
    public :
        void operator () (...)
        {
            statements
        }
};
```

We can see that the definition of a function object is longer than a regular function, but a function object has advantages:

- **As an object, a function object can be used wherever an object can be used. In particular, it can be passed to a function as an argument, and it can be returned from a function.**
- **A function object can have a state, which means that it can hold information from one call to another.**
- **We can define a class to be used as a function object and then inherit from it to create other function objects.**

Example of Using a Functor

```
#include <iostream>
using namespace std;

class Print
{
        public:
        void operator()(int value) {cout << value;}
};

int main()
{
        Print print; // Instantiation of an object of type Print
        print(45); // calling operator()
        return 0;
}
```

```
Run:
45
```

The Program shows how we can call a *functor* from another function (*main*).

We create a class named Print, overload the *operator*(), create an object of the class, and call the operator.

Note that calling the overloaded operator is simply calling the object instantiated from the class and inserting the arguments inside the parentheses.

In fact, we are calling *print.operator*()(45).

Function Objects (Functors) Part 2

Function Objects in STL Algorithms

The STL library defines many function objects in the <functional> header.

They can be divided into unary and binary.

Each function in fact simulates one of the built-in operators.

Function object	type	arity	operator
negate <T>	arithmetic	unary	-
plus <T>	arithmetic	binary	+
minus <T>	arithmetic	binary	-
multiplies <T>	arithmetic	binary	*
divides <T>	arithmetic	binary	/
modulus <T>	arithmetic	binary	%
equal_to <T>	relational	binary	==
not_equal_to <T>	relational	binary	!=
greater <T>	relational	binary	>
greater_equal <T>	relational	binary	>=
less <T>	relational	binary	<
less_equal<T>	relational	binary	<=
logical_not <T>	logical	unary	!
logical_and <T>	logical	binary	&&
logical_or <T>	logical	binary	II

Using Pointer to Function and Functor

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <functional>
using namespace std;

// User-defined print function
void print(int value)
{
        cout << value << " ";
}

int main()
{
        // Creation of a vector with four nodes
        vector <int> vec;
        vec.push_back(24);
        vec.push_back(42);
        vec.push_back(73);
        vec.push_back(92);
        // Printing the node using a pointer to user-defined function
        for_each(vec.begin(), vec.end(), print);
        cout << endl;
        // Negating the values of all nodes and print them again
        transform(vec.begin(), vec.end(), vec.begin(), negate <int>());
        for_each(vec.begin(), vec.end(), print);
        return 0;
}
```

```
Run:
24  42  73 92
-24 -42 -73 -92
```

Let us show how we use the *transform* algorithm to negate all elements in a vector. (We discuss algorithms in the next section.)

Note that we do not see any object of type *negate* in the call to transform algorithm.

The reason is that the *transform* function directly calls the default constructor of the *negate* class (with pair of parentheses).

The algorithm then calls the *operator*() in the background and passes the value returned from dereferencing iterator to the that operator.

However, this happens at the background and hidden from the user.

KU KONKUK UNIVERSITY

Algorithms

ALGORITHMS

Another piece of the STL is generic algorithms. Instead of defining these operations inside each container type, the C++ language defines template global functions that can be applied to any container type that supports the iterators required by the algorithm.

The algorithms are template global functions, but the template type does not define the type of the elements in the container; it defines the type of the iterator the algorithm itself uses.

Non-mutating Algorithms

Non-mutating algorithms, which are defined in the <algorithm> header file, do not change the order of the elements in the container that they are applied to.

```
difference_type  count(InIter first, InIter last, const T& value);
difference_type  count_if(InIter first, InIter last, Predicate pred);
InIter find(InIter first, InIter last, const T& value);
Function for_each(InIter first, outIter last, Function func);
```

The first function counts the number of elements equal to *value*.

The second function counts the element if they meet the criteria in *pred*.

The third function finds the position of an element with a given value.

The fourth function applies the parameter *func* to the member elements in the range [*first*, *last*).

Testing Some Non-mutating Algorithms

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// Definition of isEven
bool isEven(int value)
{
	return (value % 2 == 0);
}
// Definition of timesTwo
void timesTwo(int& value)
{
	value = value * 2;
}
// Definition of print
void print(int value)
{
	cout << value << " ";
}
```

```
Run:
Original values in vector
17  10  13  13  18  15  17  13  13  18

Count of 10's: 1

Count of even values: 3

Values after multiplying by 2
34 20 26 26 36 30 34 26 26 36
```

```
int main()
{
	// Instantiation of a vector of integers
	vector <int> vec ;
	// Pushing ten values into the vector
	vec.push_back(17);
	vec.push_back(10);
	vec.push_back(13);
	vec.push_back(13);
	vec.push_back(18);
	vec.push_back(15);
	vec.push_back(17);
	vec.push_back(13);
	vec.push_back(13);
	vec.push_back(18);
	// Printing original values
	cout << "Original values in vector" << endl;
	for_each(vec.begin(), vec.end(), print);
	cout << endl << endl;
	// Counting number of 10's
	cout << "Count of 10's: ";
	cout << count(vec.begin(), vec.end(), 10);
	cout << endl << endl;
	// Counting the even values
	cout << "Count of even values: ";
	cout << count_if( vec.begin(), vec.end(), isEven );
	cout << endl << endl;
	// Doubling each value and printing vector
	cout << "Values after multiplying by 2" << endl;
	for_each(vec.begin(), vec.end(), timesTwo);
	for_each(vec.begin(), vec.end(), print);
	return 0;
}
```

Mutating Algorithms

The mutating algorithms, which are defined in the <algorithm> header file, change the structure of the container they are applied to.

(*BdIter* means bidirectional iterator and *FwIter* means forward iterator).

```
void generate(BdIter first, BdIter last, gen);
void reverse(BdIter first, BdIter last);
void rotate(FwIter first, FwIter middle, FwIter last);
void random_shuffle(BdIter first, BdIter last);
outIter transform(inIter first, inIter second, outIter start, oper);
```

The first function creates a sequence with the result of running the *gen* function.

The second function reverses the order of elements in the container.

The third function rotates the elements to the left so that the middle element becomes the first element and the element before the middle element becomes the last.

The *random_shuffle* function changes the order in the container to a random order.

The transform function changes the values from the member pointed to by *first* to *second* and puts the result starting from element pointed to by *start*.

Program to Test Mutating Functions

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <iomanip>
using namespace std;

// Definition of print function
void print(int value)
{
        cout << value << " ";
}
```

```
Run:
Original vector
11 14 17 23 35 52

Vector after reversing the order
52 35 23 17 14 11

Vector after rotating the order
23 17 14 11 52 35

Vector after random shuffle
23 14 35 52 17 11
```

```
int main()
{
        // Instantiation of a vector object
        vector <int> vec ;
        // Adding six values
        vec.push_back(11);
        vec.push_back(14);
        vec.push_back(17);
        vec.push_back(23);
        vec.push_back(35);
        vec.push_back(52);
        // Printing original values
        cout << "Original vector" << endl;
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        // Reversing the values and print the vector
        cout << "Vector after reversing the order" << endl;
        reverse(vec.begin(), vec.end());
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        // Rotate the values and print the vector
        cout << "Vector after rotating the order" << endl;
        rotate(vec.begin(), vec.begin() + 2, vec.end());
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        // Random shuffle the value print the vector
        cout << "Vector after random shuffle" << endl;
        random_shuffle(vec.begin(), vec.end());
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        return 0;
}
```

Sorting and Related Algorithms

The <algorithm> header file also defines several algorithms that either sort the sequence or apply operations that are related to sorting.

```
void sort(RndIter first, RndIter last);
bool binary_search(FwIter first, FwIter last, const T& value);
FwIter min_element(FwIter first, FwIter last);
FwIter max_element(FwIter first, FwIter last);
OutIter set_difference(InIter first1, InIter last1, InIter first2, InIter last2, OutIter result);
OutIter set_intersection(InIter first1, InIter last2, InIter first2, InIter last2, OutIter result);
OutIter set_union(InIter first1,  InIter last1, InIter first2, InIter last2, OutIter result);
OutIter set_symmetric_difference(InIter first1,  InIter last1, InIter first2, InIter last2, OutIter result);
```

Using Sorting Algorithms

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

// Definition of print function
void print(int value)
{
        cout << value << " ";
}

int main()
{
        // Instantiation of a vector object
        vector <int> vec ;
        // Pushing six elements into the vector and print them
        vec.push_back(17);
        vec.push_back(10);
        vec.push_back(13);
        vec.push_back(18);
        vec.push_back(15);
        vec.push_back(11);
        cout << "Original vector" << endl;
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        // Sorting the vector in ascending order and print it
        cout << "Vector after sorting in ascending order" << endl;
        sort(vec.begin(), vec.end());
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        // Sorting the vector in descending order and print it
        cout << "Vector after sorting in descending order" << endl;
        sort(vec.begin(), vec.end(), greater <int>());
        for_each(vec.begin(), vec.end(), print);
        cout << endl << endl;
        return 0;
}
```

```
Run:
Original vector
17   10   13   18   15   11

Vector after sorting in ascending order
10   11   13   15   17   18

Vector after sorting in descending order
18   17   15   13   11   10
```

Function Objects defined in STL

Function object	type	arity	operator
negate <T>	arithmetic	unary	-
plus <T>	arithmetic	binary	+
minus <T>	arithmetic	binary	-
multiplies <T>	arithmetic	binary	*
divides <T>	arithmetic	binary	/
modulus <T>	arithmetic	binary	%
equal_to <T>	relational	binary	==
not_equal_to <T>	relational	binary	!=
greater <T>	relational	binary	>
greater_equal <T>	relational	binary	>=
less <T>	relational	binary	<
less_equal<T>	relational	binary	<=
logical_not <T>	logical	unary	!
logical_and <T>	logical	binary	&&
logical_or <T>	logical	binary	||

Using Binary Search Algorithm

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
	// Instantiation of a vector object
	vector <int> vec ;
	// Adding six elements to the vector
	vec.push_back(17);
	vec.push_back(10);
	vec.push_back(13);
	vec.push_back(18);
	vec.push_back(15);
	vec.push_back(11);
	// Sorting the vector
	sort(vec.begin(), vec.end());
	// Searching vector for two values
	cout << "Found 10 in vector? " << boolalpha;
	cout << binary_search(vec.begin(), vec.end(), 10) << endl;
	cout << "Found 19 in vector? " << boolalpha;
	cout << binary_search(vec.begin(), vec.end(), 19) << endl;
	return 0;
}
```

```
Run:
Found 10 in vector? true
Found 19 in vector? false
```

Numeric Algorithms

There are a small number of algorithms, defined in the <numeric> header file, that perform simple arithmetic operations on the elements of a container or containers.

Note that these algorithms do not have to be applied on arithmetic types(such as *int* or *double*); as long as the corresponding operation is defined for a type, these algorithms can be applied to them.

```
T accumulate(InIter first, InIter last, T init);
```

The *accumulate* algorithm finds the sum of the value in a range [*first*, *last*) and adds the result to the *init* value.

Testing Accumulate Algorithm

```
#include <vector>
#include <numeric>
#include <iostream>
using namespace std;

// A print function
void print(int value)
{
	cout << value << " ";
}

int main()
{
	// Instantiate and print a vector
	vector <int> vec ;
	vec.push_back(17);
	vec.push_back(10);
	vec.push_back(13);
	vec.push_back(13);
	vec.push_back(18);
	vec.push_back(15);
	vec.push_back(17);
	for_each(vec.begin(), vec.end(), print);
	cout << endl;
	// Calculate the sum and print it
	int sum = accumulate(vec.begin(), vec.end(), 0);
	cout << "Sum of elements: " << sum;
	return 0;
}
```

```
Run:
vector: 17  10  13  13  18  15  17
Sum of elements: 103
```

Summary

Highlights

- Pointer to Function
- Function Objects
- Non-mutating/Mutating Algorithms
- Sorting Algorithms
- Numeric Algorithms

What's Next? (Reading Assignment)

- Read Chap. 16. Input/Output Streams

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr