Lecture #7

STL Containers

chapter: 3 – 3.4

Objectives

- To know the relationships among containers, iterators, and algorithms.
- To distinguish sequence containers, associative containers, and container adapters.
- To distinguish some containers vector, map, multimap
- To use common features of some containers.
- To access elements in a container using iterators.
- To distinguish iterator types: input, output, forward, bidirectional, and random-access.
- To manipulate iterators using operators.
- To obtain iterators from containers and know the type of interators supported by containers.
- To perform input and output using istream_iterator and ostream_iterator.
- To store, retrieve and process elements in sequence containers: vector.

Three components of STL

Containers: Classes in the STL are container classes. A container object such as a vector is used to store a collection of data, often referred to as *elements*.

Iterators: The STL container classes make extensive use of iterators, which are objects that facilitate traversing through the elements in a container. Iterators are like built-in pointers that provide a convenient way to access and manipulate the elements in a container.

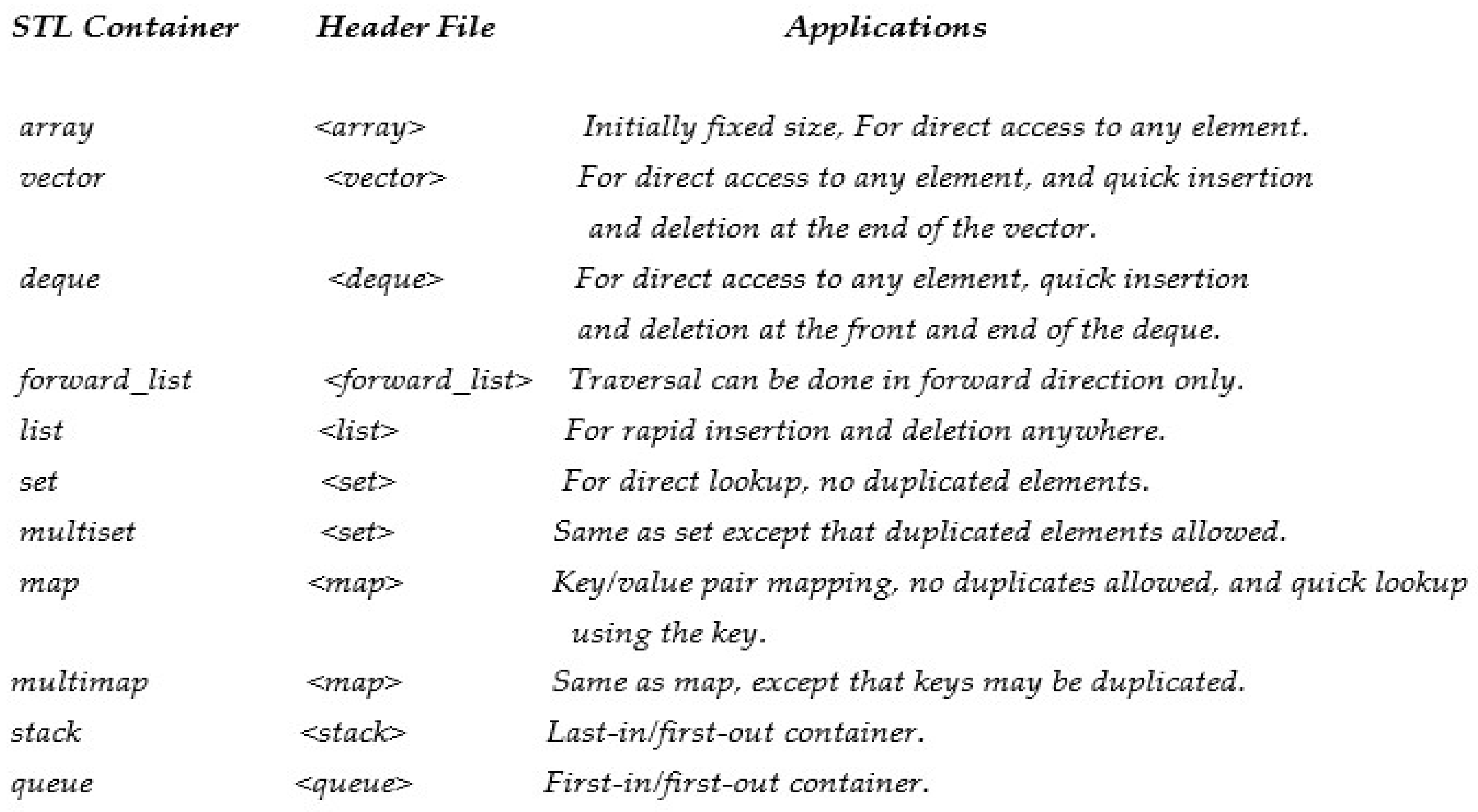
Algorithms: Algorithms are used in the functions to manipulate data such as sorting, searching, and comparing elements. There are about 80 algorithms implemented in the STL. Most of these algorithms use iterators to access the elements in the container.

Sequence, Associative Containers

- The sequence containers (also known as sequential containers) represent linear data structures. The five sequence containers are array, vector, forward_list, list and deque.

- Associative containers are non-linear containers that can locate elements stored in the container quickly. Such containers can store sets of values or *key/value* pairs (pair object). The four ordered associative containers are set, multiset, map, and multimap.

Container Classes



STL Container	Header File	Applications
array	<array>	Initially fixed size, For direct access to any element.
vector	<vector>	For direct access to any element, and quick insertion and deletion at the end of the vector.
deque	<deque>	For direct access to any element, quick insertion and deletion at the front and end of the deque.
forward_list	<forward_list>	Traversal can be done in forward direction only.
list	<list>	For rapid insertion and deletion anywhere.
set	<set>	For direct lookup, no duplicated elements.
multiset	<set>	Same as set except that duplicated elements allowed.
map	<map>	Key/value pair mapping, no duplicates allowed, and quick lookup using the key.
multimap	<map>	Same as map, except that keys may be duplicated.
stack	<stack>	Last-in/first-out container.
queue	<queue>	First-in/first-out container.

Common Functions to All Containers

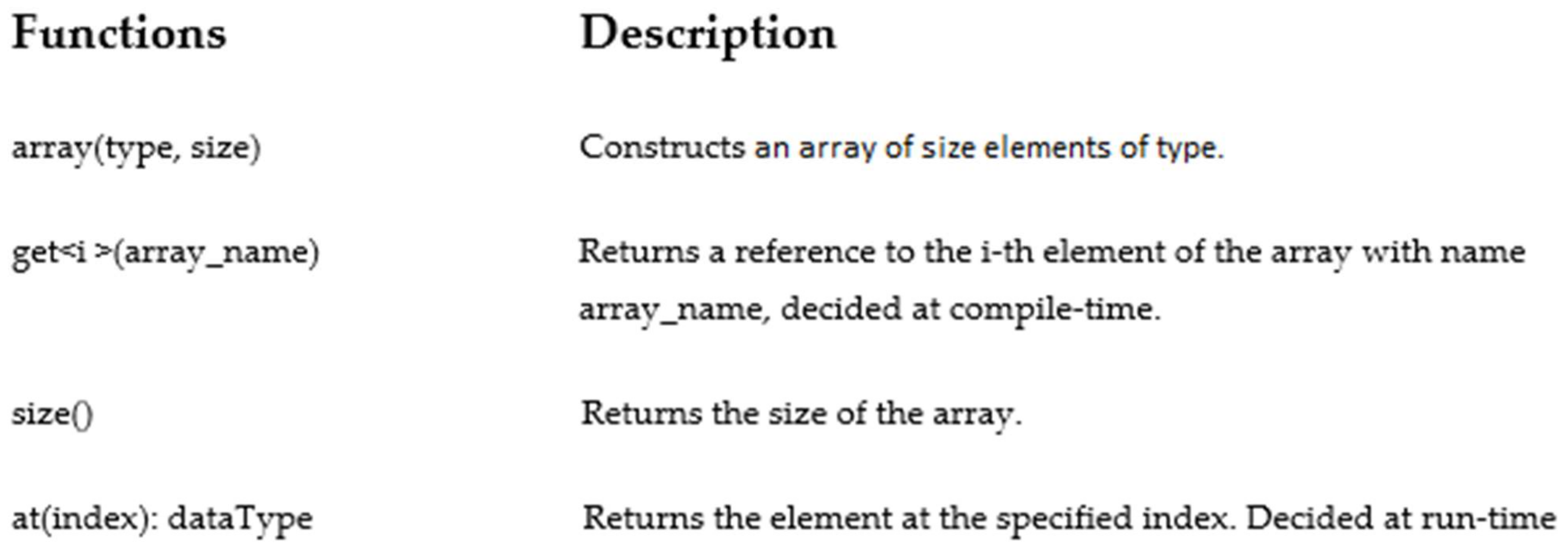
Functions	**Description**
`non-arg constructor`	Constructs an empty container.
`constructor with args`	In addition to the non-arg constructor, every container has several constructors with args.
`copy constructor`	Creates a container by copying the elements from an existing container of the same type.
`destructor`	Performs cleanup after the container is destroyed.
`empty()`	Returns true if there are no elements in the container.
`size()`	Returns the number of elements in the container.
`operator=`	Copies one container to another.
Relational operators (`<`, `<=`, `>`, `>=`, `==`, and `!=`)	The elements in the two containers are compared sequentially to determine the relation.

Common Functions to First-Class Containers

STL Functions	Description
c1.swap(c2)	Swaps the elements of two containers c1 and c2.
c.max_size()	Returns the maximum number of elements a container can hold.
c.clear()	Erases all elements from the container.
c.begin()	Returns an iterator to the first element in the container.
c.end()	Returns an iterator that refers to the next position after the end of the container.
c.rbegin()	Returns an iterator to the last element in the container for processing elements in reverse order.
c.rend()	Returns an iterator that refers to the position before the first element in the container.
c.erase(beg, end)	Erases the elements in the container from beg to end-1. Both beg and end are iterators.

Sequence Containers: array

- The sequence container array is a little bit different than the rest. Its fixed sized and it is created on the stack rather than the heap.
- Some functions defined in <arrays>:



Functions	Description
array(type, size)	Constructs an array of size elements of type.
get<i >(array_name)	Returns a reference to the i-th element of the array with name array_name, decided at compile-time.
size()	Returns the size of the array.
at(index): dataType	Returns the element at the specified index. Decided at run-time

Sequence Containers: array

```
std::array<int, 10> myArray;
std::array<int, 10> myArray2{1, 2, 3, 4, 5, 6};

for(int i= 0; i < 10;i++)
    std::cout << myArray2[i] <<" ";

for(int i= 0; i < 10;i++)
    std::cout << myArray2.at(i) <<" ";


 // is decided at run-time, index must be known
std::cout << get<0>(myArray2) <<" ";

std::cout << myArray2.size() << std::endl;

for(const auto& element:myArray2)
    std::cout << element << " ";
```

Sequence Containers: vector

Functions	**Description**
vector(n, element)	Constructs a vector filled with n copies of the same element.
vector(beg, end)	Constructs a vector initialized with elements from iterator beg to end.
vector(size)	Constructs a vector with the specified size.
at(index): dataType	Returns the element at the specified index.

Common Functions in Sequence Containers

Functions	**Description**
`assign(n, elem)`	Assign n copies of the specified element in the container.
`assign(beg, end)`	Assign the elements in the range from iterator beg to iterator end.
`push_back(elem)`	Appends an element in the container.
`pop_back()`	Removes the last element from the container.
`front()`	Returns the reference of the first element.
`back()`	Returns the reference of the last element.
`insert(position, elem)`	Inserts an element at the specified iterator.

Simple Demo

This simple example demonstrates how to create a vector

```
#include <vector>

vector<int> vector1, vector2;   //declarations
vector<int> vector3(20);   //vector3 has a size of 20

vector<double> vector4(5,3.5 ); //vector4 has a size of 5 elements filled with 3.5

vector<int> vector5(vector4.begin(),vector4.end()); //copy from vector4 to vector5

vector2.push_back(30);

vector2.emplace_back(30);

cout << "size of vector2: " << vector2.size() << endl;

cout << "maximum size of vector1: " << vector1.max_size() << endl;

vector1.swap(vector2);  //Swap the content of vector1 with vector2
```

SimpleSTLDemo

Iterators

Iterators are used extensively in the first-class containers for accessing and manipulating the elements. As you already have seen earlier, several functions (e.g., <u>begin()</u> and <u>end()</u>) in the first-class containers are related to iterators.

Types of C++ iterators:

- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

Iterator Types Supported by Containers

STL Container	Type of Iterators Supported
vector	random access iterators
deque	random access iterators
list	bidirectional iterators
set	bidirectional iterators
multiset	bidirectional iterators
map	bidirectional iterators
multimap	bidirectional iterators
stack	no iterator support
queue	no iterator support
priority_queue	no iterator support

Operators Supported by Iterators

Operator	**Description**
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
Input iterators	
`*p`	Dereference an iterator (used as rvalue).
`p1 == p2`	Evaluates true if p1 and p2 point to the same element.
`p1 != p2`	Evaluates true if p1 and p2 point to different elements.
Output iterators	
`*p`	Dereference an iterator (used as lvalue).
Bidirectionl iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment iterator p by i positions.
`p -= i`	Decrement iterator p by i positions.
`p + i`	Returns an iterator ith position after p.
`p - i`	Returns an iterator ith position before p.
`p1 < p2`	Returns true if p1 is before p2.
`p1 <= p2`	Returns true if p1 is before or equal to p2.
`p1 > p2`	Returns true if p1 is after p2.
`p1 >= p2`	Returns true if p1 is after p2 or equal to p2.
`p[i]`	Returns the element at the position p offset by i.

Iterators

```
vector<int>::iterator It; // declare a random access iterator

for (It = intVector.begin(); It != intVector.end(); It++)
    {
        cout << *It << " "; // output the current value that It is pointing to
    }

auto it = find(intVector.begin(), intVector.end(), 20);

if (it != intVector.end())
        intVector.erase(it);
```

IteratorDemo

Constant Iterator Demo (optional)

```
 vector<int> intVector1;

 intVector1.push_back(10);

 vector<int>::iterator p1 = intVector1.begin();
 vector<int>::const_iterator p2 = intVector1.begin();

 *p1 = 123; // OK
// *p2 = 123; // Not allowed

vector<int>::reverse_iterator p3;

for (p3= intVector1.rbegin(); p3!= intVector1.rend(); p3++)
{
  cout << *p3<< " ";
}
```

ConstIteratorDemo

ReverseIteratorDemo

Printing out the content of a vector

Even though Iterators are used extensively in the first-class containers for accessing and manipulating the elements. We can use other methods to print out the content of a vector.

Review VectorDemo

```
    vector<double> doubleVector(7, 1.0);

    for( auto& element: doubleVector)  //ranged based for
            cout << element << " ";

//printing out using output iterator with std::copy
copy(doubleVector.begin(), doubleVector.end(),ostream_iterator<double>(cout, " "));

//printing out using own print-function and std::for_each

std::for_each(doubleVector.begin(), doubleVector.end(),print);


void print(const int& element) {
    std::cout << element << " ";
}
```

Creating a Matrix with vector

We can use vector in a vector to create a matrix.

```
//Declaring a 2D vector means create the rows first and than create columns for each row

    vector< vector<int> > matrix(5);  //This creates the rows

    // Adding a vector for each row above

    for (int i = 0; i < ssize(matrix); i++)    //ssize() is signed size, C++20
    {
        matrix[i]=vector<int>(5);
        cout << ssize(matrix[i]) <<endl;
    }
```

TwoDimVector

For a better readability as simplicity we can use ranged based for .

Vectors and Matrix as in/out -parameters

```
//prints out the matrix

void printOutMatrix (const vector <vector<int>>& twoDimVector);

//prints out sum of each row in order

int printOutSumOfRows(const vector<int>& aMatrixRow);

//Reads into and returns a 2-D vector

const vector< vector<int> >& readIn(vector<vector<int>>& twoDimVector);
```

TwoDimVector

Reading from file into a vector

```
vector<string> text;     // holds the input
string s;               // holds a line of text from the file

ifstream infile("data.txt");     //try to open the file for reading

if(!infile.is_open())
 {
    cout <<"The file is not open for reading. Going out." <<endl;
    return 0;  //exit(0);
  }

while (getline(infile, s))  // read the input file until eof reached
        text.push_back(s);   // storing each line as an element in text

infile.close();  //we don't need the file anymore
cout << "text.size: " << text.size() << endl;

for (const auto element : text) {
     cout << element << endl;
}
```

Review ReadFileToVector

Thank you