Object-oriented Programming

Standard Template Library (STL) Part 1

YoungWoon Cha

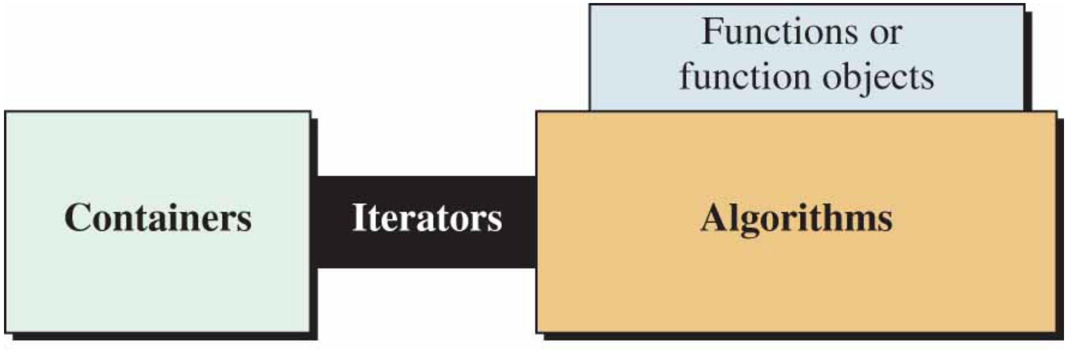
Computer Science and Engineering

Introduction

INTRODUCTION

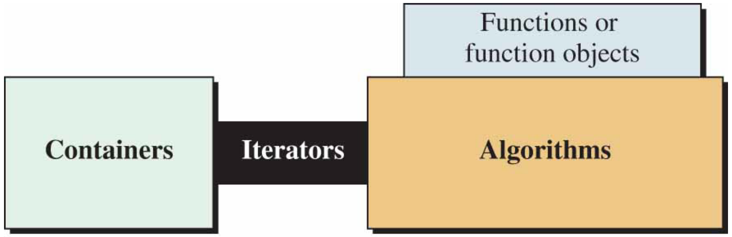
The Standard Template Library (STL) is the result of years of research to solve two important issues:
1. reusability of software
2. separation of functionality

STL is made of four components:



Four components of STL

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.

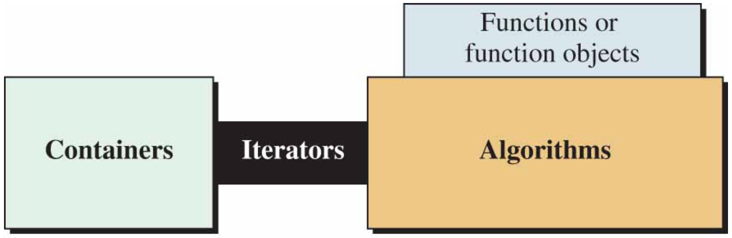
Algorithms

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.

SEQUENCE CONTAINERS



Four components of STL

A sequence container is a collection of objects in which the programmer controls the order of storing and retrieving elements.

The STL provides three sequence containers: *vector*, *deque*, and *list*.

The first two are normally implemented as dynamic arrays; the third is normally implemented as a doubly linked list.

ITERATORS

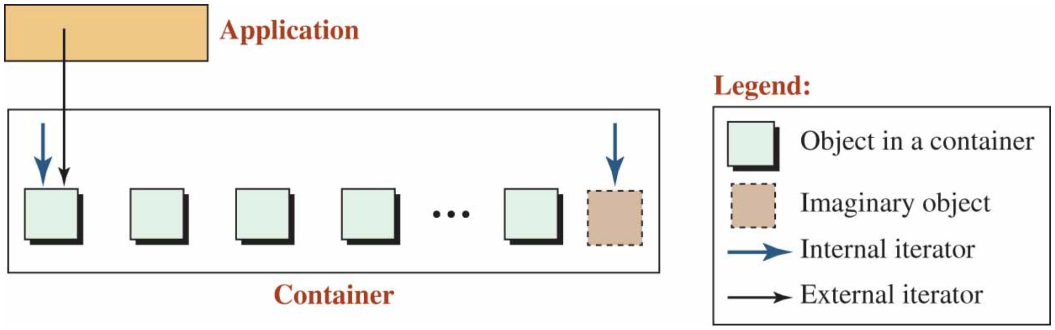
We can declare a *pointer* that points to an element of a container such as an array or a linked list.

We then use the operations for pointers to access the elements of the container one by one.

An *iterator* is an abstraction of a pointer.

It is a class type that has a pointer as a data member with predefined operations that can be applied to the pointer member.

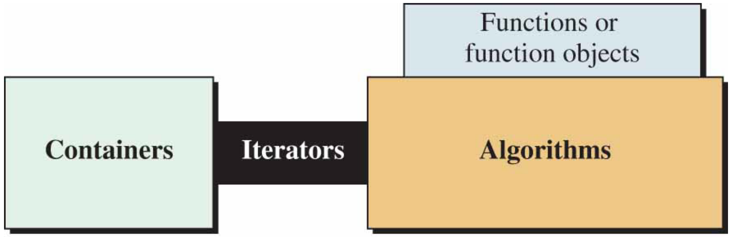
We can instantiate an object of an iterator and then apply the operations defined for it.



One advantage of an iterator is that it can hide the internal structure of a container.

Each container can define its own iterator type whose design is hidden from the user but the user can create an iterator of that type and access the objects in the container.

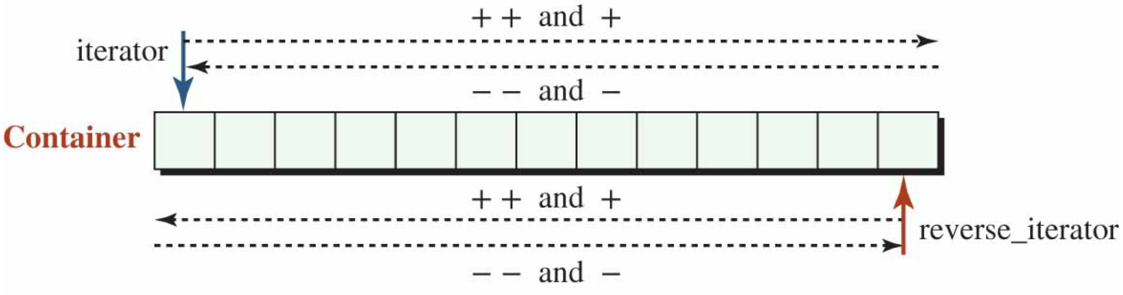
Moving Direction



Four components of STL

A container normally defines two categories of iterators: regular (called *iterator*) and reverse (called *reverse_iterator*).

The moving direction for these two types of iterators need to be understood.



Sequence Containers: vector, deque, list

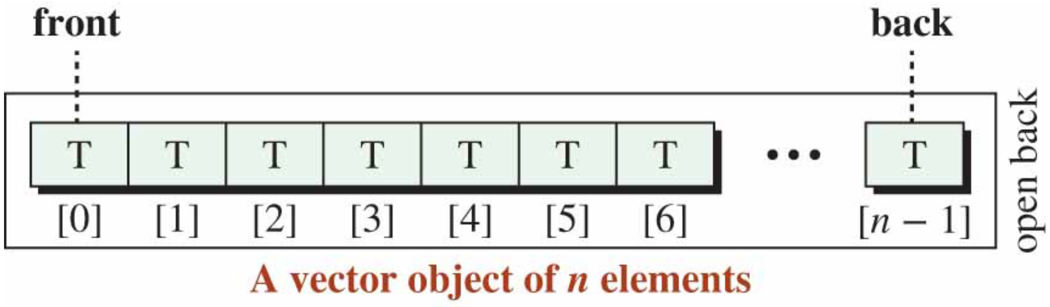
The Vector Class Part 1

The vector class has the same functionality as an array, but with some advantages.

First, it is allocated in the heap and can be resized as needed.

Second, it has a very well-defined iterator mechanism that allows us to access, insert, and erase elements.

We can use a vector anytime we need an array and benefit from its advantages.

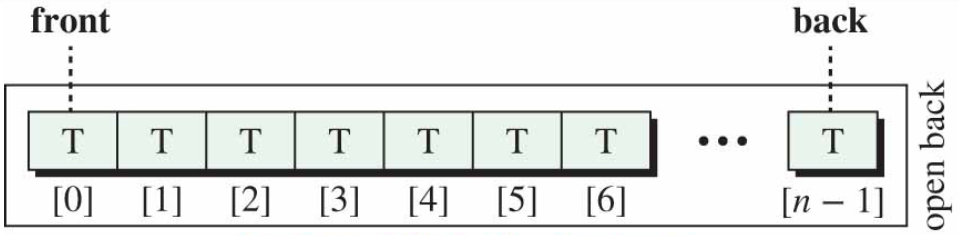


A vector object of *n* elements

The vector <T> class, defined in the <vector> header, implements a *sequence container* that provides fast random access to any element and fast insertion and deletion at the back.

This means that if we need a sequence container with a lot of insertions and deletions at the front or the middle, a vector is not the best choice.

The Vector Class Part 2



A vector object of *n* elements

Constructors and Assignment Operator

```
vector <T> vec;               // Constructs an empty vector
vector <T> vec(4 , value);    // Constructs a vector of 4 elements all the same value
vector <T> vec(from, to);     // A vector copy elements from another structure
vector <T> vec(otherVec);     // Copy constructor
vector <T> vec = otherVec;    // Assignment operator
```

Destructor

The destructor is called automatically when a vector container goes out of scope.

The destructor is designed to delete the whole array from the heap.

Size and Capacity

```
vec.size();             // Returns the current size
vec.max_size();         // Returns the maximum size
vec.resize(n, value);   // Resizes the vector
vec.empty();            // Returns true if vector is empty
vec.capacity();         // Returns potential size
vec.reserve(n);         // Reserves more memory locations
```

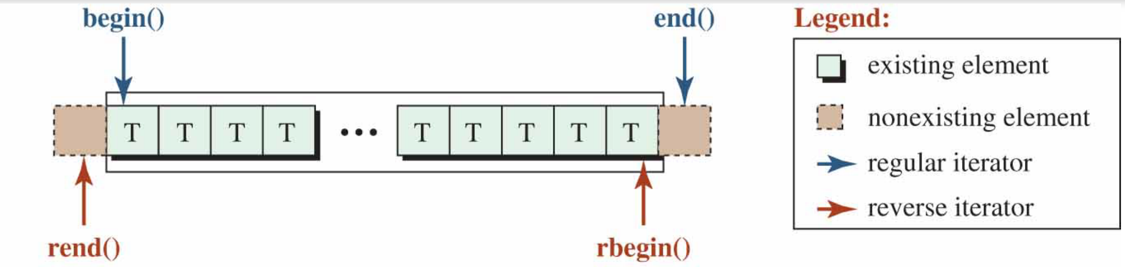
Accessing Elements (for retrieve or change)

```
vec.front();   // Access the first element
vec.back();    // Access the last element
vec [i];       // Access the element at index i
vec.at(i);     // Access the element at index i
```

The Vector Class Part 3

Iterators

The vector class defines two regular iterators and two reverse iterators.



Notes:

The **begin()** function returns a regular iterator that points to the front element.

The **end()** function returns a regular iterator to a nonexisting element after the back element.

The **rbegin()** function returns a reverse iterator that points to the back element.

The **rend()** function returns a reverse iterator to a nonexisting element before the front element.

Both iterators are random-access iterators that can jump forward and backward.

To use iterators in our program, we need first to instantiate them.

Since each container has its own iterators, we need to create one or both of the iterators for that specific class as needed.

```
vector <T> :: iterator iter;           //A regular iterator
vector <T> :: reverse_ iterator riter; //A reverse iterator
```

After instantiation of the user iterators, we need to set them.

The iterator *iter* needs to start where the iterator returned from *begin*() is set; we need to set the iterator *riter* to where the iterator returned from *rbegin*() is set as shown below:

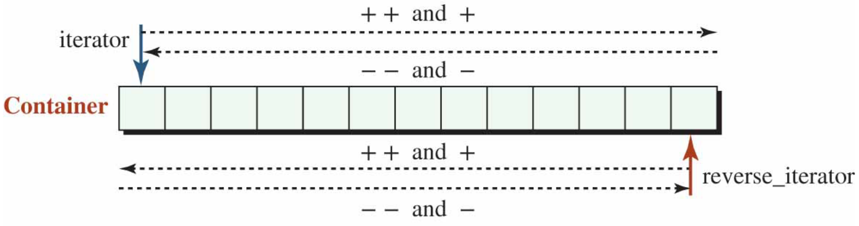
```
iter = vec.begin() ;  // iter starts when the vec.begin() points to
riter = vec.rbegin();  // riter starts when the vec.rbegin() points to
```

Testing Iterators in a Vector

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

int main()
{
      // Constructing a vector of 10 elements and two iterators
      vector <int> vec(10);
      vector <int> :: iterator iter;
      vector <int> :: reverse_iterator rIter;
      // Changing the value of elements
      for (int i = 0; i < 10; i++)
      {
           vec.at(i) = i * i;
      }
      // Printing the elements using the forward iterator
      cout << "Regular navigation: ";
      for (iter = vec.begin() ; iter != vec.end() ; ++iter)
      {
           cout << setw(4) << *iter;
      }
      cout << endl;
      // Printing the elements using reverse iterator
      cout << "Reverse navigation: ";
      for (rIter = vec.rbegin() ; rIter != vec.rend() ; ++rIter)
      {
           cout << setw(4) << *rIter;
      }
      cout << endl;
      return 0;
}
```

```
Run:
Regular navigation:  0   1    4   9  16  25  36  49  64  81
Reverse navigation: 81  64   49  36  25  16   9   4   1   0
```



We are using ++ operators for both the regular iterator and the reverse iterator.

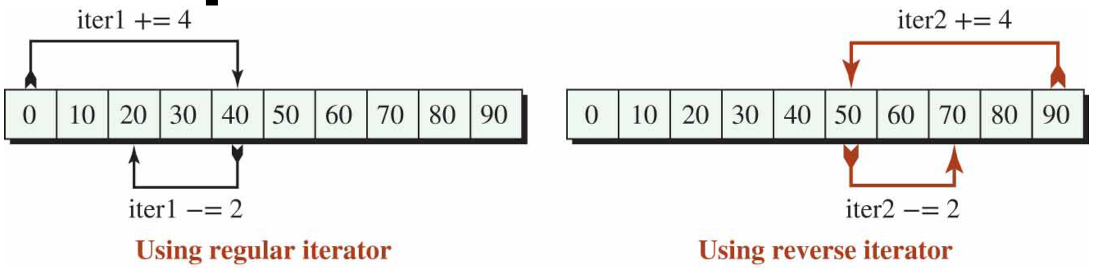
Both iterator are moving forward (from beginning to the end or from end to the beginning).

Showing Random-access Nature of Iterators

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

int main()
{
// Instantiation of a vector and two iterators
      vector <int> vec;
      vector <int> :: iterator iter1;
      vector <int> :: reverse_iterator iter2;
      // Filling the vector with 10 elements
      for (int i = 0; i < 10; i++)
      {
            vec.push_back(i * 10);
      }
// Using the regular iterator to print 40 followed by 20
      cout << "Printing 40 followed by 20" << endl;
      iter1 = vec.begin();
      iter1 += 4;
      cout << *iter1 << " ";
      iter1 -= 2;
      cout << *iter1 << endl;
      // Using the reverse iterator to print 50 followed by 70
      cout << "Printing 50 followed by 70" << endl;
      iter2 = vec.rbegin();
      iter2 += 4;
      cout << *iter2 << " ";
      iter2 -= 2;
      cout << *iter2 << endl;
      return 0;
}
```

```
Run:
Printing 40 followed by 20
40 20
Printing 50 followed by 70
50 70
```



Using regular iterator

Using reverse iterator

The Vector Class Part 4

Insertion

The vector defines several member functions to insert one or more items into the container.

The insertion at the back is very efficient and does not need relocation of the items in the vector.

Insertion in the middle and at the front require reallocation of items in memory.

Note that the last example copies the elements from the first to one before the last, but not the last element. The *first* and *last* parameters are input iterators defining the range [*first*, *last*).

```
vec.push_back(value);           // Insert value at the back
vec.insert(pos, value)          // Insert value before pos
vec.insert(pos, n, value);      // insert n copies before pos
vec.insert(pos, first, last);   // Insert another seqence
```

```
vec.pop_back();              // Erase the back(last) element
vec.erase(pos);              // Erase the element before pos
vec.erase(first, second)     // Erase elements in the range
vec.clear();                 // Erase all elements
```

Erasure

The vector defines several member function for erasure of one or more items in the container.

Erasure at the back is very efficient and does not need relocation of any items.

An erasure from the middle or from the front requires reallocation of memory and should be avoided.

The Vector Class Part 5

We can create a two-dimensional vector, which is a vector of vectors.

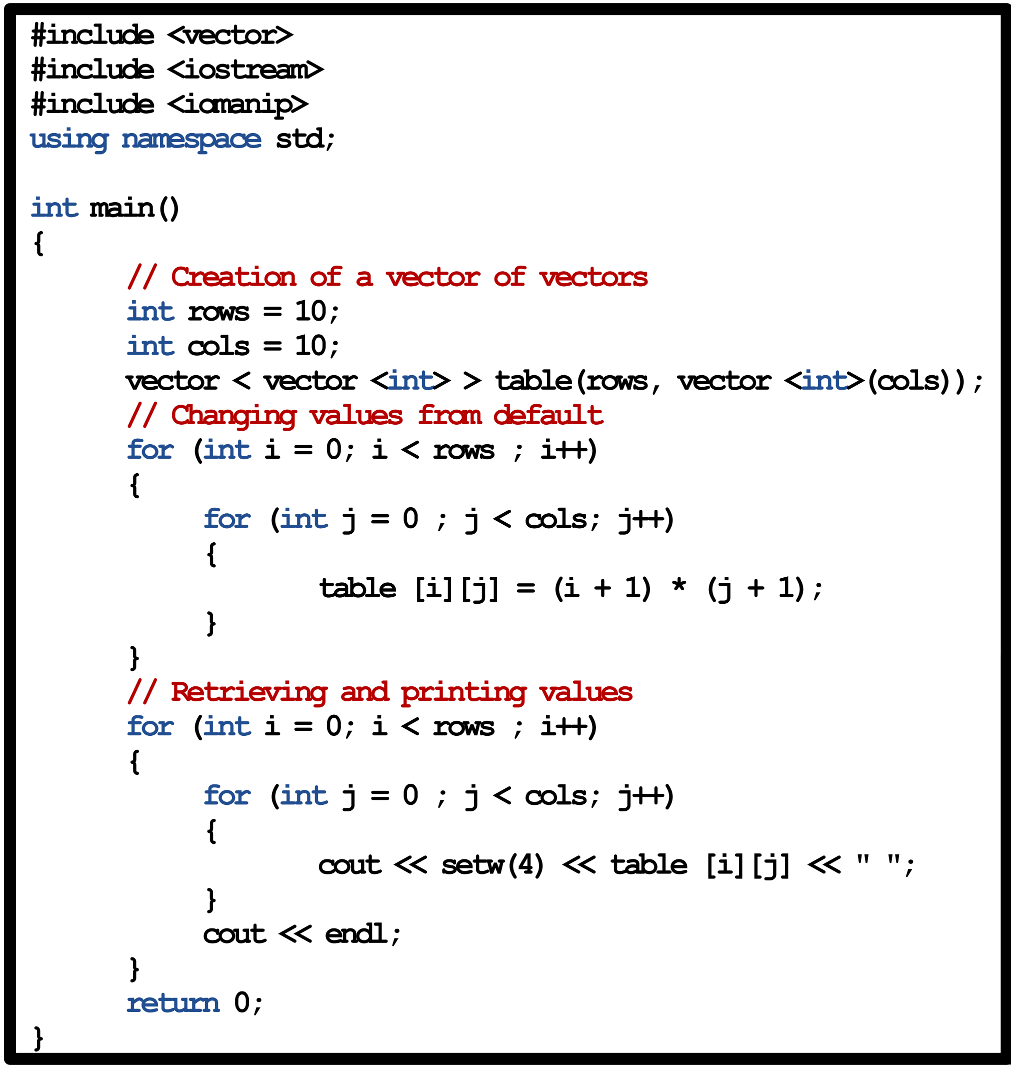
A vector of vectors has the same advantages over a two-dimensional array that a vector has over a one-dimensional array.

```
vector <type> table(rows, value);
vector <vector <int> > table(rows, vector <int>(cols));
```

In the first line, we define a vector named *table* of *rows* elements, each initialized to value.

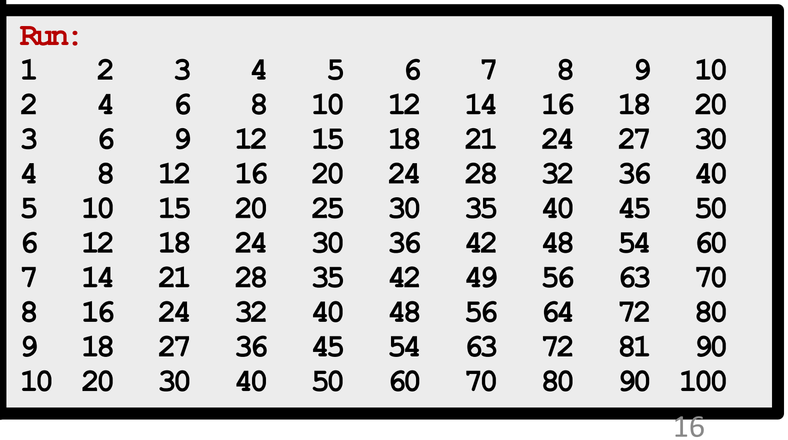
In the second line, we see that the type of the table is in fact a vector of integers. The value of each row is actually a vector of type integer and we have *cols* number of them.

A Vector of Vectors (table)



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

int main()
{
        // Creation of a vector of vectors
        int rows = 10;
        int cols = 10;
        vector < vector <int> > table(rows, vector <int>(cols));
        // Changing values from default
        for (int i = 0; i < rows ; i++)
        {
              for (int j = 0 ; j < cols; j++)
              {
                        table [i][j] = (i + 1) * (j + 1);
              }
        }
        // Retrieving and printing values
        for (int i = 0; i < rows ; i++)
        {
              for (int j = 0 ; j < cols; j++)
              {
                        cout << setw(4) << table [i][j] << " ";
              }
              cout << endl;
        }
        return 0;
}
```



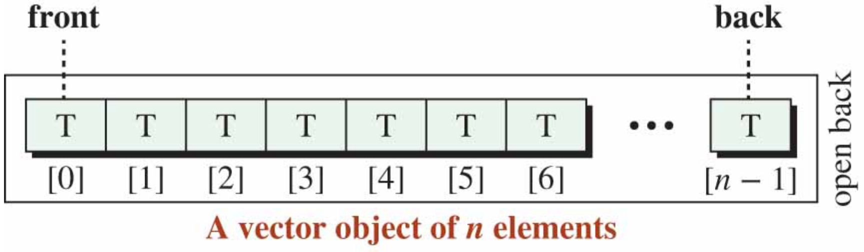
```
Run:
1     2    3    4    5    6    7    8    9   10
2     4    6    8   10   12   14   16   18   20
3     6    9   12   15   18   21   24   27   30
4     8   12   16   20   24   28   32   36   40
5    10   15   20   25   30   35   40   45   50
6    12   18   24   30   36   42   48   54   60
7    14   21   28   35   42   49   56   63   70
8    16   24   32   40   48   56   64   72   80
9    18   27   36   45   54   63   72   81   90
10   20   30   40   50   60   70   80   90  100
```

Sequence Containers: vector, **deque**, list

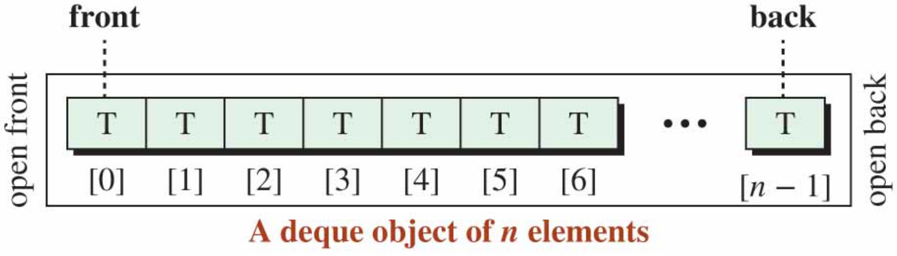
The deque Class Part 1

The *deque* <T> class, defined in the <deque> header file, is a sequence container similar to a *vector* but with two open ends.

The name *deque* is short for *double-ended queue.* This means that we can insert and erase at both ends of a *deque*.



A vector object of *n* elements



A deque object of *n* elements

The operations for inserting and erasing elements from both ends are fast.

However, the extra capability to insert at the beginning makes a *deque* less efficient than a *vector* because it requires that the system allocate extra memory to be able to extend the *deque* at either end.

The *deque* class was designed to be the foundation for the *queue adapter* class that we discuss later.

However, it can be used in any application that needs insertion and deletion at both ends.

One application which normally uses a *deque* is *rotation*, in which we have a list of data items we want to rotate *n* times.

The deque Class Part 2

Operations

The syntax of member functions for the *deque* class is very similar to the ones for the *vector* class with some minor differences. We list only the differences.

Operations added for deque

The *deque* class uses the same set of operations as the *vector*, but uses two additional operations.

```
deq.push_front(value); // Insert value at the front
deq.pop_front();       // Delete value at the front
```

Operations missing in deque

The *capacity*() and *reserve*() member function are missing in the *deque* class.

The reason is the implementation.

Since a *deque* can grow or shrink from both ends, the standard normally implements the *deque* as a set of blocks in heap memory.

Using deque to Rotate a List to Left and Right

```
#include <deque>
#include <string>
#include <iostream>
#include <iomanip>
using namespace std;

// Global print function
void print(deque <string> & deq)
{
        for (int i = 0; i < deq.size (); i++)
        {
              cout << deq.at(i) << " ";
        }
        cout << endl;
}

int main()
{
        // Create a deque of five string and print it
        deque <string> deq(5);
        string arr [5]= {"John", "Mary", "Rich", "Mark", "Tara"};
        for (int i = 0 ; i < 5; i++)
        {
              deq [i] = arr [i];
        }
        print(deq);
        // Rotate the deque clockwise one element
        deq.push_back(deq.front());
        deq.pop_front();
        print(deq);
        // Rotate the deque counter-clockwise one element
        deq.push_front(deq.back());
        deq.pop_back();
        print(deq);
        return 0;
}
```

```
Run:
John    Mary    Rich    Mark    Tara
Mary    Rich    Mark    Tara    John
John    Mary    Rich    Mark    Tara
```

Clockwise rotation requires that we remove an element from the back and insert it at the front; counter-clockwise rotation requires that we remove an element from the front and insert it at the back.

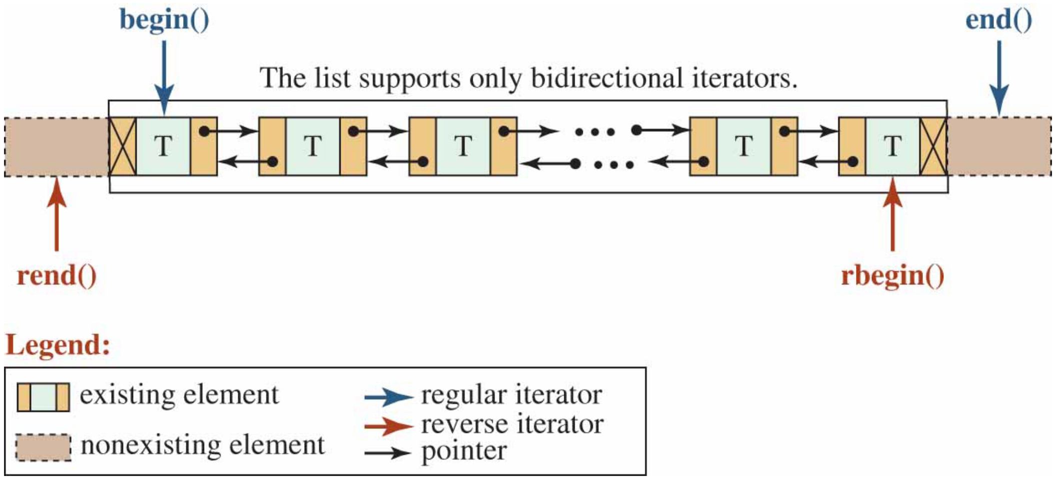
Sequence Containers: vector, deque, list

The list Class Part 1

The *list* <T> class is a sequence container with fast insertion and deletion at any point.

A *list* does not support random access for retrieving or changing the value of an element using the index operator or the *at*() member function because the *list* is implemented as a doubly linked list.

If we want to randomly access the list, we need to use an iterator that moves to the desired element and then accesses it.



Since the *list* class can easily grow or shrink from both ends, it has many applications.

The list Class Part 2

Operations

The syntax of member functions for the *list* class is similar to the ones for the *vecto*r class and *deque* class with some differences. We discuss only the differences.

Iterators

The list class supports only bidirectional iterators (not random-access).

This means that the list elements cannot be accessed using the index operator or the *at*() member function that requires the operators + and - defined for random access iterators.

Capacity and Reserve

The *capacity*() and *reserve*() member functions do not exist in the list class because the system adds an element at the front, at the back, and in the middle using the pointers in the double linked list.

Accessing Elements

The operator [ ] and the *at*() function are not supported for the list class because the list class does not support the random-access operator (only bidirectional).

We need to explicitly use an iterator to access elements.

The list Class Part 3

Erasure

The list class provide three new operations to erase an element or elements from the list as the public interface shows.

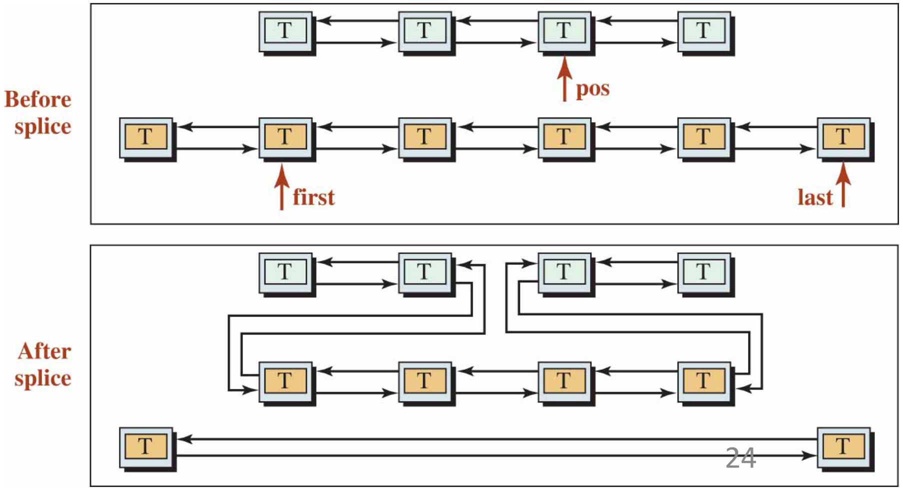
The following shows how we call these operations.

```
remove(value);          // Erases all occurrences of value
remove_if(predicate);   // Erases all occurrences if parameter is true
unique(predicate);      // Erases duplicates if parameter is true
```

Splice, Merge, and Sort

The list class defines three other operations: *splice*(), *merge*(), and *sort*() as defined in the public interface.

```
splice(pos, first, last); // Splice [first, last)
merge(other);             // Merges two sorted lists
sort();                   // Sorts the list.
```



Printing List Elements

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

int main()
{
      // Instantiation of a list object and declaration of a variable
      list <int> lst;
      int value;
      // Inputting five integers and store then in the list
      for (int i = 0; i < 5; i++)
      {
           cout << "Enter an integer: ";
           cin >> value;
           lst.push_back(value);
      }
      // Printing the list in forward direction
      cout << "Print the list in forward direction. " << endl;
      list <int> :: iterator iter1;
      for(iter1 = lst.begin(); iter1 != lst.end(); iter1++)
      {
           cout << *iter1 << " " ;
      }
      cout << endl;
      // Printing the list in backward direction
      cout << "Print the list in reverse direction. " << endl;
      list <int> :: reverse_iterator iter2;
      for (iter2 = lst.rbegin(); iter2 != lst.rend(); iter2++)
      {
           cout << *iter2 << " " ;
      }
      return 0;
}
```

```
Run:
Enter an integer: 25
Enter an integer: 32
Enter an integer: 41
Enter an integer: 72
Enter an integer: 95
Print the list in forward direction.
25    32    41    72    95
Print the list in reverse direction.
95    72    41    32    25
```

Printing Some Elements in a List

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

int main()
{
        // Instantiation of the list container and defining two iterators
        list <int> lst;
        list <int> :: iterator iter1;
        list <int> :: reverse_iterator iter2;
        // Inserting 10 integers into the list
        for (int i = 0; i < 10; i++)
        {
                lst.push_back(i * 10);
        }
        // Moving two steps forward and two steps backward using iter1
        cout << "Printing 40 followed by 20" << endl;
        iter1 = lst.begin();
        iter1++;
        iter1++;
        iter1++;
        iter1++;
        cout << *iter1 << " ";
        iter1--;
        iter1--;
        cout << *iter1 << endl;
        // Moving two steps forward and two steps backward using iter2
        cout << "Printing 50 followed by 70" << endl;
        iter2 = lst.rbegin();
        iter2++;
        iter2++;
        iter2++;
        iter2++;
        cout << *iter2 << " ";
        iter2--;
        iter2--;
        cout << *iter2 << endl;
        return 0;
}
```

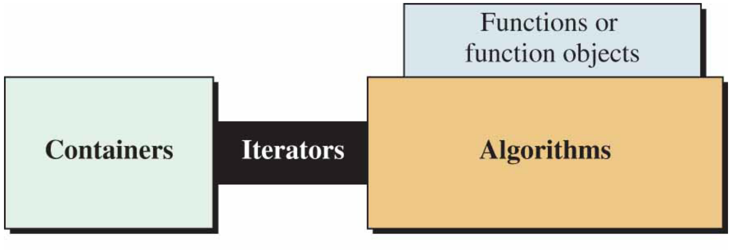
Index:	0	1	2	3	4	5	6	7	8	9
Value:	0	10	20	30	40	50	60	70	80	90

Run:

```
Printing 40 followed by 20
40          20
Printing 50 followed by 70
50          70
```

Container Adapters: stack, queue, priority_queue

CONTAINER ADAPTERS



Four components of STL

The standard library also defines three container adapters that have a smaller interface for easier use.

The container adapters defined in the library are *stack*, *queue*, and *priority_queue*.

A very important point we need to remember is that, unlike containers, we cannot apply the algorithms defined in the library to container adapters because they lack iterators; they do not provide the member functions, such as *begin* and *end*, to create iterators.

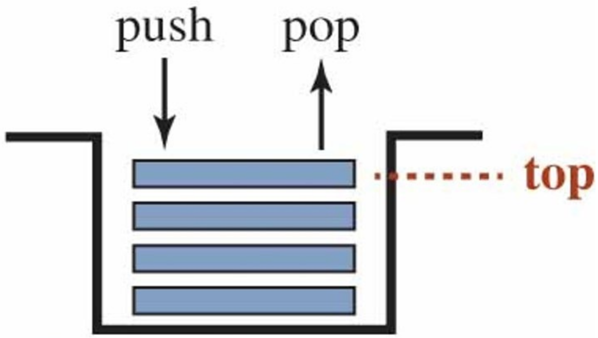
Container adapters cannot be used with algorithms because of no supports for iterators.

The stack Class

The *stack* class is a container adapter class that is designed for three simple operations: *push*, *pop* and *top*.

The stack class is designed for insertion to and erasure from one end (the top).

It is also referred to as a *last-in*, *first-out* or LIFO structure because the last item pushed into the stack is the first item popped from the stack.



Stack of data items

Operations

The stack interface creates an empty constructor. We can check the size and emptiness of a stack.

The only member function that we can use to access the elements in the stack is the *top*() member function.

The *push*() and *pop*() operations are used to insert into and erase an element from the stack respectively.

Convert a Decimal to Hexadecimal

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

int main()
{
	// Instantiation of a stack
	stack <char> stk;
	// Creation of two strings and a declaration of a variable
	string converter("0123456789ABCDEF");
	string hexadecimal;
	int decimal;
	// Inputting a decimal number
	do
	{
		cout << "Enter a positive integer: ";
		cin >> decimal;
	} while (decimal <= 0);
	// Creation of hexadecimal characters and push them into stack
	while (decimal != 0)
	{
		stk.push(converter [decimal % 16]);
		decimal = decimal / 16;
	}
	// Popping characters from stack and pushing into hex string
	while (!stk.empty())
	{
		hexadecimal.push_back(stk.top(());
		stk.pop();
	}
	cout << "The hexadecimal number : " << hexadecimal;
	return 0;
}
```

```
Run:
Enter a positive integer: 182
The hexadecimal number : B6
```

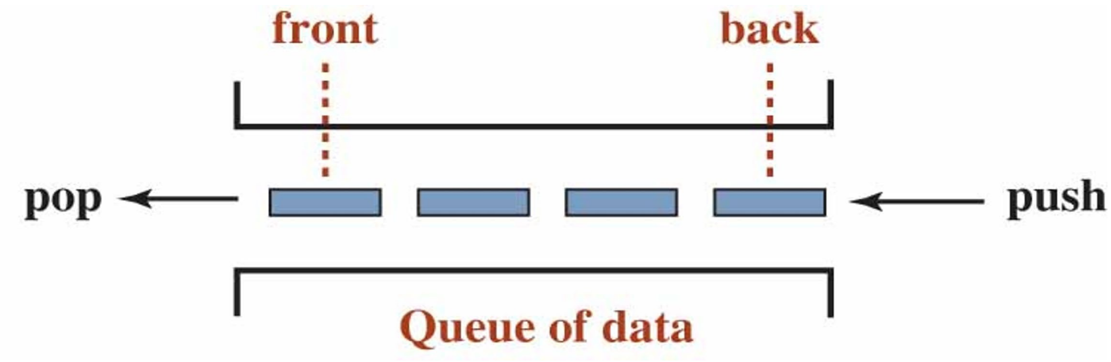
```
Run:
Enter a positive integer: 1234
The hexadecimal number : 4D2
```

```
Run:
Enter a positive integer: 23
The hexadecimal number : 17
```

The queue Class

The *queue* class, which is defined in the <queue> header file, is a container adapter class that is designed for three simple operations: insertion at one end, erasure from the other end, and accessing at both ends.

It is also referred to as *first-in*, *first-out* or FIFO structure because the first item pushed into the queue is the first item to be popped from the queue.



Operations

The queue interface allows us to create an empty queue. We can check the size and emptiness of a queue.

The only two member functions that we can use to access the elements in the queue are the *front*() and *back*() member functions each in two versions (constant and non-constant).

The *push*() and *pop*() operations are used to insert in and erase an element from the queue respectively.

A Program to Group Charity Donations

```
#include <queue>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// Definition of the print function
void print(queue <int> queue)
{
        while (!queue.empty())
        {
                cout << queue.front() << " ";
                queue.pop();
        }
        cout << endl;
}
```

```
int main()
{
        // Instantiation of five queue objects and two variables
        queue <int> queue1, queue2, queue3, queue4, queue5;
        int num;
        int donation;
        // Random creation of donation values and push them in queues
        srand(time(0));
        for (int i = 0; i < 50; i++)
        {
                num = rand();
                donation = num % (50 - 0 + 0) + 0;
                switch (donation / 10)
                {
                        case 0: queue1.push(donation);
                                break;
                        case 1: queue2.push(donation);
                                break;
                        case 2: queue3.push(donation);
                                break;
                        case 3: queue4.push(donation);
                                break;
                        case 4: queue5.push(donation);
                                break;
                }
        }
        // Printing the donations in each group
        cout << "Donations between 00 and 09: ";
        print(queue1);
        cout << "Donations between 10 and 19: ";
        print(queue2);
        cout << "Donations between 20 and 29: ";
        print(queue3);
        cout << "Donations between 30 and 39: ";
        print(queue4);
        cout << "Donations between 40 and 49: ";
        print(queue5);
        return 0;
}
```

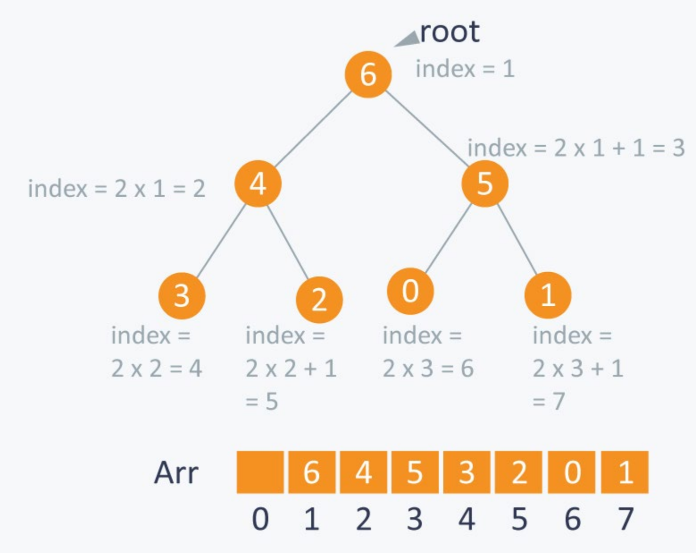
```
Run:
Donations between 00 and 09: 3 1 3 7 9 1 8 8 9 7 4 3
Donations between 10 and 19: 18 14 13 15 16 15
Donations between 20 and 29: 23 26 28 29 23 21 26 21 22 27 28 23
Donations between 30 and 39: 32 31 39 39 39 34 30 38 30 36 32
Donations between 40 and 49: 41 49 49 43 47 43 45 46 41
```

The priority_queue Class

The *priority_queue* class defined in the <queue> header file is a container adapter in which each element has a priority level.

The elements can be inserted into the *priority_queue* in any order; the elements are retrieved based on their priority.

In other words, the front element is the element with the largest priority in the container.



Using Binary (Max/Min) Heap, Binary Search Tree

Operations

Like a *queue*, we *push* at the *back* and *pop* in the *front*, but the access is limited only to the *front*, which is called *top*(), like a stack; we cannot access elements at the *back*.

An implementation difference that we can see between a *queue* and a *priority_queue* is that the *priority_queue* does not support relational operators.

In other words, two priority queues cannot be compared.

Using a Priority Queue

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

int main()
{
        // Create a priority_queue object
        priority_queue <int> line;
        // Push some elements
        line.push(4);
        line.push(7);
        line.push(2);
        line.push(6);
        line.push(7);
        line.push(8);
        line.push(2);
        // Print the elements according to their priorities
        while (!line.empty ())
        {
           cout << line.top() << " ";
           line.pop();
        }
        return 0;
}
```

```
Run:
8  7  7  6  4  2  2
```

Associative Containers: set, map

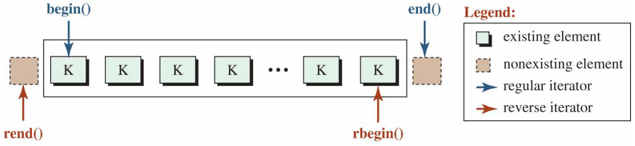
ASSOCIATIVE CONTAINERS

Elements in an *associative container* are stored and retrieved by a *key*.

To access an element in an associate container, we need to use the *key* of the element.

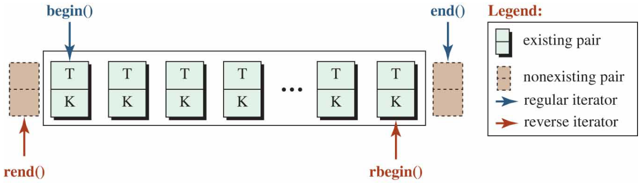
Associate containers are divided into two classes (*set* and *map*).

- ❑ A *set* is an associative container in which the *key* and *value* are the same data item.



Notes:
The keys are unique in a set.
The objects are in a nonlinear relationship with each other.
The *set* class uses bidirectional iterators.

- ❑ A *map* is an associative container in which the *key* and the *value* are separate data items.



Notes:
The keys are unique in a map.
The objects are in nonlinear relationship to each other.
The *map* class uses bidirectional iterators.

To discuss the class *set* and *map*, we need to introduce a library *struct* named *pair* defined in the <utility> header.

A *pair* defines a template *struct* with two template data members possibly of different types.

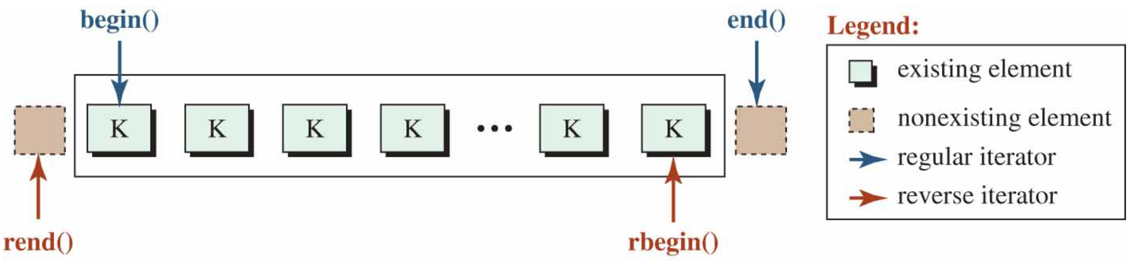
The elements are called *first* and *second* respectively.

```
template <typename T1, typename T2>
struct pair
{
     T1 first;
     T2 second;
};
```

Set Part 1

In a *set*, each element in the container stores one template value which is referred to as the *key*.

The elements are sorted in ascending order and duplicates are not allowed.



Although the objects appear to be one after another to the user, they are not linearly connected.

Notes:
The keys are unique in a set.
The objects are in a nonlinear relationship with each other.
The *set* class uses bidirectional iterators.

```
set <type> set1;             // Create an empty set
set <type>(pos1, pos2) set2; // Create a set from part of another set
set <type>(set2) set3;       // Create a set from another set
set4 = set3;                 // Assignment
```

Constructors, Destructor, and Assignment

We can create a new set using a default constructor (empty set).

We can also create a new set using a parameter constructor by copying the elements of another set in which the range is indicated by [first, last).

We can also create a set by using a copy constructor, or an assignment operator.

Set Part 2

Controlling Size

There are three member functions that we can be used to check size, maximum size, and emptiness. They are the same as discussed for sequence containers.

Iterators

The set class uses bidirectional (not random-access) iterators because it is normally implemented as a non-linear linked-list.

It provides the same eight internal iterators as the sequence containers in which four are constant and four non-constant.

```
set1.begin()   // A regular iterator to the first element
set1.end()     // A regular iterator to the element after the last
set1.rbegin()  // A reverse iterator to the last element
set1.rend()    // A reverse iterator to the element before the last
```

Searching

Since the elements in a set are sorted, searching is possible and efficient. There are four members for searching.

```
set1.count(k)          // Returns number of elements equal to k
set1.find(k)           // Returns an iterator pointing to the first k found
set1.lower_bound(k)    // Returns the first position where k can be inserted
set1.upper_bound(k)    // Returns the last position where k can be inserted
set1.equal_range(k)    // Combination of lower and upper bound
```

Set Part 3

Insertion

There are no push members to insert an element in a set.

Insertion must be done using the key or through iterators.

```
set1.insert(k)          // Returns a regular iterator to first element
set1.insert(hint, k)    // Returns a regular iterator after the last
set1.insert(pos1, pos2) // Returns an iterator to the last element
```

Erasure

There are no pop members to delete an element from a set.

Deletion must be done through the value or iterator.

```
set1.erase(k)           // Erases k and returns a pair <pos, bool>
set1.erase(pos)         // Erases an element pointed to by pos
set1.erase(first, last) // Erases elements in the range [first, last)
set1.clear()            // Returns an iterator after the last
```

Other Operations

Just like the sequence containers, the *swap*() operation is defined for sets.

Also we can compare two sets with relational operators.

Program to Handle a Set of Integers

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

int main()
{
        // Create an empty set of integers
        set <int> st;
        // Insert some keys in the set (with duplicates)
        st.insert(47);
        st.insert(18);
        st.insert(12);
        st.insert(24);
        st.insert(52);
        st.insert(20);
        st.insert(24);
        st.insert(92);
        st.insert(53);
        st.insert(77);
        st.insert(98);
        st.insert(87);
        // Print the set elements from smallest to largest
        cout << "Printing set elements from smallest to
        largest." << endl;
        set <int> :: iterator iter;
        for (iter = st.begin(); iter != st.end(); iter++)
        {
                cout << setw(4) << *iter;
        }
        cout << endl << endl;
```

```
        // Print the set elements from largest to smallest
        cout << "Printing set elements from largest to smallest."
        << endl;;
        set <int> :: reverse_iterator riter;
        for (riter = st.rbegin(); riter != st.rend(); riter++)
        {
                cout << setw(4) << *riter;
        }
        cout << endl << endl;
        // Print the element after 52
        set <int> :: iterator iter1 = st.find(52);
        iter1++;
        cout << "Element after 52: " << *iter1 << endl;
        // Print the element before 20
        set <int> :: iterator iter2 = st.find (20);

        iter2--;
        cout << "Element before 20: " << *iter2 << endl;
        return 0;
}
```

Run:

```
Printing set elements from smallest to largest.
  12  18  20  24  47  52  53  77  87  92  98

Printing set elements from largest to smallest.
  98  92  87  77  53  52  47  24  20  18  12

Element after 52: 53
Element before 20: 18
```

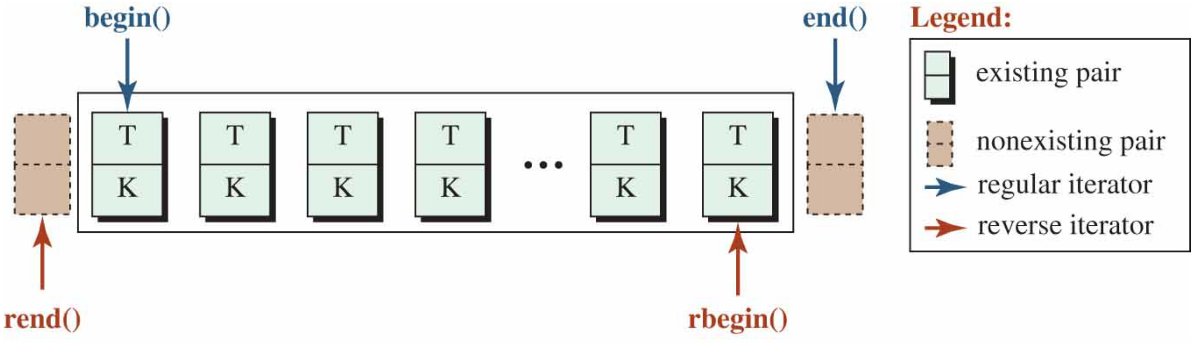
Map Part 1

A *map*, which is also called a *table*, a *dictionary*, or an *associate* array, is defined in the <map> header file.

It is a container that stores a template *pair* of *key* and *value*.

The elements are sorted in ascending order based on the key.

In a map, the keys are unique.



Notes:
The keys are unique in a map.
The objects are in nonlinear relationship to each other.
The *map* class uses bidirectional iterators.

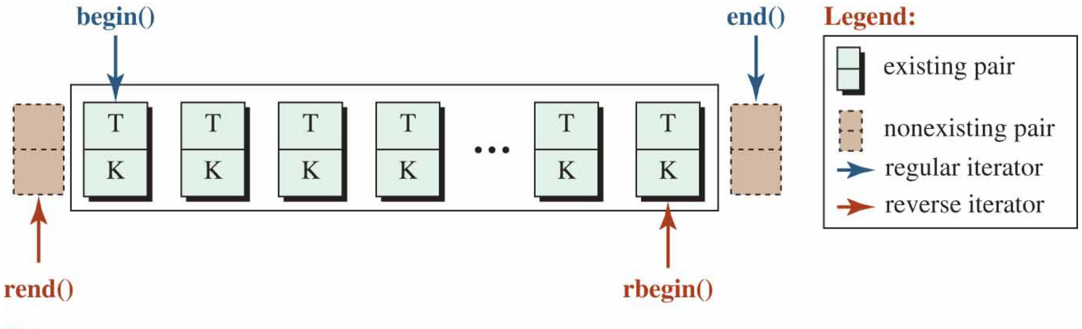
Map Part 2

Operations

The operations defined for the map class are very similar to the ones for the set class.

The main difference is the ability to access the elements in a map using the operator [ ].

This operator makes a map object look like an array in which the index is a key value instead of an integer.



Notes:
The keys are unique in a map.
The objects are in nonlinear relationship to each other.
The *map* class uses bidirectional iterators.

In other words, if we know the value of the key in an element, we can access the element using the expression map [key].

However, we need to be aware that the operator [ ] does not act like the one used for an array or a vector or a deque.

It is just a notation to create a pair of key-value. A map uses only bidirectional iterators and cannot jump from one pair to another using the + or - operator.

Student Table (name and score)

```
#include <map>
#include <iostream>
#include <iomanip>
#include <utility>
using namespace std;

int main()
{
// Creation of a map and the corresponding iterator
      map <string, int > scores;
      map <string, int > :: iterator iter;
// Inputting student name and score in the map
      scores ["John"] = 52;
      scores ["George"] = 71;
      scores ["Mary"] = 88;
      scores ["Lucie"] = 98;
      scores ["Robert"] = 77;
// Printing the names and score sorted on names
      cout << "Students names and scores" << endl;
      for (iter = scores.begin (); iter != scores.end(); iter++)
      {
           cout << setw(10) << left << iter -> first << " ";
           cout << setw(4) << iter -> second << endl;
      }
      return 0;
}
```

```
Run:
Students names and scores

George     71
John       52
Lucie      98
Mary       88
Robert     77
```

Frequency of Words in Text

```
#include <map>
#include <string>
#include <iomanip>
#include <iostream>
using namespace std;

int main()
{
      // Declaration of map, iterator, and a string
      map <string, int > freq;
      map <string, int > :: iterator iter;
      string word;
      // Reading and storing words in the map
      cout << "Enter a sentence to be parsed: " << endl;
      while (cin >> word)
      {
            ++freq [word];
      }
      // Printing the words and their frequency
      for (iter = freq.begin(); iter != freq.end(); iter++)
      {
            cout << left << setw(10) << iter -> first << iter -> second << endl;
      }
      return 0;
}
```

```
Run:
Enter a sentence to be parsed:
we are in the world of this and that and this and that
^Z
and       3
are       1
in        1
of        1
that      2
the       1
this      2
we        1
world     1
```

Summary

Highlights

- ❑ Iterators
- ❑ Sequence Containers: vector, deque, list
- ❑ Container Adapters: stack, queue, priority_queue
- ❑ Associative Containers: set, map

What's Next? (Reading Assignment)

- ❑ Read Chap. 19. Standard Template Library (STL)

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr