Unit 6 Exception Handling and Templates

Introduction to STL, STL Components, Containers- Sequence container and associative containers, container adapters, Application of Container classes: vector, list,

Algorithms- basic searching and sorting algorithms, min-max algorithm, set operations, heap sort,

Iterators- input, output, forward, bidirectional and random access. Object Oriented Programming – a road map to future.

6.1 Introduction to STL & STL Components

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

Sr.No	Component & Description
1	Containers Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
2	Algorithms Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
3	Iterators Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Let us take the following program that demonstrates the vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows –

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

int main() {

   // create a vector to store int
```

```
   vector<int> vec;
   int i;

   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++) {
      vec.push_back(i);
   }

   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++) {
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
      v++;
   }

   return 0;
}
```

Output

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

Here are following points to be noted related to various functions we used in the above example −

1. The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.
2. The size( ) function displays the size of the vector.
3. The function begin( ) returns an iterator to the start of the vector.
4. The function end( ) returns an iterator to the end of the vector.

6.2 Containers- Sequence container and associative containers

A container is an object that stores a collection of objects of a specific type. For example, if we need to store a list of names, we can use a vector.

C++ STL provides different types of containers based on our requirements

Types of STL Container in C++

In C++, there are generally 2 kinds of STL containers:

1. Sequential Containers
2. Associative Containers

1. Sequential Containers in C++

In C++, sequential containers allow us to store elements that can be accessed in sequential order.

Internally, sequential containers are implemented as arrays or linked lists data structures.

Types of Sequential Containers

1. Array
2. Vector
3. Deque
4. List
5. Forward List

Example

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

int main() {

  // initialize a vector of int type
  vector<int> numbers = {1, 100, 10, 70, 100};

  // print the vector
  cout << "Numbers are: ";
  for(auto &num: numbers) {
    cout << num << ", ";
  }

  return 0;
}
```

Output

```
Numbers are: 1, 100, 10, 70, 100,
```

In the above example, we have created sequential container `numbers` using the `vector` class.

```cpp
vector<int> numbers = {1, 100, 10, 70, 100};
```

Here, we have used a ranged for loop to print each element of the container.

In the output, we can see the numbers are shown in sequential order as they were initialized.

```
// output numbers
1, 100, 10, 70, 100,
```

2. Associative Containers in C++

In C++, associative containers allow us to store elements in sorted order. The order doesn't depend upon when the element is inserted.

Internally, they are implemented as binary tree data structures.

Types of Associative Containers

1. Set
2. Map

3. Multiset
4. Multimap

Example

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

int main() {

  // initialize a set of int type
  set<int> numbers = {1, 100, 10, 70, 100};

  // print the set
  cout << "Numbers are: ";
  for(auto &num: numbers) {
    cout << num << ", ";
  }

  return 0;
}
```

Output

```
Numbers are: 1, 10, 70, 100,
```

In the above example, we have created an associative container using the set class.

We have initialized a set named numbers with unordered integers, along with a duplicate value 100:

```
set<int> numbers = {1, 100, 10, 70, 100};
```

Then we print the content of the set using a ranged for loop.

In the output, we see that the numbers are sorted in ascending order with duplicate numbers removed. Initially, 100 was repeated twice but the set removes the duplicate number 100.

```
// output numbers
1, 10, 70, 100
```

6.3 Container Adapters in C++

In C++, Container Adapters take an existing STL container and provide a restricted interface to make them behave differently. For example,

A stack is a container adapter that uses the sequential container deque and provides a restricted interface to support push() and pop() operations only.

Types of Container Adapters

1. Stack
2. Queue
3. Priority Queue

Example

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

int main() {

  // create a stack of ints
  stack<int> numbers;

  // push into stack
  numbers.push(1);
  numbers.push(100);
  numbers.push(10);

  cout << "Numbers are: ";

  // we can access the element by getting the top and popping
  // until the stack is empty
  while(!numbers.empty()) {
    // print top element
    cout << numbers.top() << ", ";

    // pop top element from stack
    numbers.pop();
  }
```

```
  return 0;
}
```

Output

```
Numbers are: 10, 100, 1,
```

In the above example, we have created a container adaptor using the `stack` class.

```
stack <int> numbers;
```

Unlike other containers, we are not directly initializing values to this container. This is because the container adaptor restricts assigning values directly to the container.

That's why we have used the push() operation to insert elements to the stack.

Similarly, we are not allowed to directly iterate through this container. So, we have used a loop that iterates until the stack is empty.

```
while (!numbers.empty()) {
  cout << numbers.top() << ", ";
  numbers.pop();
}
```

Here,

- numbers.empty() - checks if the stack is empty
- numbers.top() - returns the element at the top of the stack
- numbers.pop() - removes the top element of the stack

6.4 Application of Container classes: vector, list

Vector

In C++, vectors are used to store elements of similar data types. However, unlike arrays, the size of a vector can grow dynamically.

That is, we can change the size of the vector during the execution of a program as per our requirements.

Vectors are part of the C++ Standard Template Library. To use vectors, we need to include the vector header file in our program.

The vector class provides various methods to perform different operations on vectors. We will look at some commonly used vector operations in this tutorial:

1. Add elements
2. Access elements
3. Change elements
4. Remove elements

Example

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

int main() {
  vector<int> num {1, 2, 3, 4, 5};

  cout << "Initial Vector: ";

  for (const int& i : num) {
    cout << i << "  ";
  }

  // change elements at indexes 1 and 4
  num.at(1) = 9;
  num.at(4) = 7;

  cout << "\nUpdated Vector: ";

  for (const int& i : num) {
    cout << i << "  ";
  }

  return 0;
}
```

Output

```
Initial Vector: 1  2  3  4  5
```

```
Updated Vector: 1  9  3  4  7
```

List

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about a doubly linked list. For implementing a singly linked list, we use a forward list. Below is the program to show the working of some functions of List

Example

```
// CPP program to show the implementation of List
#include <iostream>
#include <iterator>
#include <list>
using namespace std;

// function for printing the elements in a list
void showlist(list<int> g)
{
    list<int>::iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

// Driver Code
int main()
{

    list<int> gqlist1, gqlist2;

    for (int i = 0; i < 10; ++i) {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);

    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);

    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();

    cout << "\ngqlist1.pop_front() : ";
```

```
    gqlist1.pop_front();
    showlist(gqlist1);

    cout << "\ngqlist2.pop_back() : ";
    gqlist2.pop_back();
    showlist(gqlist2);

    cout << "\ngqlist1.reverse() : ";
    gqlist1.reverse();
    showlist(gqlist1);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);

    return 0;
}
```

Output

```
List 1 (gqlist1) is :     0    2    4    6    8    10    12    14    16    18

List 2 (gqlist2) is :     27    24    21    18    15    12    9    6    3    0

gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() :     2    4    6    8    10    12    14    16    18

gqlist2.pop_back() :     27    24    21    18    15    12    9    6    3

gqlist1.reverse() :     18    16    14    12    10    8    6    4    2

gqlist2.sort():     3    6    9    12    15    18    21    24    27
```

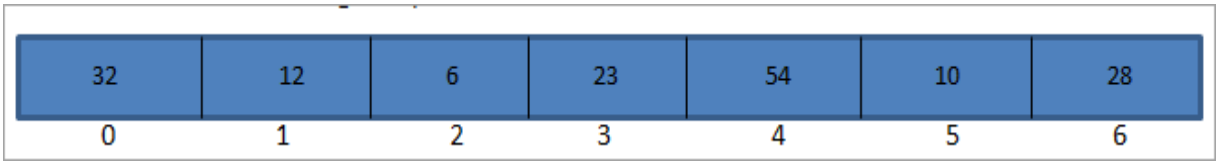
6.5 Algorithms- basic searching

We have two main searching algorithms that are mostly employed to search for information. These include:

1. Linear Search
2. Binary Search

Linear Search

This is the most basic searching technique and is easier to implement too. In a linear search, the key to be searched is compared linearly with every element of the data collection. This technique works effectively on linear data structures. Let us consider the following array.



Above is the array of seven elements. If we want to search key = 23, then starting from the 0th element, the key value will be compared to each element. Once the key element matches with the element in the array, then that particular location will be returned. In this case location, 4 will be returned as the key-value matches the value at that location.

Example

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   int myarray[10] = {21,43,23,54,75,13,5,8,25,10};
    int key,loc;
    cout<<"The input array is"<<endl;
    for(int i=0;i<10;i++){
        cout<<myarray[i]<<" ";
    }
    cout<<endl;
    cout<<"Enter the key to be searched : "; cin>>key;
    for (int i = 0; i< 10; i++)
    {
        if(myarray[i] == key)
        {
            loc = i+1;
            break;
        }
        else
        loc = 0;
    }
    if(loc != 0)
    {
        cout<<"Key found at position "<<loc<<" in the array";
    }
    else
    {
```

```
        cout<<"Could not find given key in the array";
    }

}
```

Output

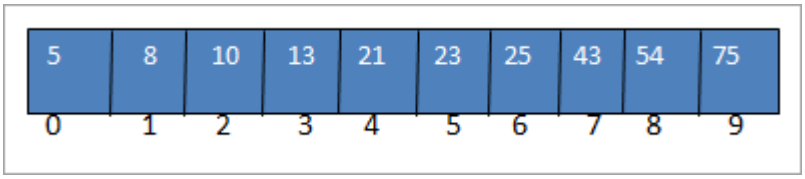
```
The input array is
21 43 23 54 75 13 5 8 25 10
Enter the key to be searched : 3
Could not find given key in the array

The input array is
21 43 23 54 75 13 5 8 25 10
Enter the key to be searched: 75
Key found at position 5 in the array
```

Binary Search

Binary search is a technique that uses “divide and conquer” technique to search for a key. It works on a sorted linear list of elements. The sorted list is the basic requirement for a binary search to work.

In the binary search method, the list is repeatedly divided into half and the key element is searched in both the halves of the list until the key is found.

For Example, let us take the following sorted array of 10 elements.



Let us say the key = 21 is to be searched in the array.

Let us calculate the middle location of the array.

Mid = 0+9/2 = 4

Example

```
#include <iostream>
#include <string>
```

```
using namespace std;
int binarySearch(int myarray[], int beg, int end, int key)
{
    int mid;
    if(end >= beg) {
        mid = (beg + end)/2;
        if(myarray[mid] == key)
        {
            return mid+1;
        }
        else if(myarray[mid] < key) {
            return binarySearch(myarray,mid+1,end,key);
        }
        else {
            return binarySearch(myarray,beg,mid-1,key);
        }
    }
    return -1;
}
int main ()
{
    int myarray[10] = {5,8,10,13,21,23,25,43,54,75};
    int key, location=-1;
    cout<<"The input array is"<<endl;
    for(int i=0;i<10;i++){
        cout<<myarray[i]<<" ";
    }
    cout<<endl;
    cout<<"Enter the key that is to be searched:"; cin>>key;
    location = binarySearch(myarray, 0, 9, key);
    if(location != -1)  {
        cout<<"Key found at location "<<location;
    }
    else   {
        cout<<"Requested key not found";
    }
}
```

Output

```
The input array is
5 8 10 13 21 23 25 43 54 75
Enter the key that is to be searched:21
Key found at location 5
```

6.6 Algorithms- basic Sorting

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

A similar approach is used by insertion sort.

Example

```
// Insertion sort in C++

#include <iostream>
using namespace std;

// Function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;

    // Compare key with each element on the left of it until an element
smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}

// Driver code
int main() {
  int data[] = {9, 5, 1, 4, 3};
```

```
  int size = sizeof(data) / sizeof(data[0]);
  insertionSort(data, size);
  cout << "Sorted array in ascending order:\n";
  printArray(data, size);
}
```

Selection Sort Algorithm

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Example

```
// Selection sort in C++

#include <iostream>
using namespace std;

// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
```

```
}

// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2};
  int size = sizeof(data) / sizeof(data[0]);
  selectionSort(data, size);
  cout << "Sorted array in Acsending Order:\n";
  printArray(data, size);
}
```

6.7 Min-max algorithm

C++ defined functions to get smallest and largest elements among 2 or in a container using different functions. But there are also functions that are used to get both smallest and largest element using a single function, "**minmax()**" function achieves this task for us. This function is defined in "**algorithm**" header file. This article would deal in its implementation and other related functions.

1. **minmax(a, b):** This function **returns a pair**, in which **1st** element is of **minimum** of the two elements and the **2nd** element is **maximum** of 2 elements.

2. **minmax(array of elements):** This function returns similarly as 1st version. Only difference is that in this version, the **accepted argument is a list of integers/strings** among which maximum and minimum are obtained. Useful in cases when we need to find maximum and minimum elements in list without sorting.

Example

```
// C++ code to demonstrate the working of minmax()

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

int main()
{

// declaring pair to catch the return value
pair<int, int> mnmx;

// Using minmax(a, b)
mnmx = minmax(53, 23);

// printing minimum and maximum values
```

```
cout << "The minimum value obtained is : ";
cout << mnmx.first;
cout << "\nThe maximum value obtained is : ";
cout << mnmx.second ;

// Using minmax((array of elements)
mnmx = minmax({2, 5, 1, 6, 3});

// printing minimum and maximum values.
cout << "\n\nThe minimum value obtained is : ";
cout << mnmx.first;
cout << "\nThe maximum value obtained is : ";
cout << mnmx.second;

}
```

Output

```
The minimum value obtained is : 23
The maximum value obtained is : 53

The minimum value obtained is : 1
The maximum value obtained is : 6
```

6.8 set operations

Sets are a type of associative containers in which each element has to be unique because the value of the element identifies it. The values are stored in a specific order.

Syntax:

```
set<datatype> setname;
```

Datatype:

Set can take any data type depending on the values, e.g. int, char, float, etc.

Example:

```
set<int> val; // defining an empty set
set<int> val = {6, 10, 5, 1}; // defining a set with values
```

Note: set<datatype, greater<datatype>> setname; is used for storing values in a set in descending order.

Example

```cpp
// CPP program to demonstrate various functions of
// Set in C++ STL
#include <iostream>
#include <iterator>
#include <set>

using namespace std;

int main()
{
    // empty set container
    set<int, greater<int> > s1;

    // insert elements in random order
    s1.insert(40);
    s1.insert(30);
    s1.insert(60);
    s1.insert(20);
    s1.insert(50);

    // only one 50 will be added to the set
    s1.insert(50);
    s1.insert(10);

    // printing set s1
    set<int, greater<int> >::iterator itr;
    cout << "\nThe set s1 is : \n";
    for (itr = s1.begin(); itr != s1.end(); itr++) {
        cout << *itr << " ";
    }
    cout << endl;

    // assigning the elements from s1 to s2
    set<int> s2(s1.begin(), s1.end());

    // print all elements of the set s2
    cout << "\nThe set s2 after assign from s1 is : \n";
    for (itr = s2.begin(); itr != s2.end(); itr++) {
        cout << *itr << " ";
    }
    cout << endl;

    // remove all elements up to 30 in s2
    cout << "\ns2 after removal of elements less than 30 "
            ":\n";
    s2.erase(s2.begin(), s2.find(30));
    for (itr = s2.begin(); itr != s2.end(); itr++) {
        cout << *itr << " ";
```

```
    }

    // remove element with value 50 in s2
    int num;
    num = s2.erase(50);
    cout << "\ns2.erase(50) : ";
    cout << num << " removed\n";
    for (itr = s2.begin(); itr != s2.end(); itr++) {
        cout << *itr << " ";
    }

    cout << endl;

    // lower bound and upper bound for set s1
    cout << "s1.lower_bound(40) : \n"
        << *s1.lower_bound(40) << endl;
    cout << "s1.upper_bound(40) : \n"
        << *s1.upper_bound(40) << endl;

    // lower bound and upper bound for set s2
    cout << "s2.lower_bound(40) :\n"
        << *s2.lower_bound(40) << endl;
    cout << "s2.upper_bound(40) : \n"
        << *s2.upper_bound(40) << endl;

    return 0;
}
```

Output

```
The set s1 is :
60 50 40 30 20 10

The set s2 after assign from s1 is :
10 20 30 40 50 60

s2 after removal of elements less than 30 :
30 40 50 60
s2.erase(50) : 1 removed
30 40 60
s1.lower_bound(40) :
40
s1.upper_bound(40) :
30
s2.lower_bound(40) :
40
s2.upper_bound(40) :
60
```

6.9 heap sort

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

The initial set of numbers that we want to sort is stored in an array e.g. [10, 3, 76, 34, 23, 32] and after sorting, we get a sorted array [3,10,23,32,34,76].

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

Example

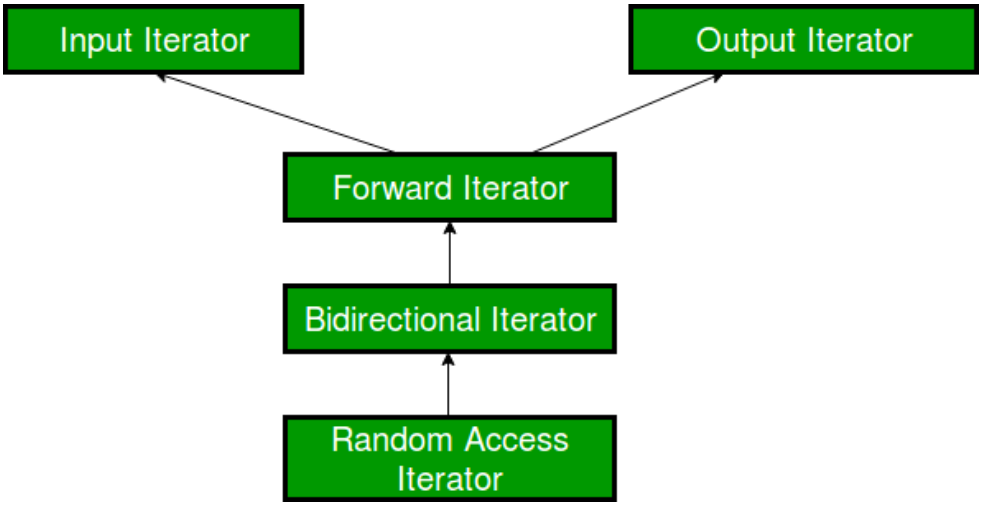
```
// Heap Sort in C++

  #include <iostream>
  using namespace std;

  void heapify(int arr[], int n, int i) {
    // Find largest among root, left child and right child
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
      largest = left;

    if (right < n && arr[right] > arr[largest])
      largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(arr[i], arr[largest]);
      heapify(arr, n, largest);
    }
  }

  // main function to do heap sort
  void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
      heapify(arr, n, i);
```

```
  // Heap sort
  for (int i = n - 1; i >= 0; i--) {
    swap(arr[0], arr[i]);

    // Heapify root element to get highest element at root again
    heapify(arr, i, 0);
  }
}

// Print an array
void printArray(int arr[], int n) {
  for (int i = 0; i < n; ++i)
    cout << arr[i] << " ";
  cout << "\n";
}

// Driver code
int main() {
  int arr[] = {1, 12, 9, 5, 6, 10};
  int n = sizeof(arr) / sizeof(arr[0]);
  heapSort(arr, n);

  cout << "Sorted array is \n";
  printArray(arr, n);
}
```

6.10 Iterators

After going through the template definition of various STL algorithms like std::find, std::equal, std::count, you must have found their template definition consisting of objects of type Input Iterator. So what are they and why are they used?

Input iterators are one of the five main types of iterators present in the C++ Standard Library, others being Output iterators, Forward iterator, Bidirectional iterator, and Random – access iterators. Input iterators are considered to be the weakest as well as the simplest among all the iterators available, based upon their functionality and what can be achieved using them. They are the iterators that can be used in sequential input operations, where each value pointed by the iterator is read-only once and then the iterator is incremented.



Input Iterator Example

```
// C++ program to demonstrate input iterator
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v1 = { 1, 2, 3, 4, 5 };

    // Declaring an iterator
    vector<int>::iterator i1;

    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Accessing elements using iterator
        cout << (*i1) << " ";
    }
    return 0;
}
```

Output

```
1 2 3 4 5
```

Output Iterator Example

```
// C++ program to demonstrate output iterator
#include<iostream>
#include<vector>
```

```
using namespace std;
int main()
{
    vector<int>v1 = {1, 2, 3, 4, 5};

    // Declaring an iterator
    vector<int>::iterator i1;

    for (i1=v1.begin();i1!=v1.end();++i1)
    {
        // Assigning elements using iterator
        *i1 = 1;
    }
    // v1 becomes 1 1 1 1 1
    return 0;
}
```

Forward Iterator Example

```
// C++ program to demonstrate forward iterator
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v1 = { 1, 2, 3, 4, 5 };

    // Declaring an iterator
    vector<int>::iterator i1;

    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Assigning values to locations pointed by iterator
        *i1 = 1;
    }

    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Accessing values at locations pointed by iterator
        cout << (*i1) << " ";
    }

    return 0;
}
```

Output

```
1 1 1 1 1
```

Bidirectional Iterator Example

```cpp
// C++ program to demonstrate bidirectional iterator
#include<iostream>
#include<list>
using namespace std;
int main()
{
    list<int>v1 = {1, 2, 3, 4, 5};

    // Declaring an iterator
    list<int>::iterator i1;

    for (i1=v1.begin();i1!=v1.end();++i1)
    {
        // Assigning values to locations pointed by iterator
        *i1 = 1;
    }

    for (i1=v1.begin();i1!=v1.end();++i1)
    {
        // Accessing values at locations pointed by iterator
        cout << (*i1) << " ";
    }

    return 0;
}
```

Output

```
1 1 1 1 1
```

Random Access Iterator Example

```cpp
// C++ program to demonstrate Random-Access iterator
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v1 = { 10, 20, 30, 40, 50 };

    // Declaring an iterator
    vector<int>::iterator i1;

    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Assigning values to locations pointed by iterator
        *i1 = 7;
    }
```

```
    for (i1 = v1.begin(); i1 != v1.end(); ++i1) {
        // Accessing values at locations pointed by iterator
        cout << (*i1) << " ";
    }

    return 0;
}
```

Output

```
7 7 7 7 7
```

6.11 Object Oriented Programming – a road map to future

Object-oriented modeling has become the de-facto standard in the early phases of a software development process during the last decade. The current state-of-the-art is dominated by the existence of the Unified Modeling Language (UML), the development of which has been initiated and pushed by industry.

The discussion on forthcoming versions of the UML will be a major task in the near future in the field of object-oriented modeling. It is obvious that a convincing solution to all open issues, can only be reached if fundamental, scientific research results will be combined in a synergetic way with industrial requirements and restriction