Unit 10

Templates

1. Introduction

C++ provides the concept of templates which allows generic programming. It is a mechanism by which we can define classes and functions without specifying the data type and only specify the behavior. Templates use a generic type as *placeholder* which will be replaced by the actual type. Hence, functions and classes become reusable components which provide the same functionality and at the same time, support different data types.

Definition: A template represents a family of functions or classes which have the same code and behavior but have parameters and members of different data types.

C++ provides two kinds of templates:

i. Function Templates

ii. Class Templates

A template is a skeleton which uses a generic datatype which is replaced by the actual type at the time the function or class is actually used. Hence, templates are also called *parameterized classes* or *parameterized functions*.

2. Function Template

Function template is a single function that represents a family of functions with the same code and different argument types. A function template is also called a *generic function*.

We could do the same thing by overloading functions which do the same task on different types. However, this results in too much repeated code and we have to define many functions to handle several data types.

For example: A function sum to find the sum of two values would be overloaded as follows;

```
int sum(int x, int y)
{
   return x+y;
}
float sum(float x, float y)
{
   return x+y;
}
```

Here, the function code is the same but only the data types differ. A function template allows us to define just one function which can work on several data types.

▶ Features of Generic Functions

i. Generic functions define the general set of operations that will be applied to various data types.

ii. The specific data type the function will operate upon is passed to it as a parameter.

iii. The compiler will automatically generate the function for the type of data i.e. actually used when the function is called.

iv. A generic function can be thought of as a function that "overloads itself".

Syntax:

```
template <class T>
return-type function-name(arguments)
{
   //body of function using arguments
}
```

- The keyword template is used to create a function template.
- The keyword class means the parameter can be of any type. It can even be a class.
- T is a placeholder for a data type used by the function. Any other valid C++ identifier can be used instead of T.
- Atleast one formal argument must be generic i.e. of the type T.

Let us consider a simple *example* for finding the maximum of two variables. If we do not know the type of the variables i.e. int, float, char etc. then we will have to define separate functions for each type. Instead,.we can define a single function template.

```
template <class T>
T max(T& a, T& b)  //C++ function template sample
{
    T ans;
    ans = a > b ? a: b;
    return ans ;
}
```

The generic type T will be replaced by either int, float or char when the function will be called.

Calling the Function Template

The function template can be called just like a regular C++ function. *For example*, the above template can be called in different ways:

```
1.  int x=10, y=20;
    cout << max(x,y);
2.  char c1 = 'B', c2 = 'S';
    cout << max(c1, c2);
3.  double d1 = 4.5, d2 = 6.7;
    cout << max(d1, d2);
```

The following program uses the template function defined above.

Program : Use of function template

```
#include<iostream.h>
template<class T>
T max( T& a, T& b)
{
  T ans;
  ans = a>b?a:b;
  return ans;
}
int main()
{
  cout << max(10,15) <<endl;
  char c1 = 'S', c2 = 'B';
  cout << max(c1,c2) <<endl;
  double d1 = 67.3, d2 = 25.3;
  cout << max(d1,d2) <<endl;
  return 0;
}// end of main
```

Output

```
15
S
67.3
```

In the above *example*, T is replaced by the specific type when the function is invoked. So naturally, both arguments will be treated of the same type. Because our template function includes only one data type (**class T**) and both arguments it takes are of the same type, we cannot call our template function with two objects of different types as parameters.

A template function can also have non-generic parameters. In the *example* below, the display function has two parameters, one generic and one int type.

```
template<class T>
void display(T data, int n)
{
   for(int i=1; i<=n; i++)
   cout<< data << endl;
}
int main()
{
   display('*',3);
   display("Hello",2);
}
```

Output

```
*
*
*
Hello
Hello
```

2.1 Generic Bubble Sort

One of the most useful applications of functions template is to implement standard algorithms which can be applied to all types of data. *For example*, searching and sorting algorithms. Let us now consider the commonly used sorting method – Bubble Sort and develop a template function for bubble sort.

The template function will take an array of type T as parameter and the number of elements.

```
template<class T>
void bubblesort(T arr[], int n)
{
   for(int pass =1; pass<=n-1; pass++)
     for(int i=0; i<=n-pass-1; i++)
      if(arr[i] > arr[i+1])
         swap(arr[i], arr[i+1]);  //swap is a generic function
}
```

The swap function will be defined as a template function as follows

```
template<class T>
void swap(T& a, T& b)
{  T temp;
   temp = a; a = b; b = temp;
}
```

The program is given below.

Program : Generic Bubble Sort

```
#include<iostream.h>
template<class T>
void swap(T& a, T& b)
{
   T temp;
   temp = a; a = b; b = temp;
}
template<class T>
void bubblesort(T arr[], int n)
{
     for(int pass =1; pass<=n-1; pass++)
     for(int i=0; i<n-pass; i++)
     if(arr[i] > arr[i+1])
     swap(arr[i], arr[i+1]);  //swap is a generic function
}
template<class T>
void display(T[] arr, int n)
{
  for(int i=0;i<n;i++)
  cout << arr[i]<< "  " ;
  cout<<endl;
}
int main()
{
  int a[]={25,57,48,37,12};
  char c[]={'P','R','O','G','R','A','M'};
  double d[] = {5.6, 3.4, -9.2, 15.1, 2.9};
  int i;
  bubblesort(a,5);
  cout<< "The sorted integer array is: ";
  display(a,5);
  bubblesort(c,7);
  cout<< " The sorted character array is: ";
  display(c,7);
  bubblesort(d,5);
  cout<< " The sorted double array is: ";
  display(d,5);
  return 0;
}
```

Output

The sorted integer array is: 12 25 37 48 57

The sorted character array is:

A G M O P R R

The sorted double array is: -9.2 2.9 3.4 5.6 15.1

2.2 Overloading a Function Template

Although a generic function "overloads itself", we can still overload it explicitly. The function template can be overloaded in two ways:

i. A non-template function
ii. Another function template

When a template function is overloaded, the compiler resolves the call in the following manner

i. Call the non-template function that has an exact match.
ii. Call the template function that can be created for an exact match.
iii. Apply type promotion, conversion to ordinary functions and call the one that is the best match.
iv. If none of these can find a match, the compiler generates an error.

1. **Overloading by a non-template function:** A function template can be overloaded by a regular non-template function. In this case, the non-template function must differ in number or type of parameters.

   Consider the following program which defines a template function f having two parameters of generic type T. This function has been overloaded by a non-template function f which takes two int arguments.

Program : Overloading function template by non-template function

```
#include<iostream.h>
template<class T>
void f(T  x , T  y)
{
     cout<<"Template function" <<endl;
}
void f(int a, int b)
{
     cout<<"Non-Template function"<<endl;
}
int main()
{
   f(1,2) ;
    f('a', 'b');
    f(1,'a');
    return 0;
}// end of main
```

Output

```
Non-Template function
Template function
Non-Template function
```

Explanation: The first call f(1,2) invokes the non-template function because a non-template function takes precedence in overload resolution.

The function call `f('a','b')` can only match the argument types of the template function. Hence, the template function is called.

Argument deduction fails for the function call `f(1,'b')`; the compiler does not generate any template function for parameters of different types. The non-template function resolves this function call after using the standard conversion from `char` to `int` for the function argument `'b'`.

2. **Overloading by a template function:** We can overload a function template by another function template which takes different number of arguments. The following program defines two template functions – max. One function takes two arguments. The other takes three.

Program : Overloading by another template function

```
#include<iostream.h>
template<class T>
T max(T a, T b)
{
   return a>b?a:b;
}
// the following template overloads the previous template
template<class T>
T max(T a, T b, T c)
{
   return max(a,b)>c?max(a,b):c;
}
int main()
{
   cout<<"Maximum of two ints: "<<max(89,34)<<endl;
   cout<<"Maximum of three ints: "<<max(34,56,12)<<endl;
   cout<<"Maximum of two chars: "<<max('a','A')<<endl;
   return 0;
}
```

Output

```
Maximum of two ints: 89
Maximum of three ints: 56
Maximum of two chars: a
```

3. Class Templates

A class template represents a family of classes having the same methods but data members of different types. It creates a skeleton or structure of a class which contains one or more types that are generic or parameterized. Hence, a class template is also called a *parameterized class*.

Syntax:

```
template<class T>
class class_name
{
     // data elements of type T
     // functions with arguments of type T
}
```

Note The member functions of a class template are template functions by default.

For example, let us create a template class to represent a stack of any data type.

```
template<class T>
class stack
{
   T data[MAXSIZE]; // holds the stack elements
   int top; // index of topmost element
public:
   stack();            //constructor
   void push(T obj); // push object on stack
   T pop();            //pop object from stack
   bool isempty();
   bool isfull();
};
```

▶ Defining Member Functions

The member functions can be defined within the class or outside it. All member functions of a generic class are automatically generic. There is no need to use the template keyword while declaring them. However, if they are defined outside the class, they will have to be defined using the syntax of template functions.

```
template <class T>
return-type class-name<T>::function-name(arguments)
{
  //function body
}
```

For example,

```
template<class T>
void stack<T>::push(T obj)
{
  data[++top] = ob;
}
```

▶ Creating Objects

After the template class has been defined, we have to create objects of the class. To do so, the following syntax should be used

```
class_name<type> object_name;
```

Here, type is the data type which will be used in place of generic type T.

For example,

1. To create a stack of type int
   ```
   stack<int> s1;
   ```
2. To create a stack of type char
   ```
   stack<char> s2;
   ```

The following program defines a template class to represent a stack of any data type. In main, we create two stacks, one of type int and one of type char and perform operations on these stacks.

Program : Class template for Stack

```
#include<iostream.h>
#define size 10
template <class T>
class stack
{
T data[size];        // holds the stack elements
int top;             // index of topmost element
 public:
  stack()            //constructor
  {
     top = -1;
  }
  void push(T obj); // push object on stack
  T pop(); //pop object from stack
```

```
    int isempty();
    int isfull();
};
template<class T>
void stack<T>::push(T obj)
{
   data[++top] = obj;
}
template<class T>
T stack<T>::pop()
{
   return data[top--];
}
template<class T>
int stack<T>::isempty()
{
    return top==-1;
}
template<class T>
int stack<T>::isfull()
{
    return top==size-1;
}
int main()
{
   stack<int> s1;          //create stack of type int
   stack<char> s2;         //stack of type char
   char str[20];
   for(int i=1;i<=5;i++)
       s1.push(i);
   while(!s1.isempty())
      cout << s1.pop() << " " ;
  cout<<"Enter a string:";
  cin>>str;
  for(i=0;str[i]!='\0';i++)
          s2.push(str[i]);
  cout<<"The reversed string is: ";
  while(!s2.isempty())
          cout<< s2.pop();
  return 0;
}
```

Output

5 4 3 2 1
Enter a string: Program
The reversed string is:
margor P

3.1 Class Template with Default Parameters

The class template parameter T can be assigned a default value. This value will be used if no data type is given during object creation.

For example,

```
template <class T = int>
class A
{
   //code
};
```

Here, the type *int* will be used if no other data type is specified when an object of type A is created. The default value cannot be given in the function template declaration or a function template definition, nor in the *template-parameter-list* of the member function of a class template.

We can also provide default values for non-generic type parameters. The following *example* is a class template with default type int and default size 100.

```
template<class T=int, int size=10>       //both default
class MyArray
{
  T data[size];
 public:
  void display()
   {
      for(int i=0; i<size; i++)
      cout<<data[i];
   }
};
```

We can create objects as follows:

```
MyArray<> obj1;            // type =int, size = 10
MyArray<char,50> obj2;     // type =char, size = 50>
MyArray<float> obj3;       //type = float, size=10
```

4. Template with Multiple Parameters

More than one generic data type can be defined in the template using a comma separated list as shown below. This is applicable to class as well as function templates.

Syntax for function template:

```
template<class T1, class T2>
return_type  function_name(arguments of type T1,T2,. . .)
{
  //Body of function
}
```

Example,

```
template<class X, class Y >
void display(X data1, Y data2)
{
    cout << data1 << data2;
}
```

Syntax for class template:

```
template<class T1, class T2>
class class_name
{
    // data elements of type T1 and T2
    // functions with arguments of type T1 and T2
}
```

Example,

```
template<class T1, class T2>
class pair
{
  T1 x;
  T2 y;
    public:
      pair(T1 a , T2  b) ; //constructor
      void display() ;
}
```

To create objects of the above template, the following syntax should be used.

```
pair<int,float> p1(10,5.6);
```

The following program defines a template class "pair" having two data members of types T1 and T2.

Program : Class template with multiple parameters

```
#include<iostream.h>
template<class T1, class T2>
class pair
{
  T1 x;
  T2 y;
    public:
      pair(T1 a , T2  b) //constructor
      {  x=a ; y=b ;
      }
      void display()
      { cout<<"(" << x <<","<< y << ")" <<endl;
      }
}
int main()
{
  pair<int,char> p1(25,'B') ;
  p1.display() ;
  pair<char,char> p2('Q','a') ;
  p2.display() ;
  return 0 ;
}
```

Output

(25,B)
(Q,a)

5. Advantages and Limitations of Templates

▶ Advantages

1. A single class Template or function template can handle different types of parameters.
2. Compiler generates classes or functions only for the data types used. If the template is instantiated for int type, compiler generates only an int version for the template class or function.
3. Templates reduce the effort on coding for different data types to a single set of code.
4. Testing and debugging efforts are reduced.
5. Requires less memory space compared to defining multiple classes or overloaded functions.

▶ Limitations

1. Its syntax is complex.
2. It can be difficult to debug code that is developed using templates. Since the compiler replaces the templates, it becomes difficult for the debugger to locate the code at runtime.
3. Automatically generated source code can become very huge.
4. Compile-time processing of templates can be extremely time consuming.
5. Templates can't be linked and distributed as a pre-compiled library. Hence all templates must be included in the header files completely.

6. Introduction to Standard Template Library (STL)

The Standard Template Library (STL) is a general-purpose C++ library which is a collection of algorithms and data structures. The data structures used in the algorithms are abstract in the sense that the algorithms can be used on (practically) every data type. The algorithms can work on these abstract data types due to the fact that they are template based algorithms. STL was developed by Alexander Stepanov and Meng Lee at Hewlett Packard.

There are six components in the STL organization. The first three can be considered the core components of the library:

i. **Containers** are data structures that manage a set of memory locations.
ii. **Algorithms** are computational procedures.
iii. **Iterators** provide a mechanism for traversing and examining the elements in a container.
iv. **Function objects** encapsulate a function as an object.
v. **Adapters** provide an existing component with a different interface.
vi. **Allocators** encapsulate the memory model of the machine.

7. Containers

Containers are STL objects that actually store data. There are several different types of containers, which are grouped into three categories namely, Sequence Container, Associative Container and Derived Containers or Container Adapters. Each container provides a set of functions and operators that can be used on the container.

The containers are organized in a hierarchy as shown below:

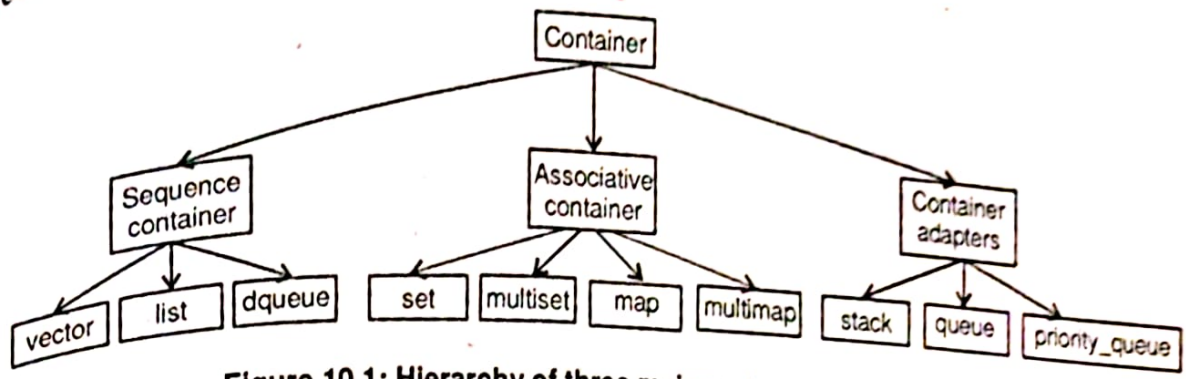

Figure 10.1: Hierarchy of three major categories of containers

The following table describes the details of the containers defined by the STL as well as the header file required to use each container.

Container	Description	Required Header File	Iterator type
Sequential Containers			
vector	A dynamic array of variables, structures or objects. Insertion/deletion of element at the end and allows direct access to any element.	<vector>	Random access
list	A linked list of variables, structures or objects. Insertion /deletion of element anywhere.	<list>	Bidirectional
deque	A double-ended queue, it is an array which supports insertion/removal of elements at beginning or end of array. It allows direct access to any element.	<deque>	Random access
Associative Containers			
set	A set in which each element is unique i.e. duplicate data is not allowed.	<set>	Bidirectional
multiset	A set in which each element is not necessarily unique i.e. duplication allowed.	<set>	Bidirectional
map	A map stores unique key/value pairs. Each key value is associated with only one value.	<map>	Bidirectional
multimap	A multimap stores key/value pairs in which one key may be associated with two or more values i.e. duplicate keys allowed.	<map>	Bidirectional
Derived Containers / Container Adapters / Sequence Adapters			
stack	A stack works in LIFO i.e. Last In First Out manner.	<stack>	No iterator
queue	A queue works in FIFO i.e. First In First Out manner.	<queue>	No iterator
priority queue	A priority queue. The element which is first out is always the highest priority element.	<queue>	No iterator

Each container class defines a set of functions that may be used to manipulate its contents. *For example*, a vector container defines functions for inserting and deleting an element and swapping the contents of two vectors. A list includes functions for inserting, deleting and merging elements.

A. Sequence Containers

A sequence is a container that stores a finite set of objects of the same type in a linear organization.

Following are the three types of sequence containers:

1. **vector**: A vector is a sequence that you can access at random. Insertion/deletion of element is fast at the end of the vector but it takes more time (i.e. slow) because they involve shifting the remaining elements. Vector allows fast random access. It supports a dynamic array.

2. **list**: A list is sequence that you can access bi-directionally i.e. at both end. Insertion/deletion of element is fast and we can insert/delete element anywhere, but provides slow sequential access.

3. **deque**: A doubly ended queue is a like a vector, except that it allows fast insertion/deletion at beginning as well as at the end of the container. Insertion/deletion of the element in the middle takes more time (i.e. slow). It allows fast random access.

b. Associative Containers

Sequences are indexed by integers; where as associative containers can be indexed by any type. Associative containers provide efficient retrieval of values based on keys. They are essentially key-value pairs and store data in trees rather than arrays or linked lists. These structures support fast random retrievals and updates.

1. **set**: set allows you to add and delete elements, query for membership, and iterate through the set.

2. **multisets:** Multisets are just like sets, except that you can have several copies of the same element (these are often called bags).

3. **maps**: Maps represent a mapping from one type (the key type) to another type (the value type). You can associate a value with a key, or find the value associated with a key, very efficiently; you can also iterate through all the keys.

4. **multimaps:** Multimaps are just like maps except that a key can be associated with several values.

c. Container Adapters

Container adapters are created from the existing sequence containers. They are derived containers.

1. **stack**: A stack is a data structure, which supports push and pop. The elements are arranged in the LIFO – Last In First Out manner. The stack template class supports two functions: push() and pop().

2. **queue** : A queue is a data structure wherein we can insert elements at the end and extract elements from the beginning. The queue template class supports two functions: push() and pop() to insert at the end and extract the first element in the data structure.

3. **priority_queue:** A priority_queue is a data structure wherein we can insert elements at the end and extract element that has the highest priority. The priority_queue template class supports two functions: push() and pop() to insert and extract elements in the data structure.

8. Algorithms

The STL algorithms are template C++ functions used to perform operations on containers. Although each container provides functions for its own basic operations, the standard algorithms provide more extended or complex actions. Also we can work with two different types of containers at the same time by using algorithms. By using `<algorithm>` header file in our program we can access the STL algorithms.

In order to work with many different types of containers, the algorithms do not take containers as arguments. Instead, they take iterators that specify part or all of a container.

The generic algorithms fall into following four categories

i. **Non-modifying Sequence Algorithms**

On a container (sequence), you may need to perform different functions that don't need to modify the contents of the container on which they work. *For example:* search(), count(), find() , equal() etc.

ii. **Mutating Sequence Algorithms**

Some types of a sequence operations results in modifying the contents of a container on which they work. *For example*: copy(), remove(), replace(), reverse(), swap(), fill() etc.

iii. **Sorting Algorithm**

STL provides various functions for sorting and sorting related operations. These include sort(), merge(), make_heap(), binary_search(), lexicographical_compare() etc.

iv. **Numeric Algorithms**

Numeric algorithms are used to perform four types of numeric calculations on the contents of a sequence. *For example:* accumulate(), partial_sum(), inner_product() etc.

9. Iterators

Iterators are used to access members of container classes. Iterators are used throughout the STL to access and list elements in a container. They are often used to traverse from one element to other, a process known as iterating through the container. They are objects that act like a pointer i.e. they specify the location for containers.

STL provides five categories of Iterators. Each category adds new features to the previous one. The iterator categories obey the following order:

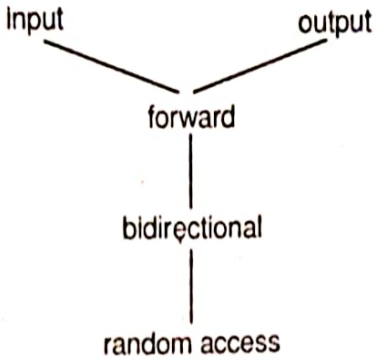

Figure 10.2: Iterator categories

The iterator categories are summarized in the following table

Iterator	Description
input_iterator	Read values with forward movement. These can be incremented, compared, and dereferenced.
output_iterator	Write values with forward movement. These can be incremented and dereferenced.
forward_iterator	Read or write values with forward movement. These combine the functionality of input and output Iterators with the ability to store the iterators value.
bidirectional_iterator	Read and write values with forward and backward movement. These are like the forward iterators, but you can increment and decrement them.
random_iterator	Read and write values with random access. These are the most powerful iterators, combining the functionality of bidirectional iterators with the ability to do pointer arithmetic and pointer comparisons.

10. Application of Container Classes

In this section we will see how to use containers. Two important and commonly used containers are Vector and List.

Creating and using a vector

A vector is like a dynamic array whose size can grow as elements are added to it. It also allows random access to any element. The vector class template is in a header file which should be included.

```
#include<vector>
```

The vector object is created as follows:

```
vector<int> v1; //creates a vector of type int
vector<char> v2; //vector of type char
```

The following table lists some commonly used member functions of the vector template class.

Function	Description
at()	Returns a reference to the element at a specified location in the vector.
back()	Returns a reference to the last element of the vector.
begin()	Returns a random-access iterator to the first element in the container.
capacity()	Returns the number of elements that the vector could contain without allocating more storage.
clear()	Erases the elements of the vector.
empty()	Returns True if the vector container is empty.
end()	Returns a random-access iterator that points just beyond the end of the vector.
erase()	Removes an element or a range of elements in a vector from specified positions.
front()	Returns a reference to the first element in a vector.
insert()	Inserts an element or a number of elements into the vector at a specified position.
max_size()	Returns the maximum length of the vector.
pop_back()	Deletes the element at the end of the vector.
push_back()	Adds an element to the end of the vector.
rbegin()	Returns an iterator to the first element in a reversed vector.
rend()	Returns an iterator to the end of a reversed vector.
reverse()	Reverses the order of the elements in the vector.
size()	Returns the number of elements in the vector.
vector()	Constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of some other vector.

In addition to member functions, the class also supports operators. These include all relational operators for comparing vectors and the subscript operator - [ ] for accessing a specific element of the vector.

The following program illustrates the use of several functions of the vector class template.

Program : Vector class template

```
#include<iostream.h>
#include<vector>
int main()
{
```

```
unsigned int i;  int a;
vector<int> v;      //creates an empty vector v of type int
cout<<"Original size of a vector is : "<<v.size()<<endl;
// putting values into the vector
v.push_back(10);
v.push_back(20);
v.push_back(30);
cout<<"Size after adding values =  "<<v.size()<<endl;
//Display the contents
cout<< "Contents of v : ";
for( i=0;i<v.size();i++)
  cout << v[i] <<" ";
vector<int>::iterator p=v.begin();  //iterator
//Inserting value 100 at 3rd location
p+=2;        // p points to 3rd element
v.insert(p,1,100);
//Display the size and contents after adding value
cout<< "\n Size after inserting value 100 = "<<v.size()<<"\n";
cout<< "Contents after insert : ";
for(i=0;i<v.size();i++)
  cout<<v[i]<<"  ";
cout<<endl;
//removing 2nd element
v.erase(v.begin()+1);
//Display the contents after deletion
cout<< "\n Size after erase : "<<v.size()<<endl;
cout<< "Contents after erase : \n";
for(i=0;i<v.size();i++)
  cout<<v[i]<<"  ";
cout<<endl;
return 0;
}
```

Output

Original size of a vector is : 0
Size after adding values = 3
Contents of v : 10 20 30
Size after inserting value 100 = 4
Contents after insert : 10 20 100 30
Size after erase : 3
Contents after erase : 10 100 30

Explanation

Creating a vector: vector<int> v; v has an initial capacity 0 i.e. initially v contains no elements.

Adding elements: v.pushback(element); Three elements are added at the end of the vector v using the push_back() function.

Displaying vector elements : You can display all elements in the vector using v[i] where the value of i goes from 0 to v.size().

Defining an iterator: The program uses an iterator to access the vector elements. It is created using: vector<int>::iterator p = v.begin();

Inserting an element: To insert, use function v.insert (iterator position, number of elements, value) .

Deleting an element: To delete an element, we can use the erase() or pop_back() function. To delete a single element, simply supply an iterator for the element as the erase() member function's single argument. If one wants to remove all the elements at once from the vector, the vector.clear() function can be used.

Creating and using a List

The list container implements the linked list data structure. The list container is defined as a template class, meaning that it can be customized to hold objects of any type. Unlike a C++ array or an STL vector, the objects it contains cannot be accessed directly using a subscript. List is a Sequence that supports both forward and backward traversal, and provide a constant time insertion and removal of elements at the beginning or the end, or in the middle. Compared to vectors, they allow fast insertions and deletions. Lists don't provide random access like an array or vector, They support bidirectional iterators.

To use STL lists, the following header file should be included:

```
#include<list>
```

Member functions

Some commonly used member functions of the list class are

Function	Description
back()	Returns a reference to the last element of the list.
begin()	Returns a random-access iterator to the first element in the list.
clear()	Erases the elements of the list.
empty()	Returns True if the list is empty.
end()	Returns a random-access iterator that points just beyond the end of the list.
erase()	Removes an element or a range of elements in a list from specified positions.
front()	Returns a reference to the first element in a list.
Insert()	Inserts an element or a number of elements into the list at a specified position.
max_size()	Returns the maximum length (the greatest number of elements that can fit) of the list.

merge()	Merge one list into other.
pop_back()	Deletes the last element of the list.
pop_front()	Deletes the first element of the list.
push_back()	Adds an element to the end of the list.
pop_front()	Adds an element at the start of the list.
rbegin()	Returns an iterator to the first element in a reversed list.
rend()	Returns an iterator to the end of a reversed list.
resize()	Specifies a new size for a list.
remove()	Removes specified elements.
remove_if	removes elements conditionally.
reverse()	Reverses the order of the elements in the list.
size()	Returns the number of elements in the list.
swap()	Exchanges the elements of two lists.
sort()	Sorts the list elements in ascending order.
splice()	Merge two lists in constant time.
swap()	Swap the contents of this list with another list.
unique()	Removes duplicate elements from the list.

In addition to these member functions, some STL algorithms (e.g., find) can be applied to the list container.

Operators

Some of the operators defined for the list container are

Operator	Meaning
=	The assignment operator copies one list to another
!=	Tests whether two lists are unequal.
<	Tests whether one list is less than another.
<=	Tests whether one list is less than or equal to another.
==	Tests whether two lists are equal.
>	Tests whether one list is greater than another.
>=	Tests whether one list is greater than or equal to another.

The following program illustrates the use of the above member functions on a list.

Program : Program using List

```
#include<list>
#include<iostream.h>
#include<cstdlib>       // for using rand() function
void display(list<int> &lst)
{
  list<int>::iterator iter;
  for(iter = lst.begin(); iter!=lst.end() ; iter++)
      cout<< *iter << "  ";
    cout<<"\n\n";
}
int main()
{
```

```
list<int>list1;          // Create an empty list
list<int>list2(5);       // Create a list having 5 elements.
int i;
//Insert the element into the 1st list
for(i=0;i<4;i++)
    list1.push_back(i+1);
cout<<"Original contents of the list 1: ";
display(list1);
//Erasing elements at the front and back of the list1
list1.pop_front();
list1.pop_back();
cout<<"The contents of the list 1 after erasing : ";
display(list1);
//Inserting elements at the end of the list1
list1.push_front(11);
list1.push_back(22);
cout<<"Now the contents of the list 1: ";
display(list1);
//Insert elements into the 2nd list
list<int>::iterator iter;
for(iter=list2.begin(); iter!=list2.end();iter++)
   *iter=rand()/100;
cout<<"The contents of the list 2:  ";
display(list2);
//Sort and Merge the two lists
list1.sort();
list2.sort();
list1.merge(list2);
cout<<"The merged sorted list is: ";
display(list1);
//Reverse a list
list1.reverse();
cout<<"Reversed  List is: ";
display(list1);
return 0;
}
```

Output

Original contents of the list 1: 1 2 3 4
The contents of the list 1 after erasing : 2 3
Now the contents of the list 1: 11 2 3 22
The contents of the list 2: 1 19 10 16 71
The merged sorted list is: 1 2 3 10 11 16 19 22 71
Reversed List is: 71 22 19 16 11 10 3 2 1

Explanation

- **Creation of the list**: The program uses two empty lists: list1 with zero length and list2 of size 5 using statements : list<int>list1; and list<int>list2(5);
- **Displaying list elements**: To display the list elements, we have defined a function called display() which uses an iterator "iter" to traverse the list from lst.begin() to lst.end(). Each element can be accessed using *iter.
- **Inserting and deleting values:** In list1 we insert values using push_back() and in list2 we use a list type iterator iter and the rand() function to generate random numbers. The begin() function gives the first position of the element and end() function gives the position immediately after the last element respectively.

  We insert the element in the list1 at both ends using push_front() and push_back() functions. At the same time remove the first and last element from list1 using the pop_front() and pop_back() function.
- **Merge, sort and reverse**: The sort() function sorts the elements of a list and the merge() function merges list1 with list2 and the result is placed in the list1 hence the list2 becomes empty. The reverse() function reverses a list.

EXERCISES

A. True or False

1. Templates create different versions of a function at runtime.
2. Template classes can work with different data types.
3. A template function can be overloaded by another template function with the same function name.
4. A function template can have more than one template argument.
5. Class template can have only class-type as parameters.

B. Review Questions

1. What is a template? List the merits and demerits of using a template in C++.
2. Define a function template.
3. Explain how a function template is defined and declared in a program.
4. What is a class template?
5. Explain how a function template can be overloaded?
6. Explain the concept of template with multiple parameters.
7. Write the syntax of template class.
8. Define a template function which returns maximum of two elements.

9. Write the syntax of defining a template class function outside the class.
10. Write a short note on STL.
11. Explain the container types in STL.
12. Write a short note on algorithms in STL.
13. What is an iterator? Explain its types.
14. With the help of an example, explain how a container class can be used.

C. Programming Exercise

1. Write a program in C++ to perform the following using the function template concepts
   a. to read a set of integers
   b. to read a set of floating point numbers
   c. to read a set of double members individually.

   Find out the average of the nonnegative integers and also calculate the deviation of the numbers.

2. Write a C++ program using a class template to read any five parameterized data type such as float and integer and print the average.

3. Write a function template to sort n integers. The function also has a parameter 'sortorder' which can be 'A' (for ascending) or 'D' (for descending). The default value is 'A'.

4. Define a class template Array to represent an array of any type and size. Define appropriate constructors for the class.

5. Define member functions to:
   i. Accept and display an Array object.
   ii. Find the largest element in the array object.

   Example: Array<int, 10> a; //creates an array of the type int and size 10.

6. Identify which of the following function template definitions are illegal.

   a.
```
template<class A,B>
void fun(A, B)
{. . .};
```

   b.
```
template<class A, class A>
void fun(A, A)
{. . .};
```

   c.
```
template<class A>
void fun(A, A)
{. . .};
```

   d.
```
template<class T, typename R>
T fun(T, R)
{. . .};
```

   e.
```
template<class A>
A fun(int *A)
{. . .};
```

7. Find error if any, in the following code segment.

```
Template <class T>
T max(T, T)
{
   . . .
};
unsigned int m;
int main()
{
   max(m, 100);
}
```

8. State which of the following definitions are illegal

a.
```
template <class T>
class city
{    ...
}
```

b.
```
template<class p, R ,class S>
class city
{ ... }
```

c.
```
class<class T, int size = 10>
class list
{
   ...    }
```

d.
```
class<class T = int , int size>
class list
{ ... }
```

e.
```
template<class T, type name S>
class city
{... }
```

Answers

A.

1. False
2. True
3. True
4. True
5. False