Introduction to STL and its Application

The **STL** or **Standard Template Library** is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Whenever a scenario arises that needs an implementation of a particular data structure or for say a binary search or any particular sorting technique, the STL library provides us with the particular function. Using this function, one does not need to write the complete code for a specific operation, instead the STL library provides the best-optimised way of doing it using STL containers and algorithm. The STL in C++ can be divided into two parts:

1. **Containers:** A container or container classes store objects and data. There are in total seven standards "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

    a. **Simple or Sequence Containers:** These implement data structures which can be accessed in a sequential manner. For eg.,
    - **Pair:** The pair container is a simple container defined in header consisting of two data elements or objects.
    - **Vector:** Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
    - **forward_list:** Forward list in STL implements singly linked list. Introduced from C++11, the forward list is useful than other containers in insertion, removal and moving operations (like sort) and allows time constant insertion and removal of elements.
    - **List:** Lists are sequence containers that allow non-contiguous memory allocation.

    b. **Container Adaptors:** These provide a different interface for sequential containers. For eg.,
    - **Stack:** Stacks are a type of container adaptors with LIFO(Last In First Out) type of work, where a new element is added at one end and (top) an element is removed from that end only.
    - **Queue:** Queues are a type of container adaptors which operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.
    - **Priority_queue:** Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non-increasing order(hence we can see that each element of the queue has a priority{fixed order}).

    c. **Associative Containers:** These implement sorted data structures that can be quickly searched (O(log n) complexity). For eg.,
    - **Set:** Sets are a type of associative containers in which each element has to be unique because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.
    - **Map:** Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have some key values.
    - **Unordered_set:** An unordered_set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized.
    - **Unordered_map:** The unordered_map is an associated container that stores elements formed by a combination of key-value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type of predefined or user-defined.

    Note: The *containers in C++ STL are templated* which means that for every container a template can be generated which will save the user from code repetition and will allow the user to use a single template for various data types. These templates accept data type as parameters. These are mainly templated so that they can accept any data types.

2. **Algorithm:** The header algorithm defines a collection of functions specially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers. For eg., binary_search, find, reverse, sort etc. The algorithms use the concept of iterators.
    An **iterator** is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of the iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator(++).
    Note: Using a begin and an end address of a container, the whole container can be traversed. The begin points to the first element and end points to the element beyond the last element. To traverse an array arr of size n, the user needs to pass, arr(beginning of the address), and (arr+n) as the end of the address. **Example:** The *find()* function can be used to find an element in the various containers like list, dequeue, vector

etc. It accepts two simple parameters, begin_address and an end_address.

Program: Take a sample array as input with elements {10, 15, 8, 20}. Sort the array and search for 8 inside the array.

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

// Drivers Method
int main()
{
    // Sample array
    int arr[] = { 10, 15, 8, 20 };

    // Calling STL sort() function
    // to sort the array
    // arr points to the beginning of
    // the array and arr+4 points to
    // the element beyond the last
    // element of the array
    sort(arr, arr + 4);

    // for loop to traverse through
    // and print the array
    for (int i = 0; i < 4; i++)
        cout << arr[i] << " ";
    cout << endl;

    // Performing binary binary_search
    // look for the element 8 in the
    // array
    if (binary_search(arr, arr + 4, 8))
        cout << "Present";
```

Run

Output:

```
8 10 15 20
Present
```

Working:

Step 1: An array has been taken input with 4 elements:

```
{10, 15, 8, 20}
```

Step 2: The sort() function has been called to sort the array. The sort() function is an algorithm is a part of the algorithm header file, which has been included at the beginning of the array.
The sort() function takes two parameters. The first parameter points to the beginning of the array and the second parameter points to the element beyond the last element of the array.
arr points to 10 and (arr+4) points to the element beyond 20.
Now the sort function can also take a third parameter which determines the order of sorting of the array.

```
sort(arr, arr+4)

arr -> 10

arr+4 -> 20
```

Step 3: Then we print the sorted array:

```
{8, 10, 15, 20}
```

Step 4: Then we perform a binary search where the binary_search() function is called and 3 parameters are passed to it. The first points

to the beginning of the array, the second points to the position beyond the last element of the array and the third parameter refer to the element that is needed to be searched in the array. This function returns a boolean true if the element is found else it returns false.

```
binary_search(arr, arr+4, 8)

arr -> 8

arr+4 -> 20

8 is the element to be searched.
```

Step 5: Finally, the search result is printed. Since the element 8 is present in the array, the output will be true.

Iterators in C++ STL

Iterators are used to point at the memory addresses of STL containers. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access the content at that particular location using them. They are primarily used in the sequence of numbers, characters etc. They reduce the complexity and execution time of the program.

Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of the iterator is a pointer. A pointer can point to elements in an array and can iterate through them using the increment operator (++). But, all iterators do not have similar functionality as that of pointers. For example, the iterators created for a forward_list, cannot be called pointers as they are not random access iterators and no mathematical computations can be performed over it. Different containers support different iterators, like vectors support Random-access iterators, while lists support bidirectional iterators. The whole list is as given below:

CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator Supported
Queue	No iterator Supported
Priority-Queue	No iterator Supported

Syntax:

```
containers_name iterator:: iterator_variable
```

Components:

- *containers_name:* This refers to the container for which the iterator is being created. Eg., List, vector, set, multiset etc.

- *data_type:* This refers to the data type of the iterator. Eg., int, double, float etc.

- *iterator_variable:* This refers to the iterator.

Now instead of writing this long statement, one can simply write **auto i** where i is the iterator.

Example:

```
1
2  // CPP program illustrating
3  // iterators
4  #include <iostream>
5  #include <vector>
6  using namespace std;
7
8  // Driver Method
9  int main()
10 {
11     // Sample vector container
12     vector<int> v = { 10, 20, 30, 40, 50 };
13
14     // Creating an iterator
15     // auto i; -> this can also be used to
16     // avoid the long statement
17     // i points to the first element
18     vector<int>::iterator i = v.begin();
19
20     // Prints the value at i
21     cout << (*i) << " ";
22
23     // Increments i by 1
24     i++;
25
26     // Prints the value at next position
27     cout << (*i) << " ";
28
29     // Now i points to the element beyond
30     // the alst element at the vector
```

Run

Output:

```
10 20 50
```

Working:

1. `vector v = {10, 20, 30, 40, 50};`

   This statement creates a vector with the mentioned elements. Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

2. `vector:: iterator i = v.begin();`

   This creates an iterator of integer type to traverse throughout the array. It acts as a pointer to point to the various elements of the array. If there would have been a list the syntax to create the iterator would have been:

   ```
   list:: iterator i
   ```

   The begin() function is used to return an iterator pointing to the first element of the vector or any container. This function returns a bidirectional iterator pointing to the first element. Here it is pointing to 10.

3. `cout << (*i) << " ";`

   Here dereferencing of the iterator have been done, to access the the value at the i-th position. Mathematical operations can be performed on this iterator like increment or decrements.

4. `i = v.end();`

Here the i points to the element beyond the last element of the vector. The end() function is used to return an iterator pointing to next to the last element of the vector container. end() function returns a bidirectional iterator.

Important Operations in Iterators:

1. **next():** This function returns the new iterator that the iterator would point after advancing the positions mentioned in its arguments.

2. **prev():** This function returns the new iterator that the iterator would point after decrementing the positions mentioned in its arguments.

Program: This shows the operation of next() and previous() function in iterators.

```
// CPP program illustrating
// iterators
#include <iostream>
#include <vector>
using namespace std;

// Driver Method
int main()
{
    // Sample vector container
    vector<int> v = { 10, 20, 30, 40, 50 };

    // Creating an iterator
    // auto i; -> this can also be used to
    // avoid the long statement
    // i points to the first element 10
    vector<int>::iterator i = v.begin();

    // Points to next value 20
    // By default next() moves one position
    // ahead
    i = next(i);

    // Prints the value 20
    cout << (*i) << " ";

    // Moves i two positions ahead
    // pointing to 40
    i = next(i, 2);
```

Run

Output:

```
20 40 30
```

3. **advance():** This function is used to increment the iterator position until the specified number mentioned in its arguments is met. It simply modifies the existing value of the iterator with the given value.

Program: This shows the operation of advance function in iterators.

```

// CPP program illustrating
// iterators
#include <iostream>
#include <vector>
using namespace std;

// Driver Method
int main()
{
    // Sample vector container
    vector<int> v = { 10, 20, 30, 40, 50 };

```

```
    // Creating an iterator
    // auto i; -> this can also be used to
    // avoid the long statement
    // i points to the first element 10
    vector<int>::iterator i = v.begin();

    // Modifies the iterator from 0 to 3
    advance(i, 3);
    cout << (*i) << " ";
    return 0;
}

```

Run

Output:

```
40
```

Types of Iterators: Based upon the functionality of the iterators, they can be classified into five major categories.

Note: Thes iterators are only logical classification of iterators.

1. **Input Iterators:** They are the weakest of all the iterators and have very limited functionality. They can only be used in single-pass algorithms, i.e., those algorithms which process the container sequentially such that no element is accessed more than once. The does not allow to write anything in the iterators. For eg., read data from iterator, prefix and postfix operations on the iterator.

2. **Output Iterators:** Just like input iterators, they are also very limited in their functionality and can only be used in the single-pass algorithm, but not for accessing elements, but for being assigned elements. These can be used only to write something on iterators but not read.

3. **Forward Iterator:** They are higher in the hierarchy than input and output iterators and contain all the features present in these two iterators. But, as the name suggests, they also can only move in the forward direction and that too one step at a time.

4. **Bidirectional Iterators:** They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions, that is why their name is bidirectional.

5. **Random-Access Iterators:** They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality is same as pointers.

Here is a list of the iterators and there properties.

ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i=	++	
Forward	->	= *i	*i=	++	==, !=
Bidirectional		= *i	*i=	++, --	==, !=,
Random-Access	->,[ ]	= *i	*i=	++, --, +=, -==, + ,-	==, !=, <,>,<=,>=

Here is a list of the various containers and what all, iterators they support.

Container Types	Containers	Supporting Iterators
Simple	forward_list(singly linked list)	Forward (only in forward directions)
Simple	list	bidirectional
Simple	vector	Random
Associate	set	Bidirectional
Associate	map	Bidirectional
Associate	Multimap	Bidirectional
Associate	Multiset	Bidirectional
Associate	unordered_set	Forward
Associate	unordered_map	Forward
Adapters	queue	Do not have iterators
Adapters	stack	Do not have iterators
Adapters	priority_queue	Do not have iterators

The input and output iterators are used in the algorithms such as find(), sort(), search etc. If any of these algorithms need a read, then they would need an input iterator and if they want to write something, they would need an output iterator. If a program is expecting an input or output operator, then it means that it can be used in all the containers.

Note: For the next(), prev() or advance() functions,

- If the iterator is a random access iterator, they are going to work in O(1) time as the increments or decrements can be done directly.
- If the iterator is a forward iterator, they are going to work in O(n) time as the increments would be done one by one. Here n can never be negative, as in forward iterators one cannot traverse back into location.
- If the iterator is a bidirectional iterator, n can both be negative and positive, to traverse back and forth.

Templates in C++ STL

A **template** is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

Important points on Templates:

- Write once and use for any data type: The user needs to generate a generic code and the compiler automatically generates the data type code when it is needed to be used for a particular data type.
- Like macros, templates are processed by the compiler, but they are better than the macros in a way that templates perform the type-checking.
- Templates are the main concept behind STL or Standard Template Library. This library has all the classes built as a template. You can pass a data type as an argument and the same function can be used for all the data types.

Types of Template

There are two types of templates:

1. **Function Templates:** We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray()

2. **Class Templates:** Like function templates, class templates are useful when a class defines something that is independent of the data type. It is used to reproduce the code for various data types. It can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

Example:

```
1
2  // CPP program illustrating templates
3  #include <iostream>
4  using namespace std;
5
6  // This can also be written as
7  // template <class T>
8  template <typename T>
9
10 // Function template
11 T myMax(T x, T y)
12 {
13     return (x > y) ? x : y;
14 }
15
16 // Drivers Method
17 int main()
18 {
19     // This function call creates a function
20     // where all the T's are replaced with int
21     cout << myMax<int>(3, 7) << endl;
22
23     // This function call creates a function
24     // where all the T's are replaced with char
25     cout << myMax<char>('c', 'g') << endl;
26     return 0;
27 }
28
```

Run

Output:

```
7
g
```

Working:

- When myMax() is called, then all the T's in the function template is replaced with the int. T is a data type parameter processed during the compile time. x and y get the integer values 3 and 7 respectively. It does the comparison and returns the larger element of the two i.e., 7
- When myMax() is called, then all the T's in the function template is replaced with char. x and y receive 'c' and 'g' respectively. The function does the comparison and returns the greater character i.e., g.

Templates VS Macros in C++

Templates	Macros

template <typename T>
T myMax1(T x, T y)
{
return (x > y) ? x : y
}
	#myMax2(x, y)(((x) > (y)) ? (x) : (y))
	
	In this the expression is replaced everywhere with the ternary operator expression.

Internal Working: When myMax1(<data_type>) is called, this code is generated internally.	**Internal Working:** Macros are the preprocessing phase in the compiler and this preprocessor, replaces every myMax2()

int myMax1(data_type x, data_type y)
{
return (x > y) ? x : y
}
	with the ternary operation in the whole program.

Does Type checking	Does not do type checking, but just search and replace.
Less prone to errors	More prone to errors
Easy to debug.	Difficult to debug.

Function Templates in C++ STL

A **Function Template** is used to write a function once and let the compiler auto-generate the code for various data types when the function is being called or used for different data types. This function is also referred to as the generic function Examples of function templates are sort(), max(), min(), printArray() etc.

Example:

```
1
2  // C++ program to illustrate
3  // Function Templates
4  #include <iostream>
5  using namespace std;
6
7  // Function that take
8  // datatype as an argument
9  // T is the variable and
10 // can be extended to multiple types
11 template <typename T>
12
13 // Function of type as this Function
14 // is going to return an array element
15 // of type T
16 T arrMax(T arr[], int n)
17 {
18     // res variable to store the max
19     // element of type T
20     T res = arr[0];
21
22     // Loop traversing through the array
23     // Fincding max element
24     for (int i = 1; i < n; i++)
25         if (arr[i] > res)
26             res = arr[i];
27
28     // Returning the maximum element
29     return res;
30 }
```

Run

Output:

```
40
30.5
```

Working:

- When the integer array(arr1) is passed during the function call of arrMax(), then the compiler generates the following code where all T is being replaced by the data type int and the operation inside arrMax() is executed.

```
1
2  int arrMax(int arr[], int n)
3  {
4      // res variable to store the max
5      // element of type int
6      int res = arr[0];
7
```

```
    // Loop traversing through the array
    // Fincding max element
    for (int i = 1; i < n; i++)
        if (arr[i] > res)
            res = arr[i];

    // Returning the maximum element
    return res;
}

```

- When the float array(arr2) is passed during the function call of arrMax(), then the compiler generates the following code where all T is being replaced by the data type float and the operation inside arrMax() is executed.

```

float arrMax(float arr[], int n)
{
    // res variable to store the max
    // element of type int
    float res = arr[0];

    // Loop traversing through the array
    // Fincding max element
    for (int i = 1; i < n; i++)
        if (arr[i] > res)
            res = arr[i];

    // Returning the maximum element
    return res;
}

```

Important Points

1. The compiler has the ability to guess the data type of an element or array but is always a good practice to specify the data type of the element being passed. This makes the code more readable and easy to understand.

2. The template can also take values as parameters alongside the data type. For example, lets put a condition into the above program, where the array will only accept array up to a particular limit of size. We can do this by passing a limit value into the template.

Note: The template can also accept non-datatype parameters as well, but the value passed into the template must be a constant, as the generic code is generated during compile time, and the compiler would need to know the value of the limit is passed. It the passed value is not predefined as constant, then the compiler will throw an error.

Example:

```

// C++ program to illustrate
// Function Templates
#include <iostream>
using namespace std;

// Function that take
// datatype as an argument
// T is the variable and
// can be extended to multiple types
// limit accept an integer value
template <typename T, int limit>

// Function of type as this Function
// is going to return an array element
// of type T
T arrMax(T arr[], int n)
{
    // Program exits if limit exceeds
    if (n > limit) {
        cout << "limit exceeded";
        return 0;
    }
    // res variable to store the max
    // element of type T
    T res = arr[0];

    // Loop traversing through the array
    // Fincding max element
    for (int i = 1; i < n; i++)
```

Run

Output:

```
40
30.5
```

- Class Template in C++ STL

A **Class Template** also follows the trend of writing code once and using it for various data types. Examples of class templates are, a Stack class can be created that might be used as integer stack, float class etc. It can also be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc. Like function templates, class templates are useful when a class defines something that is independent of the data type.

Example:

```
1
2  // C++ program to illustrate
3  // Class Templates
4  #include <iostream>
5  using namespace std;
6
7  // T is the variable and
8  // can be extended to multiple types
9  template <typename T>
10
11 // A structure can also be used
12 // instead of a class as it has
13 // public access type
14 struct Pair {
15
16     // These pairs can be of any type
17     T x, y;
18
19     // Constructor accpecting both
20     // parameters of same type
21     Pair(T i, T j)
22     {
23         x = i;
24         y = j;
25     }
26
27     // Function of T type
28     T getFirst()
29     {
30         return x;
```

Run

Output:

```
10 20
```

Working: When the object of the class or structure Pair is created then all the T's in the class gets replaced with the data type parameter that is passed into the template.

```
1
2  struct Pair {
3
4      // These pairs can be of any type
5      int x, y;
6
7      // Constructor accpecting both
8      // parameters of same type
9      Pair(int i, int j)
10     {
11         x = i;
12         y = j;
13     }
14
15     // Function of T type
```

```
    int getFirst()
    {
        return x;
    }

    // Function of T type
    int getSecond()
    {
        return y;
    }
};

```

The functions that are created inside the class can also be created outside the class with a reference to the class. This can be done in the following way.

```

// C++ program to illustrate
// Class Templates
#include <iostream>
using namespace std;

// T is the variable and
// can be extended to multiple types
template <typename T>

// A structure can also be used
// instead of a class as it has
// public access type
struct Pair {

    // These pairs can be of any type
    T x, y;

    // Constructor accpecting both
    // parameters of same type
    Pair(T i, T j)
    {
        x = i;
        y = j;
    }

    // Function of T type
    T getFirst();

    // Function of T type
```

Output:

```
10 20
```