**Delhi Technological University**

****

**Self-Study Report**

on

**C++ Standard Template Library**

by

**Navjot Singh**

DTU/2K14/MC/**045**

**4th Semester** (January, 2016 - May 2016)

**DEPARTMENT OF MATHEMATICS**

JANUARY-MAY 2016

CERTIFICATE

I (Navjot Singh) hereby solemnly affirm that the project report entitled “Advanced C++ Concepts” being submitted by me in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Mathematics & Computing, to the Delhi Technological University, is a record of the bonafide work carried out by me under the guidance of Dr. H.C. Taneja. The work reported in this report in full or in part has not been submitted to any University or Institute for the award of any degree or diploma.

Navjot Singh

(DTU/2K14/MC/045)

Place: DTU, Bawana Road, Delhi-110042 Dr. H. C. Taneja

Date: 28/04/2016

|  |  |
| --- | --- |
| Index | |
| Acknowledgment | 1 |
| Introduction | 2 |
| History | 3 |
| Generic Programming | 4 |
| * Templates | 4 |
| Standard Library | 8 |
| Standard Template Library | 9 |
| * Containers | 10 |
| * Simple Containers | 10 |
| * Sequence Containers | 11 |
| * Container Adapters | 12 |
| * Associative Containers | 12 |
| * Others | 13 |
| * Algorithms | 14 |
| * Iterators | 16 |
| String Class | 18 |
| Bibliography | 20 |

**Acknowledgement**

The successful completion of any task would be incomplete without accomplishing the people who made it possible and whose constant guidance and encouragement secured me the success.

First of all, I am grateful to the almighty for establishing me to complete this self-study assignment. I owe a debt to our faculty, Mr. H. C. Taneja for incorporating in me the idea of a creative self-study project, helping me in undertaking this project and also for being there whenever I needed their assistance.

I also place on record, my sense of gratitude to one and all, who directly or indirectly have lent their helping hand in this venture.

Last, but never the least, I thank my parents for being with me, in every sense.

**Introduction**

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming. It is regarded as a middle-level language, as it comprises a combination of both high-level and low-level language features. It is a superset of C & a legal C program is a legal C++ program.

It was designed with a bias toward system programming and embedded, resource-constrained and large systems, with performance, efficiency and flexibility of use as its design highlights. It has also been found useful in desktop applications, servers (e.g. e-commerce, web search or SQL servers) & performance-critical applications (space probes).

It is an object oriented programming language as it supports the following characteristics:

**History**

C++ was developed by Bjarne Stroustrup at AT&T Bell Labs in 1979, as an extension of the C language. He wanted an efficient and flexible language similar to C, which also provided high-level features for program organization.

Stroustrup, began work on "C with Classes", the predecessor to C++, in 1979. The motivation for creating a new language originated from the fact that Simula had features that were very helpful for large software development, but was too slow for practical use, while BCPL was fast but too low-level to be suitable for large software development. C was chosen because it was general-purpose, fast, portable and widely used.

Initially, "C with Classes" added features to the C compiler, Cpre, including classes, derived classes, strong typing, inlining and default arguments.

In 1983, *C with Classes* was renamed *C++* ("++" being the increment operator in C) adding new features including virtual functions, function name and operator overloading, references, constants, type-safe free-store memory allocation (new/delete), improved type checking, and BCPL style single-line comments with two forward slashes (//), as well as development of a compiler for C++, Cfront.

In 1989, C++ 2.0 was released. New features included multiple inheritance, abstract classes, static member functions, const member functions, and protected members. In 1990, Later feature additions included templates, exceptions, namespaces, new casts, and a boolean type.

After the 2.0 update, C++ evolved relatively slowly until, in 2011, the C++11 standard was released, adding numerous new features, enlarging the standard library further, and providing more facilities to C++ programmers. After a minor C++14 update, released in December 2014, various new additions are planned for 2017.

**Generic Programming**

Generic programming is a style of computer programming in which algorithms are written in terms of types that can be specified later that are then instantiated when needed for specific types which are provided as parameters. It permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication.

The generic programming paradigm is an approach to software decomposition whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, analogously to the abstraction of algebraic theories in abstract algebra.

**Templates**

When creating container classes in statically typed languages, it is inconvenient to have to write specific implementations for each data type contained, especially if the code for each data type is virtually identical. For example, in C++, this duplication of code can be circumvented by defining a class template.

template<typename T>

class List

{

/\* class contents \*/

};

List<Animal> list\_of\_animals;

List<Car> list\_of\_cars;

Above, T is a placeholder for whatever type is specified when the list is created. These "containers-of-type-T", commonly called templates, allow a class to be reused with different data types as long as certain contracts such as subtypes and signature are kept.

Templates can also be used for type-independent functions. This can be illustrated by the example of a program which swaps two elements of any data type. Using this function, two variables- irrespective of their data types can be swapped. Hence, different swap functions for every data type need not be defined.

template<typename T>

void Swap(T & a, T & b) //"&" passes parameters by reference

{

T temp = b;

b = a;

a = temp;

}

string hello = "world!", world = "Hello, ";

Swap( world, hello );

cout << hello << world << endl; //Output is "Hello, world!"

**Advantages of Templates over macros:**

1. Both macros and templates are expanded at compile time. Macros are always expanded inline, while templates are only expanded inline when the compiler deems it appropriate. When expanded inline, macro functions and function templates have no extraneous runtime overhead. Template functions with many lines of code will incur runtime overhead when they are not expanded inline, but the reduction in code size may help the code to load from disk more quickly or fit within RAM caches.
2. Macro arguments are not evaluated prior to expansion. The expression using the macro defined above max(0, std::rand() - 100) may evaluate to a negative number (because std::rand() will be called twice as specified in the macro, using different random numbers for comparison and output respectively), while the call to template function

std::max(0, std::rand() - 100) will always evaluate to a non-negative number.

1. As opposed to macros, templates are considered [type-safe](https://en.wikipedia.org/wiki/Type_safety); that is, they require type-checking at compile time. Hence, the compiler can determine at compile time whether the type associated with a template definition can perform all of the functions required by that template definition.
2. By design, templates can be utilized in very complex problem spaces, whereas macros are substantially more limited.

**Disadvantages of Templates:**

1. Historically, some compilers exhibit poor support for templates. So, the use of templates could decrease code portability.
2. Many compilers lack clear instructions when they detect a template definition error. This can increase the effort of developing templates, and has prompted the development of concepts for possible inclusion in a future C++ standard.
3. Since the compiler generates additional code for each template type, indiscriminate use of templates can lead to code bloat, resulting in larger executables.
4. Because a template by its nature exposes its implementation, injudicious use in large systems can lead to longer build times.
5. 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.
6. Templates of templates (nested templates) are not supported by all compilers, or might have a limit on the nesting level.
7. Templates are in the headers, which require a complete rebuild of all project pieces when changes are made.
8. No information hiding. All code is exposed in the header file. No one library can solely contain the code.

**Application of Generic Programming**

A C++ program implementing Queue Data Structure compatible with all data types can be implemented as follows-

template <typename T> //Typename is a keyword used to signify a class

class queue

{

T\* array; //array of objects of type T

int nextIndex;

int firstIndex;

int size;

int arraySize;

public:

queue(){

array = new T[10];

size = 0;

arraySize = 10;

firstIndex = -1;

nextIndex = 0;

}

int getSize()

{ return size; }

bool isEmpty(){

if (size == 0)

return true;

else

return false;

}

T front(){

if (isEmpty())

return 0;

return array[firstIndex];

}

void enqueue(T element){

if (size == arraySize) // handle array full

{

T\* temp = new T[arraySize \* 2];

int k = 0;

for (int i = 0; i < firstIndex; i++, k++)

temp[k] = array[i];

firstIndex = 0;

nextIndex = arraySize;

arraySize = arraySize \* 2;

delete [] array;

array = temp;

}

array[nextIndex] = element;

if (size == 0)

firstIndex = 0;

nextIndex = (nextIndex + 1) % arraySize;

size++;

}

T dequeue(){

if (isEmpty())

return 0;

T output = array[firstIndex];

firstIndex = (firstIndex + 1) % arraySize;

size--;

if (size == 0){

firstIndex = -1;

nextIndex = 0;

}

return output;

}

~queue() {

delete [] array;

}

};

**Standard Library**

In the C++ programming language, the C++ Standard Library is a collection of classes and functions which are written in the core language and part of the C++ ISO Standard itself.

The C++ Standard Library is based upon conventions introduced by the Standard Template Library (STL), and has been influenced by research in generic programming and developers of the STL such as Alexander Stepanov and Meng Lee. Although the C++ Standard Library and the STL share many features, neither is a strict superset of the other.

A noteworthy feature of the C++ Standard Library is that it not only specifies the syntax and semantics of generic algorithms, but also places requirements on their performance. These performance requirements often correspond to a well-known algorithm, which is expected but not required to be used. In most cases this requires linear time O(*n*) or linearithmic time O(*n* log *n*), but in some cases higher bounds are allowed, such as quasilinear time O(*n* log2 *n*) for stable sort (to allow in-place merge sort)

The C++ Standard Library underwent ISO standardization as part of the C++ ISO Standardization effort, and is undergoing further work regarding standardization of expanded functionality.

**Standard Template Library**

The C++ Standard Template Library is a powerful set of general-purpose templatized classes (data structures) and functions (algorithms) that can be used for storing and processing data. It includes implementation of common data structures like vectors, lists, queues, and stacks. It was developed by Alexander Stepanov and Meng Lee while working at HP.

STL is large and complex and using it can save considerable time & effort and lend high quality to programs. All these benefits are possible because the well written and well tested components defined in STL are being reused.

At the core of the C++ STL are following three well-structured components:

Their relationship can be seen as:

**Object 2**

**Object 1**

**Object 3**

**Containers**

As stated earlier, containers are objects that hold data of the same type. The STL provides 15 containers which can be grouped as the following:

**Simple Containers**

* **pair**

The pair container is a simple associative container consisting of a 2-tuple of data elements or objects, called 'first' and 'second', in that fixed order. The STL 'pair' can be assigned, copied and compared. The array of objects allocated in a map are of type 'pair' by default, where all the 'first' elements act as the unique keys, each associated with their 'second' value objects.

**Sequence Containers**

Sequence Containers store elements in a linear sequence. Each element is related to other elements by its position along a line.

**Container Adaptors (Derived Containers)**

These containers can be created from different sequence containers. Therefore, they do not support iterators and hence can’t be used for data manipulation. However, they support certain member functions like push and pop that help in implementing insertion and deletion operations

**Associative Containers**

These containers are designed to support direct access to elements using keys. They are not sequential and store data using the “Tree” data structure. This facilitates fast searching, deletion and insertion. However, these are very slow for random access and inefficient for sorting.

**Others**

* **bitset**

A bitset stores bits (elements with only two possible values: 0 or 1, true or false, ...). The class emulates an array of bool elements, but optimized for space allocation: generally, each element occupies only one bit (which, on most systems, is eight times less than the smallest elemental type: char).  
Each bit position can be accessed individually like a regular array accesses its elements.   
Bitsets have the feature of being able to be constructed from and converted to both integer values and binary strings (members to\_ulong and to\_string). They can also be directly inserted and extracted from streams in binary format.

* **valarray**

A valarray object is designed to hold an array of values, and easily perform mathematical operations on them. It also allows special mechanisms to refer to subsets of elements in the arrays. Most mathematical operations can be applied directly to valarray objects, including arithmetical and comparison operators, affecting all its elements.  
The valarray specification allows for libraries to implement it with several efficiency optimizations.

**Algorithms**

Algorithms are functions that can be used generally across a variety of containers for processing their contents. Although each container provides functions for its basic operations, STL provides more than sixty standard algorithms to support more extended or complex applications. STL algorithms reinforce the philosophy of reusability.

To access STL algorithms, the header **<algorithm>** must be included.

STL Algorithms can be categorized on the basis of operations they perform. Some of the major Algorithms are described below:

|  |  |
| --- | --- |
| Non-Mutating Algorithms | |
| count() | Counts occurrence of a value in a sequence |
| equal() | Returns true if 2 ranges are the same |
| find() | Finds first occurrence of a value in a sequence |
| find\_end() | Finds last occurrence of a value In a sequence |
| search() | Finds a subsequence within a sequence |
| adjacent\_find(), count\_if(), for\_each(), mismatch(), search\_n() are some other non-mutating algorithms. | |

|  |  |
| --- | --- |
| Mutating Algorithms | |
| copy() | Copies a sequence |
| fill() | Fills a sequence with a specified value |
| generate() | Replaces all elements with the result of n operation |
| Iter\_swap() | Swaps elements pointed to by the iterators |
| remove() | Deletes elements of a specified value |
| replace() | Replaces elements with a specified value |
| reverse() | Reverses the order of elements |
| rotate() | Rotates elements |
| swap() | Swaps two elements |
| copy\_backward(),fill\_n(), random\_shuffle(), remove\_copy(), remove\_if(), replace\_copy(), replace\_if(), reverse\_if(), swap\_ranges, transform() are some other mutating algorithms. | |

|  |  |
| --- | --- |
| Sorting Algorithms | |
| binary\_search() | Conducts a binary search |
| Inplace\_merge() | Merges two consecutive sorted sequences |
| merge() | Merges two sorted elements |
| nth\_element() | Puts a specific element in its proper place |
| sort() | Sorts a sequence |
| make\_heap() | Replaces elements with a specified value |
| partial\_sort() | Sorts a part of a sequence |
| equal\_range(), lower\_bound(), Partition(), push\_heap(), pop\_heap() are some other Sorting Algorithms. | |

|  |  |
| --- | --- |
| Set Algorithms | |
| includes() | Finds whether a sequence Is a subsequence of another |
| set\_difference() | Constructs a sequence that is difference of two ordered sets |
| set\_intersection() | Constructs a sequence that contains intersection of two sets |
| set\_symmetric\_difference() | Produces a set which is the symmetric difference of two sets |
| Set\_union() | Produces sorted union of two ordered sets |

|  |  |
| --- | --- |
| Relational Algorithms | |
| equal() | Finds whether two sequences are the same |
| max() | Gives maximum of two values |
| max\_element() | Finds the maximum element within the sequence |
| min() | Gives minimum of two values |
| min\_element() | Finds the minimum element within the sequence |
| mismatch() | Finds the first mismatch between the elements in 2 sequences |

|  |  |
| --- | --- |
| Numeric Algorithms | |
| accumulate() | Accumulates the results of operation on a sequence |
| adjacent\_difference() | Produces a sequence consisting of difference of adjacent elements of a given sequence |
| Inner\_product() | Accumulates the results of operation on a pair of sequences |
| partial\_sum() | Produces a sequence by operation on a sequence- resulting in cumulative answer of all elements before that element |
| iota() | Assigns to every element in a range in a sequence, successive values which may be regularly incremented |

**Iterators**

Iterators behave like pointers and are used to access container elements. They are used to traverse from one element to another. The STL implements 5 different types of iterators-

|  |  |  |  |
| --- | --- | --- | --- |
| Iterator | Access Method | Movement Direction | i/o capability |
| Input | Linear | Forward | Read |
| Output | Linear | Forward | Write |
| Forward | Linear | Forward | Read & Write |
| Bidirectional | Linear | Forward & Backward | Read & Write |
| Random | Random | Forward & Backward | Read & Write |

Following are some of the important iterator methods:

|  |  |
| --- | --- |
| begin | Returns an iterator pointing to the first element |
| end | Returns an iterator referring to the *past-the-end* element. The past-the-end element is the theoretical element that would follow the last element. It does not point to any element. |
| rbegin | Returns a *reverse iterator* pointing to the last element. It iterates backwards: increasing it moves them towards the beginning of the container. |
| rend | Returns a *reverse iterator* pointing to the theoretical element preceding the first element (which is considered its *reverse end*). |
| size | Returns the number of elements |
| clear | Returns whether the sequence is empty (i.e. whether its [size](http://www.cplusplus.com/vector::size) is 0). |
| capacity | Returns the size of the storage space currently allocated for the sequence, expressed in terms of elements. This *capacity* is not necessarily equal to the size. It can be equal or greater, with the extra space allowing to accommodate for growth without the need to reallocate on each insertion. |
| front | Returns a reference to the first element. Unlike member begin, which returns an iterator to this same element, this function returns a direct reference. |
| back | Returns a reference to the last element. Unlike member end, which returns an iterator just past this element, this function returns a direct reference. |
| at | Returns a reference to the element at position *n*. The function automatically checks whether *n* is within the bounds of valid elements, throwing an out\_of\_range exception if it is not |
| assign | Assigns new contents to a sequence, replacing its current contents, and modifying its size accordingly. |
| insert | Removes all elements from the sequence (which are destroyed), leaving the container with a size of 0. |
| vector::push\_back | Adds a new element at the end of the [vector](http://www.cplusplus.com/vector), after its current last element. The content of *val* is copied (or moved) to the new element. |
| vector::pop\_back | Removes the last element in the [vector](http://www.cplusplus.com/vector), effectively reducing the container [size](http://www.cplusplus.com/vector::size) by one. |

**String Class**

A string is a sequence of characters. An array of char type to store characters together to form a string is known as “C-String”. This string is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null. This continues to be supported in C++.

Operations on C-strings often become complex and inefficient. Therefore, the standard C++ library provides a string class type that supports all the operations of a C-String and additionally provides much more functionality.

**Constructors**

Commonly used string constructors are:

* String(); //For creating an empty string
* String(const char \*str); //For creating string object from a C-String
* String(const string & str); //For creating string object from other string object

**Important Iterators on String Class**

|  |  |
| --- | --- |
| begin | Return iterator to beginning |
| end | Return iterator to end |
| rbegin | Return reverse iterator to end |
| rend | Return reverse iterator to beginning |
| cbegin, cend, crbegin, crend are some other iterators on String Class | |

**Important String Characteristics**

|  |  |
| --- | --- |
| size | Returns length of string |
| resize | Resize String but does not alter capacity |
| capacity | Returns size of allocated storage |
| reserve | Request a change in capacity |
| clear | Clear string |
| empty | Test if string Is empty |
| length, max\_size, shrink\_to\_fit are some other iterators on String Class | |

**Functions to access elements of String Objects**

|  |  |
| --- | --- |
| [] | Get character of string at mentioned index |
| at | Get character of string at mentioned position |
| back | Access last character |
| front | Access first character |
| substr | Generates substring within two positions |

**Functions to modify/act on String Objects**

|  |  |
| --- | --- |
| append | Append to string |
| push\_back | Append character to string |
| assign | Assign content to string |
| insert | Insert into string |
| erase | Erase characters from string |
| replace | Replace portion of string |
| swap | Swap string values |
| pop\_back | Delete last character |
| copy | Copy sequence of characters from string |
| find | Find content in string |
| rfind | Find last occurrence of content in string |
| compare | Compares two strings |

**Bibliography**

Books:

1. Object Oriented Programming with C++ (Fourth Edition) by E. Balaguruswamy)

Websites:

1. https://en.wikipedia.org/wiki/C%2B%2B
2. http://www.tutorialspoint.com/cplusplus/cpp\_overview.htm
3. https://en.wikipedia.org/wiki/C%2B%2B\_Standard\_Library
4. https://en.wikipedia.org/wiki/Standard\_Template\_Library
5. http://www.cplusplus.com/reference/stl/
6. https://cal-linux.com/tutorials/STL.html
7. http://www.tutorialspoint.com/cplusplus/cpp\_stl\_tutorial.htm
8. http://www.tutorialspoint.com/cplusplus/cpp\_strings.htm
9. http://www.cplusplus.com/reference/string/string/