IDHAYA COLLEGE FOR WOMEN
KUMBAKONAM - 612 001

PG & RESEARCH DEPARTMENT OF COMPUTER SCIENCE

ACADEMIC YEAR : 2019-2020
SEMESTER : II
CLASS : I - B.Sc (CS)
SUBJECT INCHARGE : J.SATHYA
SUBJECT NAME : PROGRAMMING IN C++
SUBJECT CODE : 16SCCCS2

Unit V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

The C++ Standard Template Library

- What is STL?
- Generic Programming: Why Use STL?
- Overview of STL concepts & features
  - *e.g., helperclass & function templates, containers, iterators, generic algorithms, function objects, adaptors*
- A Complete STL Example
- References for More Information on STL

What is STL?

• *The Standard Template Library provides a set of well structured* **generic** *C++ components that work together in a* **seamless** *way.*

What is STL (cont'd)?

A collection of composable class & function templates

- Helper class & function templates: operators, pair
- Container & iterator class templates
- Generic algorithms that operate over *iterators*
- Function objects
- Adaptors

Generic Programming: Why Use STL?

- **Reuse: "write less, do more"**
  - **STL hides complex, tedious & error prone details**
  - The programmer can then focus on the problem at hand
  - *Type-safe* plug compatibility between STL components
- **Flexibility**
  - **Iterators decouple algorithms from containers**
  - Unanticipated combinations easily supported
- **Efficiency**
  - **Templates avoid virtual function overhead**
  - Strict attention to time complexity of algorithms

STL Features: Containers, Iterators, & Algorithms

Containers

- *Sequential*: vector, deque, list
- *Associative*: set, multiset, map, multimap
- *Adapters*: stack, queue, priority queue

Iterators

- **Input, output, forward, bidirectional, & random access**
- Each container declares a trait for the type of iterator it provides

Generic Algorithms

- **Mutating, non-mutating, sorting, & numeric**

Types of STL Containers

There are three types of containers

Sequential containers that arrange the data they contain in a linear manner

- **Element order has nothing to do with their value**
- Similar to builtin arrays, but needn't be stored contiguous

Associative containers that maintain data in structures suitable for fast associative operations

- **Supports efficient operations on elements using keys ordered by operator**
- Implemented as balanced binary trees

Adapters that provide different ways to access sequential & associative containers

- *e.g.*, stack, queue, & priority queue

STL Container Overview

STL containers are *Abstract Data Types* (ADTs)

All containers are parameterized by the type(s) they contain

Each container declares various traits

- *e.g.*, iterator, const iterator, value type, *etc.*

Each container provides factory methods for creating iterators:

- begin()/end() for traversing from front to back
- rbegin()/rend() for traversing from back to front

STL Vector Sequential Container

A std::vector is a dynamic array that can grow & shrink at the end

- *e.g., it provides (pre—re)allocation, indexed storage,* push back(), pop back()
- Supports *random access*

iterators

- Similar to—but more powerful than—built-in C/C++ arrays
- **A std::deque (pronounced "deck") is a double-ended queue**
- It adds efficient insertion & removal at the *beginning* & *end* of the sequence via push front() &
- pop_front()

STL Associative Container: Map

An **std::map** associates a value with each unique key

- – a student's id number
- Its value type is implemented as pair<const Key, Data>

STL Associative Container: Set

An std::set is an ordered collection of unique keys

- *e.g., a set of student id numbers*

```
#include <set>
int main ()
{
    std::set<int> myset;
}
```

STL Associative Container: MultiSet & MultiMap

An **std::multiset** or an **std::multimap** can support multiple equivalent (non-unique) keys

- *e.g., student first names or last names*

Uniqueness is determined by an *equivalence* relation

- *e.g.*, strncmp() *might treat last names that are distinguishable by* strcmp() *as being the same*
- performance

- Trade-off: does not offer a *random access* iterator
- Implemented as doubly-linked list

STL Associative Container: MultiSet Example

```
#include <set> #include <iostream> #include <iterator>

int main()

{

  const int N = 10;

int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};

int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

std::multiset<int> A(a, a + N); std::multiset<int> B(b, b + N); std::multiset<int> C;

std::cout << "Set A: ";

std::copy(A.begin(), A.end(), std::ostream_iterator<int>(std::cout, " ")); std::cout <<
std::endl;

std::cout << "Set B: ";

std::copy(B.begin(), B.end(), std::ostream_iterator<int>(std::cout, " ")); std::cout <<
std::endl;

}
```

STL Iterator Categories

- Iterator *categories* depend on type parameterization rather than on inheritance: allows algorithms to operate seamlessly on both native (i.e., pointers) & user-defined iterator types.
- Iterator categories are hierarchical, with more refined categories adding constraints to more general ones.

-*Forward* iterators are both *input* & *output* iterators, but not all *input* or *output* iterators are *forward* iterators.

-*Bidirectional* iterators are all *forward* iterators, but not all *forward* iterators are *bidirectional* iterators.

-All *random access* iterators are *bidirectional* iterators, but not all

- *bidirectional* iterators are *random access* iterators.
- Native types (i.e., pointers) that meet the requirements can be used as iterators of various kinds.

STL Input Iterators

- *Input* iterators are used to read values from a sequence
- They may be dereferenced to refer to some object & may be incremented to obtain the next iterator in a sequence
- An *input* iterator must allow the following operations
  - ✓Copy ctor & assignment operator for that same iterator type.
  - ✓Operators == & != for comparison with iterators of that type. (AND Operator)
  - ✓Operators * (can be const) & ++ (both prefix & postfix).

STL Output Iterators

- *Output* iterator is a type that provides a mechanism for storing (but not necessarily accessing) a sequence of values
- *Output* iterators are in some sense the converse of Input Iterators, but have a far more restrictive interface:
  - Operators = & == & != need not be defined (but could be)
  - Must support non-const operator * (*e.g.*, **iter = 3)*
- Intuitively, an *output* iterator is like a tape where you can write a value to the current location.
- And you can advance to the next location, but you cannot read values & you cannot back up or rewind

STL Forward Iterators

- *Forward* iterators must implement (roughly) the union of requirements for *input* & *output* iterators, plus a default factor.
- The difference from the *input* & *output* iterators is that for two.
- *forward* iterators r & s, r==s implies ++r==++s.
- A difference to the *output* iterators is that operator* is also valid on the left side of operator= (*it = v is valid) & that the number of assignments to a *forward* iterator is not restricted.

Ex:

// Copy a file to cout via a loop. std::ifstream ifile ("example_file"); int tmp;

while (ifile >> tmp) std::cout << tmp;

//

STL Generic Algorithms

- Each container declares an iterator & const iterator as a trait.
  - ❖vector & deque declare *random access* iterators.
  - ❖list, map, set, multimap, & multiset declare *bidirectional*.
- Each container declares factory methods for its iterator type:
  - ❖begin()
  - ❖end()
  - ❖rbegin()
  - ❖rend()
- Composing an algorithm with a container is done simply by invoking the algorithm with iterators for that container.
- Templates provide compile-time type safety for combinations of containers, iterators, & algorithms.

Categorizing STL Generic Algorithms

- There are various ways to categorize STL algorithms, *e.g.*:
  - **Non-mutating**, which operate using a range of iterators, but don.t change the data elements found.
  - **Mutating**, which operate using a range of iterators, but can change the order of the data elements.
  - **Sorting & sets**, which sort or searches ranges of elements & act on sorted ranges by testing values.
  - **Numeric**, which are mutating algorithms that produce numeric results.
- In addition to these main types, there are specific algorithms within each type that accept a predicate condition.
  - Predicate names end with the if suffix to remind us that they require an “if” test.s result (true or false), as an argument; these can be the result of function calls.

Benefits of STL Generic Algorithms

- STL algorithms are decoupled from the particular containers they operate on & are instead parameterized by iterators.
- All containers with the same iterator type can use the same algorithms.
- Since algorithms are written to work on iterators rather than components, the software development effort is drastically reduced.

  -*e.g.*, instead of writing a search routine for each kind of container, one only write one for each iterator type & apply it any container.
- Since different components can be accessed by the same iterators, just a few versions of the search routine must be implemented.

STL Function Objects

- Function objects (aka *functors*) declare & define operator().
- STL provides helper base class templates unary function .
- binary_function to facilitate user-defined function objects.
- STL provides a number of common-use function object class templates:
  - ✓**Arithmetic**: plus, minus, times, divides, modulus, negate.
  - ✓**comparison**: equal to, not equal to, greater, less, greater equal, less equal.
  - ✓**logical**: logical and, logical or, logical not.
- A number of STL generic algorithms can take STL-provided or

user-defined function object arguments to extend algorithm behavior.

STL Function Objects Example

```
#include <vector>
 #include <algorithm>
 #include <iterator>
#include <functional>
 #include <string>
int main (int argc, char *argv[])
{
std::vector <std::string> projects;
for (int i = 0; i < argc; ++i) projects.push_back (std::string (argv [i]));
std::greater<std::string> ());
return 0;
}
```

STL Adaptors

- **STL adaptors implement the *Adapter* design pattern.**
  -*i*.e., they convert one interface into another interface clients expect.
- Container adaptors include stack, queue, priority queue.
- Iterator adaptors include reverse iterators .
- back_inserter() iterators.
- Function adaptors include negators & binders.
- STL adaptors can be used to *narrow* interfaces (*e.g.*, *a* stack *adaptor for* vector*).*

Strings:

- One of the most useful data types supplied in the C++ libraries is the string.
- A string is a variable that stores a sequence of letters or other characters, such as "Hello" or "May 10th is my birthday!".
- Just like the other data types, to create a string we first declare it, then we can store a value in it.

  - string testString; testString = "This is a string.";

- We can combine these two statements into one line:

  - string testString = "This is a string.";

- Often, we use strings as output, and cout works exactly like one would expect:

```
cout << testString << endl;
```

will print the same result as

```
cout << "This is a string." << endl;
```

Passing, returning, assigning strings:

- C++ strings are designed to behave like ordinary primitive types with regard to assignment.
- Assigning one string to another makes a deep copy of the character sequence.

```
string str1 = "hello";

string str2 = str1; // makes a new copy str1[0]  = 'y';     // changes str1, but not str2.
```

- Passing and returning strings from functions clones the string.
- If you change a string parameter within a function, changes are not seen in the calling function unless you have specifically passed the string by reference (e.g. using that & trick we learned about in the Queen Safety example.) .

Function & Purpose

1. strcpys1,s2;

- Copies string s2 into string s1.

2 .strcats1,s2;

- Concatenates string s2 onto the end of string s1.

3 .strlens1;

- Returns the length of string s1.

4 .strcmps1,s2;

- Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

5 .strchrs1,ch;

- Returns a pointer to the first occurrence of character ch in string s1.

Object–Oriented Analysis

- Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.
- The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions.
- They are modelled after real-world objects that the system interacts with.
- In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

Object–Oriented Design

- Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis.
- In OOD, concepts in the analysis model, which are technology−independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.
- The implementation details generally include:
  - Restructuring the class data (if necessary),
  - Implementation of methods, i.e., internal data structures and algorithms.

Object–Oriented Analysis

- In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects.
- The analysis produces models on how the desired system should function and how it must be developed.
- The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Object

- An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has:
  - Identity that distinguishes it from other objects in the system.

Class

- A class represents a collection of objects having same characteristic properties that exhibit common behavior.
- It gives the blueprint or description of the objects that can be created from it.
- Creation of an object as a member of a class is called instantiation. Thus, object is an instance of a class.
- The constituents of a class are:
- A set of attributes for the objects that are to be instantiated from the class.
- Generally, different objects of a class have some difference in the values of the attributes.

Object–Oriented Design

- Object-oriented design includes two main stages, namely, system design and object design.

System Design

- In this stage, the complete architecture of the desired system is designed.
- The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes.
- System design is done according to both the system analysis model and the proposed system architecture.
- Here, the emphasis is on the objects comprising the system rather than the processes in the system.