4/4 B.Tech. FIRST SEMESTER
LARGE SCALE C++ LAB

CS7L1 **Credits: 2**

(Common to CSE/IT)
Required

Lecture: - **Internal assessment: 25 marks**
Tutorial: 6 period /week **Semester end examination: 50 marks**

Course Context and Overview: This course introduces the fundamental concepts of Large Scale C++ Lab. With this foundation students can gain knowledge on Large Scale C++ and how the software components are organized using C++

Prerequisites: C LANGUAGE, I/O ANALOG AND DIGITAL INTERFACING, AND PERIPHERALS

Learning outcomes:

Ability to:

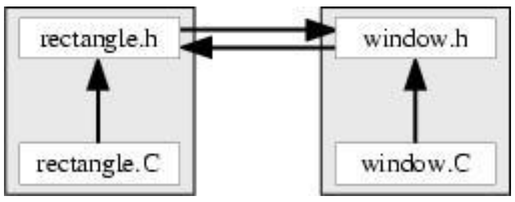
1. Implement programs to access data referred to by an iterator using STL a,b 25%
2. Create a simple container with a list of values using STL a,b 25%
3. Implement operations like inserting / manipulating on containers a,b 25%
4. Implement programs to access data referred by iterators to associate data to items in a data structure a,b 25%.

Introduction:

If we develop a large application or library, we have to consider a new level of organization: How to distribute classes and functions over files -- and also bigger units of organization. We distinguish between the logical design and the physical design. The logical design describes how to write a class or a function and how to relate them to each other. Physical design describes how to organize the code in files. In large systems, it is crucial to keep the complexity manageable. One measure of complexity is the dependency between different units. Specifically cyclic dependencies increase the complexity. Complex systems are hard to understand and hard to test. Compilation times and link times can grow unmanageable large for complex systems.

Programs

1. Compute prime numbers at compile time. Use typename and explicit,Implicit instantiation and its consequences for member functions.
2. Implementation of back_inserter, Use STL (Standard Template Library). Iterators, generic functions, iterator adaptors, function objects and iterator traits.
3. Implement graph data structures with nodes and edges: Eliminate cyclic dependencies in the dependency graph using templates The dependency graph looks like this:



4. Implement a program to access data referred to by iterators , to associate (additional) data to items in a data structure.

   Use Property maps (Boost Graph library). Exception mechanism and exception safety.
5. Implement simple STL containers (queues, vectors,...), iterators, and algorithms (find, for_each,...).
6. Implement B-Trees: THE external memory search tree data structure. can be used to implement the STL sorted container classes (multi)map and (multi)set
7. Implement Priority Queues.
8. Implement Sorting Strings
9. Implement Suffix Array Construction.
10. Implement a class Relation that supports the typical operation of a relational database (select, join, project).
11. Implement External memory numerics for sparse matrices, solving partial differential equations,...
12. Implement External memory minimum spanning trees
13. Implement External memory graph partitioning for planar graphs

In object-oriented design we would start with a common base class for all iterators. Let us assume that the value type is fixed to char for now.

```
struct Iterator {
   virtual void advance()            = 0;
   virtual bool equal( Iterator* i)  = 0;
   virtual char get()                = 0;
   virtual ~Iterator() {}
};
```

14. Write a concrete class for an iterator over C-arrays of characters using the following skeleton.

```
class Array_iterator : public Iterator {
   char* s;          public:          ...          };
```

15. Write a generic function that finds a character in a range of two iterators. The range is defined as in the STL to be the half-open interval including begin but excluding last. The function returns true if the character exists in the range [first, last).

    `bool contains( Iterator* first, Iterator* last, char c) {   ...  }  }`
16. Turn in both programs; the one with templates and the first one without templates.
17. Implement a program to break a cycle using Callback functions
18. Implement a program to change A HasA relationship to a HoldsA relationship. The cost for this is usually dynamic memory management.