A COMPARISON OF DATA STRUCTURES IN C++

WEINING ZHOU

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2001

0-612-59346-0

ABSTRACT

A Comparison of Data Structures in C++

Weining Zhou

Since many C++ libraries have become widely-used by programmers. Pointers are one of the most powerful and flexible features used in C++. Pointers are particularly important for the analysis and design of data structures. The intention of this project is to compare the two implementation methods, "Closed" and "Open" by using two C++ Libraries. In this project, I designed and implemented a phonebook application by using a doubly linked list and a binary search tree as common data structures in the C++ libraries: C++ Standard Template Library (STL) and C++ Data Object Library (OrgC++). In order to explore the performance of the two styles of building data structure libraries in C++, I performed an objective comparison of two implementation methods in terms of ease of coding, time and space efficiency as well as the reliability. The applications were coded in C++.

Acknowledgments

Through many face-to-face discussions as well as exchanges of electronic mail, a number of individuals have shaped my ideas on the project. For their contributions, first and foremost, I gratefully acknowledge the encouragement, assistance and valuable opinions of my supervisor, Dr. Peter Grogono, whose enthusiastic support and guidance enabled me to successfully understand and complete the project on time. I also thank Yang Wang, Weidong Sun, Xieming Hao and, Zhan Zhang for their valuable feedback and their help in finding ways to present the ideas more clearly. Thanks to Chunbo Yang for helping to straighten out the installation problems with the libraries. Thanks also to my friends David Wilkins and Alan Yu for their encouragement during this project. Finally, thanks to my dog, Muffy, who kept me company on many days and late nights of doing the project.

Table of Contents

LIST OF FIGURES VII

LIST OF TABLES VII

CHAPTER 1 INTRODUCTION 1

1.1 OBJECTIVES 1
1.2 MOTIVATION 2
1.3 OUTLINE OF THE PROJECT 2

CHAPTER 2 BACKGROUND 4

2.1 PROBLEMS WITH EXISTING SOFTWARE 4
2.2 CHALLENGES 5
2.3 THE BASIC CONCEPTS 5

CHAPTER 3 DESIGN 7

3.1 EVALUATION CRITERIA DEFINITION 7
3.2 THE APPLICATION 9
3.3 THE DATA STRUCTURES 10
3.4 THE LIBRARIES 11
3.4.1 Standard Template Library (STL) 12
3.4.2 C++ Data Object Library (OrgC++) 13
3.5 TEST CORRECTNESS 14
3.6 COMPARING TIME AND SPACE EFFICIENCY 15
3.6.1 Calculating Time 16
3.6.2 Calculating Space 16
3.6.3 Definition of Time and Space Ratio 16

CHAPTER 4 IMPLEMENTATION 18

4.1 ENVIRONMENT 18
4.2 "OPEN" IMPLEMENTATION METHOD 19
4.2.1 Implementing a Doubly Linked List 20

4.2.1 Implementing a Binary Search Tree 22
4.3 "CLOSED" IMPLEMENTATION 23
4.3.1 Implementing a Doubly Linked List 23
4.3.2 Implementing a Binary Search Tree 26
4.4 USER INTERFACE 30
4.4.1 Suspected Problems with OrgC++ 30
4.4.2 UI for "Open" Implementation in STL 31
4.4.3 UI for "Closed" Implementation in OrgC++ 35

CHAPTER 5 RESULTS 36

5.1 USABILITY - EASE OF CODING 36
5.2 TIME AND SPACE 38
5.2.1 Build Time Efficiency 38
5.2.2 Access Time Efficiency 39
5.2.3 Space Efficiency 42
5.3 INTERACTION WITH MFC 43
5.4 OTHER COMPARISONS 43

CHAPTER 6 CONCLUSIONS 45

6.1 THE ADVANTAGES OF STL AND ORGC++ 45
6.2 FUTURE WORK 47

CHAPTER 7 BIBLIOGRAPHY 48

A. BOOKS AND ARTICLES 48
B. INFORMATION ON THE WEB 49

APPENDIX 50

A STLFUNCTOR.H 50
B STLPERSON.H 51
C STLPERSONHASIT.H 53
D ORGCFUNCTOR.H 54
E ORGCPERSON.H 55
F ORGCPERSONHASIT.H 57

List of Figures

Figure 1: Lists in STL 20
Figure 2: DOUBLE_RING in OrgC++ 23
Figure 3: DOUBLE_TREE in OrgC++ 26
Figure 4: The Main Window of PhoneBook in STL 31
Figure 5: UI for Searching a record in STL 32
Figure 6: UI for inserting a record in STL 33
Figure 7: UI for deleting a record in STL 34
Figure 8: UI in OrgC++ 35

List of Tables

Table 1: Build Time ratio 39
Table 2: Access Time Ratio 40
Table 3: Access Time 41
Table 4: Space Ratio 42
Table 5: Comparison of the two Libraries in other criteria 44

Chapter 1 Introduction

As we know, there are many C++ libraries that are used by programmers. But each library works better in different situations. Data structures are of interest since there are some important performance problems in designing scientific class libraries. The experience of working with various data structures may offer some useful insights about performance of these data structures. In this section, I describe objectives of the project and why I chose this topic. I also introduce the structure of this report.

1.1 Objectives

This project is an attempt to find out whether pointers should be inside objects or in separate nodes. For example, with a linked list, there are two ways to implement it. In the "closed" implementation, each node has a pointer to the next node. In the "open" implementation, the list nodes are separated from the data nodes; the data nodes have no pointers. In order to explore the performance of two styles of building data structure libraries in C++, I present an objective comparison of two methods of implementation in terms of ease of coding, size of node, security, time/space efficiency and reliability. I designed several applications using Linked List and Tree as the common data structures on base class libraries such as STL and OrgC++ and not on UI libraries (such as InterViews, StarViews, etc.).

1.2 Motivation

When Dr. Peter Grogono asked me in May 2000 whether I would feel like writing the project to compare data structures in some C++ libraries, my first thought was: is this a very interesting topic? Many books about data structures and algorithms were already out at that time, but you could hardly find any references about comparison of these data structures. I simply questioned the need for doing this project. I tried to look at this aspect in different libraries to see if there was any related material, mostly in vain. Unfortunately, it is very hard to find any books that compare data structures with C++ libraries. About two weeks later, I must admit, he convinced me. This project will be very interesting and new to us. In summary, I enjoyed the project and hope it can provide some valuable reference to people who are interested in further research of data structures in C++ libraries.

1.3 Outline of the Project

The project starts off with an introduction in Chapter 1. It describes the objectives and motivation of the project. Chapter 2 describes the background of the project, and what the challenges are to the current technology as well as describing the existing problems with software and the concepts of the "Open" and "Closed" implementation methods. In Chapter 3, I present details about experimental design from many aspects including: the application design, selection of data structures, and libraries. In addition, I define the evaluation criteria which are used to compare the performance of the two methods later in chapter 5. Chapter 4 gives the details of the implementation of the phonebook in the different libraries and Chapter 5 presents the test results of the different implementations. To compare the performance of each implementation I used the predefined criteria. I also present the User Interface in Chapter 5. At the end I conclude the project and summarize the two implementation methods in Chapter 6 and. I also give some recommendations for further work on this project. References are listed in the Chapter 7.

Chapter 2 Background

In recent years there has been a proliferation of C++ libraries used by programmers but there are few published comparisons of data structures with them. With most class libraries today data structures are using "Open" implementation, such as STL. There are few libraries using "Closed" implementation, such as OrgC++ from the Code Farm. Definitions of "Open" and "Closed" implementation methods are in section 2.3.

2.1 Problems with existing software

Most of the problems with today's software stem from poor management of internal data. Data related errors such as dangling pointers may stay in the code unnoticed for a long time and are often difficult to find. There are no generic, fully typed libraries for data structures in C, and container based C++ libraries hide the critical data organization (data structures) thus making the resulting code difficult to debug. They also cause inefficiencies at run-time. Some libraries offer persistent objects but require custom coded IO functions for every new class. Even highly recognized libraries such as the STL or tools.h++ do not protect the programmer from accidentally destroyed objects or messed up linked lists. For example, you cannot add/delete objects from a list while traversing it with an iterator.

2.2 Challenges

The major challenge for present technology is to generate efficient software more efficiently. Computers are being used in essentially all areas of human activities; even products that seemingly have nothing to do with computers contain a significant portion of computer related cost.

In general, more efficient software means three things:

- Coding and debugging programs faster;
- Generating programs with less errors,
- Producing cleaner, more readable code, which is easier to maintain.

2.3 The Basic Concepts

Here, I present a brief explanation of some concepts I used in this project.

With **"Open" implementation method,** as the name implies, the list nodes are separated from the data nodes; the data nodes have no pointers to substructures. The data structures are formed by auxiliary objects, usually called links, that point to the application objects but do not add any pointers or other data to them. Each Link has a pointer that leads to the object that participates in the list.

With **"Closed" implementation method**, the data structures are formed by pointers stored inside the application objects. For example, in a linked list, each node has a pointer to the next node.

All this seems so trivial that you may wonder why I are spending so much time discussing it but there is more here than meets the eye. The problem is that the choice of implementation has numerous consequences which are often neglected. For example, a "closed" implementation is computationally more efficient, and permits bi-directional access between the objects involved. On the other hand, lists using "open" implementation method are easy to implement using templates, and they form the basis of almost every existing container class library.

Chapter 3 Design

In this project my major interest is to explore the performance of two styles of building data structure libraries in C++. I present an objective comparison of two methods of implementation in terms of ease of coding, size of node, security, time/space efficiency and reliability. Firstly, I need to choose the kind of application and what libraries and common data structures are to be considered in this project. In addition, I also need to consider two other important aspects which, are the correctness of the implementation and comparison of the time and the space of each data structure in the two libraries. In order to evaluate the performance of each implementation, I define evaluation criteria for the experiments.

3.1 Evaluation Criteria Definition

The criteria defined are applicable to a class library in general or to its component classes in particular to this project. The following have been defined:

- **Usability**

By this term I mean several aspects which, although difficult to quantify, are important, such as the set of classes provided, the ease of use of a class, the kind of services provided, and the simplicity and clarity of the interface. Of course, the more the interface is simple and clear, the better.

- **Time and Space**

For a given data structures such set, list, tree, graph, etc., the programs could check the time needed to build a list and search elements or the space occupied to build a list.

- **Others**

In deciding which C++ class library to use, other criteria can be defined, such as the following:

- is it shareware or commercial (do we have to pay for it, is it provided in source or object format) ?
- if it is provided in source format, are there problems to compile it with the compiler I have ?
- is the class library supported by the vendor/developer ?

Of course, without vendor/developer support I have to take into account the risk of finding bugs I then have to fix by myself.

- is it easy to install ?
- is there a suite of tests to be run in order to make sure that no problems will occur when using the library ?
- is it reliable? Is it easy to crash when you do a large search?

There is something to add to the discussion made so far. The criteria defined till now are somehow context-independent, in the sense that they are general criteria against which a class library can be evaluated and found to be more or less "desirable".

However, other criteria could be defined which are more context-dependent, and could arise from particular needs related to the kind of application that has to use the class library. The definition of such criteria is not included in the scope of this project but ought to be considered when making the final decision on which library (libraries) to use.

3.2 The Application

The application selected was a phone book. The phone book includes first names, last names and phone numbers of people. I designed this application specifically for the purpose of testing and comparing the performance of the selected C++ libraries.

It is not overly complex. I chose this application for two reasons:

- First, simplicity made it possible to write implementations in several C++ libraries in a reasonable time.
- Second, the simplicity enhances data structures comparison because all the libraries considered provide support for the set of features needed by the application.

The application can carry out the following basic functions:

- Insert: Input a record which includes first name, last name and telephone number of a person into the Phone Book.
- Search: Given a selected phone number, search for a person's record (First name & Last name.)

- Delete: Given a selected phone number, delete record that contains a person's first name, last name and this phone number. The phone number is used as a key to associate with the first name and last name of a person from the phonebook.

3.3 The Data Structures

The application is implemented in different libraries using Double Linked List and binary tree.

The reasons why I choose the above structures are:

(1) Both data structures are supported by both STL and Org C++.

(2) They are very common data structures in any class library. In other words, they can represent many data structures.

(3) Due to time concerns, they are easy to implement.

(4) A linked list and a binary search tree require pointers for allocating their nodes dynamically.

(5) I also considered other data structures, but found that the implementation of these data structures do not add significant new aspects over the data structures I have chosen. For example, the implementation of a single linked list is similar to a double linked list; the implementation of a hash tale or a graph can use a single or double liked list.

3.4 The Libraries

There are several libraries for containers and algorithms in C++. Each of them has different characteristics of its data structures in the implementation. After searching and studying many references about different libraries from different resources, I selected the libraries considering the following factors:

- Availability: The libraries can be easily obtained. They can be either part of most recent C++ implementations or can be downloaded free from web-sites.
- Common data structures: All libraries have the common support data structures such as list, tree, etc.
- Different implementation: The libraries are able to represent "Open" and "Closed" implementation methods.

In addition, I considered other criteria. In the end, based on these factors, I decided to use STL and OrgC++ libraries in the project to do the comparison. You may wonder why I selected STL and OrgC++. I describe them in the following sections.

Here, I will mention another Library, LEDA (Library of Efficient Data types and Algorithms). LEDA is a library of the data types and algorithms of combinatorial computing. At the beginning of the library selection, I considered LEDA. It is similar to STL. Data structures of LEDA also are implemented using "Open" Implementation. It is not public domain, for this reason and also concern about time, I dropped it from the study.

It took quite a long time for me to get a complete "feeling" of the different and multifold aspects of each library, although a "flavour" of it may be had in a shorter time.

3.4.1 Standard Template Library (STL)

As is well known, the C++ Standard Template Library (STL) is provided as a part of most recent C++ implementations. It is a C++ library of container classes, algorithms, and iterators. It provides many of the basic algorithms and data structures of computer science. As its name suggests, the Standard Template Library is based on the comparatively new subject of templates. The STL is a generic library, as implied by the name, meaning that its components are heavily parameterized. Almost every component in the STL is a template. The basic idea of templates is that they allow us to write functions and classes in a very general way and then specialize them when they are actually put to use.

The Standard Template Library is defined as a set of class interfaces. Individual vendors provide their own implementations of the STL. Strictly speaking, the comparison I describe in this report is between *a particular implementation* of STL (namely, the implementation provided by Microsoft) and OrgC++.

The organization and design of the STL differs in almost all respects from the design of most other C++ libraries. The most important difference between STL and all other C++ container class libraries is that most STL algorithms are generic; they work on a variety of containers and even on ordinary C++ arrays. A key factor in the library design is the consistent use of iterators, which generalize C++ pointers, as intermediaries between algorithms and containers. Containers make iterators available, algorithms use them, and this leads to a separation which allows an exceptionally clear design.

Therefore, iterators are central to generic programming because they are an interface between containers and algorithms. Algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write a generic algorithm that operates on many different kinds of containers, even containers as different as a vector and a doubly linked list.

3.4.2 C++ Data Object Library (OrgC++)

C++ Data Object Library includes the most extensive data structures on the market for fast, automatic persistence and are ideal for large, complex C++ projects.

The C++ Data Object Library (OrgC++), is much more than just a class library. It is based on the new concept of hyper-objects which are objects that store their data in other objects, called carriers, carriers passively keep the data, but hyper-objects provide the methods. OrgC++ provides an additional abstract layer for the management of data. We do not have to assemble the organization from library objects. The organization is automatically generated in a way which provides optimum run-time performance. Combined with automatic persistence and version control, OrgC++ provides a set of powerful tools which are useful for a variety of tasks ranging from the mangagement of internal data to the design of fast memory resident databases.

Compared to the C++ library, STL, OrgC++ is a library of intrusive data structures which are formed by pointers stored inside the application objects, unlike container where auxilliary objects form the required data structure and point only to the application objects without adding any pointers or other data to them. Therefore, OrgC++ uses the "closed" implementation and STL uses the "open" implementation.

3.5 Test Correctness

The main program of this project is the *phonebook.cpp*. A phonebook is defined as a list and binary search tree in STL and as *Double_Ring* and *Double_Tree* in OrgC++. It is used to test all the data structures implemented in this project, which include:

- **Person**: Defines a record of a person that includes the person's *firstname, lastname*, and *phonenumber*. It is used as a user defined data type later.
- **Functor**: Defines a random number generator. It is used to initialize a stream by a given seed. In other words, it is used to generate each random data stream of *firstname, lastname* and *phonenumber* of persons.
- **PersonHasIt**: is used to get a person's record from a given *phonenumber*.

The following functionality is tested in the applications using STL:

- To search a record of a *person* in the existing phonebook. It returns an iterator to that object if the given *phonenumber* is found, then I can get the record of that person.

- To insert a record of a person into the *phonebook* and provide confirmation whether the record is inserted. I can also later use search to verify if the record is correctly inserted.
- To delete a record of a *person* from the existed *phonebook* and check whether it can be correctly removed from the *phonebook*.

3.6 Comparing Time and Space Efficiency

In order to analyze the time and space performance of data structures when building and accessing a large tree and a list in both STL and OrgC++, I compare the results in STL to equivalent results in OrgC++. I used the following code to create a random number generator. It generates some large random numbers. I use them as records for the phonebook. The large number is also used as the number of nodes for building trees and lists and the number of times to access trees and lists.

```
long unsigned Functor::random_number()
{
   ran = (ran * multiplier)%module;
         if (ran <1000000)
            ran += 3000000;
         return ran;
}
```

3.6.1 Calculating Time

I use the function *_ftime()* to get the current time of the system: to measure the time for building and accessing trees and lists.

To get the build time of a tree or a list, I call *_ftime()* right before and right after building the tree or the list. Then I calculate the time difference between the two calls of *_ftime()*. Similarly, I get the access time of a tree or a list by calling *_ftime()* right before and right after accessing the tree or the list for n times.

3.6.2 Calculating Space

I use *sizeof*(*this) to measure the space needed for each tree node or list node. I implement a *size()* function in each full node class and calculate the space needed for an entire tree or list.

3.6.3 Definition of Time and Space Ratio

In order to compare time and space performance of data structures in each library easily and clearly, I define time and space ration as following:

Given a number n, I define

$$\textbf{BuildTimeRatio (n)} = \frac{\text{Time to build a data structure with nodes using STL}}{\text{Time to build a data structure with nodes using OrgC++}}$$

$$\textbf{AccessTimeRatio (n)} = \frac{\text{Time to access a data structure with n nodes 10000 times using STL}}{\text{Time to access a data structure with n nodes 10000 using OrgC++}}$$

$$\textbf{SpaceRatio (n)} = \frac{\text{Space needed to build a data structure with n nodes using STL}}{\text{Space needed to build a data structure with n nodes using OrgC++}}$$

I use *BuildTimeRatio (n)*, *AccessTimeRatio (n)*, and *SpaceRatio (n)* to compare the time and space efficiency of using STL and OrgC++. If the ratio is less than1, it means that using STL requires less time or space than using OrgC++.

I use 10000 loop iterations because I want to make the comparison of the performance of two implementations in the same level. Although it takes a long time to access data in the "Open" implementation, it is almost instant in "Closed" implementation if the number of the loop times used is small.

Chapter 4 Implementation

In this section, I will give the implementation details of the phonebook in different libraries. In order to show how the implementation can be generalized to work with any data type without conflicts between STL (Standard Template Library) and OrgC++ (C++ Data object Library) in this project, all the classes head file names in STL start with *'stl'* and in OrgC++ start with the *'orgc'*.

There are three main classes defined through this application.

- *Class Functor*: It defines a random number generator. It is used to initialize a stream by a given *seed.*
- *Class Person*: It defines a person's *firstname*, *lastname* and *phonenumber*. *Class ORGCPerson* is used as a template parameter type in the file *ORGCphonebook.cpp*.
- *Class PersonHasIt*: It defines the function that is used to get a person's record by a given *phonenumber*.

For more implementation details of the classes, please refer to *functor.h*, *person.h* and *personhasit.h* on page 50 to 57.

4.1 Environment

The different libraries have been installed on Windows 95. The tests have been carried out on Windows95 using Microsoft Visual C++ 6.0.

4.2 "Open" Implementation Method

The Standard Template Library is a C++ library of container classes, algorithms, and iterators. Container classes are very useful classes of data abstractions in the STL. A container contains values of some kind or references to values of some kind. For example, *list* is a kind of container and *set* is another kind of container. The difference between a list and a set is that a list imposes a physical, though not necessarily a logical ordering of the elements it contains, and a set imposes nothing on the values it contains other than the fact of containment. STL also includes a large collection of algorithms that manipulate the data stored in containers.

Another important data abstraction provided by the Standard Template Library are called iterators. They are used to refer to the individual elements of containers and to provide the means of applying operations to the contents of containers.

From the previous chapter 3.4.1, I know that a key factor in STL design is the consistent use of iterators. Pointers themselves are iterators. So what is the purpose of iterators? In the STL, very few of the algorithms needed to manipulate an abstraction are implemented in the class corresponding to that abstraction. Instead, the class defines functions that make certain information about the abstraction - here a list - available in such as way that I can implement any needed algorithm without modifying the structure itself. In fact, it is possible to build such algorithms in such as way that they work with other data abstractions as well as the one for which they were originally designed. This is the purpose of iterators.

In terms of inheritance, the Standard Template Library does not depend heavily on object-oriented features of C++. Relatively little inheritance is involved in this library. In contrast, other libraries use inheritance extensively, some to the extent that every class is derived from a common base class. In fact, some other object-oriented languages (Smalltalk, Modula-3, Java) make this a requirement.

4.2.1 Implementing a Doubly Linked List

In this section I shall look at an implementation of *a doubly linked list* that uses two pointers in each node as shown in figure 1. *A doubly linked list*, is a sequence that supports both forward and backward traversal, and provide constant time insertion and removal of elements at the beginning or the end, or in the middle.

Figure 1 shows the structure of Lists in the STL.

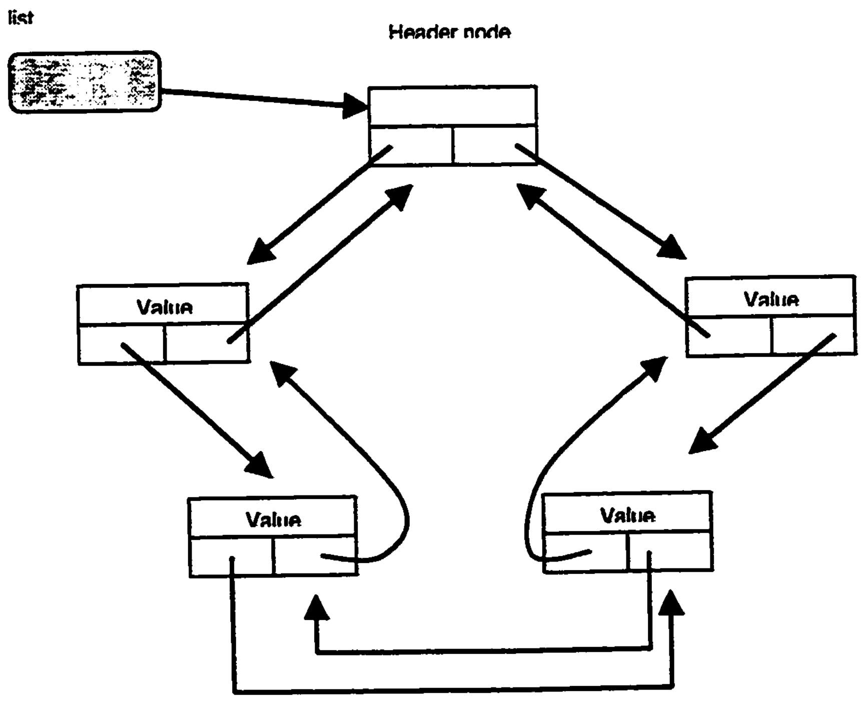


Figure 1: Lists in STL

In the project, when a phonebook is created, I use container class List:

```
list<Person> phoneBook;
list<Person>::iterator phoneBookIterator;
```

I also define phonebookIterator.

As we know iterators are the mechanism that makes it possible to decouple algorithms from containers; algorithms are templates, and are parameterized by the type of iterator, so they are not restricted to a single type of container.

Hence, in the project, when I search a record of a person in the existing phonebook, I use the STL's *find_if()* algorithm to perform this search through the phonebook list. It shows:

```
PhoneBookIterator = find_if(phoneBook.begin(), phoneBook.end(), personHasIt(pn));

if(phoneBookIterator==phoneBook.end())
   cout << "Record not found in list" << endl;

  else
        cout << *phoneBookIterator << endl;
```

It completes the first time *PersonHasIt()* returns true for any object. It returns an iterator to that object, or if *PersonHasIt()* never returned true, *find_if()* returns *end()*.

To insert a record of a person into the phonebook and to provide confirmation to check whether the record is inserted, an *Insert iterator* is used to implement this. Here is the definition of insertion into a container by means of:

```
phoneBook.insert(phoneBook.end(), aPerson);
```

Its effect is to insert the record of *aPerson* immediately before the *iterator phoneBook.end()*.

To delete a record of a person from the existing phonebook and check whether it can be correctly removed from the phonebook. I use STL's algorithm *remove()* to perform this function:

```
if (phoneBookIterator == find_if(phoneBook.begin(), phoneBook.end(),
PersonHasIt(pn)))

        phoneBook.remove(aPerson);
```

Remove removes from the range [*phoneBook.begin(), phoneBook.end()*] all elements that are equal to the value of *PerHasIt(pn)*.

4.2.1 Implementing a Binary Search Tree

A tree structure is used to store data in sorted order. It consists of nodes that contain data. Nodes have links to other nodes. Any tree has a single root node to which all other nodes are linked. A node data must have a key which can be compared and sorted. *A binary tree* is the simplest type of a tree structure; a *binary search tree* is a particular kind of binary tree. Each node in a binary search tree contains data and two links. The left link connects to all nodes with lesser data values; and the right link connects to all nodes with greater data values. If one datum is greater or less than another datum it can be determined by comparing the key values of the two data.

In STL, I use *Set* to represent tree to compare to the *Tree* organization in OrgC++. Set is one of the associative Containers in STL. Also STL sets are always sorted and they are implemented using tree structures. In this project, I use a binary search tree. It holds data that can be compared with something like *operator* <. The build-in types of C++ have such a comparison, though the comparison for pointers to strings (or indeed any pointers) is quite meaningless. Therefore, the STL permits the user to define alternate comparison operations using function objects. A comparison object is either a binary function returning *bool*, or an object in a class that has such an *operator()* defined.

In the project, I defined the following comparison function to compare two nodes:

```
class PersonLess
{   bool operator() (const Person& x, const Person& y) const {
        return x.getPhonenumber() < y.getPhonenumber();
        }
}
```

4.3 "Closed" Implementation

In the C++ Data Object Library (OrgC++), all organizations are based on a ring-type arrangement, not on a NULL-ending list.

4.3.1 Implementing a Doubly Linked List

In order to make a comparison to STL list, I can use *double_ring* organization to represent doubly linked list because they behave similarly.

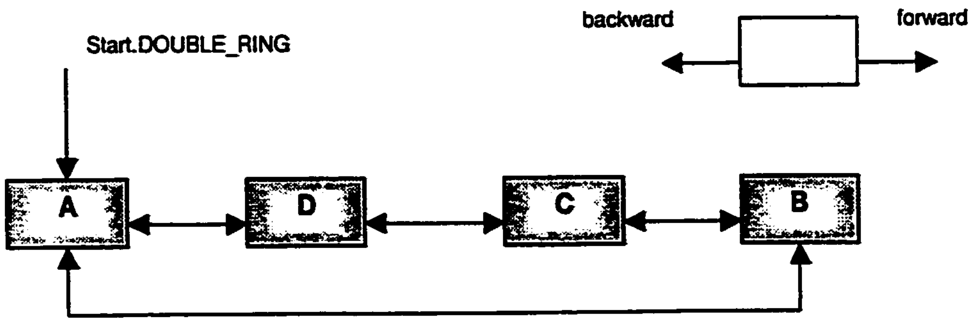


Figure 2: DOUBLE_RING in OrgC++

As I described in the previous chapter 3.4.2, OrgC++ is based on new concepts of Hyper-objects which are objects that store their data in other objects, called carriers. Carriers passively keep the data but hyper-objects provide the methods.

OrgC++ uses generic functions. For example, *add()* can be used to add an object to a ring, to a tree, or to a graph, without loosing the advantage of full type checking. *del()* for deleting (disconnecting) an object. In OrgC++, a ring (or circular list) is a structure existing on a set of objects without any start/tail pointer encapsulated in a special class. Since the entry to the ring is not encapsulated I have to keep it externally, otherwise the ring would be there, but I would not know how to get to it. The entry is also important if I am concerned about the order of the objects in the ring. The entry to the ring will be returned as the last element when traversing the ring.

Here I defined a phonebook using *double_ring* organization.

ZZ_HYPER_DOUBLE_RING(phoneBook,Person); // declare a DOUBLE_RING

Before I start to use this ring, I have to set *entry=NULL.*

*Person *nstart=NULL;* // initialize start before using the DOUBLE_RING.

I also define *Person* phoneBook*_ptr. OrgC++ doesn't provide operation *find_if()* in DOUBLE_RING, in order to compare to STL algorithm *find_if()*, here I also define it as the following:

```
Person* find_if(PersonHasIt& another_person) {
        Person *temp;
        int count = 0;
  ZZ_A_TRAVERSE(phoneBook,nstart,temp){
                if(another_person(*temp)) break;
  }ZZ_A_END;
        return temp;
}
```

Then when I search a record of a person in the existed phonebook, I defined:

```
    phoneBook_ptr = find_if(PersonHasIt(pn));

        if(!phoneBook_ptr)
            cout << "Record not found in list" << endl;

        else
              cout << *phoneBook_ptr << endl;
```

To insert a record of a person into the phonebook and provide confirmation to check whether the record is inserted. I use OrgC++ generic function *add()*:

```
        phoneBook.add(nstart, aPerson);
```

Adds new object, a record of *aPerson* to the ring with entry point *nstart* and returns a new entry.

I also use OrgC++ generic function *del()* to move a record of a person from the existing phonebook and check whether if it can be correctly removed from the phonebook.

```
    if(phoneBook_ptr = find_if(PersonHasIt(pn)))
       {
         phoneBook.del(nstart,phoneBook_ptr);
               delete(phoneBook_ptr);
       }
```

4.3.2 Implementing a Binary Search Tree

In OrgC++, I use *DOUBLE_TREE* to compare to the STL *Sets* as illustrated in figure 3

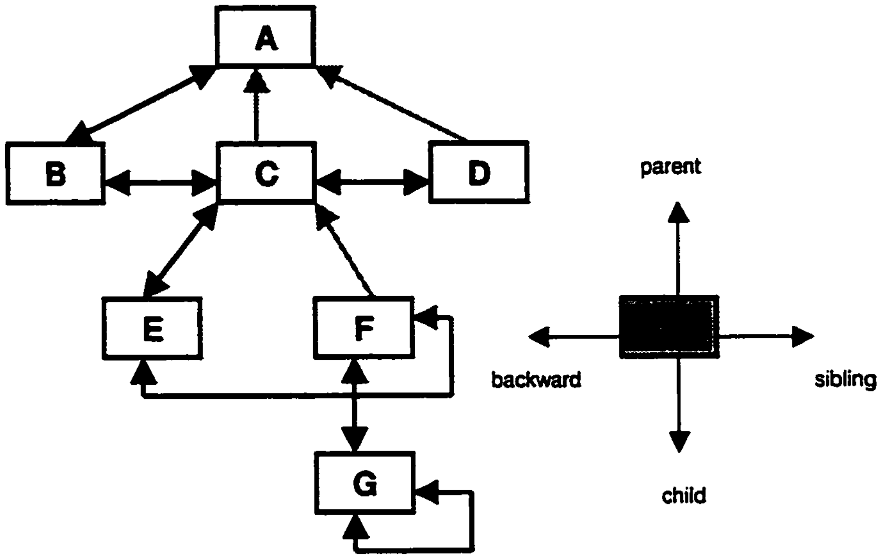


Figure 3: DOUBLE_TREE in OrgC++

Here I defined a phonebook using *DOUBLE_TREE* organization:

```
ZZ_HYPER_DOUBLE_TREE(phoneBook,Person);
Person *nstart=NULL;
```

Because OrgC++ doesn't contain *find_if()* algorithm, as I described in 4.3.1, I therefore, have to define *find_if()* here to search for a person's record.

```
Person* find_if(PersonHasIt& another_person) {

        Person *tmp_parent = nstart;
        Person *first_child, *second_child;

        while (tmp_parent) {
          if (another_person(*tmp_parent)) break;
           /* No child */
           if (!(first_child=phoneBook.child(tmp_parent))) {
                tmp_parent = NULL;
                break;
                }
           /* One child */
          else if ((second_child=phoneBook.fwd(first_child))==first_child) {
         if ((another_person.getPhonenumber()>=tmp_parent->getPhonenumber())
                &&(first_child->getPhonenumber()<tmp_parent->getPhonenumber()))
                {
                        tmp_parent = NULL;
                        break;
                } else
                    if ((another_person.getPhonenumber()<tmp_parent->getPhonenumber())
                        &&(first_child->getPhonenumber()>=tmp_parent->getPhonenumber()))
                        {
                                tmp_parent = NULL;
                                break;
                        } else
                                tmp_parent = first_child;
                }
           /* Two children */
                else {
                        if (another_person.getPhonenumber()>=tmp_parent->getPhonenumber())
                                tmp_parent = second_child;
                        else
                                tmp_parent = first_child;
                }
        } /* while loop */
        return tmp_parent;
}
```

Then I can use *find_if* to search for a person record of the phonebook using:

```
phoneBook_ptr = find_if (PersonHasIt(pn));

if (!phoneBook_ptr) {
   cout << "Record not found in list" << endl;
  }
  else {
        cout << *phoneBook_ptr << endl;
        }
```

To remove a record of a person from phonebook, here I also defined *phoneBookdel():*

```
void phoneBookdel(Person* del_person) {

        phoneBook.del(del_person);
        if (del_person==nstart) nstart=NULL;
        resortAllChildren(del_person);
        delete del_person;
}
```

As seen in figure 3, all the children of any node form a RING. The child pointer from the parent represents the start pointer of the RING. Repeated use of *add()* loads the nodes in reverse order. When I insert a record of a person into the phonebook, the following OrgC++ contained operations are used to define *phoneBookadd()*.

Add()	adds a new child;
App()	appends a new sibling to the right of a given node.
Ins()	inserts a new sibling to the left of a given node

Here I defined *phoneBookadd()* to add a node into the phonebook.

```
void phoneBookadd(Person* new_person) {
        Person *tmp_parent = nstart;
        Person *first_child, *second_child;

        if(!nstart) {
                //phoneBook.add(nstart,new_person);
                nstart = new_person;
        }

        while (tmp_parent) {

          /* No child */
          if(!(first_child=phoneBook.child(tmp_parent))) {
                phoneBook.add(tmp_parent, new_person);
                return;
        }

          /* One child */
          else if((second_child=phoneBook.fwd(first_child))==first_child) {
                if((new_person->getPhonenumber()>=tmp_parent->getPhonenumber())
                   &&(first_child->getPhonenumber()<tmp_parent->getPhonenumber()))
                     {
                        phoneBook.app(first_child, new_person);
                        // appends a new sibling to the right of a given node.
                        return;
                     } else
                        if((new_person->getPhonenumber()<tmp_parent->getPhonenumber())
                        &&(first_child->getPhonenumber()>=tmp_parent->getPhonenumber()))
                        {
                                phoneBook.ins(first_child, new_person);
                                return;
                        } else
                                tmp_parent = first_child;
          }

                /* Two children */
                else {
                        if(new_person->getPhonenumber()>=tmp_parent->getPhonenumber())
                                tmp_parent = second_child;
                        else
                                tmp_parent = first_child;
                }

        } /* while loop */
}
```

4.4 User Interface

In order to compare the selected data structures easily in STL and OrgC++, I originally wanted to use Microsoft Foundation Class (MFC) as an "application framework" to design and implement Windows Style User Interface for the project. Because MFC is accepted and used by many professional programmers, it's very natural for Windows to have a C++ programming interface. MFC Library application framework includes many general-purpose classes such as collection classes for lists, arrays and maps.

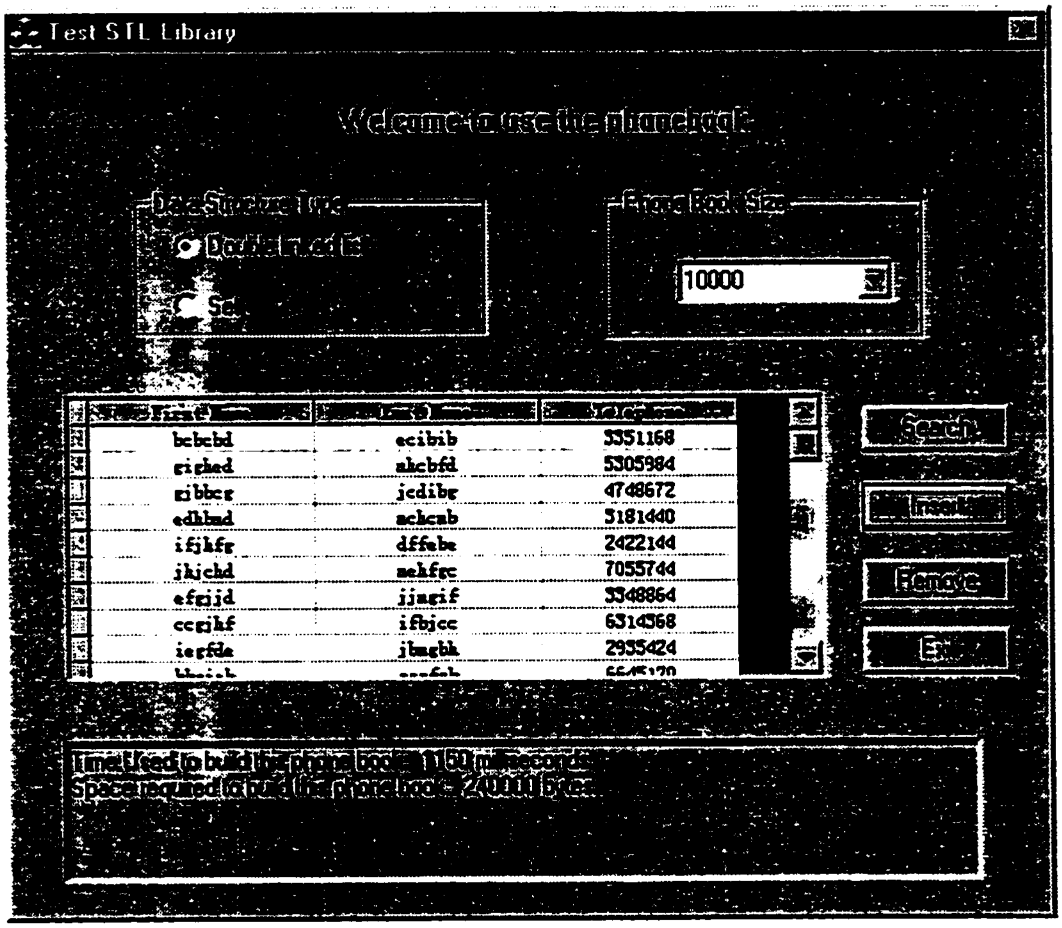
I tried using MFC to design UI for both implementations in the project, but because of difficulties with OrgC++ class generator, I decided against it although a lot of time was spent in the consideration. From the perspective of the main purpose of this project, the interface is not a critical factor in the completion of the project to compare data structures in STL and OrgC++. Therefore, I used MFC Library to implement UI for "Open" implementation only for STL, and UI for OrgC++ interactive windows under MS-DOS was used.

4.4.1 Suspected Problems with OrgC++

The problem is that OrgC++ consists of a class generator and a special library. The class generator typically just reads the header file with the declarations of objects and organization, and it creates two files: an include file and a source file with all the access routines required by the program. These two files are used to compile and link the original source code. Hence, when modifying data objects, I have to call the class generator every time before re-compilation. All used classes have to be modified when using OrgC++; and in addition, might involve changes to the MFC base classes.

4.4.2 UI for "Open" Implementation in STL

Figure 4 below shows the main interface of the Phone Book system using STL.



10000

First Name	Last Name	Telephone
bcbcbd	ecibib	5351168
gighed	ahcbfd	5305984
gibbcg	jcdibg	4748672
edhbad	achcab	5181440
ifjhfg	dffabe	2422144
jhjchd	achfgc	7055744
efgijd	jjagif	5348864
ccgjhf	ifbjcc	6314368
iegfda	jbagbh	2935424
[illegible]	[illegible]	[illegible]

Time Used to build the phone book = 1150 milliseconds
Space required to build the phone book = 240000 bytes

Figure 4: The Main Window of PhoneBook in STL

As shown in Figure 4, I can select a data structure using either doubly linked list or set and also select a number of records (nodes) to generate a phonebook.

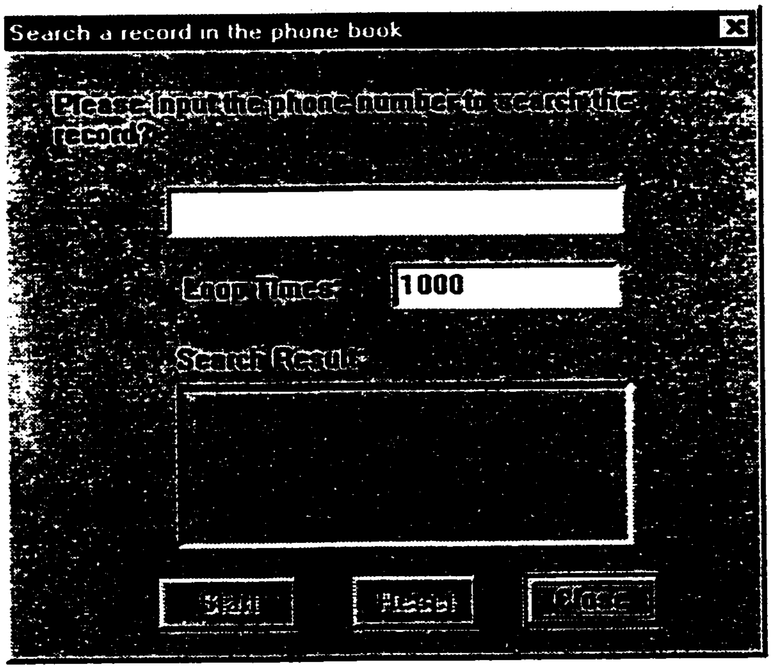
This main window also shows two results:

- The time used to build this phonebook in milliseconds
- The space required to build this phonebook in bytes

In addition there are four buttons on the right side of the screen. Any one of the following options can be selected :

- Search: to search a record of a person in the phonebook.
- Insert: to insert a record of a person into the phonebook.
- Remove: to delete a record of a person from the phonebook.
- Exit: to exit the phonebook system.

The following sub-window will be popped up after pressing the Search button.


Search a record in the phone book

Please input the phone number to search the record?

Loop Times: 1000

Search Result:

FIGURE 5: UI for Searching a record in STL

This sub-window allows us to input a phone number to search for a record of a person whose phone number is the input phone number. It also shows the loop times when I perform a search function. The search result will also show in the box after I press the start button. In each sub-window I have three buttons to select: Start, Reset and Cancel. Start and Reset buttons are disable in the initial sub-screen but are activated if you input a phone number in the box field:

- Start: to start search / insert / remove a record in / to / from a phonebook
- Reset: to reset all data in the screen if I have done a search.
- Close: to exit this screen and go back to the main window of the application

If I press the insert button, the following window is shown. It is similar to the sub-screen of the search screen except there are three box fields for us to input a new person's first name, last name, and phone number into the existing phonebook.

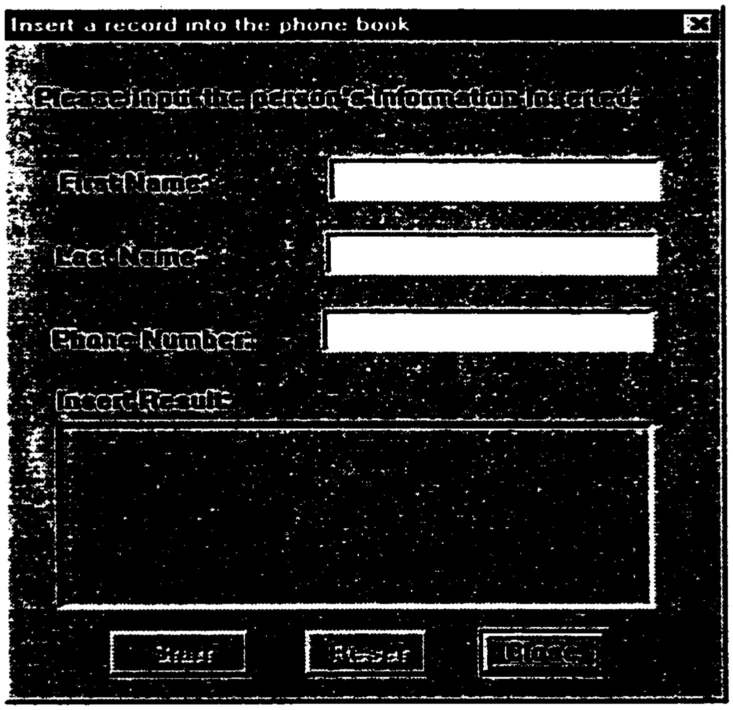

Insert a record into the phone book
Please input the person's information inserted:
First Name:
Last Name:
Phone Number:
Insert Result:
Start
Reset
Close

Figure 6: UI for inserting a record in STL

If I press the Remove button, it will bring us to this screen:

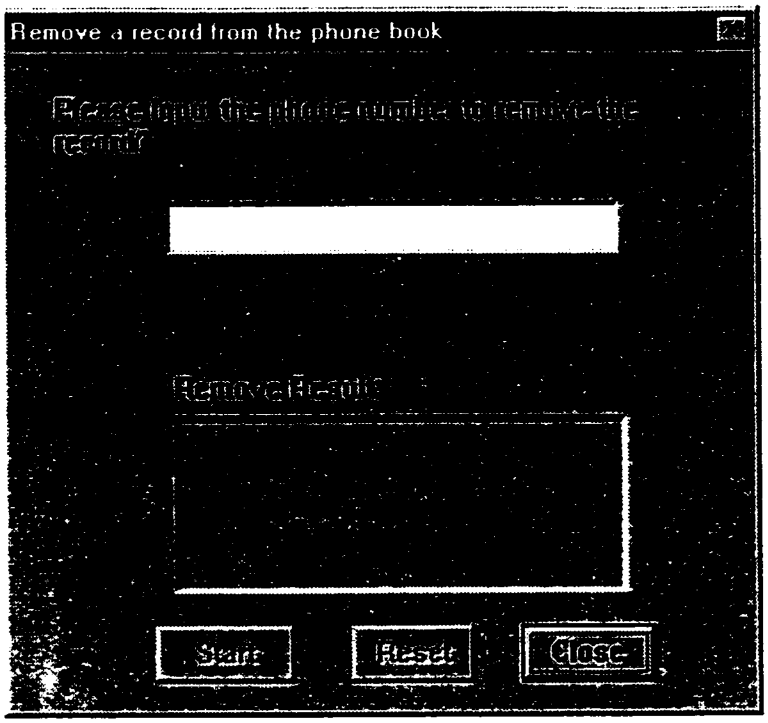
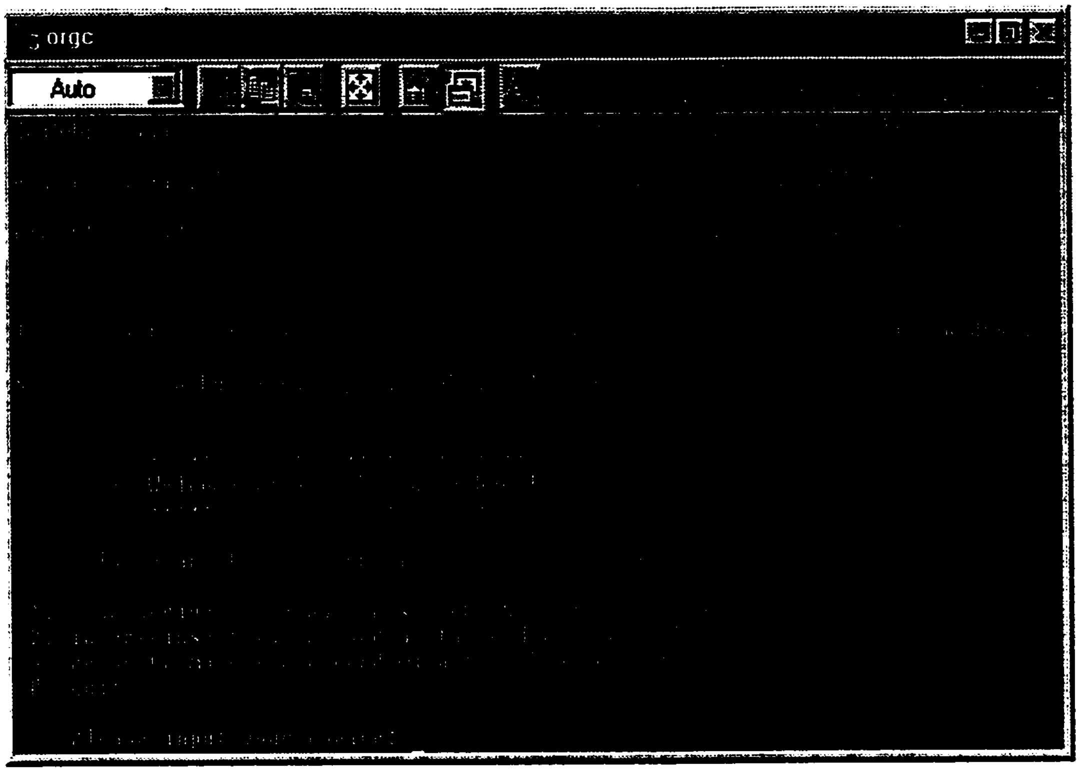

Remove a record from the phone book
Please input the phone number to remove the record?
Remove Result
Start
Reset
Close

Figure 7: UI for deleting a record in STL

It is very similar to the Search sub-screen. (See Figure 5) It also shows the *remove result* after I press the Start button on this screen.

4.4.3 UI for "Closed" Implementation in OrgC++

I described the problems with OrgC++ in the section 4.2.1 and, therefore, only show the interactive windows for this implementation. See Figure 8.



[illegible]

Figure 8: UI in OrgC++

As we can see, four selection options are provided. I can key in the number of the selected option and press *Enter* key to show the result. Actually it is similar to the four buttons shown on the main window of the "Open" implementation (See figure 4).

Chapter 5 Results

In this section I evaluate and analyze the performance of both implementations of data structures considered. I also present the results of each experimental test. It is important to note that the results in this section reflect the performance of a particular implementation of the data structures and do not necessarily reflect performance limitations that exist in the data structures independent of these implementations in each library.

According to the evaluation criteria defined in the section 3.1, here I present the results for each item.

5.1 Usability - Ease of Coding

After I experienced implementation of the data structures in both "Open" and "Closed" implementations, I found both STL and OrgC++ Libraries provide sets of classes. There are some differences.

- Difference in class declaration

  In STL: can define a class just as I do in C++ to define all data and member functions of a class.

  In OrgC++: when you declare an object (STL called class), only the attributes that do not relate to the organization of data are declared. Objects which will be involved in the automatic handling of data must contain a line of the form

ZZ_EXT_<objectType>. This statement marks the place where the transparent pointers will be inserted.

Here is how I defined a class Person in both implementations.

In STL	In OrgC++

class PersonHasIt
{
public:
  PersonHasIt(); // constructor
  PersonHasIt(int& phoneNumber);
  // person has the phonenumber.
 	
  bool operator() (Person& aPerson); /
 	
private:
  int number;
};
	class PersonHasIt
	{
	
	ZZ_EXT_PersonHasIt
	
	Public:
	  PersonHasIt(); // constructor
	  PersonHasIt(int& phoneNumber);
	  // person has the phonenumber.
	
	  bool operator() (Person& aPerson);
	
	private:
	  int number;
	};

- STL contains more generic algorithms.

STL provides *find_if()* algorithm. I can easily use it to search for a record in the phonebook. OrgC++ doesn't contain this function so I have to define a function called find_if() in the program. Then I can use it to perform a search as I would with STL. To see the implementation of this, please refer to section 4.3, on page 23.

- STL and OrgC++ have very similar interface attributes.

They both have similar iterators and similar functions that manipulate the data structures but they are not identical. In STL, data and their relations are interwoven with the classes that carry the data. In OrgC++, data and their relations are orthogonal.

The following shows the declaration of a phonebook in each library:

In STL	In OrgC++
list<Person> phoneBook;	*ZZ_HYPER_DOUBLE_RING(phoneBook,Person);*

5.2 Time and Space

In this section I show the test results of building time ratio, access time ratio, and space ratio for "Open" and "Closed" implementations as I defined earlier in section 3.6.3. Each time the test results corresponding to different node numbers are generated.

5.2.1 Build Time Efficiency

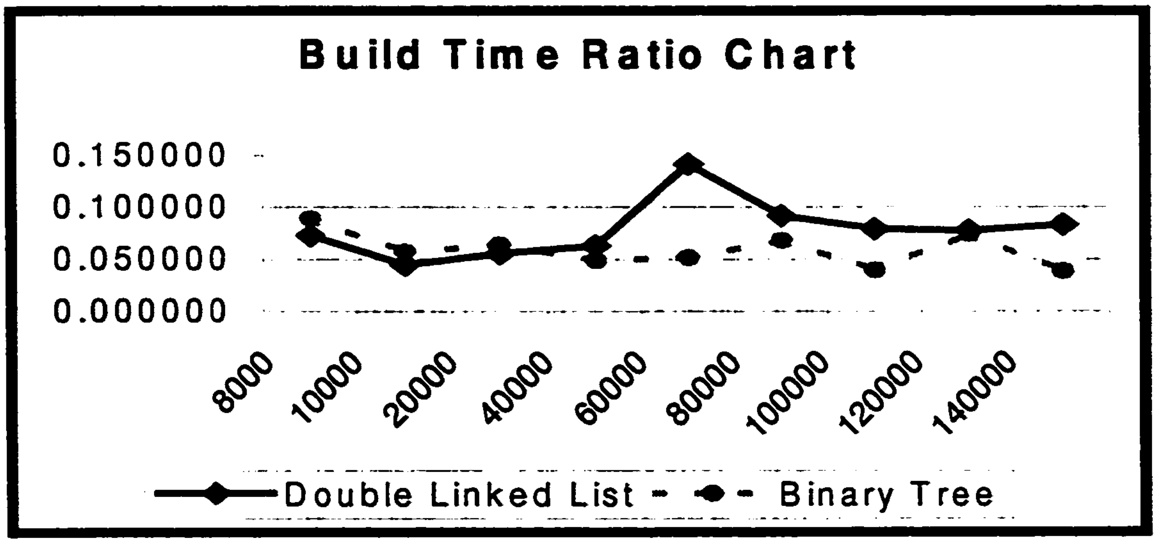
The build time ratios I got from both implementations are shown in Table 1. From the table, we can see that the results of build time ratio for doubly linked list and binary search tree structure are similar. They indicate that the time used to build a phonebook in the "Closed" implementation is greater than in the "Open" implementation no matter what data structure is selected.

The time range of building doubly linked list with nodes from 8000 to 140000:

• "Open" implementation:	440 ~ 10110 milliseconds
• "Closed" implementation:	6040 ~ 121390 milliseconds

Table 1: Build Time ratio

	Test No.	No.of Node	Double Linked List	Binary Tree
Build TimeRatio	1	8000	0.072848	0.088792
	2	10000	0.044534	0.056995
	3	20000	0.055423	0.063584
	4	40000	0.062898	0.048596
	5	60000	0.141103	0.051179
	6	80000	0.091810	0.068187
	7	100000	0.078645	0.039478
	8	120000	0.077346	0.073972
	9	140000	0.083285	0.038643

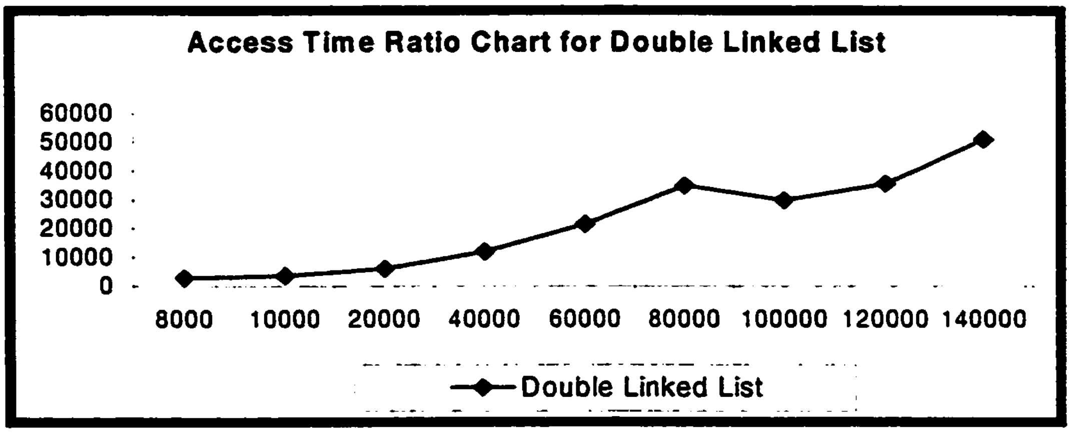


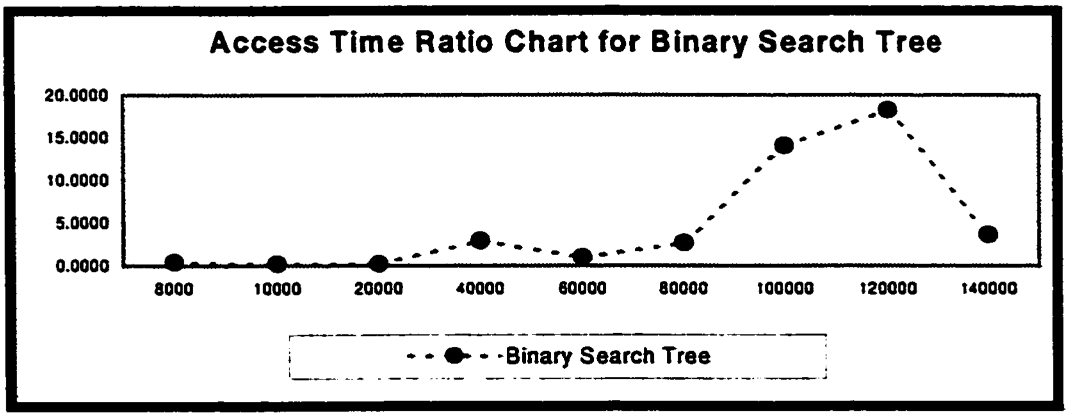
5.2.2 Access Time Efficiency

In this section, I show the results of time ratios in the following table:

Table 2: Access Time Ratio

	Test No.	No. of Nodes	Double Linked List	Binary Search Tree
Access Time Ratio	1	8000	2870.400000	0.350649
	2	10000	3555.800000	0.182796
	3	20000	5959.500000	0.187500
	4	40000	11923.500000	2.941176
	5	60000	21483.600000	0.940299
	6	80000	34925.000000	2.640884
	7	100000	29763.166667	14.060000
	8	120000	35703.500000	18.227273
	9	140000	51062.000000	3.638498





From the previous section 5.2.1, we know that it takes more time to build a phonebook with "Closed" implementation. You may then wonder about access time. The test results are surprising as shown in Table 2.

The raw data, that is what I actually measured, is shown in Table 3.

Table 3: Access Time

	Test	No.	STL		OrgC++	
	No.	of Nodes	list	ring	set	tree
Access Time	1	8000	143520	50	270	770
	2	10000	177790	50	170	930
	3	20000	357570	60	330	1760
	4	40000	715410	60	500	170
	5	60000	1074180	50	1260	1340
	6	80000	1746250	50	9560	3620
	7	100000	1785790	60	7030	500
	8	120000	2142210	60	8020	440
	9	140000	2553100	50	7750	2130

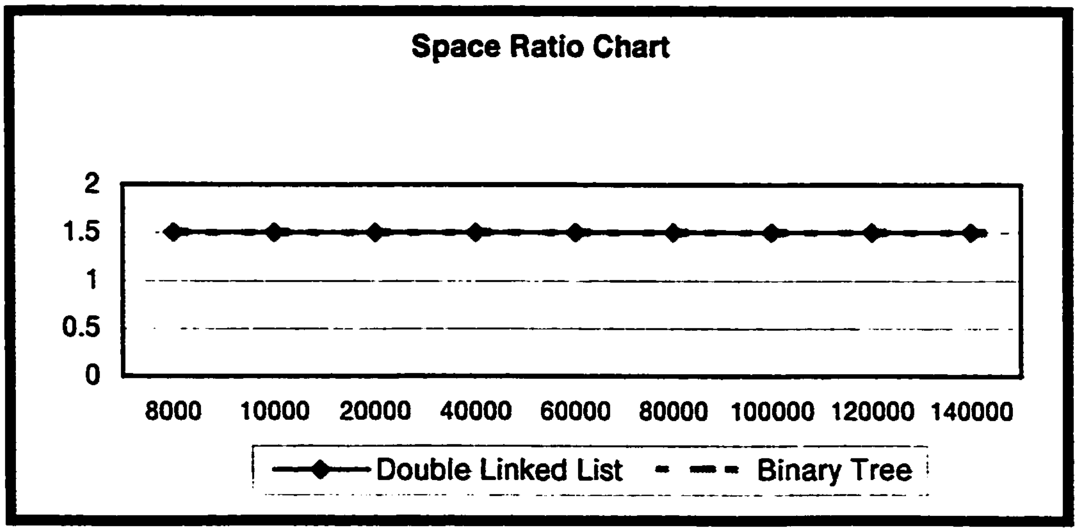
From the test results we see it required much longer to search for a record in the phonebook using doubly linked list in the "Open" implementation. It appears that accessing data is very efficient in the "Closed" implementation. The results of access time ratio for the binary search tree are very close. The "Open" implementation method is slightly slower than the "Closed" implementation method. Why are the results so different? The exact cause is unclear, but it is probably due to differences in the organization of STL and OrgC++ data structures. All organizations in OrgC++ are based on a ring-type arrangement, not on a NULL-terminated list. Another possibility is that OrgC++ reorganizes the list so that, when a node is accessed, it becomes the first node of the list. In addition, the allocation of links for STL lists may create lot of overhead because traversing lists in STL requires an additional pointer jump and periodic reallocation and copying of the links.

5.2.3 Space Efficiency

I show the test results of space ratio in this section. From the following table 4, we can see that the space ratios of all the data structures I implemented are 1.5. The required space to build a phonebook in each implementation with both selected data structures are the same, hence, the space efficiency of the two implementations are the same.

Table 4: Space Ratio

	Test No.	*No. of Nodes*	*Double Linked list*	*(Set) Binary Tree*
Space Ratio	1	8000	1.5	1.5
	2	10000	1.5	1.5
	3	20000	1.5	1.5
	4	40000	1.5	1.5
	5	60000	1.5	1.5
	6	80000	1.5	1.5
	7	100000	1.5	1.5
	8	120000	1.5	1.5
	9	140000	1.5	1.5



From the table, it also shows more space was required in the "Open" implementation than in the "Closed" implementation.

5.3 Interaction with MFC

As described in section 4.4, at the beginning of design phase, I wanted to use MFC for both implementations to create UI but there are some problems with OrgC++ so I had to give up using MFC for "Closed" Implementation.

5.4 Other Comparisons

I also did some comparisons of other criteria which I defined in the evaluation criteria in Chapter 3. Here, I show the summarized results of comparison on the two Libraries in the following table.

Table 5: Comparison of the two Libraries in other criteria

Compare criteria	**Standard Template Library**	**C++ Data Object Library**
Is it shareware or commercial?	The standard Template Library is one of the standardized components of the C++ language.	It is commercial. Full source of Data Object Library is not available free at this time.
Are there problems to compile on PC?	It's quite easy to compile.	It's not easy to compile. I had a lot of problems to compile it on my PC at the beginning.
Is it supported by the vendor/developer?	Yes.	No. Free binary version of the library does not include any support or guidance.
Is it easy to install?	No worry about installation. It comes with Visual C++.	No. It took a quite of time to install it on my PC.
Is it a suite of tests?	Yes.	Yes
Is it easy to crash when doing a large search?	No	It crashed a few times when doing a large search.

Chapter 6 Conclusions

In this chapter, I conclude the experimental results and summarize the advantages of using STL and OrgC++. I also discuss the research of comparison performance of data structures for further directions in which the project could be taken.

In summary, this project is an experiment of exploring the performance of data structures in different implementation methods. Although the application and data structures selected are not considered perfect, it may provide us some interesting findings on data structures for future research since there is not much published in this area.

Through the project, I know the "Closed" implementation provides a new way to look at pointer problems. It uses a code generator to build a meta model of all application classes and it inserts the required pointers using generated macros, not inheritance, This is a dirty but efficient method which results in faster code but is not as elegant as templates. Compared to the "Open" implementation method, the phonebooks are easy to implement using templates, and they form the basis of almost every existing container class library.

6.1 The advantages of STL and OrgC++

I have implemented our Phonebook in both libraries, there are some different features in each of them:

<u>Standard Template Library</u>

- It's easy to use. It has an elegant, consistent, and easy to comprehend architecture. The different components of the STL can be plugged together which makes the programmer more productive.
- STL algorithms are generic. They work on a variety of containers and even on ordinary C++ arrays.
- The use of STL is likely to make software more reliable, more portable and more general and will reduce the cost of producing it.
- It interacts well with other libraries such as MFC.

<u>C++ Data Object Library</u>

- Data structures can be protected against most pointer errors by initializing all pointers to NULL, and setting them back to NULL when the object is disconnected from the data structure, because all organizations are based on a ring-type arrangement, not on a NULL-terminated list. An object cannot be destroyed before all its pointers are NULL.
- It makes the data structures automatically persistent by taking Code generator to find the locations of all the pointers, and without the user coding and supporting serialization functions for every class.
- It is possible to perform a fast run-time check for the integrity of the list.
- It is more efficient sometimes to do random search, insert or remove operations from a doubly linked list.

6.2 Future Work

There is not much work published in this area. The study of data structures in different C++ implementation methods has generally been neglected. Therefore, there is the opportunity for much more research in this area.

In this project, I presented the design and implementation of a phonebook system with both "Closed" and "Open" implementations in Standard Template Library and C++ Data Object Library. Although the implementation was for only two libraries and for a specific application, there are some findings recommendations for future work to expand the research.

- Select more libraries other than STL and OrgC++ such as LEDA, Rogue Wave, etc.
- Design and implement other applications to explore the performance of data structures.
- Select more data structures other than doubly link lists and trees.
- Implement the application on other platforms such as UNIX, Windows NT, etc.
- Make comparisons using different criteria than were used here.

Chapter 7 Bibliography

A. Books and articles

[1] Joseph Bergin. *Data Structure Programming with The Standard Template Library in C++*. Springer-Verlag New York Berlin, Inc, 1998

[2] Ulrich Breymann. *Designing Components with the C++ STL: A New Approach to Programming*. Addison-Wesley, 1998

[3] Timothy A. Budd. *Multiparadigm Programming in LEDA*. Addison-Wesley Publishing Company, Inc., 1995

[4] Robert Robson. *Using the STL: The C++ Standard Template Library*. Springer-Verlag New York, Inc., 1998

[5] Leen Ammeraal. *STL for C++ Programmers*. John Wiley & Sons Ltd., 1997

[6] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991

[7] Mitchell L Model. *Data Structures, Data Abstraction - A Contemporary Introduction Using C++*. Prentice Hall, Inc. Englewood Cliffs, New Jersey. 1994

[8] Glenn W. Rol. *Introduction to Data Structures and Algorithms with C++*. Prentice Hall Europe, 1997

[9] Ian Sommerville. *Software Engineering*. Fifth edition. Addison-Wesley, 1995

[10] Cameron Hughes and Tracey Hughes. *Collection and Container Classes in C++*. John Wiley & Sons, Inc., 1996

[11] Frank M. Carrano. *Data Abstraction And Problem Solving With C++*. The Benjamin/Cummings Publishing Company, Inc., 1995

[12] J. P. Pardoe and M. J. King. *Object Oriented Programming Using C++*. Macmillan Press Ltd., 1997

B. Information on the web

[1] Jiri Soukup. C++ Data Object Library and Persistent Data. http://www.codefarms.com/

[2] LEDA. http://www.mpi-sb.mpg.de/LEDA/leda.html

[3] Standard Template Library Programmer's Guide. http://www.sgi.com/Technology/STL/

[4] Standard Template Library. http://www.cs.rpi.edu/~musser/stl.html

Appendix

A Stlfunctor.h

//stlfunctor.h: The declaration of class Functor in STL, it defines a random number
//generator. It is used to initilize a stream by a given seed.

```
#ifndef STLFunctor_H
#define STLFunctor_H

#include <iostream.h>
#include <list>

// constants used in random number generator.
const unsigned long int module = 10000000LU;
const unsigned long int multiplier = 169874;

//class Functor defines a random number generator.
class Functor
{
//private:
  unsigned long ran;            //used to represent the random data stream.

  unsigned long random_number(); //define function random_number().

public:
// constructor initilizes a data stream by a seed.
  Functor(int i): ran(i){}

//Get a random number.
  unsigned long getRandom() { return random_number(); }

//operator() overloading.
  unsigned long int operator()() { return random_number();}

 };

#endif STLFunctor_H
```

B Stlperson.h

// stlperson.h: the declaration of class Person in STL. It defines a person's firstname,
// lastname and phonenumber. Class ORGCPerson

```
#ifndef STLPerson_H
#define STLPerson_H

#include <iostream.h>
#include <list>

class Person
{
private:
  char *firstName;  //define person's firstname
  char *lastName;  //define person's lastname
        int  phoneNumber; //define person's phonenumber

public:
        Person();        // Constructor
        ~Person();       // Destructor
        Person(char *f, char *l, int i=0);

   const char* getLastname() const { return lastName; }
   const char* getFirstname() const { return firstName; }
//  void setLastname(l);
//  void setFirstname(f)
//  void setPhonenumber(i);

  int size();
  int getPhonenumber() const;            // Get a person's phonenumber.
  void setPerson(char *s1, char *s2, int i=0); // Set a person's record.

//operators
  friend ostream& operator<<(ostream& out, const Person& p);
        bool operator==(const Person& p2);
        bool operator!=(const Person& p2);
        bool operator<(const Person& p2);
        bool operator>(const Person& p2);

 };
```

```
class PersonLess
{
public:
        bool operator() (const Person& x, const Person& y) const {
#ifdef IND_NAME
                int temp = strcmp(x.getLastname(), y.getLastname());
                if ( temp < 0)
                        return true;
                else if (temp > 0)       return false;

                temp = strcmp(x.getFirstname(), y.getFirstname());
                if ( temp < 0)
                        return true;
                else return false;
#else
                return x.getPhonenumber() < y.getPhonenumber();
#endif
        }
};

#endif STLPerson_H
```

C stlpersonhasit.h

// stlpersonhasit.h: contains declaration of class ORGCPersonHasIt in OrgC++. It
// defines the function that is used to get a person's record by a given phonenumber.

```
#ifndef STLPersonHasIt_H
#define STLPersonHasIt_H

#include "stlperson.h"

class PersonHasIt
{
public:
 PersonHasIt(); // constructor
 PersonHasIt(int& phoneNumber); // person has the phonenumber.
 bool operator() (Person& aPerson);

private:
 int number;
};

#endif STLPersonHasIt_H
```

D Orgcfunctor.h

// orgcfunctor.h: The declaration of class class Functor in OrgC++, it defines a random
// number generator. It is used to initilize a stream by a given seed.

```
#ifndef ORGCFunctor_H
#define ORGCFunctor_H

#include <iostream.h>
#include "zzincl.h"

// constants used in random number generator.
const unsigned long int module = 10000000LU;
const unsigned long int multiplier = 169874;

//class Functor defines a random number generator.
class Functor
{

ZZ_EXT_Functor

//private:
  unsigned long ran; //used to represent the random data stream.

  unsigned long random_number(); //define function random_number().

 public:
// constructor initilizes a data stream by a seed.
  Functor(int i): ran(i){}

//Get a random number.
  unsigned long getRandom() { return random_number(); }

//operator() overloading.
  unsigned long int operator()() { return random_number();}

 };

 #endif ORGCFunctor_H
```

E orgcperson.h

```
//orgcperson.h: the declaration of class Person in OrgC++. It defines a person's
// firstname, lastname and phonenumber.

#ifndef ORGCPerson_H
#define ORGCPerson_H

#include <iostream.h>
#include "zzincl.h"

class PersonhasIt;

class Person
{
friend int phoneBooksize();
friend Person* find_if(PersonHasIt& another_person);

ZZ_EXT_Person

private:
  char *firstName;  //define person's firstname
  char *lastName;  //define person's lastname
        int  phoneNumber; //define person's phonenumber

public:
        Person();         // Constructor
        ~Person();        // Destructor
        Person(char *f, char *l, int i=0);

  const char* getLastname() const { return lastName; }
  const char* getFirstname() const { return firstName; }

  int GetSize();
  int getPhonenumber() const;           // Get a person's phonenumber.
  void setPerson(char *s1, char *s2, int i=0); // Set a person's record.

//operators
  friend ostream& operator<<(ostream& out, const Person& p);
        bool operator==(const Person& p2);
        bool operator!=(const Person& p2);
        bool operator<(const Person& p2);
        bool operator>(const Person& p2);
};

class PersonLess
{

ZZ_EXT_PersonLess;

public:
        bool operator() (const Person& x, const Person& y) const {
#ifdef IND_NAME
                int temp = strcmp(x.getLastname(), y.getLastname());
                if ( temp < 0)
                        return true;
                else if (temp > 0)      return false;

                temp = strcmp(x.getFirstname(), y.getFirstname());
                if ( temp < 0)
                        return true;
                else return false;
#else
                return x.getPhonenumber() < y.getPhonenumber();
#endif
        }
};

#endif ORGCPerson_H
```

F orgcpersonhasit.h

```
// orgcpersonhasit.h: contains declaration of class ORGCPersonHasIt in OrgC++. It
// defines the function that is used to get a person's record by a given phonenumber.

#ifndef ORGCPersonHasIt_H
#define ORGCPersonHasIt_H

#include "orgcperson.h"
#include "zzincl.h"

class PersonHasIt
{

ZZ_EXT_PersonHasIt

public:
  PersonHasIt(); // constructor
  PersonHasIt(int& phoneNumber); // person has the phonenumber.
  bool operator() (Person& aPerson);

private:
  int number;       // define phonenumber as integer.
 };

#endif ORGCPersonHasIt_H
```