4 Laboratory

Ordered List ADT

Objectives

In this laboratory you

- implement the Ordered List ADT using an array to store the list data items and use a binary search to locate data items.
- use inheritance to derive a new class from an existing one.
- explore a number of issues that relate to programming with inheritance.
- analyze the efficiency of your implementation of the Ordered List ADT.

ADT Overview

In an ordered list the data items are maintained in ascending (or descending) order based on the data contained in the list data items. Typically, the contents of one field are used to determine the ordering. This field is referred to as the key field, or the key. In this laboratory, we assume that each data item in an ordered list has a key that uniquely identifies the data item—that is, no two data items in any ordered list have the same key. As a result, you can use a data item's key to efficiently retrieve the data item from a list.

There is a great deal of similarity between the Ordered List ADT and the List ADT. In fact, with the exception of the insert, retrieve, and replace operations, these ADTs are identical. Rather than implementing the Ordered List ADT from the ground up, you can take advantage of these similarities by using your array implementation of the List ADT from Laboratory 3 as a foundation for an array implementation of the Ordered List ADT.

C++ Concepts Overview

Inheritance: Inheritance is the major new topic in this lab. C++ supports the ability for a class to extend another class's functionality. This is done by having the new class (the derived class) inherit the traits of the existing class (the base class) and adding new traits to the derived class. All of the following concepts relate to inheritance.

Access control: As a reminder, access to data and member functions from outside the class is governed by declaring them in one of three sections of the class: public, private, and protected. Items in the `private` section are not accessible to derived classes. Member functions and data in the `protected` section are accessible to derived classes, but are protected from any other access.

Virtual functions: To indicate which functions the base class intends to allow a derived class to override, the keyword `virtual` is placed in each function's signature in the base class declaration. It is also standard to use virtual in the derived class in order to 1) provide documentation, and 2) allow the derived class to be used as a base class for a third class.

Base class constructors: When a constructor of a derived class is called, the default behavior of the C++ compiler is to call the default constructor for the base class. This will often produce incorrect results, so a mechanism is provided to specify which base class constructor to execute.

Virtual destructors: If a class is designed to serve as a base class, its destructor should be declared virtual to allow any derived class's destructor to function correctly.

Ordered List ADT

Data Items:

The data items in an ordered list are of generic type DataType. Each data item has a key of the generic type KeyType that uniquely identifies the data item. Data items usually include additional data. Type DataType must provide a function called `getKey` that returns a data item's key.

Structure:

The list data items are stored in ascending order based on their keys. For each list data item *i*, the data item that precedes *i* has a key that is less than *i*'s key and the data item that follows *i* has a key that is greater than *i*'s key. The cursor in a nonempty list always marks one of the list's items. You iterate through the list using operations that change the position of the cursor.

Operations:

```
OrderedList ( int maxNumber = MAX_LIST_SIZE )
```

Requirements:
None

Results:
Constructor. Creates an empty list. Allocates enough memory for a list containing `maxNumber` data items.

```
~OrderedList ()
```

Requirements:
None

Results:
Destructor. Deallocates (frees) the memory used to store a list.

```
void insert ( const DataType& newDataItem ) throw ( logic_error )
```

Requirements:
List is not full.

Results:
Inserts `newDataItem` in its appropriate position within a list. If a data item with the same key as `newDataItem` already exists in the list, then updates that data item with `newDataItem`. Moves the cursor to mark `newDataItem`.

```
bool retrieve ( const KeyType& searchKey,
                DataType& searchDataItem ) const
```

Requirements:
None

Results:
Searches a list for the data item with key `searchKey`. If the data item is found, moves the cursor to the data item, copies it to `searchDataItem`, and returns `true`. Otherwise, returns `false` without moving the cursor and with `searchDataItem` undefined.

```
void remove () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Removes the data item marked by the cursor from a list. If the resulting list is not empty, then the cursor points to the data item that followed the deleted data item. If the deleted data item was at the end of the list, then moves the cursor to the beginning of the list.

```
void replace ( const DataType& newDataItem ) throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Replaces the data item marked by the cursor with `newDataItem`. Note that this entails removing the data item at the cursor inserting `newDataItem` at the proper location. Moves the cursor to `newDataItem`.

```
void clear ()
```

Requirements:
None

Results:
Removes all the data items in a list.

```
bool isEmpty () const
```

Requirements:
None

Results:
Returns `true` if a list is empty. Otherwise, returns `false`.

```
bool isFull () const
```

Requirements:
None

Results:
Returns `true` if a list is full. Otherwise, returns `false`.

```
void gotoBeginning () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Moves the cursor to the data item at the beginning of the list.

```
void gotoEnd () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Moves the cursor to the data item at the end of the list.

```
bool gotoNext () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
If the cursor is not at the end of a list, then moves the cursor to the next data item in the list and returns `true`. Otherwise, returns `false`.

```
bool gotoPrior () throw ( logic_error )
```

Requirements:
List is not empty.

Results:
If the cursor is not at the beginning of a list, then moves the cursor to the preceding data item in the list and returns `true`. Otherwise, returns `false`.

```
DataType getCursor () const throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Returns the value of the data item marked by the cursor.

```
void showStructure () const
```

Requirements:
None

Results:
Outputs the keys of the data items in a list. If the list is empty, outputs "Empty list". Note that this operation is intended for testing/debugging purposes only. It only supports keys that are one of C++'s predefined data types (`int`, `char`, and so forth) or for which the << operator has been overloaded.

Implementation Notes

Inheritance: The following declaration

```
class OrderedList : public List<DataType> {
    // Rest of derived class declaration here
}
```

indicates that OrderedList is derived from List. The keyword "public" specifies that this is a public inheritance—that is, OrderedList does not change List's protection mechanisms. Although rarely used, it is also possible to do "protected" or "private" inheritance, which do change the protection mechanism of the base class. We will always use public inheritance.

Access control: If you want the member functions in OrderedList to be able to refer to List's internal data members, List's data members must be declared in the protected section as shown below. If they were in the private section, OrderedList would not be able to access them.

```
class List
{
  ...

  protected:
    // Data members
    int maxSize,          // Maximum number of data items in the list
        size,             // Actual number of data items in the list
        cursor;           // Cursor array index
    DataType* dataItems; // Array containing the list data items
};
```

Virtual functions: Recall the class declaration of List. Remember that the `insert` and `replace` member functions are declared `virtual`.

```
class List
{
  public:
    ...
    // List manipulation operations
    virtual void insert ( const DataType& newDataItem )
                        throw ( logic_error );
    virtual void replace ( const DataType& newDataItem )
                         throw ( logic_error );
    ...
};
```

We also declare `insert` and `replace` virtual in the derived class declaration.

```
class OrderedList : public List<DataType>
{
  public:
    ...
    // Modified list manipulation operations
    virtual void insert ( const DataType& newDataItem )
                        throw ( logic_error );
    virtual void replace ( const DataType& newDataItem )
                         throw ( logic_error );
    ...
};
```

Note: in both the base and the derived class, we do use the `virtual` keyword only in the class declaration, not in the class implementation.

Multiple template parameters: Template classes can have multiple generic data types. In the OrderedList, the key type is a second generic type. The syntax is to add a second parameter to the template list. As an example, the insert function implementation would begin as follows.

```
template < typename DataType, typename KeyType >
void OrderedList<DataType,KeyType>::insert(const DataType& newDataItem)
         throw ( logic_error )
```

Note that the second template type appears in two places.

1. template <typename DataType, typename KeyType>
2. OrderedList<DataType,KeyType>::

Base class constructors: In a derived class constructor, you indicate which base class constructor to use through member initialization. The format is to identify the specific constructor by passing parameters that will match the correct base class constructor. For example, in the OrderedList constructor, we wish to pass the maximum size of the list to the List constructor. We do so as follows.

```
template < typename DataType, typename KeyType >
OrderedList<DataType,KeyType>:: OrderedList ( int maxNumber )
  : List<DataType>(maxNumber)
{
    // Empty body: all initialization is done by the base class
    // constructor
}
```

Templates and accessing base class members: Inheritance gives a derived class access to all base class member functions and data that are in the public or protected sections. In C++, references to base class members from the derived class are normally transparent. For instance, to reference the variable *size* located in the base class, code in the derived class just uses the name *size*, as though it were in the derived class. However, with templated classes, you need to prefix references to base class members with scope resolution information. So to reference the *size* variable, you would need to type the following every time you reference *size*.

```
List<DataType>::size
```

To avoid having to prefix every reference to *size* with the string "List<DataType>::", you can type the following once in the derived class declaration (in *OrderedList.h*) and then use *size* without scope resolution syntax.

```
using List<DataType>::size;
```

This must be repeated for each base class member that is to be referenced transparently within the derived class.

Using a key field: The following example illustrates how to declare an appropriate data type to be used with the ordered list. It meets the Ordered List `DataType` requirements because it has both a key field (`accountNum`) and a `getKey` function that returns the value of the key field. All other labs with a `KeyType` have the same requirements.

```
// lab4-example1.cpp

#include "OrderedList.cpp"
class Account {
  public:
      int accountNum;              // (Key) Account number
      float balance;               // Account balance
      int getKey () const
          { return accountNum; }   // Returns the key
};
int main()
{
    OrderedList<Account, int> accounts(20);  // List of accounts
    Account acct;                            // A single account
    // Rest of program processes accounts ...
}
```

The Account structure includes the `getKey` function that returns an account's key field—its account number. This function is used by the OrderedList class to order the accounts as they are inserted. Insertion is done using the OrderedList class `insert` function, but list traversal is done using the inherited List class `gotoBeginning` and `gotoNext` functions.

Step 1: Implement the operations in the Ordered List ADT using the array representation of a list. Base your implementation on the official declaration from the file *OrderedList.h*.

Note that you only need to create implementations of the constructor, insert, replace, and retrieve operations for the Ordered List ADT—the remainder of the operations are provided or are inherited from your array implementation of the List ADT from Lab 3. Your implementations of the insert and retrieve operations should use the `binarySearch()` function to locate a data item. An implementation of the binary search algorithm is given in the file *search.cpp*. An implementation of the showStructure operation is given in the file *show4.cpp*.

Step 2: Save your implementation of the Ordered List ADT in the file *OrderedList.cpp*. Be sure to document your code.

Compilation Directions

Because of how templated classes are compiled, you compile your implementation of the Ordered List ADT by compiling *test4.cpp*, which then includes *OrderedList.cpp*.

Testing

Test your implementation of the Ordered List ADT by using the program in the file *test4.cpp*. That test program allows you to interactively test your ADT implementation using the commands in the following table.

Command	Action
`+key`	Insert (or update) the data item with the specified key.
`?key`	Retrieve the data item with the specified key and output it.
`-`	Remove the data item marked by the cursor.
`@`	Display the data item marked by the cursor.
`=key`	Replace the data item marked by the cursor.
`N`	Go to the next data item.
`P`	Go to the prior data item.
`<`	Go to the beginning of the list.
`>`	Go to the end of the list.
`E`	Report whether the list is empty.
`F`	Report whether the list is full.
`C`	Clear the list.
`Q`	Quit the test program.

Step 1: Download the online test plans for Lab 4.

Step 2: Prepare Test Plan 4-1 for your implementation of the Ordered List ADT. Your test plan should cover the application of each operation to data items at the beginning, middle, and end of lists (where appropriate). Complete Test Plan 4-1 by filling in the test cases, commands, and expected results.

Step 3: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Programming Exercise 1

When a communications site transmits a message through a network like the Internet, it does not send the message as a continuous stream of data. Instead, it divides the message into pieces, called packets. These packets are sent through the network to a receiving site, which reassembles the message. Packets may travel to the receiving site along different paths. As a result, they often arrive out of sequence. In order for the receiving site to reassemble the message correctly, each packet must include the relative position of the packet within the original message. This identifier is called the sequence number.

For example, if we break the message "A SHORT MESSAGE" into packets each holding five characters and preface each packet with a number denoting the packet's position in the message, the result is the following set of packets.

```
1A SHO
2RT ME
3SSAGE
```

No matter in what order these packets arrive, a receiving site can correctly reassemble the message by placing the packets in ascending order based on their position numbers.

Step 1: Create a program that simulates reassembling an incorrectly sequenced message by reading in the packets from a text file, resequencing them, and outputting the original message. Your program should use the Ordered List ADT to assist in reassembling the packet. Each packet in the message file contains a position number and five characters from the message (the packet format shown previously). Ideally, you would base your program on the following declarations.

```
class Packet {
  public:
    static const int PACKET_SIZE = 6;  // Number of characters in a packet
                                       // including null ('\0') terminator,
                                       // but excluding the key (1st char in each line).
    int position;                      // (Key) Packet's position w/in message
    char body[PACKET_SIZE]; // Characters in the packet
    int getKey () const
        { return position; }   // Returns the key field
};
```

Read each successive packet in the input file into a Packet object and insert the Packet object into the ordered list. Then iterate through the entire list from beginning to end, retrieving each message and printing out the message body. Save your program in *packet.cpp*.

Step 2: Prepare a test plan for your implementation of the packet resequencer. Your test plan should be developed by visually inspecting the data file and determining the expected output. Complete Test Plan 4-2 by filling in the expected results.

Step 3: Test your program using Test Plan 4-2. The message data is in the text file *message.dat*. If you discover mistakes in your program, correct them and execute your test plan again.

Programming Exercise 2

Suppose you wish to combine the data items in two ordered lists of similar size. You could use repeated calls to the insert operation to insert the data items from one list into the other. However, the resulting process would not be very efficient. A more effective approach is to use a specialized merge operation that takes advantage of the fact that the lists are ordered.

```
void merge ( const OrderedList<DataType,KeyType>& other )
     throw ( logic_error )
```

Requirements:
The list being merged into has enough free space. The two Lists have no keys in common.

Results:
Merges the data items in `other` into this list. Does not change `other`.

Even before you begin to merge the lists, you already know how much larger the list will become (remember, no key is in both lists). By traversing the lists in parallel, starting with their highest keys and working backward, you can perform the merge in a single pass. Given two ordered lists alpha and beta containing the keys

```
alpha : a d j t
beta  : b e w
```

the call

```
alpha.merge(beta);
```

produces the following results.

```
alpha : a b d e j t w
beta  : b e w
```

Step 1: Implement this operation and add it to the file *OrderedList.cpp*. A prototype for this operation is included in the declaration of the Ordered List class in the file *OrderedList.h*.

Step 2: Activate Test 1 in the test program *test4.cpp* by changing the definition of LAB4_TEST1 in *config.h* from 0 to 1 and recompiling.

Step 3: Prepare Test Plan 4-3 (merge operation) that covers lists of various lengths, including empty lists and lists that combine to produce a full list.

Step 4: Execute your test plan. If you discover mistakes in your implementation of the merge operation, correct them and execute your test plan again.

Programming Exercise 3

A set of objects can be represented in many ways. If you use an unordered list to represent a set, then performing set operations such as intersection, union, difference, and subset require up to $O(N^2)$ time. By using an ordered list to represent a set, however, you can reduce the execution time for these set operations to $O(N)$, a substantial improvement.

Consider the subset operation described below. If the sets are stored as unordered lists, this operation requires that you traverse the list once for *each* data item in `other`. But if the sets are stored as ordered lists, only a single traversal is required. The key is to move through the lists in parallel.

```
bool subset ( const OrdList& other ) const
```

Requirements:
None

Results:
Returns `true` if every key in `other` is also in this list. Otherwise, returns `false`. Does not change `other`.

Given four ordered lists `alpha`, `beta`, `gamma`, and `delta` containing the keys

```
alpha : a b c d
beta  : a c x
gamma : a b
delta : <empty list>
```

the function call `alpha.subset(beta)` yields the value `false` (beta is not a subset of alpha), the call `alpha.subset(gamma)` yields the value `true` (gamma is a subset of alpha), and the calls `alpha.subset(delta)` and `beta.subset(delta)` yield the value `true` (the empty set is a subset of every set).

Step 1: Implement this operation and add it to the file *OrderedList.cpp*. A prototype for this operation is included in the declaration of the Ordered List class in the file *OrderedList.h*.

Step 2: Activate Test 2 in the test program *test4.cpp* by changing the definition of LAB4_TEST2 in *config.h* from 0 to 1 and recompiling.

Step 3: Prepare Test Plan 4-4 for the subset operation that covers lists of various lengths, including empty lists.

Step 4: Execute your test plan. If you discover mistakes in your implementation of the subset operation, correct them and execute your test plan again.

Analysis Exercise 1

Part A

Given an ordered list containing N data items, develop worst-case, order-of-magnitude estimates of the execution time of the steps in the insert operation, assuming this operation is implemented using an array in conjunction with a binary search. Briefly explain your reasoning behind each estimate.

Array Implementation of the Insert Operation

Find the insertion point	O( )
Insert the data item	O( )
Entire operation	O( )

Explanation:

Part B

Suppose you had implemented the Ordered List ADT using a linear search, rather than a binary search. Given an ordered list containing N data items, develop worst-case, order-of-magnitude estimates of the execution time of the steps in the insert operation. Briefly explain your reasoning behind each estimate.

Linked List Implementation of the Insert Operation

Find the insertion point	O( )
Insert the data item	O( )
Entire operation	O( )

Explanation:

Analysis Exercise 2

In specifying the Ordered List ADT, we assumed that no two data items in an ordered list have the same key. What changes would you have to make to your implementation of the Ordered List ADT in order to support ordered lists in which multiple data items have the same key?