Ordered List ADT

objectives

In this laboratory you will:

- Implement the Ordered List ADT using an array to store the list data items and a binary search to locate data items
- Use inheritance to derive a new class from an existing one
- Create a program that reassembles a message that has been divided into packets
- Use ordered lists to create efficient merge and subset operations
- Analyze the efficiency of your implementation of the Ordered List 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.

Ordered List ADT

Data Items

The data items in an ordered list are of generic type DataType. Each data item has a key (of type `char`) 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 *E*, the data item that precedes *E* has a key that is less than *E*'s key, and the data item that follows *E* has a key that is greater than *E*'s key. At any point in time, one data item in any nonempty list is marked using the list's cursor. You travel through the list using operations that change the position of the cursor.

Operations

```
List ( int maxNumber = defMaxListSize ) throw ( bad_alloc )
```

Requirements:
None

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

```
~List ()
```

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's nonkey fields with newDataItem's nonkey fields. Moves the cursor to mark newDataItem.

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

Requirements:
None

Results:
Searches a list for the data item with key `searchKey`. If the data item is found, then 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 moves the cursor 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 and inserting `newDataItem`. 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 a copy 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).

Laboratory 4: Cover Sheet

Name ______________________________ Date ____________________

Section ______________________________

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory 4: Prelab Exercise

Name ____________________ Date ____________

Section ____________________

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.

A key feature of C++ is the ability to derive a new class from an existing one through **inheritance**. The **derived class** inherits the member functions and data members of the existing **base class** and can have its own member functions and data members as well. The following declaration from the file *ordlist.h* derives a class called OrdList from the List class.

```
class OrdList : public List
{
  public:

    // Constructor
    OrdList ( int maxNumber = defMaxListSize );

    // Modified (or new) list manipulation operations
    virtual void insert ( const DataType &newDataItem )
        throw ( logic_error );
    virtual void replace ( const DataType &newDataItem )
        throw ( logic_error );
    bool retrieve ( char searchKey, DataType &searchDataItem );

    // Output the list structure — used in testing/debugging
    void showStructure () const;

  private:

    // Locates a data item (or where it should be) based on its key
    bool binarySearch ( char searchKey, int &index );
};
```

The declaration

```
class OrdList : public List
```

indicates that OrdList is derived from List. The keyword “public” specifies that this is a **public inheritance**—that is, OrdList inherits access to List’s public member functions, but not to its private data members (or private member functions).

You want the member functions in OrdList to be able to refer to List's private data members, so you must change the data members in the List class declaration from private to protected, as follows.

```
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
};
```

In a public inheritance, private List data members can only be accessed by List member functions. Protected List data members, on the other hand, can be accessed by the member functions in any class that is derived from List: OrdList, in this case.

The OrdList class supplies its own constructor, as well as a pair of new member functions: a public member function `retrieve()` that retrieves a data item based on its key, and a private member facilitator function `binarySearch()` that locates a data item in the array using a binary search. The OrdList class also includes its own versions of the `insert()` and `replace()` public member functions. The redefinition of these functions is indicated by the use of the keyword `virtual` in their function prototypes. Note that you must change this pair of functions to virtual in the List class declaration as well.

```
class List
{
  public:

    ...
    // List manipulation operations
    virtual void insert ( const DataType &newDataItem )  // Insert
        throw ( logic_error );
    virtual void replace ( const DataType &newDataItem ) // Replace
        throw ( logic_error );
    ...

  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
};
```

A class declaration for the List class containing the changes specified above is given in the file *listarr2.h*.

An OrdList object can call any of the List public member functions, as well as any of its own member functions. The following program reads in the account number and balance for a set of accounts and outputs the accounts in ascending order based on their account numbers.

```
#include <iostream>

struct DataType
{
    int acctNum;                // (Key) Account number
    float balance;              // Account balance

    int getKey () const
        { return acctNum; }     // Returns the key
};

#include “ordlist.cpp”

const int maxNameLength = 15;

void main()
{
    OrdList accounts;           // List of accounts
    DataType acct;              // A single account

    // Read in information on a set of accounts.

    cout << endl << “Enter account information (EOF to end) : ”
         << endl;
    while ( cin >> acct.acctNum >> acct.balance )
        accounts.insert(acct);

    // Output the accounts in ascending order based on their account
    // numbers.

    cout << endl;
    if ( !accounts.isEmpty() )
    {
       accounts.gotoBeginning();
       do
       {
           acct = accounts.getCursor();
           cout << acct.acctNum << “ “ << acct.balance << endl;
       }
       while ( accounts.gotoNext() );
    }
};
```

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

Another change from the Laboratory 3 List class implementation is that we will not use `typedef` any more. We just declare a `struct` or `class` in the application program (see preceding example) and name it `DataType`. To ensure that the code in the OrdList class and in the List class have access to the type information, we include the file *ordlist.cpp*—instead of *ordlist.h*—near the start of the application program on a line below where `DataType` is declared. This goes against the principles of modular coding, but it simplifies the implementation. Furthermore, we will need to use this approach when we implement the next laboratory, the Stack ADT, using C++ templates.

We are introducing one more C++ programming technique that helps avoid the accidental—and sometimes unavoidable—multiple inclusion of class implementation files. For instance, assume that an application program needs ordered lists. It includes the OrdList class, which in turn includes the List class. The programmer might now include another class derived from the List class. The result is that the compiler encounters the List class implementation twice and signals an error. The solution is to use the C++ preprocessor to enable **conditional compilation**. Conditional compilation allows the programmer to control what parts of a file the compiler will try to compile. In this case, we ensure that the compiler only tries to compile the contents of each file once, regardless of how many times the file has been included.

We protect *ordlist.h* by inserting three lines around the entire code contents of the file as follows:

```
#ifndef ORDLIST_H
#define ORDLIST_H

...     // The previous contents of the file

#endif
```

The line "`#ifndef ORDLIST_H`" means that if the string "`ORDLIST_H`" has not yet been defined, then compile all the code up to the matching "`#endif`" line. The first thing that happens within that block is that the identifier `ORDLIST_H` is defined. If the compiler encounters the file again during that compilation, `ORDLIST_H` will be already defined and all code up through the `#endif` will be ignored. The tradition for identifier names is that they should match the file name. The period character ('`.`') is not valid for identifiers, so it is replaced by the underscore ('`_`') character. Conditional compilation by means of C++ preprocessor definitions becomes progressively more useful as programs become larger and more complex.

Step 1: The `showStructure()` and `find()` functions from your array implementation of the List ADT are designed for use with lists that are composed of one of C++'s built-in types. Comment out these functions and save the resulting implementation of the List ADT in the file *listarr2.cpp*.

Step 2: Implement the operations in the Ordered List ADT using the array representation of a list. Base your implementation on the following declaration from the file *ordlist.h*.

```
#include "listarr2.cpp"

class OrdList : public List
{
  public:

    // Constructor
    OrdList ( int maxNumber = defMaxListSize );
    // Modified (or new) list manipulation operations
    virtual void insert ( const DataType &newDataItem )
        throw ( logic_error );
    virtual void replace ( const DataType &newDataItem )
        throw ( logic_error );
    bool retrieve ( char searchKey, DataType &searchDataItem );

    // Output the list structure -- used in testing/debugging
    void showStructure () const;

  private:

    // Locates a data item (or where it should be) based on its key
    bool binarySearch ( char searchKey, int &index );
};
```

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 inherited from your array implementation of the List ADT. 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 3: Save your implementation of the Ordered List ADT in the file *ordlist.cpp*. Be sure to document your code.

Laboratory 4: Bridge Exercise

Name ______________________________ Date ____________________

Section ______________________________

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

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

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: Prepare a test plan 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). A test plan form follows.

Step 2: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Ordered List ADT

Test Case	*Commands*	*Expected Result*	*Checked*

Laboratory 4: In-lab Exercise 1

When a communications site transmits a message through a packet-switching network, 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 be transmitted to the receiving site along different paths. As a result, they are likely to 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 message.

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

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

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 reassembles the packets contained in a text file and outputs the corresponding message. Your program should use the Ordered List ADT to assist in reassembling the packets in a message. Assume that each packet in the message file contains a position number and five characters from the message (the packet format shown in the preceding paragraph). Base your program on the following declarations.

```
const int packetSize = 6;  // Number of characters in a packet
                           // including null (‘\0’) terminator

struct DataType
{
    int position;              // Packet’s position w/in the message
    char body[packetSize];     // Characters in the packet

    int key () const
        { return position; }   // Returns the key field
};
```

Store your program in the file *packet.cpp*.

Step 2: Test your program using the message in the text file *message.dat*.

Test Plan for the Message Processing Program

Test Case	*Checked*
Message in the file *message.dat*	