laboratory 3

Array Implementation of the List ADT

Objectives

In this laboratory you will:

- Implement the List ADT using an array representation of a list, including development of an iteration scheme that allows you to move through a list data item by data item
- Create a program that analyzes the genetic content of a DNA sequence
- Analyze the efficiency of your array implementation of the List ADT

Overview

The list is one of the most frequently used data structures. Although all programs share the same definition of **list**—a sequence of homogeneous data items—the type of data item stored in lists varies from program to program. Some use lists of integers, others use lists of characters, floating-point numbers, points, and so forth. You normally have to decide on the data item type when you implement the ADT. If you need a different data item type, there are several possibilities.

1. You could edit the class code (the declaration file, *classname.h*, and the definition file, *classname.cpp*—in this case of the List ADT, *listarr.h* and *listarr.cpp*) and replace every reference to the old data type by the new data type. This is a lot of work, tedious, and error-prone.
2. A simpler solution is to use a made up data type name throughout the class, such as `DataType`, and then use the C++ `typedef` statement at the beginning of the class declaration file to specify what `DataType` really is. To specify that the list data items should be characters, you would type

   ```
   typedef char DataType;
   ```

   This approach does work, and changing the data item type is much easier than the first solution. We will be using this approach for the List ADT and the Ordered List ADT.

   This approach does, however, have drawbacks. A major problem with this method is that a given program can have `DataType` set to only one particular type. For instance, you cannot have both a list of characters and a list of integers; `DataType` must be either `char` or `int`. You could make separate copies of the List ADT and define `DataType` differently in each copy. Because you cannot have multiple classes in a program with the same name, you would also need to change every occurrence of the class name `List` to something like `CharList` or `IntList`. This works, but it gets messy and the whole process must be repeated every time you need a list with a new data item data type.
3. Fortunately, C++ has a third solution, **templates**, which we will explain in the Stack ADT in Laboratory 5. When using templates, you do not need to create a different list implementation for each type of list data item. Instead, you create a list implementation in terms of list data items of some **generic** type rather like solution 2 above. We will use the arbitrary string "DT" for the data type. This requires just one copy of the class code. You can then ask the compiler to make any number of lists in which the data items are an arbitrary data type by adding a simple piece of information when you declare a list in your code.

   ```
   List<int> samples;      // Create a list of integers
   List<char> line;        // Then create a list of characters
   ```

   We will return to templates in the Stack ADT.

If an ADT is to be useful, its operations must be both expressive and intuitive. The following List ADT provides operations that allow you to insert data items in a list, remove data items from a list, check the state of a list (is it empty, or is it full?), and iterate through the data items in a list. Iteration is done using a **cursor** that you move through the list much as you move the cursor in a text editor or word processor. In the following example, the List ADT's gotoBeginning operation is used to move the cursor to the beginning of the list. The cursor is then moved through the list data item by

data item by repeated applications of the gotoNext operation. Note that the data item marked by the cursor is shown in bold.

After gotoBeginning: **a** b c d

After gotoNext: a **b** c d

After gotoNext: a b **c** d

After gotoNext: a b c **d**

List ADT

Data Items

The data items in a list are of generic type DataType.

Structure

The data items form a linear structure in which list data items follow one after the other, from the beginning of the list to its end. The ordering of the data items is determined by when and where each data item is inserted into the list and is *not* a function of the data contained in the list data items. 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` into a list. If the list is not empty, then inserts `newDataItem` after the cursor. Otherwise, inserts `newDataItem` as the first (and only) data item in the list. In either case, moves the cursor to `newDataItem`.

```
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, the cursor should now be marking 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 data item at 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`. The cursor remains at `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:
If a list is not empty, then moves the cursor to the data item at the beginning of the list.

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

Requirements:
List is not empty.

Results:
If a list is not empty, then 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 () ( 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 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 list data items that are one of C++'s predefined data types (`int`, `char`, and so forth).

Laboratory 3: 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 3: Prelab Exercise

Name _____ Date _____

Section _____

You can implement a list in many ways. Given that all the data items in a list are of the same type, and that the list structure is linear, an array seems a natural choice. You could declare the size of the array at **compile-time** (as you did with the logbook array in Laboratory 1), but your List ADT will be more flexible if you specify the size of the array at **run-time** and **dynamically allocate** the memory required to store it.

Memory allocation for the array is done by the **constructor**. The constructor is invoked whenever a list declaration is encountered during the execution of a program. Once called, the constructor allocates an array using C++'s `new` operator. The constructor outlined below, for example, allocates an array of `maxNumber` data items and assigns the address of the array to the pointer `dataItems`, where `dataItems` is of type `DataType*`.

```
List:: List ( int maxNumber )
{
   . . .
   dataItems = new DataType[maxNumber];
}
```

Whenever you allocate memory, you must ensure that it is deallocated when it is no longer needed. The **destructor** is used to deallocate the memory storing the array. This function is invoked whenever a list is destroyed—that is, if the function containing the corresponding list declaration terminates or if the list is explicitly destroyed by the programmer. The fact that the call to the destructor is made automatically eliminates the possibility of you forgetting to deallocate the list. The destructor outlined below frees the memory used by the array that you allocated above.

```
List:: ~List ()
{
    delete [] dataItems;
}
```

Another significant implementation issue is what to do when a logbook function such as `insert()` or `remove()` is called with parameters or preconditions that do not meet the stated requirements. Using the `remove()` function as an example, it is a logic error to request removal of an item from an empty list. Although there are many possible ways of dealing with this situation, the standard C++ method for dealing with bad parameters and other difficult—or impossible—situations is to `throw` an **exception**. Throwing an exception causes the currently active function to stop execution and return to the calling function. Unless that function or one of its callers takes special steps to handle the exception, the program will be halted. By using the C++ try and catch instructions, callers can decide what to do when an exception is thrown. The `try` instruction is used when trying something that might cause an exception. The `catch` instruction is used to specify what to do if an exception did occur inside the preceding try code block. Common catch responses to an exception include one or more of the following: 1) print out a helpful explanation of what went wrong, 2) try to work around the problem, and 3) halt the program. The empty list

problem in the `remove()` function example can be dealt with by using the following code snippet at the beginning of the function to enforce the "List is not empty" requirement.

```
if ( size == 0 )
        throw logic_error("Remove: list is empty");
```

The C++ exception handling mechanism is quite complicated. For the purposes of this book, you will almost always be throwing a `logic_error` exception or one of just a few other exception types. We have written a longer discussion of exceptions and program validation techniques in Appendix 1. Please read it before completing Laboratory 3.

Also, note that for the purpose of demonstrating the usefulness of exceptions, we have included catch statements in the *test3.cpp* program to handle any exceptions that are thrown. This is the only test program that uses them heavily because they tend to clutter the code and make it less readable. We decided not to sacrifice the readability of the rest of our small test programs by adding complete error-handling capabilities. However, these capabilities are mandatory in large complex programs.

Step 1: Implement the operations in the List ADT using an array to store the list data items. Lists change in size; therefore, you need to store the maximum number of data items the list can hold (`maxSize`) and the actual number of data items in the list (`size`), along with the list data items themselves (`dataItems`). You also need to keep track of the array index (`cursor`). Base your implementation on the following declarations from the file *listarr.h*. An implementation of the showStructure operation is given in the file *show3.cpp*.

```
const int defMaxListSize = 10;   // Default maximum list size

typedef char DataType;

class List
{
  public:

    // Constructor
    List ( int maxNumber = defMaxListSize ) throw ( bad_alloc );

    // Destructor
    ~List ();

    // List manipulation operations
    void insert ( const DataType &newDataItem )   // Insert after cursor
        throw ( logic_error );
    void remove () throw ( logic_error );         // Remove data item
    void replace ( const DataType &newDataItem )  // Replace data item
        throw ( logic_error );
    void clear ();                                // Clear list

    // List status operations
    bool isEmpty () const;                        // List is empty
    bool isFull () const;                         // List is full

    // List iteration operations
    void gotoBeginning () throw ( logic_error );  // Go to beginning
    void gotoEnd () throw ( logic_error );        // Go to end
```

```
    bool gotoNext () throw ( logic_error );       // Go to next data item
    bool gotoPrior () throw ( logic_error );      // Go to prior data item
    DataType getCursor () const throw ( logic_error ); // Return data item

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

  private:

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

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

The code in the file *listarr.cpp* provides a template (or framework) for a set of implementations of the List ADT. The type of the list data items is deliberately left unspecified in this framework and is only made specific where the data type is declared in the class declaration in *listarr.h*.

The following program uses the array implementation of the operations in the List ADT to read in a list of integer samples and compute their sum.

```
// For this example, set DataType to "int" in listarr.h,

#include <iostream>
#include "listarr.h"   // Include the class declaration file

using namespace std;

void main ()
{
    List samples(100);        // Set of samples
    int newSample,            // Input sample
        total = 0;            // Sum of the input samples

    // Read in a set of samples from the keyboard.

    cout << "Enter list of samples (end with eof) : ";
    while ( cin >> newSample )
        samples.insert(newSample);

    // Sum the samples and output the result.

    if ( !samples.isEmpty() )            // Verify that list has data
    {
       samples.gotoBeginning();          // Go to beginning of list
       do
         total += samples.getCursor();   // Add element to running sum
       while ( samples.gotoNext() );     // Go to next element (if any)
    }

    cout << "Sum is " << total << endl;
}
```

Laboratory 3: 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 programs that you used in Laboratory 1 consisted of a series of tests that were hard-coded into the programs. Adding a new test case to this style of test program requires changing the test program itself. In this and subsequent laboratories, you use a more flexible kind of test program to evaluate your ADT implementations, one in which you specify a test case using commands, rather than code. These interactive, command-driven test programs allow you to check a new test case by simply entering a series of keyboard commands and observing the results.

The test program in the file *test3.cpp*, for instance, supports the following commands.

Command	Action
`+x`	Insert data item `x` after the cursor.
`-`	Remove the data item marked by the cursor.
`=x`	Replace the data item marked by the cursor with data item `x`.
`@`	Display 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.

Suppose you wish to confirm that your array implementation of the List ADT successfully inserts a data item into a list that has been emptied by a series of calls to the remove operation. You can test this case by entering the following sequence of keyboard commands.

Command	`+a`	`+b`	`-`	`-`	`+c`	`Q`
Action	Insert `a`	Insert `b`	Remove	Remove	Insert `c`	Quit

It is easy to see how this interactive test program allows you to rapidly examine a variety of test cases. This speed comes with a price, however. You must be careful not to violate the preconditions required by the operations that you are testing. For instance, the commands

Command	`+a`	`+b`	`-`	`-`	`-`
Action	Insert `a`	Insert `b`	Remove	Remove	Error (exception)

cause the test program to fail during the last call to the remove operation. The source of the failure does not lie in the implementation of the List ADT, nor is the test program flawed. The failure occurs because this sequence of operations creates a state that violates the preconditions of the remove operation (the list must *not* be empty when the remove operation is invoked). The speed with which you can create and evaluate test cases using an interactive, command-driven test program makes it very easy to produce this kind of error. It is very tempting to just sit down and start entering commands. A much better strategy, however, is to create a test plan, listing the test cases you wish to check and then to write out command sequences that generate these test cases. Of course, you should also deliberately violate function requirements in order to test your exception handling.

Step 1: Compile and link the test program in the file *test3.cpp*. Note that when compiling this program you need to ensure that `DataType` is defined as `char` and that you also compile and link in your array implementation of the List ADT (in the file *listarr.cpp*) to produce the correct executable for a list of characters.

Step 2: Complete the following test plan by adding test cases that check whether your implementation of the List ADT correctly handles the following tasks:

- Insertions into a newly emptied list
- Insertions that fill a list to its maximum size
- Deletions from a full list
- Determining whether a list is empty
- Determining whether a list is full

Assume that the output of one test case is used as the input to the following test case, and note that although expected results are listed for the final command in each command sequence, you should confirm that *each* command produces a correct result.

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

Test Plan for the Operations in the List ADT

Test Case	*Commands*	*Expected Result*	*Checked*
Insert at end	`+a +b +c +d`	`a b c d`	
Travel from beginning	`< N N`	`a b c d`	
Travel from end	`> P P`	`a b c d`	
Delete middle data item	`-`	`a c d`	
Insert in middle	`+e +f +f`	`a c e f f d`	
Remove last data item	`>-`	`a c e f f`	
Remove first data item	`<-`	`c e f f`	
Display data item	`@`	Returns `c`	
Replace data item	`=g`	`g e f f`	
Clear the list	`C`	Empty list	

Note: The data item marked by the cursor is shown in **bold**.

Step 4: Change the list in the test program from a list of characters to a list of integers by replacing the declaration for `DataType` in *listarr.h* and `testDataItem` in *test3.cpp* with

```
typedef int DataType;
int testDataItem;     // List data item
```

Step 5: Recompile and relink the test program. Note that recompiling the program will compile your implementation of the List ADT (in the file *listarr.cpp*) to produce an implementation for a list of integers.

Step 6: Replace the character data in your test plan ('`a`' to '`g`') with integer values.

Step 7: Execute your revised test plan using the revised test program. If you discover mistakes in your implementation of the List ADT, correct them and execute your revised test plan again.

Laboratory 3: In-lab Exercise 2

Name _____ Date _____

Section _____

In many applications, the ordering of the data items in a list changes over time. Not only are new data items added and existing ones removed, but data items are repositioned within the list. The following List ADT operation moves a data item to a new position in a list.

```
void moveToNth ( int n ) throw ( logic_error )
```

Requirements:
List contains at least `n+1` data items.

Results:
Removes the data item marked by the cursor from a list and reinserts it as the *n*th data item in the list, where the data items are numbered from beginning to end, starting with zero. Moves the cursor to the moved data item.

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

Step 2: Activate the `'M'` (move) command in the test program *test3.cpp* by removing the comment delimiter (and the character `'M'`) from the lines that begin with "`//M`".

Step 3: Complete the following test plan by adding test cases that check whether your implementation of the `moveToNth` operation correctly processes moves within full and single data item lists.

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

Test Plan for the `moveToNth` Operation

Test Case	*Commands*	*Expected Result*	*Checked*
Set up list	`+a +b +c +d`	`a b c` **`d`**	
Move first data item	`< M2`	`b c` **`a`** `d`	
Move data item back	`M0`	**`a`** `b c d`	
Move to end of list	`M3`	`b c d` **`a`**	
Move back one	`M2`	`b c` **`a`** `d`	
Move forward one	`M3`	`b c d` **`a`**	

Note: The data item marked by the cursor is shown in **bold**.

Laboratory 3: In-lab Exercise 3

Name _____ Date _____

Section _____

Finding a particular list data item is another very common task. The following operation searches a list for a specified data item. The fact that the search begins with the data item marked by the cursor—and not at the beginning of the list—means that this operation can be applied iteratively to locate all of the occurrences of a specified data item.

```
bool find ( const DataType &searchDataItem ) throw ( logic_error )
```

Requirements:
List is not empty.

Results:
Searches a list for `searchDataItem`. Begins the search with the data item marked by the cursor. Moves the cursor through the list until either `searchDataItem` is found (returns `true`) or the end of the list is reached without finding `searchDataItem` (returns `false`). Leaves the cursor at the last data item visited during the search.

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

Step 2: Activate the `'?'` (find) command in the test program *test4.cpp* by removing the comment delimiter (and the character `'?'`) from the lines that begin with "`//?`".

Step 3: Complete the following test plan by adding test cases that check whether your implementation of the `find` operation correctly conducts searches in full lists, as well as searches that begin with the last data item in a list.

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

Test Plan for the `find` Operation

Test Case	*Commands*	*Expected Result*	*Checked*
Set up list	`+a +b +c +a`	`a b c` **a**	
Successful search	`< ?a`	Search succeeds **a** `b c a`	
Search for duplicate	`N ?a`	Search succeeds `a b c` **a**	
Successful search	`< ?b`	Search succeeds `a` **b** `c a`	
Search for duplicate	`N ?b`	Search fails `a b c` **a**	

Note: The data item marked by the cursor is shown in **bold.**