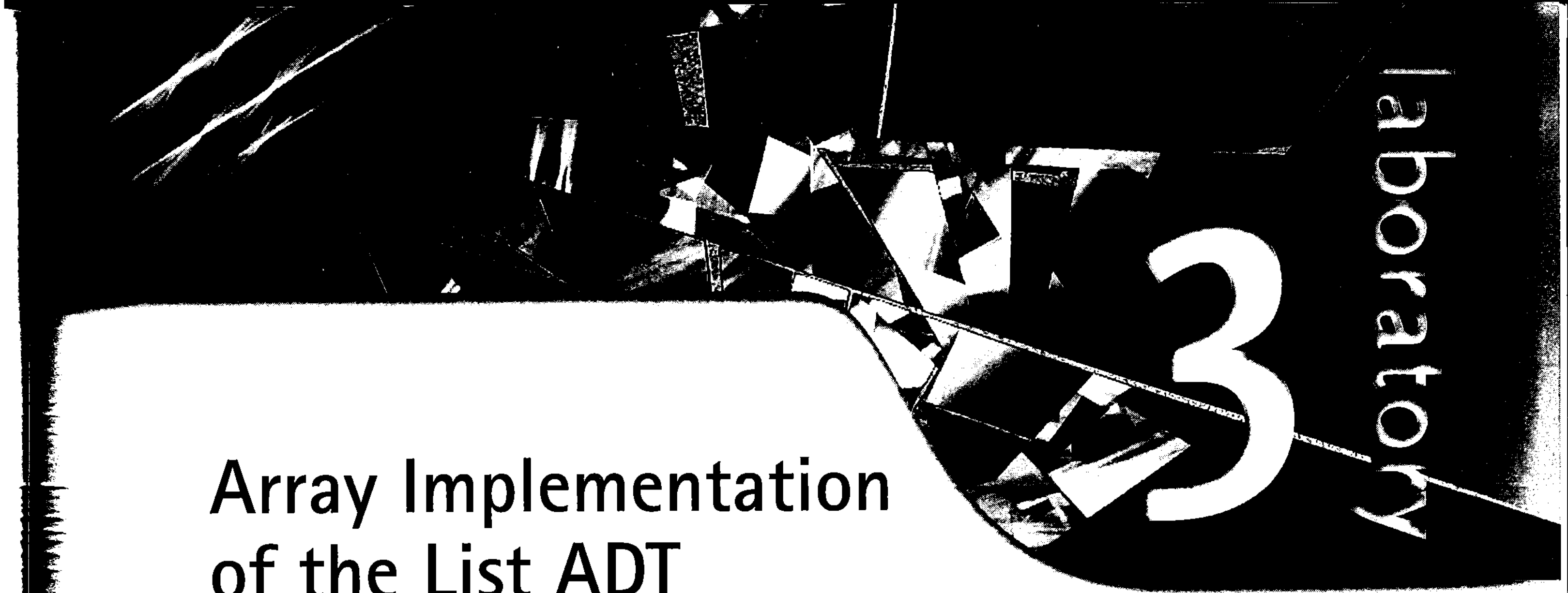

Array Implementation of the List ADT

objectives

In this laboratory you

- implement the List ADT using an array representation of a list.
- develop an iteration scheme that allows you to move through a list data item-by-data item.
- learn how to use C++ templates to create generic data types.
- are exposed to implementing base classes in an inheritance hierarchy.
- analyze the algorithmic complexity of your array implementation of the List ADT.

ADT Overview

If an ADT is to be useful, its operations must be both expressive and intuitive. The List ADT described below 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. Iteration functions typically allow the cursor to be moved to the beginning, the end, the next, or the prior item in a data structure. 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 italics.

After gotoBeginning:	***a***	b	c	d
After gotoNext:	a	***b***	c	d
After gotoNext:	a	b	***c***	d
After gotoNext:	a	b	c	***d***

C++ Concepts Overview

Templates: C++ supports the concept of a generic data type through templates. Generic data types allow a class to be implemented in a generic way such that one or more data types in the class do not need to be specified when the class is implemented. The code is written independent of the data type that will eventually be specified when objects of the class are instantiated.

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's type at the time that you implement the ADT. If you need a different data item type, there are several possibilities.

1. You could edit the class code (e.g., the declaration file, *classname.h*, and the definition file, *classname.cpp*—in the case of this List ADT, *ListArray.h* and *ListArray.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—e.g., `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 DataType char;
   ```

   This approach does work and changing the data item data type is much easier than the first solution; you just change the `typedef` line and recompile.

   It does however, have problems. For instance, a given program can only have `DataType` set to one particular type. You cannot have both a list of characters and a list of integers in the same program; `DataType` must be either `char` or `int`. You could make entire copies of the List ADT and define a new `DataType` differently in each copy. Because you cannot have multiple classes in a program with exactly the same name, you must also change every occurrence of the class name `List` to something like `CharList` or `IntList`. This will work, but it takes you back to solution #1 and is generally unsatisfactory.

3. Fortunately, C++ has a solution: templates. Using templates, you do not need to create a different list implementation for each type of data item. Instead, you create a list implementation in terms of list data items of some generic type rather like solution two above. This requires just one copy of the class implementation source code. You can then ask the compiler to make any number of lists whose data items are an arbitrary data type by adding a simple piece of information—the data type inside angle brackets, e.g. `List<float>`—when you declare a list in your code. For instance, to create a list of integers and one of characters in the same program, you would write the following:

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

Inheritance: In Lab 4, we are going to introduce the concept of inheritance. Lab 3 is the foundation on which Lab 4 will be based. The primary requirements are declaring some class members protected instead of private, and declaring functions `virtual`.

Virtual functions: By declaring a function `virtual`, we are stating that it may be overridden in another class that is connected to this class through inheritance.

List ADT Specification

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 = MAX_LIST_SIZE )`

Requirements:
None

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

```
List ( const List& other )
```

Requirements:
None

Results:
Copy constructor. Initializes the list to be equivalent to the `other` List object parameter.

```
List& operator= ( const List& other )
```

Requirements:
None

Results:
Overloaded assignment operator. Sets the list to be equivalent to the `other` object parameter and returns a reference to the object.

```
virtual ~List ()
```

Requirements:
None

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

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

Requirements:
List is not full.

Results:
Inserts `newDataItem` into the 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 the 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 the cursor marks the data item at the beginning of the list. This operation preserves the order of the remaining data items in the list.

```
virtual 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 the list.

```
bool isEmpty () const
```

Requirements:
None

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

```
bool isFull () const
```

Requirements:
None

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

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

Requirements:
List is not empty.

Results:
If the 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 the 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 the 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 the 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 data items in the 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) or that have had the << operator overloaded for an `ostream`.

Implementation Notes

Templates: A template is something that serves as a pattern. The pattern is not the final product, but is used to enable faster production of a final product. Every place in your code where you would normally have to specify the data type, you instead use an arbitrary string to represent any actual data type that you might later wish to use. We will use the arbitrary string `"DataType"` to represent the generic data type. Nowhere does either the class declaration—the *class.h* file—or the class definition—the *class.cpp* file—specify the actual C++ data type. You can defer specifying the actual data type until it is time to instantiate—create—an object of that class.

Following are a few simple rules for creating and using a template class.

- The string "`template < typename DataType >`" must go right before the class declaration and before every class member function definiton. Remember, `DataType` is our arbitrary identifier that will represent any data type in the template class. So the lines

  ```
  class List
  {
    public:
      ...
  ```

  are changed to

  ```
  template < typename DataType >
  class List
  {
    public:
      ...
  ```

  The start of a function definition that used to be

  ```
  List:: List ( int maxNumber )
  ```

  now becomes

  ```
  template < typename DataType >
  List<DataType>:: List ( int maxNumber )
  ```

- Every use of the class name now must include the generic data type name enclosed in angle brackets. Every instance of the string "`List`" becomes "`List<DataTtype>`". In the example constructor definition, the class resolution "`List::`" becomes "`List<DataType>::`". Also note that the exception to the rule is that the constructor name is not modified—it remains just "`List`".

  ```
  template < typename DataType >
  List<DataType>:: List ( int maxNumber )
  ```

- When it is time to instantiate an object of that class, the real data type—inside angle brackets—is appended to the class name. So the following lines create a list of 10 integers and a list of 80 characters

  ```
  // We tell the compiler to make a copy of the generic list just
  // for integers and to make another just for characters.
  List<int>  samples(10);
  List<char> line(80);
  ```

Note: C++ originally used the syntax "`<class DataType>`" before templated classes and functions. The use of "class" was confusing to programmers since DataType could be a built-in type. The syntax was eventually changed to be "`<typename DataType>`". C++ continues to permit the use of "class" in this context, but "typename" is preferred.

Class constants: We have a fairly strong opinion about declaring class constants (and reasons for our preference). We prefer to declare class constants inside the class—as opposed to somewhere outside the class declaration—and to make them "static" variables. Because they are part of the class, declaring them inside the class associates them more closely with a class. This also allows programmers to reference the constants from anywhere by using the `ClassName::CONST_NAME` syntax (assuming the class protection policy allows access), without having to first create an object of that class. Declaring it to be `static` also allows initialization of the constant within the class, something otherwise illegal. Last, declaring a variable or constant identifier as `static` guarantees that all objects of that class will share the same variable or constant, thus reducing program memory requirements.

```
class X {
    ...
    static const int MY_MAX = 10;
    ...
};
```

Exceptions: The use of exceptions is explained in Lab 2. If you skipped Lab 2, please go back and read the explanations in the "C++ Concepts" and "Implementation Notes" sections.

Writing classes for inheritance: Inheritance is one of the significant characteristics of object-oriented programming. Given a class—let's call it Base—we can write another class—call it Derived—that can be derived from the base class by inheriting data and member functions from the class. The important thing to be aware of when developing a base class is that anything in its private section is truly private, even from a derived class. The solution is to put anything that is to be kept private from code outside the inheritance hierarchy, but that must be accessible to derived classes, in a *protected* section. The syntax is similar to how a private section is declared, only the word "protected" is substituted for "private". Whether a base class will also have a private section depends on whether there are operations or data values that must be kept private even from derived classes.

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.

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 cursor array index (`cursor`). Base your implementation on the declarations from the file *ListArray.h*. An implementation of the `showStructure` operation is given in the file *show3.cpp*. Please insert the contents of *show3.cpp* into your *ListArray.cpp* file.

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

Compilation Directions

Compiling programs that use templated classes requires a change in what files are included using the `#include` preprocessor directive, and in how the program is compiled. Because of how C++ compilers process templated code, the program that creates objects of the classes (e.g., *test3.cpp*) must include the class implementation file, not the class declaration file. That is, it must do `#include "ClassName.cpp"` instead of the usual `#include "ClassName.h"`. The rule is in effect for the rest of this book. Because the main implementation file does a `#include` of the class implementation code, the class implementation code is not compiled separately.

Testing

The test programs that you used in Laboratories 1 and 2 consisted of a series of tests that were hardcoded 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	Causes Error

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.

Step 1: Download the online test plans for Lab 3.

Step 2: Complete the Test Plan 3-1 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.

Step 4: Activate Test 1 by changing the value of LAB3_TEST1 from 0 to 1 in the *config.h* file. The second test changes the data type being used by the List from a character to an integer.

Step 5: Recompile the test program.

Step 6: Replace the character data in your test plan with integer values to create Test Plan 3-2.

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.

Programming Exercise 1

The genetic information encoded in a strand of deoxyribonucleic acid (DNA) is stored in the purine and pyrimidine bases (adenine, guanine, cytosine, and thymine) that form the strand. Biologists are keenly interested in the bases in a DNA sequence because these bases determine what the sequence does.

By convention, DNA sequences are represented by using lists containing the letters 'A', 'G', 'C', and 'T" (for adenine, guanine, cytosine, and thymine, respectively). The following function computes one property of a DNA sequence–the number of times each base occurs in the sequence.

```
void countBases ( List& dnaSequence, int& aCount,
                  int& cCount, int& tCount, int& gCount )
```

Input parameters:
dnaSequence: contains the bases in a DNA sequence encoded using the characters
    `'A'`, `'C'`, `'T'`, and `'G'`.

Output parameters:
aCount, cCount, tCount, gCount: the number of times the corresponding base appears in the DNA sequence.

Step 1: Copy the file *test3dna.cs* with the shell program to the file *test3dna.cpp*. Implement this function and add it to the program in the file *test3dna.cpp*. Your implementation should manipulate the DNA sequence using the operations in the List ADT.

Step 2: The program in the file *test3dna.cpp* reads a DNA sequence from the keyboard, calls the `countBases` function, and outputs the resulting base counts. Complete Test Plan 3-3 by adding DNA sequences of different lengths and various combinations of bases.

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

Programming Exercise 2

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 (because `n=0` represents the first position).

Results:
Removes the data item marked by the cursor from the 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 *ListArray.cpp*. A prototype for this operation is included in the declaration of the List class in the file *ListArray.h*.

Step 2: Activate Test 1 in the test program *test3.cpp* by changing the definition of LAB3_TEST1 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete the Test Plan 3-4 by adding test cases that check whether your implementation of the moveToNth operation correctly processes moves within various sized lists.

Step 4: Execute Test Plan 3-4. If you discover mistakes in your implementation of the moveToNth operation, correct them and execute your test plan again.

Programming Exercise 3

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 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 *ListArray.cpp*. A prototype for this operation is included in the declaration of the List class in the file *ListArray.h*.

Step 2: Activate Test 2 in the test program *test3.cpp* by changing the definition of LAB3_TEST2 from 0 to 1 in *config.h* and recompiling.

Step 3: Complete the Test Plan 3-5 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 the test plan. If you discover mistakes in your implementation of the find operation, correct them and execute your test plan again.

Analysis Exercise 1

A full-page version of these exercises with space for writing in answers is available in the online supplements for Lab 3.

Given a list containing *N* data items, develop worst-case, order-of-magnitude estimates of the execution time of the following List ADT operations, assuming they are implemented using an array. Briefly explain your reasoning behind each estimate.

gotoNext O( )

Explanation:

gotoPrior O( )

Explanation:

insert O( )

Explanation:

remove O( )

Explanation:

Analysis Exercise 2

Part A

Give a declaration for a list of floating-point numbers called `echoReadings`. Assume that the list can contain no more than fifty floating-point numbers.

Part B

Give the declarations required for a list of (x, y, z)-coordinates called `coords`. Assume that x, y, and z are floating-point numbers and that there will be no more than twenty coordinates in the list.

Part C

Are the declarations that you created in Parts A and B compatible with the operations in your implementation of the List ADT? Briefly explain why or why not.