Chapter 6

LISTS AND STRINGS

1. List Specifications

2. List Implementations
   (a) Class Templates
   (b) Contiguous
   (c) Simply Linked
   (d) Simply Linked with Position Pointer
   (e) Doubly Linked

3. Strings

4. Application: Text Editor

5. Linked Lists in Arrays

6. Application: Generating Permutations
**Definition** A *list* of elements of type $T$ is a finite sequence of elements of $T$ together with the following operations:

1. **Construct** the list, leaving it empty.
2. Determine whether the list is *empty* or not.
3. Determine whether the list is *full* or not.
4. Find the *size* of the list.
5. **Clear** the list to make it empty.
6. **Insert** an entry at a specified position of the list.
7. **Remove** an entry from a specified position in the list.
8. **Retrieve** the entry from a specified position in the list.
9. **Replace** the entry at a specified position in the list.
10. **Traverse** the list, performing a given operation on each entry.

**Comparison with Standard Template Library:**

- The STL list provides only those operations that can be implemented efficiently in a List implementation known as doubly linked, which we shall study shortly.
- The STL list does not allow random access to an arbitrary list position.
- The STL vector, does provide some random access to a sequence of data values, but not all the other capabilities we shall develop for a List.
- In this way, our study of the List ADT provides an introduction to both the STL classes list and vector.
Method Specifications

List::List();

Post: The List has been created and is initialized to be empty.

void List::clear();

Post: All List entries have been removed; the List is empty.

bool List::empty() const;

Post: The function returns true or false according to whether the List is empty or not.

bool List::full() const;

Post: The function returns true or false according to whether the List is full or not.

int List::size() const;

Post: The function returns the number of entries in the List.
To find an entry in a list, we use an integer that gives its position within the list.

We shall number the positions in a list so that the first entry in the list has position 0, the second position 1, and so on.

Locating an entry of a list by its position is superficially like indexing an array, but there are important differences. If we insert an entry at a particular position, then the position numbers of all later entries increase by 1. If we remove an entry, then the positions of all following entries decrease by 1.

The position number for a list is defined without regard to the implementation. For a contiguous list, implemented in an array, the position will indeed be the index of the entry within the array. But we will also use the position to find an entry within linked implementations of a list, where no indices or arrays are used at all.

Error code List::insert(int position, const List_entry &x);

Post: If the List is not full and 0 ≤ position ≤ n, where n is the number of entries in the List, the function succeeds: Any entry formerly at position and all later entries have their position numbers increased by 1, and x is inserted at position in the List.
Else: The function fails with a diagnostic error code.
Error_code List:: remove(int position, List_entry &x);

Post: If $0 \leq \text{position} < n$, where $n$ is the number of entries in the List, the function succeeds: The entry at position is removed from the List, and all later entries have their position numbers decreased by 1. The parameter $x$ records a copy of the entry formerly at position. Else: The function fails with a diagnostic error code.

Error_code List:: retrieve(int position, List_entry &x) const;

Post: If $0 \leq \text{position} < n$, where $n$ is the number of entries in the List, the function succeeds: The entry at position is copied to $x$; all List entries remain unchanged. Else: The function fails with a diagnostic error code.

Error_code List:: replace(int position, const List_entry &x);

Post: If $0 \leq \text{position} < n$, where $n$ is the number of entries in the List, the function succeeds: The entry at position is replaced by $x$; all other entries remain unchanged. Else: The function fails with a diagnostic error code.

void List:: traverse(void (*visit)(List_entry &));

Post: The action specified by function $\ast \text{visit}$ has been performed on every entry of the List, beginning at position 0 and doing each in turn.
A C++ template construction allows us to write code, usually code to implement a class, that uses objects of an arbitrary, generic type.

In template code we utilize a parameter enclosed in angle brackets < > to denote the generic type.

Later, when a client uses our code, the client can substitute an actual type for the template parameter. The client can thus obtain several actual pieces of code from our template, using different actual types in place of the template parameter.

Example: We shall implement a template class List that depends on one generic type parameter. A client can then use our template to declare several lists with different types of entries with declarations of the following form:

```cpp
List<int> first_list;
List<char> second_list;
```

Templates provide a new mechanism for creating generic data structures, one that allows many different specializations of a given data structure template in a single application.

The added generality that we get by using templates comes at the price of slightly more complicated class specifications and implementations.
template <class List_entry>
class List {
public:
  // methods of the List ADT
  List( );
  int size() const;
  bool full() const;
  bool empty() const;
  void clear( );
  void traverse(void (*visit)(List_entry &));
  Error_code retrieve(int position, List_entry &x) const;
  Error_code replace(int position, const List_entry &x);
  Error_code remove(int position, List_entry &x);
  Error_code insert(int position, const List_entry &x);

protected:
  // data members for a contiguous list implementation
  int count;
  List_entry entry[max_list];
};

In processing a contiguous list with $n$ entries:
- insert and remove operate in time approximately proportional to $n$.
- List, clear, empty, full, size, replace, and retrieve operate in constant time.
Simply Linked Implementation

Node declaration:

```cpp
template <class Node_entry>
struct Node {
   // data members
   Node_entry entry;
   Node<Node_entry> *next;
   // constructors
   Node();
   Node(Node_entry, Node<Node_entry> *link = NULL);
};
```

List declaration:

```cpp
template <class List_entry>
class List {
public:
   // Specifications for the methods of the list ADT go here.
   // The following methods replace compiler-generated defaults.
   ~List();
   List(const List<List_entry> &copy);
   void operator = (const List<List_entry> &copy);
protected:
   // Data members for the linked list implementation now follow.
   int count;
   Node<List_entry> *head;
   // The following auxiliary function is used to locate list positions
   Node<List_entry> *set_position(int position) const;
};
```
Actions on a Linked List

(a) Stacks → are → lists.

(b) Stacks → are → lists. → simple

(c) Stacks → are → list structures. → simple
   but → important → data

(d) Stacks → are → structures. → simple
   but → important → data
Function set_position takes an integer parameter position and returns a pointer to the corresponding node of the list.

Declare the visibility of set_position as protected, since set_position returns a pointer to, and therefore gives access to, a Node in the List. To maintain an encapsulated data structure, we must restrict the visibility of set_position. Protected visibility ensures that it is only available as a tool for constructing other methods of the List.

To construct set_position, we start at the beginning of the List and traverse it until we reach the desired node:

```cpp
template <class List_entry>
Node<List_entry> *List<List_entry>::set_position(int position) const
/*/ Pre: position is a valid position in the List; 0 ≤ position < count. 
 Post: Returns a pointer to the Node in position. */
{
    Node<List_entry> *q = head;
    for (int i = 0; i < position; i++) q = q->next;
    return q;
}
```

If all nodes are equally likely, then, on average, the set_position function must move halfway through the List to find a given position. Hence, on average, its time requirement is approximately proportional to \( n \), the size of the List.
In processing a linked List with $n$ entries:

- clear, insert, remove, retrieve, and replace require time approximately proportional to $n$.
- List, empty, full, and size operate in constant time.
Variation: Keeping the Current Position

- Suppose an application processes list entries in order or refers to the same entry several times before processing another entry.

- *Remember* the last-used position in the list and, if the next operation refers to the same or a later position, start tracing through the list from this last-used position.

- Note that this will not speed up *every* application using lists.

- The method `retrieve` is defined as `const`, but its implementation will need to alter the last-used position of a `List`. To enable this, we use the `mutable` qualifier. *Mutable* data members of a `class` can be changed, even by constant methods.

```cpp
template <class List_entry>
class List {
    public:
        // Add specifications for the methods of the list ADT.
        // Add methods to replace the compiler-generated defaults.

    protected:
        // Data members for the linked-list implementation with
        // current position follow:
        int count;
        mutable int current_position;
        Node<List_entry> *head;
        mutable Node<List_entry> *current;

        // Auxiliary function to locate list positions follows:
        void set_position(int position) const;
};
```

Data Structures and Program Design In C++

Trans. 12, Sect. 6.2, Implementation of Lists

All the new class members have protected visibility, so, from the perspective of a client, the class looks exactly like the earlier implementation.

The current position is now a member of the class List, so there is no longer a need for set_position to return a pointer; instead, the function simply resets the pointer current directly within the List.

```cpp
template <class List_entry>
void List<List_entry>::set_position(int position) const

/* Pre: position is a valid position in the List: 0 \leq position < count. */
/* Post: The current Node pointer references the Node at position. */
{
    if (position < current_position) {
        // must start over at head of list
        current_position = 0;
        current = head;
    }
    for (; current_position != position; current_position++)
    {
        current = current->next;
    }
}
```

For repeated references to the same position, neither the body of the if statement nor the body of the for statement will be executed, and hence the function will take almost no time.

If we move forward only one position, the body of the for statement will be executed only once, so again the function will be very fast.

If it is necessary to move backwards through the List, then the function operates in almost the same way as the version of set_position used in the previous implementation.
Doubly Linked Lists

Node definition:

```
template <class Node_entry>
struct Node {
    // data members
    Node_entry entry;
    Node<Node_entry> *next;
    Node<Node_entry> *back;

    // constructors
    Node();
    Node(Node_entry, Node<Node_entry> *link_back = NULL, 
         Node<Node_entry> *link_next = NULL);
};
```
List definition:

```cpp
template <class List_entry>
class List {
public:
    // Add specifications for methods of the list ADT.
    // Add methods to replace compiler generated defaults.

protected:
    // Data members for the doubly-linked list implementation follow:
    int count;
    mutable int current_position;
    mutable Node<List_entry> *current;

    // The auxiliary function to locate list positions follows:
    void set_position(int position) const;
};
```

- We can move either direction through the List while keeping only one pointer, current, into the List.
- We do not need pointers to the head or the tail of the List, since they can be found by tracing back or forth from any given node.
To find any position in the doubly linked list, we first decide whether to move forward or backward from the current position, and then we do a partial traversal of the list until we reach the desired position.

```cpp
template <class List_entry>
void List<List_entry>::set_position(int position) const
/**
 * Pre: position is a valid position in the List: 0 \leq position < count.
 * Post: The current Node pointer references the Node at position. */
{
    if (current_position <= position)
        for (; current_position != position; current_position++)
            current = current->next;
    else
        for (; current_position != position; current_position--)
            current = current->back;
}
```

The cost of a doubly linked list is the extra space required in each Node for a second link, usually trivial in comparison to the space for the information member entry.
Insertion into a doubly linked list
### Comparison of Implementations

Contiguous storage is generally preferable
- when the entries are individually very small;
- when the size of the list is known when the program is written;
- when few insertions or deletions need to be made except at the end of the list; and
- when random access is important.

Linked storage proves superior
- when the entries are large;
- when the size of the list is not known in advance; and
- when flexibility is needed in inserting, deleting, and re-arranging the entries.

To choose among linked list implementations, consider:
- Which of the operations will actually be performed on the list and which of these are the most important?
- Is there _locality_ of reference? That is, if one entry is accessed, is it likely that it will next be accessed again?
- Are the entries processed in sequential order? If so, then it may be worthwhile to maintain the last-used position as part of the list structure.
- Is it necessary to move both directions through the list? If so, then doubly linked lists may prove advantageous.
1. Use C++ templates to implement generic data structures.

2. Don’t confuse contiguous lists with arrays.

3. Choose your data structures as you design your algorithms, and avoid making premature decisions.

4. Always be careful about the extreme cases and handle them gracefully. Trace through your algorithm to determine what happens when a data structure is empty or full.

5. Don’t optimize your code until it works perfectly, and then only optimize it if improvement in efficiency is definitely required. First try a simple implementation of your data structures. Change to a more sophisticated implementation only if the simple one proves too inefficient.

6. When working with general lists, first decide exactly what operations are needed, and then choose the implementation that enables those operations to be done most easily.

7. In choosing between linked and contiguous implementations of lists, consider the necessary operations on the lists. Linked lists are more flexible in regard to insertions, deletions, and rearrangement; contiguous lists allow random access.

8. Contiguous lists usually require less computer memory, computer time, and programming effort when the items in the list are small and the algorithms are simple. When the list holds large data entries, linked lists usually save space, time, and often programming effort.
9. Dynamic memory and pointers allow a program to adapt automatically to a wide range of application sizes and provide flexibility in space allocation among different data structures. Static memory (arrays and indices) is sometimes more efficient for applications whose size can be completely specified in advance.

10. For advice on programming with linked lists in dynamic memory, see the guidelines in Chapter 4.

11. Avoid sophistication for sophistication’s sake. Use a simple method if it is adequate for your application.

12. Don’t reinvent the wheel. If a ready-made class template or function is adequate for your application, consider using it.