Chapter 19

Custom Templatized Data Structures

OBJECTIVES

- Form linked data structures using pointers, self-referential classes and recursion
- Create and manipulate dynamic data structures such as linked lists

Topics

- 19.1 Introduction
- 19.2 Self-Referential Classes
- 19.3 Linked Lists

19.1 Introduction

- Create our own custom templatized dynamic data structures

- Linked list
  - Collection of data items logically “lined up in a row”-insertions and removals are made anywhere in a linked list

Topics

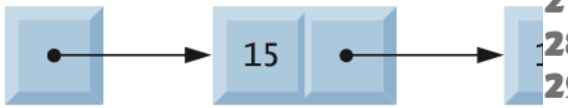
- 19.1 Introduction
- **19.2 Self-Referential Classes**
- 19.3 Linked Lists

```
class Node
{
public:
   explicit Node( int ); // constructor
   void setData( int ); // set data member
   int getData() const; // get data member
   void setNextPtr( Node * ); // set pointer to next Node
   Node *getNextPtr() const; // get pointer to next Node
private:
   int data; // data
   Node *nextPtr; //
}; // end class Node
```

ListNode<int> n1(1);

ListNode<int> *ptr1=new ListNode<int> (1);

```
8
9
10 template< typename NODETYPE >
11 class ListNode
12 {
13
14
15 public:
16    explicit ListNode( const NODETYPE &info ) // constructor
17       : data( info ), nextPtr( nullptr )
18    {
19       // empty body
20    } // end ListNode constructor
21
22    NODETYPE getData() const; // return data in node
23    {
24       return data;
25    } // end function getData
26 private:
27    NODETYPE data; // data
28    ListNode< NODETYPE > *nextPtr; // next node in list
29 }; // end class ListNode
```



Topics

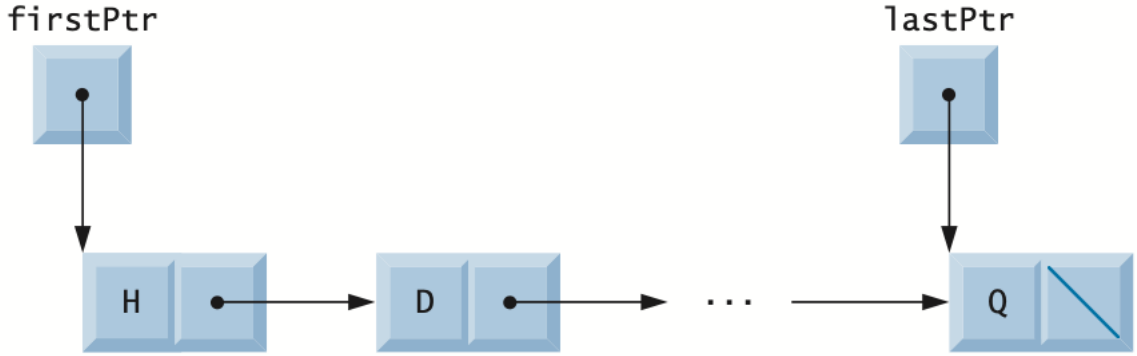
- 19.1 Introduction
- 19.2 Self-Referential Classes
- **19.3 Linked Lists**

19.3 Linked Lists

- A linked list is a linear collection of self-referential class objects, called nodes, connected by pointer links.
- A node can contain data of any type, including objects of other classes.

19.3 Linked Lists

- Linked-list nodes typically are not stored contiguously in memory, but logically they appear to be contiguous
- Linked list vs vector



19.3 Linked Lists

- List 类模板需求
  - insert a value at the beginning of the List
  - insert a value at the end of the List
  - delete a value from the beginning of the List
  - delete a value from the end of the List
  - print all elements in List
- List< int > integerList;

  integerList.insertAtFront(2);

  integerList.print();

  integerList.removeFromFront(value);

19.3 Linked Lists

```
 8  template< typename NODETYPE > class List;
 9
10  template< typename NODETYPE >
11  class ListNode
12  {
13     friend class List< NODETYPE >; // make List a friend
14
15  public:
16     explicit ListNode( const NODETYPE &info ) // constructor
17        : data( info ), nextPtr( nullptr )
18     {
19        // empty body
20     } // end ListNode constructor
21
22     NODETYPE getData() const; // return data in node
23     {
24        return data;
25     } // end function getData
26  private:
27     NODETYPE data; // data
28     ListNode< NODETYPE > *nextPtr; // next node in list
29  }; // end class ListNode
```

- List< int > integerList;
  integerList.inseertAtFront(2);
  integerList.print();
  integerList.removeFromFront(value);

```
template< typename NODETYPE >
class List
{
private:
    ListNode< NODETYPE > *firstPtr; // pointer to first node
    ListNode< NODETYPE > *lastPtr; // pointer to last node

    // utility function to allocate new node
    ListNode< NODETYPE > *getNewNode( const NODETYPE &value )
    {
       return new ListNode< NODETYPE >( value );
    } // end function getNewNode
public:
    // default constructor
   List() : firstPtr( nullptr ), lastPtr( nullptr ){ } // end List constructor
```

19.3 Linked L

- List< int > integerList;
  integerList.inseertAtFront(2);
  integerList.print();
  integerList.removeFromFront(value);

```
// insert node at front of list
void insertAtFront( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      newPtr->nextPtr = firstPtr; // point new node to old 1st node
      firstPtr = newPtr; // aim firstPtr at new node
   } // end else
} // end function insertAtFront

// is List empty?
bool isEmpty() const
{
   return firstPtr == nullptr;
} // end function isEmpty
```

19.3 Linked L

- List< int > integerList;
  integerList.inseertAtFront(2);
  integerList.print();
  integerList.removeFromFront(value);

```
// insert node at back of list
void insertAtBack( const NODETYPE &value )
{
   ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
   {
      lastPtr->nextPtr = newPtr; // update previous last node
      lastPtr = newPtr; // new last node
   } // end else
} // end function insertAtBack
```

19.3 Linked L

- List< int > integerList;
  integerList.inseertAtFront(2);
  integerList.print();
  integerList.removeFromFront(value);

```
// delete node from front of list
bool removeFromFront( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = firstPtr; // hold item to delete

      if ( firstPtr == lastPtr )
         firstPtr = lastPtr = nullptr; // no nodes remain after removal
      else
         firstPtr = firstPtr->nextPtr; // point to previous 2nd node

      value = tempPtr->data; // return data being removed
      delete tempPtr; // reclaim previous front node
      return true; // delete successful
   } // end else
} // end function removeFromFront
```

- List< int > integerList;
  integerList.inseertAtFront(2);
  integerList.print();
  integerList.removeFromFront(value);

```
// delete node from back of list
bool removeFromBack( NODETYPE &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
   {
      ListNode< NODETYPE > *tempPtr = lastPtr; // hold item to delete

      if ( firstPtr == lastPtr ) // List has one element
         firstPtr = lastPtr = nullptr; // no nodes remain after removal
      else
      {
         ListNode< NODETYPE > *currentPtr = firstPtr;

         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr; // move to next node

         lastPtr = currentPtr; // remove last node
         currentPtr->nextPtr = nullptr; // this is now the last node
      } // end else

      value = tempPtr->data; // return value from old last node
      delete tempPtr; // reclaim former last node
      return true; // delete successful
   } // end else
} // end function removeFromBack
```

19.3 Linked Lists

```
// display contents of List
void print() const
{
   if ( isEmpty() ) // List is empty
   {
      std::cout << "The list is empty\n\n";
      return;
   } // end if

   ListNode< NODETYPE > *currentPtr = firstPtr;

   std::cout << "The list is: ";

   while ( currentPtr != nullptr ) // get element data
   {
      std::cout << currentPtr->data << ' ';
      currentPtr = currentPtr->nextPtr;
   } // end while

   std::cout << "\n\n";
} // end function print
```

19.3 Linked Lists

```
// destructor
~List()
{
   if ( !isEmpty() ) // List is not empty
   {
      cout << "Destroying nodes ...\n";
      ListNode< NODETYPE > *currentPtr = firstPtr;
      ListNode< NODETYPE > *tempPtr;
      while ( currentPtr != nullptr ) // delete remaining nodes
      {
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
      } // end while
   } // end if
   cout << "All nodes destroyed\n\n";
} // end List destructor
```

```
// function to test a List
template< typename T >
void testList( List< T > &listObject, const string &typeName )
{
   cout << "Testing a List of " << typeName << " values\n";
   instructions(); // display instructions
   int choice; // store user choice
   T value; // store input value
   do // perform user-selected actions
   {
      cout << "? ";
      cin >> choice;
      switch ( choice )
      {
         case 1: // insert at beginning
            cout << "Enter " << typeName << ": ";
            cin >> value;
            listObject.insertAtFront(
            listObject.print();
            break;
         case 2: // insert at end
            cout << "Enter " << typeNa
            cin >> value;
            listObject.insertAtBack( v
            listObject.print();
            break;
```

```
int main()
{
   // test List of int values
   List< int > integerList;
   testList( integerList, "integer" );

   // test List of double values
   List< double > doubleList;
   testList( doubleList, "double" );
} // end main
```

```
         case 3: // remove from beginning
            if ( listObject.removeFromFront( value ) )
               cout << value << " removed from list\n";
            listObject.print();
            break;
         case 4: // remove from end
            if ( listObject.removeFromBack( value ) )
               cout << value << " removed from list\n";
            listObject.print();
            break;
      } // end switch
   } while ( choice < 5 ); // end do...while
   cout << "End list test\n\n";
} // end function testList
```

19.3 Linked Lists

- circular, singly linked list
- doubly linked list