Linked list (unsorted)
Header file

```
typedef desired-type-of-list-item listItemType;

struct Node        // a node on the list
{
    listItemType item;  // a data item on the list
    nodePtr      next;  // pointer to next node
};  // end struct
typedef Node* nodePtr;  // pointer to node

class listClass
{
public:
// constructors and destructor:
   listClass();                    // default constructor
   listClass(const listClass& L); // copy constructor
   ~listClass();                   // destructor

// list operations:
   bool ListIsEmpty() const;
   int ListLength() const;
   void ListInsert(int NewPosition, listItemType NewItem,  bool& Success);
   void ListDelete(int Position, bool& Success);
   void ListRetrieve(int Position, listItemType& DataItem, bool& Success) const;

private:
   int     Size;  // number of items in list
   nodePtr Head;  // pointer to linked list of items

   nodePtr PtrTo(int Position) const;
   // Returns a pointer to the Position-th node
   // in the linked list.
}; // end class
// End of header file.
```

Implementation file

```
#include "ListP.h"     // header file
#include <cstddef>    // for NULL
#include <cassert>    // for assert()
using namespace std;

listClass::listClass(): Size(0), Head(NULL)
{
}  // end default constructor

listClass::listClass(const listClass& L): Size(L.Size)
{
  if (L.Head == NULL)
    Head = NULL;  // original list is empty

  else
  {
    // copy first node
    Head = new Node;
    assert(Head != NULL);  // check allocation
    Head->item = L.Head->item;

    // copy rest of list
    nodePtr NewPtr = Head;  // new list pointer

    // NewPtr points to last node in new list
    // OrigPtr points to nodes in original list
    for (nodePtr OrigPtr = L.Head->next;   OrigPtr != NULL;  OrigPtr = OrigPtr->next)
    {
      NewPtr->next = new Node;
      assert(NewPtr->next != NULL);
      NewPtr = NewPtr->next;
      NewPtr->item = OrigPtr->item;
    }  // end for

    NewPtr->next = NULL;
  }  // end if
}  // end copy constructor

listClass::~listClass()
{
  bool Success;

  while (!ListIsEmpty())
    ListDelete(1, Success);
} // end destructor
```

```
bool listClass::ListIsEmpty() const
{
  return bool(Size == 0);
}  // end ListIsEmpty

int listClass::ListLength() const
{
  return Size;
}  // end ListLength

nodePtr listClass::PtrTo(int Position) const
// --------------------------------------------------
// Locates a specified node in a linked list.
// Precondition: Position is the number of the desired node.
// Postcondition: Returns a pointer to the desired node. If Position < 1 or Position > the number of
// nodes in the list, returns NULL.
// --------------------------------------------------
{
  if ( (Position < 1) || (Position > ListLength()) )
    return NULL;

  else  // count from the beginning of the list
  {  nodePtr Cur = Head;
    for (int Skip = 1; Skip < Position; ++Skip)
      Cur = Cur->next;
    return Cur;
  }  // end if
}  // end PtrTo

void listClass::ListRetrieve(int Position, listItemType& DataItem, bool& Success) const
{
  Success = bool( (Position >= 1) && (Position <= ListLength()) );

  if (Success) // get pointer to node, then data in node
  {
    nodePtr Cur = PtrTo(Position);
    DataItem = Cur->item;
  }  // end if
} // end ListRetrieve
```

```
void listClass::ListInsert(int NewPosition, listItemType NewItem,  bool& Success)
{
  int NewLength = ListLength() + 1;

  Success = bool( (NewPosition >= 1) &&  (NewPosition <= NewLength) );

  if (Success) // create new node and place NewItem in it
  {
    nodePtr NewPtr = new Node;
    Success = bool(NewPtr != NULL);
    if (Success)
    {
      Size = NewLength;
      NewPtr->item = NewItem;

      // attach new node to list
      if (NewPosition == 1) // insert new node at beginning of list
      {
        NewPtr->next = Head;
        Head = NewPtr;
      }

      else
      {  nodePtr Prev = PtrTo(NewPosition-1);   // insert new node after node to which Prev points
        NewPtr->next = Prev->next;
        Prev->next = NewPtr;
      }  // end if
    }  // end if
  }  // end if
} // end ListInsert

void listClass::ListDelete(int Position,   bool& Success)
{
  nodePtr Cur;

  Success = bool( (Position >= 1) &&  (Position <= ListLength()) );

  if (Success)
  {  --Size;
    if (Position == 1) // delete the first node from the list
    {
      Cur = Head;  // save pointer to node
      Head = Head->next;
    }
    else
    {  nodePtr Prev = PtrTo(Position-1);         // delete the node after the node to which Prev points
      Cur = Prev->next;  // save pointer to node
      Prev->next = Cur->next;
    }  // end if
```

```
      // return node to system
      Cur->next = NULL;
      delete Cur;
      Cur = NULL;
   }  // end if
}  // end ListDelete
```