

## The list ADT is back ...

### mytest3.cc

```
#include <iostream.h>
#include <stdlib.h>
#include "ListP.h"

. . .

int main(void)
{
    listClass mylist;
    . . .
    m.ListInsert( ... );
    . . .
    m.ListRetrieve( ... );
    . . .
    return EXIT_SUCCESS;
}
```

### ListP.h

```
. . .
class listClass
{ . . .
};
```

ADT  
specification

ADT  
implementation

### ListP.cc

```
#include "ListP.h"

. . .
void listClass::ListInsert(...)
{
    Success = bool . . .
    if (Success)
        . . .
}
. . .
```

## Here is ListP.h for the ADT list

```
typedef double listItemType; // SPECIFY TYPE OF LIST ITEM

struct listNode; // linked list node
typedef listNode* ptrType; // pointer to node

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

    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
    ptrType Head; // pointer to linked list of items

    ptrType PtrTo(int Position) const;
    // Returns a pointer to the Position-th node in the linked list.
};
```

## ListP.cc

```
// *****  
// Implementation file ListP.cpp for the ADT list.  
// Pointer-based implementation.  
// *****  
#include "ListP.h" // header file  
#include <stddef.h> // for NULL  
#include <assert.h> // for assert()  
  
struct listNode // a node on the list  
{ listItemType Item; // a data item on the list  
  ptrType Next; // pointer to next node  
}; // end struct  
  
listClass::listClass(): Size(0), Head(NULL)  
{  
} // end default constructor
```

## ListP.cc (continued)

```
listClass::listClass(const listClass& L): Size(L.Size)  
{  
  if (L.Head == NULL)  
    Head = NULL; // original list is empty  
  else  
  { // copy first node  
    Head = new listNode;  
    assert(Head != NULL); // check allocation  
    Head->Item = L.Head->Item;  
  
    // copy rest of list  
    ptrType NewPtr = Head; // new list pointer  
    // NewPtr points to last node in new list  
    // OrigPtr points to nodes in original list  
    for (ptrType OrigPtr = L.Head->Next;  
         OrigPtr != NULL;  
         OrigPtr = OrigPtr->Next)  
    { NewPtr->Next = new listNode;  
      assert(NewPtr->Next != NULL);  
      NewPtr = NewPtr->Next;  
      NewPtr->Item = OrigPtr->Item;  
    } // end for  
  
    NewPtr->Next = NULL;  
  } // end if  
} // end copy constructor
```

## ListP.cc (continued)

---

```
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
```

## ListP.cc (continued)

---

```
ptrType 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
    { ptrType Cur = Head;
      for (int Skip = 1; Skip < Position; ++Skip)
          Cur = Cur->Next;
      return Cur;
    } // end if
} // end PtrTo
```

## ListP.cc (continued)

---

```
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
        ptrType Cur = PtrTo(Position);
        DataItem = Cur->Item;
    } // end if
} // end ListRetrieve
```

## ListP.cc (continued)

---

```
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
        ptrType NewPtr = new listNode;
        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;
          }
        }
    }
}
```

## ListP.cc (continued)

```
void listClass::ListInsert(int NewPosition,
                           listItemType NewItem,
                           bool& Success)
{
    .....
    .....
    else
    { ptrType 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 ListInsert
```

## ListP.cc (continued)

```
void listClass::ListDelete(int Position,
                           bool& Success)
{
    ptrType 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
      { ptrType 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
```