# Example of the C++ "exception class" standard protocol with the ListP class

C++ provides a standard "class exception" design that is usually used for coding exception handlers.

- The base exception class has a constructor that accepts a single error-message string value, which is stored in a member variable.
- The base exception class has an access method "what( )", that simply returns a reference to the error message string. e.g.,
  cout << ec.what( ) << endl;
- Each different type of exception for a problem is coded as a different new class.

There are several "standard exceptions" that may occur.

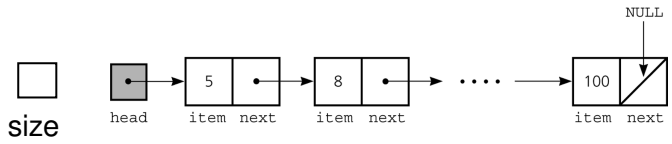
- **exception** is the predefined base class that represents all types.
- **out_of_range:exception** is derived from the base exception class, and typically represents errors that involve values that exceed their specified limits.

- or, you can code up your own custom "Exception" class and throw/catch it!

## ListExceptions.h

```
#include <string>
using namespace std;

class ListException {      // inline implementation
public:
   ListException(const string mssg = "")
      : message(mssg)
   { }

   string what()  { return message; }

private:
   string message;
};

class ListIndexOutOfRangeException {
public:
   ListIndexOutOfRangeException(const string mssg = "")
      : message(mssg)
   { }

   string what()  { return message; }

private:
   string message;
};
```

```
//*****************************************************
// Header file ListP.h for the ADT list.
// Pointer-based implementation, with Exceptions.
//*****************************************************
#include "ListExceptions.h"

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

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

// list operations:
   bool isEmpty() const;
   int  getLength() const;          // Access method

//   bool insert(int index,   ListItemType newItem);
//   void insert(int index,   ListItemType newItem, bool &success);
   void insert(int index, ListItemType newItem);


   void remove(int index);


   void retrieve(int index, ListItemType& dataItem) const;

private:
   struct ListNode                  // a node on the list
   {
      ListItemType  item;           // a data item on the list
      ListNode     *next;           // pointer to next node
   };

   int size;           // number of items in list
   ListNode *head;     // pointer to linked list of items

   ListNode *PtrTo(int index) const;
   // Returns pointer to the index-th node in list (1 .. k)
};
```




Deleted the explicit "success" parameter or return value. Throw exception instead.

```
//****************************************************
// Implementation file ListP.cpp for the ADT list.
// Pointer-based implementation.
//****************************************************
#include "ListP.h"          // header file
#include <cstddef>          // for NULL
#include <cassert>          // for assert()
```

---

```
List::List()                        // Default Constructor
    : size(0), head(NULL)
{
}
```

---

```
List::~List()                       // Destructor
{
   while (!isEmpty())
      remove(1);
}
```

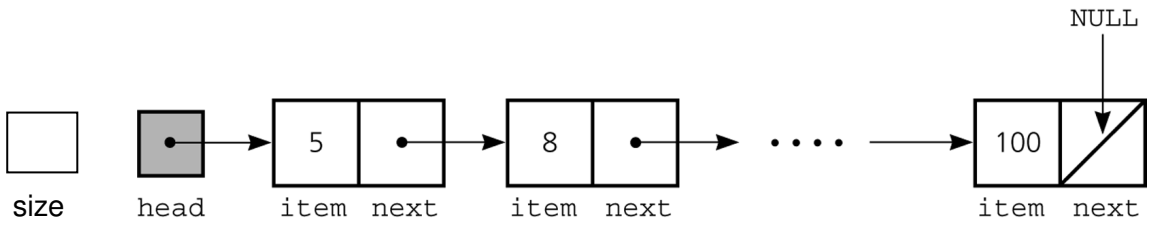
---

```
bool List::isEmpty() const
{
   return bool(size == 0);
}
```

---

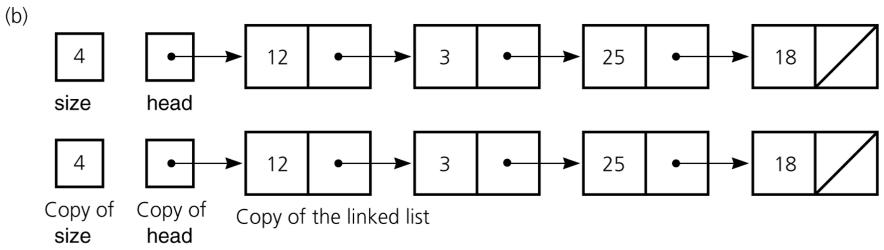
```
int List::getLength() const    // Access method
{
   return size;
}
```

---

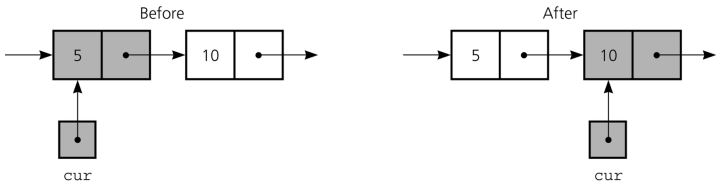


```
// Copy Constructor: Make DEEP Copy
List::List(const List& aList)
{
   size = aList.size;

   if (aList.head == NULL)
   {
      head = NULL; // original list is empty
   }
   else
   {
      // copy first node
      head = new ListNode;
      assert(head != NULL); // check allocation

      head->item = aList.head->item;
      head->next = NULL;

      // copy rest of list
      ListNode *newPtr = head; // new list pointer

      // newPtr points to last node in new list
      // origPtr points to nodes in original list

      for (ListNode *origPtr = aList.head->next;
                     origPtr != NULL;
                     origPtr = origPtr->next)
      {
         newPtr->next = new ListNode;
         assert(newPtr->next != NULL);
         newPtr = newPtr->next;

         newPtr->item = origPtr->item;
         newPtr->next = NULL;
      }
   }
}
```

(b)

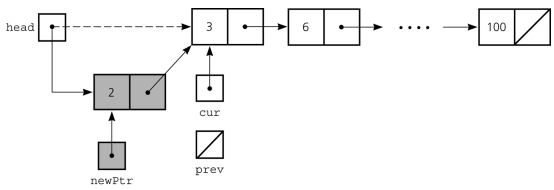


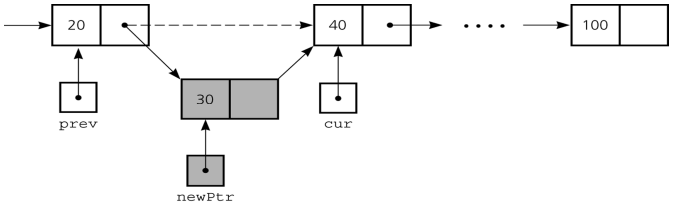
```
List::ListNode *List::PtrTo(int index) const
// ------------------------------------------------
// Locates a specified node in a linked list.
// Precondition: index is the number of the
// desired node.  Valid range is (1 .. size).
// Postcondition: Returns a pointer to the desired node.
// If index < 1 or index > the number of nodes in the list,
//    returns NULL.
// ------------------------------------------------
{
   if ( (index < 1) || (index > size) )
      return NULL;

   else // count from the beginning of the list
   {
      ListNode *cur = head;

      for (int skip = 1; skip < index; ++skip)
         cur = cur->next;

      return cur;
   }
}
```




```
void List::retrieve(int index,
                    ListItemType& dataItem) const
{
   if ((index < 1) || (index > getLength()))
      throw ListIndexOutOfRangeException(
        "ListOutOfRangeException: "
        "retrieve index out of range" );
   else
   {
      // get pointer to node, then data in node
      ListNode *cur = PtrTo(index);

      dataItem = cur->item;
   }
}
```

```
void List::insert(int index, ListItemType newItem)
{
   if ((index < 1) || (index > size + 1))
      throw ListIndexOutOfRangeException(
         "ListOutOfRangeException: "
         "insert index out of range");

   else
   {
      // create new node and place newItem in it
      ListNode *newPtr = new ListNode;

      if (newPtr == NULL)
         throw ListException(
           "ListException: insert cannot allocate memory");
      else
      {
         ++size;

         newPtr->item = newItem;

         // attach new node to list
         if (index == 1)
         {
            // insert new node at beginning of list
            newPtr->next = head;
            head = newPtr;
         }
         else
         {
            // insert new node after node
            // to which prev points
            ListNode *prev = PtrTo(index-1);

            newPtr->next = prev->next;
            prev->next = newPtr;
         }
      }
   }
}
```



```
void List::remove(int index)
{
   ListNode *cur;

   if ((index < 1) || (index > getLength()))
      throw ListIndexOutOfRangeException(
         "ListOutOfRangeException: "
         "remove index out of range");

   else
   {
      --size;

      if (index == 1)
      {
         // delete the first node from the list

         // Locate node to be deleted: cur
         cur = head;

        // Remove cur node from the list
         head = head->next;
      }

      else
      {
         // Locate previous node
         ListNode *prev = PtrTo(index-1);

         // Locate node to be deleted: cur
         cur = prev->next;

        // Remove cur node from the list
         prev->next = cur->next;
      }

      // return node to system
      cur->next = NULL;          // safety
      delete cur;
      cur = NULL;                // safety
   }
}
```