```
// RList.cpp: implementation of the RList class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "RList.h"
#include "Node.h"
#include <iostream>
#include <iomanip>
//#include "LogNode.h"
#include <conio.h> // cprintf

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

      //          Next->                Next->        Next-> NULL
      //          |                         |             |
      //           ____            ____          ____
      //          |node|                |node|        |node|
      //          |____|          |____|        |____|            NULL
      //
      //        |               |             |
      // NULL     <-Previous      <-Previous    <-Previous
      //            ^                       ^             ^            ^
      //            |                       |             |              |
      //            head                  cursor          tail
buffer
      //            nodePtr         nodePtr         nodePtr        nodePtr

      //~maa~20011113

// This class constructor is responsible for initializing class
// data members. It does so by calling the Initialize function.
RList::RList()
{
      this->Initialize();
}

RList::~RList()
{
      this->Destroy();
}

// This class method is responsible for "queueing" a node from the
// buffer onto the tail of the linked list.
// Preconditions:
// Postconditions: The new node is either add to the end of an existing
// liked list, or a new list is created with pnode as its only member.
bool RList::Queue(CNode *pNode)
{
      bool bQueued = false;

      //~bvs~20021208:  Indicate that this list is being used
      // now in a FIFO manner.
      this->SetType(FIFO);
      if (pNode)
      {
            if(this->m_pTail!=NULL)
            {
                  // There already was a tail, so make this one the new tail.
                  this->m_pTail->m_pNext=pNode;
                  pNode->m_pPrevious = this->m_pTail;
                  pNode->m_pNext = NULL;
                  this->m_pTail = pNode;
            }
            else
            {
                  // Was "empty" so this will result in a list of one node.
                  // In this case, all pointers should point to the new node.
                  this->m_pCursor=this->m_pHead=this->m_pTail = pNode;
            }

            //~bvs~20040806:  Connect the node to this list.
            pNode->m_pList=this;

            bQueued = true;
      }
      return bQueued;
      //~maa~20011113
      //~bvs~20030429
}

// This class method returns a pointer to whatever was in the buffer
// and then empties the buffer (i.e., by pointing the buffer to NULL).
// Preconditions:
// Postconditions: The function points pNode to whatever the
// "buffer" pointer points to and returns a node pointer.
CNode* RList::TakeFromBuffer()
{
      // The purpose of RList is to manage a "linked list"
      // of nodes.  However, sometimes it is either
      // convenient or necessary to manage a node that is
      // not currently part of the linked list.  Hence the
      // need for a "buffer" member that points to an
      // isolated node.
      //
      // |--------|     |----| <----|----| <----|----|
      // |  LIST  |     |NODE|            |NODE|            |NODE|
      // |________|     |----|----> |----|----> |----|
      // |        |     |             |                 |
      // |head--->|----/                    |                 |
      // |cursor->|----------------/                    |
      // |tail--->|----------------------------/
      // |        |
      // |________|           |----|
      // |buffer->|----->     |NODE|
      // |--------|           |----|
      //
      // The function has two purposes.  The first is to
      // give the caller access to the node pointed to by
      // the buffer member.  The second purpose is to
      // "empty the buffer" by making the buffer member
```

```
    // point to NULL.

    CNode *pNode = this->m_pBuffer;
    this->m_pBuffer = NULL;

    return pNode;

    //~maa~20011113
    //~bvs~20020518
}

// This function attempts to "initiate" a new list with pNode.
// Preconditions: This class method "works" only when the linked list
// is empty.
// Postconditions: Provided pNode is not NULL, a new list will be initiated
// with pNode as its only member. If pNode is NULL,an empty list will be
// initiated with whatever is in the buffer.If there is nothing in the
// buffer, the empty list will remain empty. If a new list is successfully
// created, the boolean initiated is set to true and returned to the caller.
bool RList::InitiateList(CNode *pNode)
{
    // Declare and initialize local variables
    bool initiated = false;

    // When there is already something in the list,
    // the list can not be initiated.
    if (this->IsSomethingInTheList())
    {
    }
    else
    {
        // If pNode is not NULL it will become the only node in the list
        if (pNode)
            this->m_pHead = this->m_pCursor = this->m_pTail = pNode;
        // Otherwise the whatever is in the buffer will become the only
        // member of the list. If the buffer is empty all pointer are set
        // to NULL.
        else
            this->m_pHead = this->m_pCursor = this->m_pTail = this-
>TakeFromBuffer();
        initiated = true;
    }
    return initiated;

    //~maa~20011113
    //~maa~20011114
}

// This class method returns a boolean value to indicate the
// status of the linked list.
// Postconditions: A boolean value is returned indicating true if there
// if something in the list, false otherwise.
bool RList::IsSomethingInTheList()
{
    return this->m_pHead !=NULL;

    //~maa~20011113
```

```
}

// This class method returns a boolean value to indicate the
// status of the linked list.
// Postconditions:If there are no links in the
// list, then a boolean TRUE is returned, otherwise FALSE.
bool RList::IsSomethingInTheBuffer()
{
      //
      // |--------|     |----| <----|----| <----|----|
      // |  LIST  |     |NODE|            |NODE|            |NODE|
      // |________|     |----|----> |----|----> |----|
      // |        |     |             |                 |
      // |head--->|----/                    |                 |
      // |cursor->|----------------/                    |
      // |tail--->|----------------------------/
      // |        |
      // |________|           |----|
      // |buffer->|----->     |NODE|
      // |--------|           |----|
      //

      return this->m_pBuffer !=NULL;

      //~maa~20011113
}

// This class method returns a boolean true if
// the cursor points to a valid Node.
bool RList::IsCursorValid()
{
      //
      // |--------|     |----| <----|----| <----|----|
      // |  LIST  |     |NODE|            |NODE|            |NODE|
      // |________|     |----|----> |----|----> |----|
      // |        |     |             |                 |
      // |head--->|----/                    |                 |
      // |cursor->|----------------/                    |
      // |tail--->|----------------------------/
      // |        |
      // |________|           |----|
      // |buffer->|----->     |NODE|
      // |--------|           |----|
      //


      return this->m_pCursor !=NULL;

      //~maa~20011113
      //~maa~20020523
}

// This function removes any nodes from the buffer.
// Preconditions: There must be something in the buffer. I.E. the
// buffer pointer must be pointing to something.
// Postconditions: If there is something in the buffer, it is deleted
// and its pointer is set to NULL. A boolean true is returned to the
// caller indicating the success of the function. If there was nothing
// to delete, false is returned.
bool RList::PurgeBuffer()
{
      //
      // |--------|            ----  ----  ----  ----
      // |  LIST  |           |NODE|  |NODE|    |NODE|      |NODE|
      // |        |          ----    ----  ----    ----
      // |        |         |          |                       |
      // |head--->|--------/           |                       |
      // |cursor->|----------------/                           |
      // |tail--->|--------------------------------/
      // |        |
      // |        |            ----
      // |buffer->|----->     |NODE|
      // |--------|            ----
      //
      // Check to see if there was anything in the buffer.
      if(this->m_pBuffer !=NULL)
      {
            // If so delete it and set the pointer to NULL.
            // Return true to the caller. Otherwise return false.
            delete this->m_pBuffer;
            this->m_pBuffer = NULL;
            return true;
      }
      return false;

      //~maa~20011113


}

// This class method will destroy the linked list and re-set
// all link pointers to initialized (NULL) values.
// Preconditions: There must be something in the list.
// Postconditions: All of the nodes in the list are popped off and
// destroyed. A counter keeps track of just how many
// nodes are destroyed and the count is returned to the caller.
int RList::Destroy()
{
      // Declare and initialize local variables.
      int numberOfNodesDestroyed=0;

      // {programming hint:  Repeatedly "pop off" the head node
      // and delete it, until the linked list becomes empty.  Note
      // that "popping" a node places it into the buffer, so the
      // buffer must subsequently be purged}

      // If there is already a node object in the buffer, it will be
      // destroyed and counted as deleted.
      if (this->PurgeBuffer() == true)
            numberOfNodesDestroyed++;

      // Any nodes in the list will be popped into the buffer and purged.
      while(this->Pop())
      {
            numberOfNodesDestroyed++;
      //logNode.Detail("Number of Nodes
Destroyed(%d)",numberOfNodesDestroyed);

      }

      this->PurgeBuffer();
      //logNode.Detail("RList::Destroy");
      return numberOfNodesDestroyed;

      //~maa~20001013
      //~maa~20011114
}

// This class method will take the node object from the
// buffer and "push it" (i.e., insert it) onto the front
// of the linked list.
// Preconditions:
// Postconditions: If there is something in the list, the new node is
// added to the front of the list. If there is nothing in the list, the
// function tries to initiate the list with whatever is in the buffer.
// If either aspect of the function is successful, a boolean true is
// returned to the caller.
bool RList::Push()
{
      return this->PushFromBufferIntoList();
}

// This class method will:
// (1) "pop" (i.e., remove) the first (i.e., head) node object from
// the linked list,
// (2) purge whatever is currently in the buffer, and then
// (3) place the "popped" node into the buffer.
// Precondtions: The node that is to be popped must not already be
// in the buffer.
// Postconditions: The first node of a list is placed into the buffer
// and the rest of the list's pointers are adjusted accordingly.
bool RList::Pop()
{

      return this->PlaceIntoBuffer(this->m_pHead);

      //~maa~20011114
}

// This class method "cuts" the node object that the "cursor"
// points to from the linked list and and places the node
// object into the "buffer."
// Preconditions: What is to be cut must not already be in the buffer.
// Postconditions: The node that the cursor points to is removed from the
// linked list and placed into the buffer.
bool RList::Cut()
{
      return this->PlaceIntoBuffer(this->m_pCursor);

      //~maa~20011114
```

```
}

// This class method "pastes" the node object from the "buffer"
// into the linked list at the location pointed to by the "cursor."
// Note that this will empty the buffer and insert the pasted
// link before the object pointed to by the cursor.
// Preconditions:
// (1) There must be something in the buffer.
// (2) There must be something in the list.
// (3) The cursor must be pointing to a node.
// Postconditions:
// (1) If all the preconditions are met, the new node
// is inserted into the list as the cursor's previous.
// (2) If the cursor is not valid the new node is added at the
// end of the list and the function interprets the call as a "queue."
// (3) If the list is empty, the function will interpret the call as an
// "initiate" and try to create a new list with what's in the buffer.
// If any of the function calls are successful a boolean true is returned
// to the caller.

//
      // |--------| ----       ----  ----
      // |  LIST  ||NODE|     |NODE|  |NODE|
      // |        | ----       ----    ----
      // |        |         | |       |
      // |head--->|--------/  |       |
      // |cursor->|----------/        |
      // |tail--->|----------------/
      // |        |
      // |        |            ----
      // |buffer->|----->     |NODE|
      // |--------|            ----
      //
//

void RList::Reverse()
{
//LinkedListNode start = linkedList.Head;
      this->m_pCursor = this->m_pHead;

  //LinkedListNode temp = null;
      this->m_pBuffer = NULL;

// ------------------------------------------------------------
// Loop through until null node (next node of the latest node) is found
// ------------------------------------------------------------

//while (start != null)
      while(this->m_pCursor !=NULL)
{
// ------------------------------------------------------------
// Swap the “Next” and “Previous” node properties
// ------------------------------------------------------------

//temp = start.Next;
      this->m_pBuffer = this->m_pCursor->m_pNext;
//start.Next = start.Previous;;
```

```
      this->m_pCursor->m_pNext = this->m_pCursor->m_pPrevious;
//start.Previous = temp;
      this->m_pCursor->m_pPrevious = this->m_pBuffer;

// ------------------------------------------------------------
// Head property needs to point to the latest node
// ------------------------------------------------------------

//if (start.Previous == null)
if(this->m_pCursor->m_pPrevious ==NULL)
{
//linkedList.Head = start;
      this->m_pHead = this->m_pCursor;
}

// ------------------------------------------------------------
// Move on to the next node (since we just swapped
// “Next” and “Previous”
// “Next” is actually the “Previous”
// ------------------------------------------------------------

//start = start.Previous;
this->m_pCursor = this->m_pCursor->m_pPrevious;
}
}

bool RList::Paste()
{
      // Declare and initialize local variables.
      bool wasPasted = false;

      if (this->IsSomethingInTheBuffer())
      {
            if (this->IsSomethingInTheList())
            {
                  if (this->IsCursorValid())
                  {
                        // The list is not empty and the cursor is valid,
                        // so insert the buffered node before the cursored
node.
                        wasPasted = this->m_pCursor->SetAsPrevious(this-
>TakeFromBuffer());

                        // In this case, the cursor was pointing to the head
and
                        // the new node was placed previous to the head
pointer.
                        // So adjust the head pointer as needed.
                        if(this->m_pHead->m_pPrevious != NULL)
                              this->m_pHead = this->m_pHead-
>GetPointerTo(CNode::FIRST);
                  }
                  else
                  {
                        // The cursor is not valid (i.e., it points to NULL)
                        // and the list is not empty. Interpret this as a
"Queue"
```

```
                        // request.
                        wasPasted = this->Queue();
                  }
            }
            else
            {
                  // The list is empty, so it needs to be initiated.
                  wasPasted = this->InitiateList();
            }
      }

      return wasPasted;

      //~maa~20011114
}

// This class method places a given node into the buffer.
// Preconditions: The node to be placed in the buffer must not already
// be there...
// Postconditions:
// (1) If the buffer contains something,then its contents will
// be purged (i.e., destroyed).
// (2) If the provided node is currently a member of the linked list,
// then this list's pointers (i.e., head, tail, and cursor) will be
// adjusted accordingly.
// (3) The given node will be unlinked (i.e. removed) from the list.
// (4) Finally, the buffer will point to the given node.
bool RList::PlaceIntoBuffer(CNode *pNode)
{
      //Declare and initialize local variables.
      bool nodePlacedIntoBuffer = false;

      if (pNode != this->m_pBuffer)
      {
            // (Step 1) Empty the buffer.
            this->PurgeBuffer();

            if (pNode)
            {
                  // (Step 2) A node object has been specified, so the list's
pointers
                  // may need updating.

                  // So if pNode is the first node, the head pointer will be
moved.
                  if (this->m_pHead == pNode)
                        // This should set the head to NULL
                        this->m_pHead = pNode->m_pNext;

                  // If the given node is the last node, then the tail
                  // pointer will be moved.
                  if (this->m_pTail == pNode)
                        this->m_pTail = pNode->m_pPrevious;

                  // If the given node is the cursored node, then the
                  // cursor will be moved.
                  if (this->m_pCursor == pNode)
                  {
                        // Take the next link if possible;
                        if (pNode->m_pNext)
                              this->m_pCursor = pNode->m_pNext;
                        else
                              // otherwise, take the previous (if possible).
                              this->m_pCursor = pNode->m_pPrevious;
                  }

                  // (Step 3) Now if the link was part of the linked list, it
will be
                  // detached from the list.
                  pNode->Unlink();

                  //~bvs~20040806:  While the node is not formally in the
list, it is
                  // still connected to it.
                  pNode->m_pList=this;

                  nodePlacedIntoBuffer = true;
            }

            // The buffer will point to the link that pNode pointed to (step
4).
            // (Note that pNode might point to NULL)
            this->m_pBuffer = pNode;
      }
      else if (pNode != NULL)
      {
            //~bvs~20020312
            // It's already in the buffer.
            nodePlacedIntoBuffer = true;
      }

      return nodePlacedIntoBuffer;

      //~maa~20011114
      //~bvs~20020312
}

// This class method will return a node pointer according
// the type of pointer specified in the "eTypeOfPointer"
// parameter.
// Postconditions: A pointer of the type specified by the caller
// is returned.
CNode* RList::GetPointerTo(EnumListPointerType eTypeOfPointer) const
{
      // Declare and initialize local pointers.
      CNode* pNode=NULL;

      // {programming hint:  use a switch statement with eTypeOfPointer
      // and then use the class member pointers to return a pointer to
      // the proper CLink object.}
      switch(eTypeOfPointer)
      {
      case HEAD:
            pNode= this->m_pHead;
            break;
        case TAIL:
            pNode = this->m_pTail;
            break;
        case CURSOR:
            pNode= this->m_pCursor;
            break;
        case BUFFER:
            pNode = this->m_pBuffer;
            break;
        default:
            break;
        }
        return pNode;

        //~maa~20011114
}

// This class method is responsible for "queueing" a link from
// the buffer onto the tail of the linked list.
// Postconditions: If the list had a tail, and there was something in
// the buffer, the buffer node is added to the end of the list.
// If the list was empty, the function attempts to initiate a new list
// with whatever was in the buffer. If any of these operations is
// successful the boolean inserted is set to true before returning to the
// caller.
bool RList::Queue()
{
        // Delcare and initialize local variables.
        bool inserted = false;

        //~bvs~20021208:  Indicate that this list is being used
        // now in a FIFO manner.
        this->SetType(FIFO);

        // If the list has a tail...
        if(this->m_pTail)
        {
            // Add the new node to the end of the list in effect
            // "queueing" it. Adjust the tail pointer to point to the
            // new node. If that operation is successful, set the boolean
            // inserted to true and return it to the caller.
            inserted = this->m_pTail->SetAsNext(this->TakeFromBuffer());
            if (inserted)
                this->m_pTail=this->m_pTail->m_pNext;
            else
            {
                //Beep(200,200);
            }
        }
        else
        {
            // If the list has does not have a tail and is in fact empty,
            // try to initiate the list with whatever was in the buffer.
            // If that operation is successful, set the boolean
            // inserted to true and return it to the caller.
            inserted = this->InitiateList();
```

```
      }
      return inserted;

      //~maa~20011114
}

// This function counts how many nodes are in the list.
// Precondtions: There must be something in the list.
// Postconditions: The list is traversed and each node is
// counted. The resulting number is returned to the caller.
int RList::GetCount() const
{
      // Delcare and initialize local variables
      // and pointers.
      int count = 0;
      CNode* pNode;

      // Traverse the list and count all its nodes. Return the
      // number of nodes to the caller.
      for(pNode = this->m_pHead;(pNode); pNode = pNode->m_pNext)
            count++;

      return count;

      //~maa~20011114
}

int RList::PrintList()
{
      CNode* nodePtr = NULL;
      int count = 0;
//    CString str;

      for(nodePtr=this->GetHead();nodePtr;count++,nodePtr=nodePtr->GetNext())
      {
            printf("\n %s \n",nodePtr->GetName());
            count++;
      }

      return count;
}

// This function returns a pointer to the first node in the list.
// Postconditons: A pointer to the whatever the head pointer points
// to is returned.
CNode * RList::GetHead() const
{
      //Delcare and initialize local pointers.
      CNode *pNode = NULL;

      // Assign the local pointer to the first node in the list
      // and return it to the caller.
      pNode = this->m_pHead;

      return pNode;

      //~maa~20011114
```

```
}

// This function allows the caller to retrieve a node in a list by
// supplying its name.
// Preconditions:
// (1) The character pointer passed to the function must not be NULL.
// (2) There must be something in the list.
// Postconditons: The function either returns a pointer to a matching
// node or else it returns NULL.
CNode* RList::GetNodeByName(const char *name)
{
      // Declare and initialize local pointers.
      CNode* pNode = NULL;
      int count=this->GetCount();
      // If the name passed to the function is not NULL...
      if (name)
      {
            // Traverse the list to find a match.
            for(pNode = this->m_pHead; (pNode); pNode = pNode->m_pNext)
            {
                  // If a match is found, return a pointer to it to the
                  // caller. Otherwise return NULL.
                  if (pNode->GetName())
                  {
                        //~bvs~20021221:  Now make a case-insensitive
comparison.
                        if (stricmp(name, pNode->GetName())==0)
                        {
                              break;
                        }
                  }
            }
      }
      return pNode;

      //~maa~20011114
}

// This function allows the caller to find a node by supplying its
// id.
// Preconditions: There must be something in the list.
// Postconditions: A pointer to the node whose id matches that supplied
// by the caller is returned.
CNode* RList::GetNodeById(int id)
{
      // Declare and initialize local pointers.
      CNode* pNode = NULL;
      int thisId=0;

      if (this->IsSomethingInTheList()) {
            // Traverse the list comparing the id's of all the nodes to
            // the id passed in by the caller.
            for(pNode = this->m_pHead; (pNode); pNode = pNode->m_pNext)
            {
                  // ~dv~20020909
                  // For debugging purpose
                  thisId=pNode->GetId();
                  // If a match is found, return a pointer to it.
                  // Otherwise return NULL.
                  if (thisId == id)
                  {
                        break;
                  }
            }
      }
      return pNode;

      //~maa~20011114
}

// This function returns a pointer to whatever the cursor points to.
CNode * RList::GetCursor()
{
      return this->m_pCursor;

      //~maa~20011114
}

RList& RList::operator << (RList &sourceList)
{
      while (sourceList.Pop())
      {
            this->Queue(sourceList.TakeFromBuffer());
      }
      return *this;
}

// This function allows the caller to point the cursor to a node
// by supplying its id.
// Precondtions: There must be a node in the list whose id matches
// that supplied by the caller.
// Postcondtions: The cursor is set to point to the node whose id
// matches that supplied by the caller.
bool RList::SetCursor(int iNodeID)
{
      //Declare and initialize local variables.
      bool done=false;

      // Search for the node by its id. If its found make the
      // list's cursor point to it. If the function is successful
      // set the boolean done to true and return it to the caller.
      CNode *pNode = this->GetNodeById(iNodeID);
      if (pNode)
      {
            this->m_pCursor = pNode;
            done = true; //~bvs~20010826:  forgot to do this before.
      }
      return done;

      //~maa~20011114
}

// This function allows the caller to point the cursor to whatever
// the head is pointing to.
bool RList::PointCursorAtHead()
{
      this->m_pCursor = this->m_pHead;
      return this->m_pCursor ? true : false;
      //~bvs~20010831
      //~maa~20011114
}

// This function allows the caller to point the cursor to its
// next.
// Precondtions: The cursor's next must not be NULL.
// Postconditions: The cursor is pointed to its next and
// the boolean dDone is set to true before being returned to the
// caller.
bool RList::PointCursorAtNext()
{
      // Declare and initialize local varibles
      bool bDone = false;

      // If the cursor points to something...
      if (this->m_pCursor)
      {
            // If the cursor's next is not NULL...
            if (this->m_pCursor->GetNext())
            {
                  // Point the cursor to its next and set the boolean
                  //bDone to true before its returned to the caller.
                  this->m_pCursor = this->m_pCursor->GetNext();
                  bDone = true;
            }
      }
      return bDone;

      //~bvs~20010831
      //~maa~20011114
}

// This function allows the caller to point the cursor at the
// the list's tail.
// Preconditions: The list must not be empty.
// Postconditions: The cursor is pointed at what the tail
// points to and true is returned to the caller.
bool RList::PointCursorAtTail()
{
      this->m_pCursor = this->m_pTail;
      return this->m_pCursor ? true : false;

      //~bvs~20010831
      //~maa~20011114
}

bool RList::Delete()
{
      bool bDeleted=false;
      if (this->Cut())
      {
            this->PurgeBuffer();
            bDeleted=true;
      }
      return bDeleted;
      //~bvs~20010831
}

void RList::Initialize()
{
      this->m_pBuffer = NULL;
      this->m_pCursor = NULL;
      this->m_pHead = NULL;
      this->m_pTail = NULL;
      this->m_flags = 0;
      this->m_eTypeOfQueue = FIFO;
}

RList& RList::operator >>(CNode **pNode)
{
      if (pNode)
      {
            this->Pop();
            *pNode = this->TakeFromBuffer();
      }
      return *this;
      //~bvs~20020312
}

RList& RList::operator <<(CNode *pNode)
{
      if (pNode)
      {
            this->PlaceIntoBuffer(pNode);
            this->Push();
      }
      return *this;
      //~bvs~20020312
}

CNode* RList::SetCursor(CNode *pNode)
{
      CNode *pOldCursor=this->m_pCursor;
      if (pNode)
      {
            this->m_pCursor = pNode;
      }
      return pOldCursor;
}

// Preconditions: The SortIn function must be called.
// Postconditions: The list into which the node is to be added
// is traversed to locate the appropriate position for the new
// node. Once found the node is then added. If the node cannot
// be sorted, it is placed at the end of the list. A boolean
// true is returned.
bool RList::SortInFromBuffer()
{
      // Declare and initialize local variables.
      bool bSortedIn=false;

      //~bvs~20021208:  Indicate that this list is being used
      // now in a PRIORITY manner.
      this->SetType(PRIORITY);

      // If there is something in the buffer...
      if (this->m_pBuffer)
      {
            CNode *pAnchor=NULL, *pOldCursor=NULL;

            // Get the head of the list, then traverse it.
            for (pAnchor=this->GetHead(); pAnchor; pAnchor=pAnchor-
>GetNext())
            {
                  // If what's in the buffer should come after the next node
                  // paste what's in the buffer into the list
                  if (*this->m_pBuffer < *pAnchor)
                  {
                        // Keep a pointer to the cursor's position.
                        pOldCursor = this->SetCursor(pAnchor);

                        // If the cursor is valid the paste function takes
                        // what's in the buffer and places it before the node
                        // pointed to by the cursor.
                        if (this->Paste())
                              bSortedIn = true;

                        this->SetCursor(pOldCursor);
                        break;
                  }
            }
            // If the node cannot be sorted it is placed at the
            // end of the list.
            if (!bSortedIn)
            {
                  if (RList::Queue())
                        bSortedIn = true;
            }
      }
      return bSortedIn;

      //~maa~20020523
}

// Pre-conditions: The caller must want to add a node
// to a sorted list.
// Postconditions: The node is added to the list.
// (i) If the list is not empty the node is added and sorted.
// (ii) If the cursor is not valid, the node is added to
// the end of the list.
// (iii) If the list is empty it is initialized with the
// new node.
bool RList::SortIn(CNode *pNode)
{
      // Declare and initialize local variables.
      bool bSortedIn=false;

      // Place the new node into the buffer and add it
      // to a sorted list by calling the SortInFromBuffer
      // routine which calls the RList Paste() function.
      if (pNode && this->PlaceIntoBuffer(pNode))
      {
            // If the node is either successfully sorted,
            // queued or used to initialize the list,
            // return true.
            if (this->SortInFromBuffer())
                  bSortedIn = true;
      }
      return bSortedIn;

      //~maa~20020523
}

bool RList::ContainsNode(CNode *pNode)
{
      bool bDoesContainNode=false;
      if (pNode)
      {
            for (CNode *p=this->m_pHead; p; p=p->GetNext())
            {
                  if (pNode == p)
                  {
                        bDoesContainNode = true;
                        break;
                  }
            }
      }
      return bDoesContainNode;
      //~bvs~20030406:  Needs Documentation
}

//DEL bool RList::QueueFIFO(CNode *pNode)
//DEL {
//DEL
//DEL }

// Preconditions: There must be something in the list.
// Postcondtions: A node is removed from the list and the
// pointers are adjusted accordingly.
CNode* RList::DeQueueFIFO()
{
      // Declare and initialize local variables.
      CNode* pNode = NULL;

      // There must be something in the list for there
      // to be a "dequeing"...
      if(this->m_pHead)
      {
            if(this->HasOnlyOne())
            {
                  pNode = this->m_pHead;
                  this->m_pHead= NULL;
                  this->m_pTail= NULL;
                  this->m_pCursor = NULL;
            }

            else if(this->m_pHead == this->m_pCursor)
            {
                  pNode = this->m_pHead;
                  this->m_pHead=this->m_pHead->m_pNext;
                  this->m_pCursor = this->m_pCursor->m_pNext;
            }
            else
            {
                  pNode = this->m_pHead;
                  this->m_pHead=this->m_pHead->m_pNext;
            }

      }
      return pNode;
}


CNode* RList::GetTail() const
{
      CNode* pNode = NULL;
      pNode = this->m_pTail;

      return pNode;
}

/*bool RList::EnterAndLockTheDoor()
{
      // Critical sections are useful when only one
      // thread at a time can be allowed to modify data
      // or some other controlled resource. For example,
      // adding nodes to a linked list is a process that
      // should only be allowed by one thread at a time.
      // By using a CCriticalSection object to control
      // the linked list, only one thread at a time can
      // gain access to the list.
      DWORD dwTimeOut=INFINITE; // This is ignored anyway
      bool bEntered=false;
      if (this->m_criticalSection.Lock(dwTimeOut))
      {
            bEntered=true;
      }
      return bEntered;

      //~bvs~20020518
      //~bvs~20020722
}*/

/*bool RList::LeaveAndUnlockTheDoor()
{
      // The critical section is like a dead bolt.
      // It is secure and works for race conditions.
      bool bLeft=false;
      if (this->m_criticalSection.Unlock())
      {
            bLeft=true;
      }
      return bLeft;

      //~bvs~20020518
      //~bvs~20020722
}*/

bool RList::Push(CNode *pNode)
{
      bool bPushed=false;
      if (pNode)
      {
            this->PlaceIntoBuffer(pNode);
            bPushed=this->PushFromBufferIntoList();
      }
      return bPushed;
}

// This class method will take the node object from the
// buffer and "push it" (i.e., insert it) onto the front
// of the linked list.
// Preconditions:
// Postconditions: If there is something in the list, the new node is
// added to the front of the list. If there is nothing in the list, the
// function tries to initiate the list with whatever is in the buffer.
// If either aspect of the function is successful, a boolean true is
// returned to the caller.
bool RList::PushFromBufferIntoList()
{
      // Delcare and initialize local variables.
      bool pushed=false;

      //~bvs~20021208:  Indicate that this list is being used
      // now in a LIFO manner.
      this->SetType(LIFO);

      // {programming hint:  if there is nothing in the buffer, the
      // routine returns false.  Otherwise, the object is taken from
      // the buffer (i.e., the m_pBuffer pointer will point to NULL
      // when we are done) and inserted at the head of the list.}
      if(this->IsSomethingInTheList())
      {
            // There is something in the list.  So, insert the buffered node
            //before the head of the list and then adjust the head pointer.
            pushed = this->m_pHead->SetAsPrevious(this->TakeFromBuffer());
            if (pushed)
                  this->m_pHead = this->m_pHead->m_pPrevious;
      }
      else
      {
            // The list is empty, so try to initiate the list with
            // whatever is in the buffer.
            pushed = this->InitiateList();
      }
      return pushed;

      //~maa~20011114
}

/*bool RList::Lock()
{
      // Is the "door" to the critical region already locked?
      if (!this->IsLocked())
      {
            // The door is not locked yet, so we will lock it now.
            m_flags |= IS_LOCKED;
      }
      // Now, attempt to enter the critical region.  If it was locked
      // already, we will have to wait our turn.
      return EnterAndLockTheDoor();

      //~bvs~20020722
      //~bvs~20020104
}

bool RList::UnLock()
{
      bool bUnlocked=LeaveAndUnlockTheDoor();
      if (bUnlocked)
            m_flags &= ~IS_LOCKED;
      else
            Beep(588,100);
      return bUnlocked;

      //~bvs~20020722
}*/

bool RList::Delete(CNode *pNode)
{
      bool bDeleted=false;
      if (pNode)
      {
            if (this->PlaceIntoBuffer(pNode))
            {
                  this->PurgeBuffer();
                  bDeleted=true;
                  //CLogNode logNode
("RList::Delete(%s)",bDeleted?"Deleted":"Not Deleted");
            }
      }

      return bDeleted;
      // ~dv~20020725
}

CNode* RList::GetLast()
{
      return this->m_pTail;
      //~dv~20020905
}
```

```
CNode* RList::GetBuffer()
{
      return this->m_pBuffer;
      //~dv~20020906
}

CNode* RList::GetNodeByIndex(int index)
{
      CNode *pNode=NULL;
      if (index>=0 && index<this->GetCount())
      {
            int i=0;
            for (pNode=this->GetHead(); pNode; pNode=pNode->GetNext(), ++i)
            {
                  if (i==index)
                        break;
            }
      }
      return pNode;

      //~bvs~20021027
}

bool RList::Chop()
{
      bool bChopped=false;
      if (!this->IsEmpty())
      {
            // Save the current cursor position, then move the cursor to the
            // last node in the list.
            CNode *pNode=this->SetCursor(this->m_pTail);
            bool bCursorWasPointingToTail=(pNode==this->m_pTail)?true:false;

            // Take the last node out of the list and put it into the buffer.
            if (this->Cut())
            {
                  // Record that the last node was "chopped."
                  bChopped=true;
            }
            if (!bCursorWasPointingToTail)
            {
                  // Since the cursor was not previously pointing at the
tail,
                  // restore the cursor to where it was before.
                  this->SetCursor(pNode);
            }
            else
            {
                  // Note that if the cursor was pointing at the tail, the
                  // Chop method would move the cursor to the new tail.
            }
      }
      return bChopped;
      //~bvs~20021130
}

bool RList::UnLink(CNode *pNode)
```

```
{
      bool bUnLinked=false;
      if (pNode)
      {
            // If the given node is in the list, cut it out
            // and put it in the buffer.
            if (this->ContainsNode(pNode))
            {
                  CNode *pOldCursor=this->SetCursor(pNode);
                  this->Cut();
                  this->SetCursor(pOldCursor);
            }

            // If the given node is in the buffer, take it
            // out of the buffer and let the caller deal with it.
            if (this->GetBuffer()==pNode)
            {
                  this->TakeFromBuffer();
                  bUnLinked=true;
            }
            pNode->m_pList=NULL;
      }
      return bUnLinked;
      //~bvs~20040808
}
```