Homework 1 Solution

Problems 1, 2, and 3:

In this solution, the functions with small, fast implementations are inlined. Alternatively, the `inline` keyword can be removed and the function definitions moved to Sequence.cpp. (`inline` will be mentioned at some point in class, so don't worry if you've never seen it before.)

Notice which member functions are const, and observe the use of the typedef name `ItemType`.

```
// Sequence.h

#ifndef SEQUENCE_INCLUDED
#define SEQUENCE_INCLUDED

  // Later in the course, we'll see that templates provide a much nicer
  // way of enabling us to have Sequences of different types.  For now,
  // we'll use a typedef.

typedef unsigned long ItemType;

const int DEFAULT_MAX_ITEMS = 200;

class Sequence
{
  public:
    Sequence();           // Create an empty sequence (i.e., one whose size()
is 0).
    bool empty() const;  // Return true if the sequence is empty, otherwise
false.
    int size() const;    // Return the number of items in the sequence.

    bool insert(int pos, const ItemType& value);
      // Insert value into the sequence so that it becomes the item at
      // position pos.  The original item at position pos and those that
      // follow it end up at positions one higher than they were at before.
      // Return true if 0 <= pos <= size() and the value could be
      // inserted.  (It might not be, if the sequence has a fixed capacity,
      // (e.g., because it's implemented using a fixed-size array) and is
      // full.)  Otherwise, leave the sequence unchanged and return false.
      // Notice that if pos is equal to size(), the value is inserted at the
      // end.

    int insert(const ItemType& value);
      // Let p be the smallest integer such that value <= the item at
      // position p in the sequence; if no such item exists (i.e.,
```

```
      // value > all items in the sequence), let p be size().  Insert
      // value into the sequence so that it becomes the item at position
      // p.  The original item at position p and those that follow it end
      // up at positions one higher than before.  Return p if the value
      // was actually inserted.  Return -1 if the value was not inserted
      // (perhaps because the sequence has a fixed capacity and is full).

    bool erase(int pos);
      // If 0 <= pos < size(), remove the item at position pos from
      // the sequence (so that all items that followed this item end up at
      // positions one lower than they were at before), and return true.
      // Otherwise, leave the sequence unchanged and return false.

    int remove(const ItemType& value);
      // Erase all items from the sequence that == value.  Return the
      // number of items removed (which will be 0 if no item == value).

    bool get(int pos, ItemType& value) const;
      // If 0 <= pos < size(), copy into value the item at position pos
      // in the sequence and return true.  Otherwise, leave value unchanged
      // and return false.

    bool set(int pos, const ItemType& value);
      // If 0 <= pos < size(), replace the item at position pos in the
      // sequence with value and return true.  Otherwise, leave the sequence
      // unchanged and return false.

    int find(const ItemType& value) const;
      // Let p be the smallest integer such that value == the item at
      // position p in the sequence; if no such item exists, let p be -1.
      // Return p.

    void swap(Sequence& other);
      // Exchange the contents of this sequence with the other one.

  private:
    ItemType m_data[DEFAULT_MAX_ITEMS];  // the items in the sequence
    int      m_size;                     // number of items in the sequence

      // At any time, the elements of m_data indexed from 0 to m_size-1
      // are in use.

    void uncheckedInsert(int pos, const ItemType& value);
      // Insert value at position pos, shifting items to the right to make
      // room for it.  Assume pos is valid and there's room.
};

// Inline implementations

inline
int Sequence::size() const
{
    return m_size;
}

inline
bool Sequence::empty() const
```

```
{
    return size() == 0;
}

#endif // SEQUENCE_INCLUDED
=====================================================================
// Sequence.cpp

#include "Sequence.h"

Sequence::Sequence()
 : m_size(0)
{}

bool Sequence::insert(int pos, const ItemType& value)
{
    if (pos < 0  ||  pos > size()  ||  size() == DEFAULT_MAX_ITEMS)
        return false;
    uncheckedInsert(pos, value);
    return true;
}

int Sequence::insert(const ItemType& value)
{
    if (size() == DEFAULT_MAX_ITEMS)
        return -1;
    int pos;
    for (pos = 0; pos < size()  &&  value > m_data[pos]; pos++)
        ;
    uncheckedInsert(pos, value);
    return pos;
}

bool Sequence::erase(int pos)
{
    if (pos < 0  ||  pos >= size())
        return false;
    for ( ; pos < size() - 1; pos++)
        m_data[pos] = m_data[pos+1];
    m_size--;
    return true;
}

int Sequence::remove(const ItemType& value)
{
    int keepPos = find(value);
    if (keepPos == -1)
        return 0;
    int count = 1;
    for (int examinePos = keepPos+1; examinePos < size(); examinePos++)
    {
        if (m_data[examinePos] == value)
            count++;
        else
        {
            m_data[keepPos] = m_data[examinePos];
            keepPos++;
```

```
        }
    }
    m_size -= count;
    return count;
}

bool Sequence::get(int pos, ItemType& value) const
{
    if (pos < 0  ||  pos >= size())
        return false;
    value = m_data[pos];
    return true;
}

bool Sequence::set(int pos, const ItemType& value)
{
    if (pos < 0  ||  pos >= size())
        return false;
    m_data[pos] = value;
    return true;
}

int Sequence::find(const ItemType& value) const
{
    for (int pos = 0; pos < size(); pos++)
        if (m_data[pos] == value)
            return pos;
    return -1;
}

void Sequence::swap(Sequence& other)
{
      // Swap elements.  Since the only elements that matter are those up to
      // m_size and other.m_size, only they have to be moved.

    int maxSize = (m_size > other.m_size ? m_size : other.m_size);
    for (int k = 0; k < maxSize; k++)
    {
        ItemType tempItem = m_data[k];
        m_data[k] = other.m_data[k];
        other.m_data[k] = tempItem;
    }

      // Swap sizes

    int tempSize = m_size;
    m_size = other.m_size;
    other.m_size = tempSize;
}

void Sequence::uncheckedInsert(int pos, const ItemType& value)
{
    for (int k = size(); k > pos; k--)
        m_data[k] = m_data[k-1];
    m_data[pos] = value;
    m_size++;
}
```

Problem 4:

Here's one implementation of ScoreList that uses an unsorted Sequence.

```
// ScoreList.h

#ifndef SCORELIST_INCLUDED
#define SCORELIST_INCLUDED

#include "Sequence.h"  // ItemType is typedef'd to unsigned long
#include <limits>

const unsigned long NO_SCORE = std::numeric_limits<unsigned long>::max();

class ScoreList
{
  public:
    ScoreList();       // Create an empty ScoreList

    bool add(unsigned long score);
      // If the score is valid (a value from 0 to 100), add it to the
      // score list and return true.  Otherwise, leave the score list
      // unchanged and return false.

    bool remove(unsigned long score);
      // Remove one instance of the specified score from the score list.
      // Return true iff a score was removed.

    int size() const;  // Return the number of scores in the list.

    unsigned long minimum() const;
      // Return the lowest score in the score list.  If the list is
      // empty, return NO_SCORE.

    unsigned long maximum() const;
      // Return the highest score in the score list.  If the list is
      // empty, return NO_SCORE.

  private:
    Sequence m_scoreSeq;
       // The scores in m_scoreSeq are in no particular order.
};

// Inline implementations

inline
int ScoreList::size() const
{
    return m_scoreSeq.size();
}

#endif // SCORELIST_INCLUDED
================================================================
// ScoreList.cpp

#include "Sequence.h"
```

```
#include "ScoreList.h"

  // Actually, we did not have to declare and implement the default
  // constructor:  If we declare no constructors whatsoever, the compiler
  // writes a default constructor for us that would do nothing more than
  // default construct the m_scoreSeq data member.

ScoreList::ScoreList()
{}

bool ScoreList::add(unsigned long score)
{
    if (score > 100)
        return false;
    return m_scoreSeq.insert(size(), score);
}

bool ScoreList::remove(unsigned long score)
{
    int pos = m_scoreSeq.find(score);
    if (pos == -1)  // not found
        return false;
    return m_scoreSeq.erase(pos);
}

unsigned long ScoreList::minimum() const
{
    if (m_scoreSeq.empty())
        return NO_SCORE;
    unsigned long result;
    m_scoreSeq.get(0, result);
    for (int pos = 1; pos < size(); pos++)
    {
        unsigned long v;
        m_scoreSeq.get(pos, v);
        if (v < result)
            result = v;
    }
    return result;
}

unsigned long ScoreList::maximum() const
{
    if (m_scoreSeq.empty())
        return NO_SCORE;
    unsigned long result;
    m_scoreSeq.get(0, result);
    for (int pos = 1; pos < size(); pos++)
    {
        unsigned long v;
        m_scoreSeq.get(pos, v);
        if (v > result)
            result = v;
    }
    return result;
}
```

Here's another implementation of ScoreList that uses a sorted Sequence.

```
// ScoreList.h

#ifndef SCORELIST_INCLUDED
#define SCORELIST_INCLUDED

#include "Sequence.h"  // ItemType is typedef'd to unsigned long
#include <limits>

const unsigned long NO_SCORE = std::numeric_limits<unsigned long>::max();

class ScoreList
{
  public:
    ScoreList();       // Create an empty ScoreList

    bool add(unsigned long score);
      // If the score is valid (a value from 0 to 100), add it to the
      // score list and return true.  Otherwise, leave the score list
      // unchanged and return false.

    bool remove(unsigned long score);
      // Remove one instance of the specified score from the score list.
      // Return true iff a score was removed.

    int size() const;  // Return the number of scores in the list.

    unsigned long minimum() const;
      // Return the lowest score in the score list.  If the list is
      // empty, return NO_SCORE.

    unsigned long maximum() const;
      // Return the highest score in the score list.  If the list is
      // empty, return NO_SCORE.

  private:
    Sequence m_scoreSeq;
       // It is always the case that the scores in m_scoreSeq are sorted.
};

// Inline implementations

inline
int ScoreList::size() const
{
    return m_scoreSeq.size();
}

#endif // SCORELIST_INCLUDED
```

```
// ScoreList.cpp

#include "Sequence.h"
#include "ScoreList.h"
```

```
  // Actually, we did not have to declare and implement the default
  // constructor:  If we declare no constructors whatsoever, the compiler
  // writes a default constructor for us that would do nothing more than
  // default construct the m_scoreSeq data member.

ScoreList::ScoreList()
{}

bool ScoreList::add(unsigned long score)
{
    if (score > 100)
        return false;
    return m_scoreSeq.insert(score) != -1;
      // Since all insertions into m_scoreSeq use this form of insert,
      // m_scoreSeq is guaranteed to be sorted.
}

bool ScoreList::remove(unsigned long score)
{
    int pos = m_scoreSeq.find(score);
    if (pos == -1)  // not found
        return false;
    return m_scoreSeq.erase(pos);
}

unsigned long ScoreList::minimum() const
{
    if (m_scoreSeq.empty())
        return NO_SCORE;
    unsigned long result;
    m_scoreSeq.get(0, result);
    return result;
}

unsigned long ScoreList::maximum() const
{
    if (m_scoreSeq.empty())
        return NO_SCORE;
    unsigned long result;
    m_scoreSeq.get(size()-1, result);
    return result;
}
```

Problem 5:

The few differences from the Problem 3 solution are indicated in boldface.

```
// newSequence.h

#ifndef NEWSEQUENCE_INCLUDED
#define NEWSEQUENCE_INCLUDED

  // Later in the course, we'll see that templates provide a much nicer
  // way of enabling us to have Sequences of different types.  For now,
  // we'll use a typedef.
```

```
typedef unsigned long ItemType;

const int DEFAULT_MAX_ITEMS = 200;

class Sequence
{
  public:
    Sequence(int capacity = DEFAULT_MAX_ITEMS);
      // Create an empty sequence with the given capacity

    bool empty() const;  // Return true if the sequence is empty, otherwise
false.
    int size() const;    // Return the number of items in the sequence.

    bool insert(int pos, const ItemType& value);
      // Insert value into the sequence so that it becomes the item at
      // position pos.  The original item at position pos and those that
      // follow it end up at positions one higher than they were at before.
      // Return true if 0 <= pos <= size() and the value could be
      // inserted.  (It might not be, if the sequence has a fixed capacity,
      // (e.g., because it's implemented using a fixed-size array) and is
      // full.)  Otherwise, leave the sequence unchanged and return false.
      // Notice that if pos is equal to size(), the value is inserted at the
      // end.

    int insert(const ItemType& value);
      // Let p be the smallest integer such that value <= the item at
      // position p in the sequence; if no such item exists (i.e.,
      // value > all items in the sequence), let p be size().  Insert
      // value into the sequence so that it becomes the item at position
      // p.  The original item at position p and those that follow it end
      // up at positions one higher than before.  Return p if the value
      // was actually inserted.  Return -1 if the value was not inserted
      // (perhaps because the sequence has a fixed capacity and is full).

    bool erase(int pos);
      // If 0 <= pos < size(), remove the item at position pos from
      // the sequence (so that all items that followed this item end up at
      // a position one lower than before), and return true.  Otherwise,
      // leave the sequence unchanged and return false.

    int remove(const ItemType& value);
      // Erase all items from the sequence that == value.  Return the
      // number of items removed (which will be 0 if no item == value).

    bool get(int pos, ItemType& value) const;
      // If 0 <= pos < size(), copy into value the item at position pos
      // in the sequence and return true.  Otherwise, leave value unchanged
      // and return false.

    bool set(int pos, const ItemType& value);
      // If 0 <= pos < size(), replace the item at position pos in the
      // sequence with value and return true.  Otherwise, leave the sequence
      // unchanged and return false.

    int find(const ItemType& value) const;
      // Let p be the smallest integer such that value == the item at
```

```
      // position p in the sequence; if no such item exists, let p be -1.
      // Return p.

    void swap(Sequence& other);
      // Exchange the contents of this sequence with the other one.

      // Housekeeping functions
    ~Sequence();
    Sequence(const Sequence& other);
    Sequence& operator=(const Sequence& rhs);

  private:
    ItemType* m_data;        // dynamic array of the items in the sequence
    int       m_size;        // the number of items in the sequence
    int       m_capacity;    // the maximum number of items there could be

      // At any time, the elements of m_data indexed from 0 to m_size-1
      // are in use.

    void uncheckedInsert(int pos, const ItemType& value);
      // Insert value at position pos, shifting items to the right to make
      // room for it.  Assume pos is valid and there's room.
};

// Inline implementations

inline
int Sequence::size() const
{
    return m_size;
}

inline
bool Sequence::empty() const
{
    return size() == 0;
}

#endif // NEWSEQUENCE_INCLUDED
=====================================================================
// newSequence.cpp

#include "newSequence.h"
#include <iostream>
#include <cstdlib>

Sequence::Sequence(int capacity)
 : m_size(0), m_capacity(capacity)
{
    if (capacity < 0)
    {
        std::cout << "A Sequence capacity must not be negative." <<
std::endl;
        std::exit(1);
    }
    m_data = new ItemType[m_capacity];
}
```

```
bool Sequence::insert(int pos, const ItemType& value)
{
    if (pos < 0  ||  pos > size()  ||  size() == m_capacity)
        return false;
    uncheckedInsert(pos, value);
    return true;
}

int Sequence::insert(const ItemType& value)
{
    if (size() == m_capacity)
        return -1;
    int pos;
    for (pos = 0; pos < size()  &&  value > m_data[pos]; pos++)
        ;
    uncheckedInsert(pos, value);
    return pos;
}

bool Sequence::erase(int pos)
{
    if (pos < 0  ||  pos >= size())
        return false;
    for ( ; pos < size() - 1; pos++)
        m_data[pos] = m_data[pos+1];
    m_size--;
    return true;
}

int Sequence::remove(const ItemType& value)
{
    int keepPos = find(value);
    if (keepPos == -1)
        return 0;
    int count = 1;
    for (int examinePos = keepPos+1; examinePos < size(); examinePos++)
    {
        if (m_data[examinePos] == value)
            count++;
        else
        {
            m_data[keepPos] = m_data[examinePos];
            keepPos++;
        }
    }
    m_size -= count;
    return count;
}

bool Sequence::get(int pos, ItemType& value) const
{
    if (pos < 0  ||  pos >= size())
        return false;
    value = m_data[pos];
    return true;
}
```

```
bool Sequence::set(int pos, const ItemType& value)
{
    if (pos < 0  ||  pos >= size())
        return false;
    m_data[pos] = value;
    return true;
}

int Sequence::find(const ItemType& value) const
{
    for (int pos = 0; pos < size(); pos++)
        if (m_data[pos] == value)
            return pos;
    return -1;
}

void Sequence::swap(Sequence& other)
{
      // Swap pointers to the elements.

    ItemType* tempData = m_data;
    m_data = other.m_data;
    other.m_data = tempData;

      // Swap sizes

    int tempSize = m_size;
    m_size = other.m_size;
    other.m_size = tempSize;

      // Swap capacities

    int tempCapacity = m_capacity;
    m_capacity = other.m_capacity;
    other.m_capacity = tempCapacity;
}

Sequence::~Sequence()
{
    delete [] m_data;
}

Sequence::Sequence(const Sequence& other)
 : m_size(other.m_size), m_capacity(other.m_capacity)
{
    m_data = new ItemType[m_capacity];

      // Since the only elements that matter are those up to m_size, only
      // they have to be copied.

    for (int k = 0; k < m_size; k++)
        m_data[k] = other.m_data[k];
}

Sequence& Sequence::operator=(const Sequence& rhs)
{
```

```
    if (this != &rhs)
    {
        Sequence temp(rhs);
        swap(temp);
    }
    return *this;
}

void Sequence::uncheckedInsert(int pos, const ItemType& value)
{
    for (int k = size(); k > pos; k--)
        m_data[k] = m_data[k-1];
    m_data[pos] = value;
    m_size++;
}
```