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 Set.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.

```
// Set.h

#ifndef SET_INCLUDED
#define SET_INCLUDED

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

typedef unsigned long ItemType;

const int DEFAULT_MAX_ITEMS = 100;

class Set
{
  public:
    Set();               // Create an empty set.
    bool empty() const;  // Return true if the set is empty, otherwise false.
    int size() const;    // Return the number of items in the set.

    bool insert(const ItemType& value);
      // Insert value into the set if it is not already present.  Return
      // true if the value was actually inserted.  Leave the set unchanged
      // and return false if the value was not inserted (perhaps because it
      // is already in the set or because the set has a fixed capacity and
      // is full).

    bool erase(const ItemType& value);
      // Remove the value from the set if present.  Return true if the
      // value was removed; otherwise, leave the set unchanged and
      // return false.

    bool contains(const ItemType& value) const;
      // Return true if the value is in the set, otherwise false.

    bool get(int i, ItemType& value) const;
      // If 0 <= i < size(), copy into value an item in the set and
      // return true.  Otherwise, leave value unchanged and return false.
```

```
    void swap(Set& other);
      // Exchange the contents of this set with the other one.

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

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

    int find(const ItemType& value) const;
      // Return the position of value in the m_data array, or m_size if
      // it is not in the array.
};

// Inline implementations

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

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

inline
bool Set::contains(const ItemType& value) const
{
    return find(value) != m_size;
}

#endif // SET_INCLUDED
=====================================================================
// Set.cpp

#include "Set.h"

Set::Set()
 : m_size(0)
{}

bool Set::insert(const ItemType& value)
{
```

```
    if (m_size == DEFAULT_MAX_ITEMS  ||  contains(value))
        return false;
    m_data[m_size] = value;
    m_size++;
    return true;
}

bool Set::erase(const ItemType& value)
{
    int pos = find(value);
    if (pos == m_size)
        return false;
    m_size--;
    m_data[pos] = m_data[m_size];
    return true;
}

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

void Set::swap(Set& 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 minSize = (m_size < other.m_size ? m_size : other.m_size);
    for (int k = 0; k < minSize; k++)
    {
        ItemType tempItem = m_data[k];
        m_data[k] = other.m_data[k];
        other.m_data[k] = tempItem;
    }

      // If the sizes are different, assign the remaining elements from the
      // longer one to the shorter.

    if (m_size > minSize)
        for (int k = minSize; k < m_size; k++)
            other.m_data[k] = m_data[k];
    else if (other.m_size > minSize)
        for (int k = minSize; k < other.m_size; k++)
            m_data[k] = other.m_data[k];
```

```
      // Swap sizes

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

int Set::find(const ItemType& value) const
{
    int pos = 0;
    for ( ; pos < m_size  &&  m_data[pos] != value; pos++)
        ;
    return pos;
}
```

Problem 4:

```
// SSNSet.h

#ifndef SSNSET_INCLUDED
#define SSNSET_INCLUDED

#include "Set.h"  // ItemType is typedef'd to unsigned long

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

    bool add(unsigned long ssn);
      // Add an SSN to the SSNSet.  Return true if and only if the SSN
      // was actually added.

    int size() const;  // Return the number of SSNs in the SSNSet.

    void print() const;
      // Write every SSN in the SSNSet to cout exactly once, one per
      // line.  Write no other text.

  private:
    Set m_SSNs;
};

// Inline implementations

  // 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_SSNs data member.
```

```
inline
SSNSet::SSNSet()
{}

inline
bool SSNSet::add(unsigned long ssn)
{
    return m_SSNs.insert(ssn);
}

inline
int SSNSet::size() const
{
    return m_SSNs.size();
}

#endif // SSNSET_INCLUDED
```

```
// SSNSet.cpp

#include "Set.h"
#include "SSNSet.h"
#include <iostream>
using namespace std;

void SSNSet::print() const
{
    for (int k = 0; k < m_SSNs.size(); k++)
    {
        unsigned long x;
        m_SSNs.get(k, x);
        cout << x << endl;
    }
}
```

Problem 5:

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

```
// newSet.h

#ifndef NEWSET_INCLUDED
#define NEWSET_INCLUDED

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

```
typedef unsigned long ItemType;

const int DEFAULT_MAX_ITEMS = 100;

class Set
{
  public:
    Set(int capacity = DEFAULT_MAX_ITEMS);
      // Create an empty set with the given capacity.

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

    bool insert(const ItemType& value);
      // Insert value into the set if it is not already present.  Return
      // true if the value was actually inserted.  Return false if the
      // value was not inserted (perhaps because it is already in the set
      // or because the set has a fixed capacity and is full).

    bool erase(const ItemType& value);
      // Remove the value from the set if present.  Return true if the
      // value was removed; otherwise, leave the set unchanged and
      // return false.

    bool contains(const ItemType& value) const;
      // Return true if the value is in the set, otherwise false.

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

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

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

  private:
    ItemType* m_data;        // dynamic array of the items in the set
    int       m_size;        // the number of items in the set
    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.

    int find(const ItemType& value) const;
```

```
      // Return the position of value in the m_data array, or m_size if
      // it is not in the array.
};

// Inline implementations

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

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

inline
bool Set::contains(const ItemType& value) const
{
    return find(value) != m_size;
}

#endif // NEWSET_INCLUDED
```

```
// newSet.cpp

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

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

bool Set::insert(const ItemType& value)
{
    if (m_size == m_capacity  ||  contains(value))
        return false;
    m_data[m_size] = value;
```

```
    m_size++;
    return true;
}

bool Set::erase(const ItemType& value)
{
    int pos = find(value);
    if (pos == m_size)
        return false;
    m_size--;
    m_data[pos] = m_data[m_size];
    return true;
}

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

void Set::swap(Set& 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;
}

Set::~Set()
{
    delete [] m_data;
}
```

```
Set::Set(const Set& 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];
}

Set& Set::operator=(const Set& rhs)
{
    if (this != &rhs)
    {
        Set temp(rhs);
        swap(temp);
    }
    return *this;
}

int Set::find(const ItemType& value) const
{
    int pos = 0;
    for ( ; pos < m_size  &&  m_data[pos] != value; pos++)
        ;
    return pos;
}
```