**Homework 1 Solution**

[Problems 1, 2, and 3](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/1/solution.html#P123)  
[Problem 4](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/1/solution.html#P4)  
[Problem 5](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/1/solution.html#P5)

**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 = 200;

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. 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:**

// StudentSet.h

#ifndef STUDENTSET\_INCLUDED

#define STUDENTSET\_INCLUDED

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

class StudentSet

{

public:

StudentSet(); // Create an empty StudentSet

bool add(unsigned long id);

// Add a student id to the set. Return true if and only if the

// id was actually added.

int size() const; // Return the number of ids in the set.

void print() const;

// Print every student id in the set exactly once, one per line.

private:

Set m\_idSet;

};

// Inline implementations

inline

bool StudentSet::add(unsigned long id)

{

return m\_idSet.insert(id);

}

inline

int StudentSet::size() const

{

return m\_idSet.size();

}

#endif // STUDENTSET\_INCLUDED

===================================================================

// StudentSet.cpp

#include "Set.h"

#include "StudentSet.h"

#include <iostream>

using namespace std;

// 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\_idSet data member.

StudentSet::StudentSet()

{}

void StudentSet::print() const

{

for (int k = 0; k < m\_idSet.size(); k++)

{

unsigned long x;

m\_idSet.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 = 200;

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;

}