```cpp
// FILE: IntSet.cpp - header file for IntSet class
//        Implementation file for the IntStore class
//         (See IntSet.h for documentation.)
// INVARIANT for the IntSet class:
// (1) Distinct int values of the IntSet are stored in a 1-D,
//      compile-time array whose size is IntSet::MAX_SIZE;
//      the member variable data references the array.
// (2) The distinct int value with earliest membership is stored
//      in data[0], the distinct int value with the 2nd-earliest
//      membership is stored in data[1], and so on.
//      Note: No "prior membership" informationm is tracked; i.e.,
//            if an int value that was previously a member (but its
//            earlier membership ended due to removal) becomes a
//            member again, the timing of its membership (relative
//            to other existing members) is the same as if that int
//            value was never a member before.
//      Note: Re-introduction of an int value that is already an
//            existing member (such as through the add operation)
//            has no effect on the "membership timing" of that int
//            value.
// (4) The # of distinct int values the IntSet currently contains
//      is stored in the member variable used.
// (5) Except when the IntSet is empty (used == 0), ALL elements
//      of data from data[0] until data[used - 1] contain relevant
//      distinct int values; i.e., all relevant distinct int values
//      appear together (no "holes" among them) starting from the
//      beginning of the data array.
// (6) We DON'T care what is stored in any of the array elements
//      from data[used] through data[IntSet::MAX_SIZE - 1].
//      Note: This applies also when the IntSet is empry (used == 0)
//            in which case we DON'T care what is stored in any of
//            the data array elements.
//      Note: A distinct int value in the IntSet can be any of the
//            values an int can represent (from the most negative
//            through 0 to the most positive), so there is no
//            particular int value that can be used to indicate an
//            irrelevant value. But there's no need for such an
//            "indicator value" since all relevant distinct int
//            values appear together starting from the beginning of
//            the data array and used (if properly initialized and
//            maintained) should tell which elements of the data
//            array are actually relevant.

#include "IntSet.h"
#include <iostream>
#include <cassert>
using namespace std;

//This initializes an empty set
IntSet::IntSet(): data(), used(0)
{
```

```cpp
}

//Returns how many elements are in the initilized IntSet
int IntSet::size() const
{
    return used;
}

//This function checks if the initilized IntSet is empty or not.
bool IntSet::isEmpty() const
{
    if (used == 0){
        return true;
    }
    else{
        return false;
    }
}

//This checks the initilized IntSet to see if there is an "int" in the set
//and will return a true or false
bool IntSet::contains(int anInt) const
{
    for(int i = 0; i < used; i++){
        if(data[i] == anInt){
            return true;
        }
    }
    return false;
}

//This checks if all the elements in IntSet are also in otherIntSet.
//This will return a true or false value.
bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
{
    if (this->isEmpty()){
        return true;
    }
    else{
        for(int i = 0; i < this->size(); i++){
            if(!otherIntSet.contains(this->data[i])){
                return false;
            }
        }
        return true;
    }
}

void IntSet::DumpData(ostream& out) const
{   // already implemented ... DON'T change anything
```

```cpp
    if (used > 0)
    {
        out << data[0];
        for (int i = 1; i < used; ++i)
            out << "  " << data[i];
    }
}

//This will unionize the IntSet with the otherIntSet
//and return an IntSet representing that union.
IntSet IntSet::unionWith(const IntSet& otherIntSet) const
{
    assert(size() + (otherIntSet.subtract(*this)).size() <= MAX_SIZE);
    IntSet unionIntSet = (*this);
    for(int i = 0; i < otherIntSet.size(); i++){
        if(!unionIntSet.contains(otherIntSet.data[i])){
            unionIntSet.add(otherIntSet.data[i]);
        }
    }
    return unionIntSet;
}

//This will return an Inset that represent the intersected part of the IntSet
// and otherIntSet
IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
    IntSet interSect = (*this);

    for(int i = 0; i < used; i++){
        if(otherIntSet.contains(data[i]) == 0){
            interSect.remove(data[i]);
            i--;
        }
    }
    return interSect;
}

//This will subtract the otherIntSet from the
//intSet and will return an intSet representing the results of the operation.
IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
    IntSet subSet = (*this);

    for(int i = 0; i < used; i++){
        if(otherIntSet.contains(data[i]) == 1){
            subSet.remove(data[i]);
            i--;
        }
    }
    return subSet;
}
```

```cpp
//This resets the intSet and makes it empty again
void IntSet::reset()
{
    used = 0;
}

//This checks whether an element is already in the IntSet or not,
//if not it will add that element to the intSet.
bool IntSet::add(int anInt)
{
    if(!contains(anInt)){
        if(used < MAX_SIZE){
            data[used] = anInt;
            used++;
            return true;
        }
    }
    return false;
}

//This checks whether an element is in the IntSet or not,
//if the element is in the IntSet it will remove that element.
bool IntSet::remove(int anInt)
{
    bool found = false;
    if(contains(anInt) == 0){
        return false;
    }
    for(int i = 0; i < used; i++){
        if(data[i] == anInt){
            found = true;
        }
        if(found == true && i != used - 1){
            data[i] = data[i + 1];
        }
    }
    used--;
    return true;
}

//This checks two IntSet is the same/equal.
bool equal(const IntSet& is1, const IntSet& is2)
{
    if(is1.IntSet::isSubsetOf(is2) && is2.IntSet::isSubsetOf(is1)) {
            return true;
        }
        return false;
}
```