```cpp


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


void IntSet::resize(int new_capacity)
{
   // Validating new capacity value
   if (used != 0 && new_capacity < used )
   {
      capacity = used;
   }
   else if (new_capacity <= 0)
   {
      capacity = DEFAULT_CAPACITY;
   }
   else
   {
      capacity = new_capacity;
   }

   // Creating temp dynamic array with new capacity value.
   int * temp_array = new int[capacity];

   // Transferring old elements to new dynamic array.
   for (int i = 0; i < used; i++)
   {
      temp_array[i] = data[i];
   }

   // Deallocating old array.
   delete [] data;

   // Assigning data to new array.
   data = temp_array;

   // Removing temp_array pointer because it is no longer needed
   temp_array = NULL;
   delete temp_array;
}


IntSet::IntSet(int initial_capacity) : capacity(initial_capacity), used(0)
{
   // Initializing capacity to DEFAULT_CAPACITY if initial_capacity < 1
   if (initial_capacity < 1)
   {
      capacity = DEFAULT_CAPACITY;
   }

   // Assingning 'data' to a new intance of a dynamic array of size 'capacity'
   data = new int[capacity];
}


IntSet::IntSet(const IntSet& src) : capacity(src.capacity), used(src.used)
{
   // Assingning 'data' to a new intance of a dynamic array of size 'capacity'
   data = new int[capacity];

   // Copying every element in src to data
   for (int i = 0; i < src.used; i++)
   {
      data[i] = src.data[i];
   }
}
```

```cpp


IntSet::~IntSet()
{
   // Deallocating dynamic variables
   delete [] data;
   data = NULL;
}


IntSet& IntSet::operator=(const IntSet& rhs)
{
   // Allocating space in temp array to hold elements in rhs
   int * temp_array = new int[rhs.capacity];
   for (int i = 0; i < rhs.used; i++)
   {
      temp_array[i] = rhs.data[i];
   }

   // Deleting current dynamic array pinted to by data
   // and assigning data to temp_array
   delete [] data;
   data = temp_array;

   // Copying over all properties from rhs to data
   capacity = rhs.capacity;
   used = rhs.used;

   // Deletingt temp_array pointer because is no longer needed
   temp_array = NULL;
   delete temp_array;

   return *this;
}


int IntSet::size() const
{
   // Returning # of distinct int values the invoking IntSet currently contains
   // which is stored in the member variable used.
   return used;
}


bool IntSet::isEmpty() const
{
   // Returning true if used is equal to 0.
   return (used == 0);
}


bool IntSet::contains(int anInt) const
{
   if (used > 0)
   {
      for (int i = 0; i < used; i++)
      {
         // if (data[i] == anInt) // Differnt version
         if (*(data + i) == anInt)
         {
            return true;
         }
      }
   }

   return false;
}


```

```cpp
bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
{
   // An empty set is a subset of any set.
   if (isEmpty())
   {
      return true;
   }

   // Check for all elements of invoking set, if any one element is not
   // contained return false.
   else
   {
      for (int i = 0; i < used; i++)
      {
         if (!(otherIntSet.contains(data[i])))
         {
            return false;
         }
      }
   }

   return true;
}

void IntSet::DumpData(ostream& out) const
{
   // already implemented ... DON'T change anything
   if (used > 0)
   {
      out << data[0];
      for (int i = 1; i < used; ++i)
         out << "  " << data[i];
   }
}

IntSet IntSet::unionWith(const IntSet& otherIntSet) const
{
   // Instanciating IntSet unionSet = *this to hold union elements of both
   // this and otherIntSet
   IntSet unionSet = *this;

   // Copying over unique elements from ohterIntSet since all
   // elements from *this are already contained
   for (int i = 0; i < otherIntSet.used; i++)
   {
      unionSet.add(otherIntSet.data[i]);
   }

   return unionSet;
}

IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
   // IntSet representing the intersection of the invoking IntSet
   // and otherIntSet that will be returned
   IntSet interSet = *this;

   // Removing all elements not contained in otherIntSet
   // from the interSet
   for (int i = 0; i < used; i++)
   {
      if (!otherIntSet.contains(interSet.data[i]))
      {
         interSet.remove(data[i]);
      }
   }

   return interSet;
}
```

```cpp


IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
   // IntSet representing the difference between the
   // invoking IntSet and otherIntSet
   IntSet subtractSet = *this;

   // Removing all elemts in otherIntSet that
   // are contained in subtractSet
   for (int i = 0; i < otherIntSet.used; i++)
   {
      // Calling remove() right away because the remove function
      // does the contain() check already
      subtractSet.remove(otherIntSet.data[i]);
   }

   return subtractSet;
}


void IntSet::reset()
{
   // Deleting all array data, creating new empty array
   // and reseting used to 0
   delete [] data;
   data = new int[DEFAULT_CAPACITY];
   used = 0;
   capacity = DEFAULT_CAPACITY;
}


bool IntSet::add(int anInt)
{
   // Validating new value, if not new then false is returned
   if (!contains(anInt))
   {
      // Validating that array is not full,
      // if so then calling the resize function
      if (used >= capacity)
      {
         // Resizing to atleast capacity + 1.
         resize(int(1.5 * capacity) + 1);
      }

      // Adding new element
      data[used] = anInt;
      used++;

      return true;
   }
   else
   {
      return false;
   }
}


bool IntSet::remove(int anInt)
{
   // Checking to see if anInt is contained in 'data'
   // if contained, we proceed with removal
   if (contains(anInt))
   {
      for (int i = 0; i < used; i++)
      {
         if (data[i] == anInt)
         {
            // Sifting all elements to the left after the
            // reqested element was removed
            for (int j = i; j < used - 1; j++)
            {
               data[j] = data[j+1];
            }
            --used;
         }
      }
      return true;
   }

   return false;
}


bool operator==(const IntSet& is1, const IntSet& is2)
{
   // If both sets are empty then they already equal to each other
   // This saves a little bit of time
   if (is1.isEmpty() && is2.isEmpty())
   {
      return true;
   }
   // if they are both subsets of eachother then they are equal
   else if (is1.isSubsetOf(is2) && is2.isSubsetOf(is1))
   {
      return true;
   }
   else
   {
      return false;
   }
}
```