Program Evaluation Form

Student Name: Ruben Suarez	**Course:** CS 3358 **Section #:** 004
Assignment #: 1 **Part # (if applicable):**	**Due Date:** 9-11-19
Extended/Relaxed Due Date (if applicable):	**Date Submitted:** 9-11-19

Success Summary. Indicate the status of your program – compiles successfully? runs without errors? gives expected (correct) results? specific difficulties not overcome? specific requirements not met? *etc.*

Assignment codes compiled and ran successfully. All expected output is provided.

(A rather broad/general guide - may supplement/replace with something more detailed tailored for program involved.)

Point Deduction Description	***Actual % Deducted***	***Possible % Deduction***
E-mail submission (not received, asked to resubmit, *etc.*) ~~emailed the [illegible] 9/13/19 (8:24)~~ [illegible]		-1 to -40
Compilation error		-70
Runtime error		-5 to -50
Logic error (incorrect output)		-5 to -50
Program testing (input/output submission, adequacy, *etc.*)		-1 to -40
Fulfillment of requirements/specifications		-1 to -100
Poor alignment and/or indentation and/or spacing		-1 to -10
Did not use meaningful identifiers		-1 to -5
Other readability woes (line wraps, *etc.*)		-1 to -5
Did not follow good practice (global variables, go to, *etc.*)		-1 to -10
Others: See 2nd sheet	6	(case dependent) **(below is your grade)**
100% -	6	= 94/100

Evaluator's Comments:
Compiled/tested ok

Assignment 1 Penalty Tally Sheet (Page 1 of 1)

■ **Not Heeding Style Guide (up to 30 points total penalty)**

- *Naming Convention:* (penalty - up to 3 per type of violation)
  - not using meaningful name
  - not starting a variable name in lowercase
  - not using separating underscore(s) or camel style in multi-word name
  - name of constant not in all-uppercase or name of variable in all-uppercase
  - function not doing, or doing more than, what it's name suggests
- Other:
- *Readability Woes:* (penalty - depends on severity)
  - poor indentation
  - poor spacing
  - poor alignment
  - line wrap
  - using `l` or `o` (looks like `1` or `0`) as variable name
  - not using monospaced font

■ **General Shortcomings**

- Code in hardcopy not gibing with code in softcopy. [depends on severity]
- Leaving behind irrelevant comments, debugging code, *etc.* [½ 1 1½ 2]
- Removing as-provided documentation (*esp.* class invariant) at top of *IntSet.cpp* [1]
- Shortcomings checking preconditions: expose implementation details, all-in-1 lumping, *etc.* [½ 1]
- Unreacheable code. [1 per function; 3 max]
- Other:

■ **Function Specific Penalties**

- `IntSet::IntSet()`
  - ▶ Not using *initializer list* [✓ [½]]
  - ▶ Not observing *class invariant* [1]
  - ▶ Not setting `used` [2½]
  - ▶ Other:
- `int IntSet::size() const`
- `bool IntSet::isEmpty() const`
  - ▶ Unnecessarily traversing/processing array (algorithmically correct or otherwise) [1 2]
  - ▶ Other:
- `int IntSet::contains(int anInt) const`
  - ▶ Traversing *entire* array and using an algorithm not in line with *class invariant* [3]
  - ▶ Out-of-bound (in general) traversing array(s) [2]
  - ▶ Logic error (role reversal, *etc.*) [2½]
  - ▶ Other:
- `bool IntSet::isSubsetOf(const IntSet& otherIntSet) const`
  - ▶ Traversing *entire* array and using an algorithm not in line with *class invariant* [3]
  - ▶ Out-of-bound (in general) traversing array(s) [2]
  - ▶ Logic error (role reversal, *etc.*) [2½]
  - ▶ Other:
- `void IntSet::reset()`
  - ▶ Not observing *class invariant* [1]
  - ▶ Not setting `used` [2½]
  - ▶ Other:
- `bool IntSet::add(int anInt)`
  - ▶ Various flaws or not implementing [[✓ ½] 1 1½ 2 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7]
- `bool IntSet::remove(int anInt)`
  - ▶ Various flaws or not implementing [½ 1 1½ [✓ 2] 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7]
- `IntSet IntSet::unionWith(const IntSet& otherIntSet) const`
  - ▶ Various flaws or not implementing [½ 1 1½ [✓ 2] 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7]
- `IntSet IntSet::intersect(const IntSet& otherIntSet) const`
  - ▶ Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7]
- `IntSet IntSet::subtract(const IntSet& otherIntSet) const`
  - ▶ Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7]
- `bool equal(const IntSet& is1, const IntSet& is2)`
  - ▶ Various flaws or not implementing [½ 1 1½ 2 2½ 3 3½ 4 4½ 5 5½ 6 6½ 7]

■ **Test Result**

- Inadequate cases demonstrated [up to 10 for not including any hardcopy output]
- Turning in output not generated by softcopy. [5]

■ **Other Issues**

Some lines cut off @ bottom of printout 1

```
// 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" information 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;

IntSet::IntSet() ← use initializer list?
{
   used = 0;
}

int IntSet::size() const
{
   // should be same as size.
   return used;
}

bool IntSet::isEmpty() const
{
   // if 'used'/'size' is greater than 0
   // then the intSet is not empty, else is empty.
   if (used > 0)
   {
      return false;
   }
   else
   {
      return true;
   }
}

bool IntSet::contains(int anInt) const
{
   // Check that IntSet is not empty then check for
   // anInt in the IntSet, return true if present,
   // else return false.
   if (used > 0)
   {
      for (int i=0; i < used; i++)
      {
         if(data[i] == anInt) return true;
      }
   }
   return false;
}

bool IntSet::isSubsetOf(const IntSet& otherIntSet) const
{

   //Check size of 'this' IntSet first, if 0 then subset is true.
   if (isEmpty())
   {
      return true;
   }
   else
   {
      for (int i = 0; i < used; i++)
      {
         if (!otherIntSet.contains(data[i]))
         {
            return false;
         }
      }
   }
   return true;
}

void IntSet::DumpData(ostream& out) const
```

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

IntSet IntSet::unionWith(const IntSet& otherIntSet) const
{
   int shared = 0;
   IntSet unionSet = *this;
   //Identify ~~# of~~ shared values
   for (int i=0; i<=otherIntSet.size(); i++)
```

? do the inner loop otherIntSet.size()+1 times?

```
   {
      for (int j=0; j<=used; j++)
      {
         if (otherIntSet.contains(data[j])) shared++;
```

data[j] is out of bounds when j == used

```
      }
   }
   //Verify that size of combination does not go above MAX SIZE
   if (((used + otherIntSet.size()) - shared) <= MAX_SIZE)
```

won't be checking the pre

```
   {
      for (int i = 0; i < otherIntSet.size(); i++)
      {
         if (!unionSet.contains(otherIntSet.data[i]))
         {
            unionSet.add(otherIntSet.data[i]);
         }
      }
   }
   return unionSet;
}

IntSet IntSet::intersect(const IntSet& otherIntSet) const
{
   // Creating retrun IntSet
   IntSet intersectSet = *this;

   // Removing every item that is not present
   // in otherIntSet
   for (int i = 0; i < size(); i++)
   {
      if (!otherIntSet.contains(data[i]))
      {
         intersectSet.remove(data[i]);
      }
   }

   return intersectSet;
}

IntSet IntSet::subtract(const IntSet& otherIntSet) const
{
   IntSet subtractedSet = *this;
   // Removing every item that is present
```

line(s) cut off

```
   {
      if (otherIntSet.contains(data[i]))
      {
         subtractedSet.remove(data[i]);
      }
   }

   return subtractedSet;
}

void IntSet::reset()
{
   // Reseting 'used' to 0
   used = 0;
}

bool IntSet::add(int anInt)
{
   // Check that the IntSet is not full
   if (used < MAX_SIZE)
   {
      // Check contains() for int value
      // if contains() returns true, do
      // NOT add new value and return false
      if(contains(anInt))
      {
         return false;
      }

      // if contains() returns false,
      // add new value and return true
      else if(!contains(anInt))
      {
         data[used] = anInt;
         used++;
         return true;
      }
   }

   return false;

}

bool IntSet::remove(int anInt)
{
   bool erased = true;
   int flag = 0;
   //Checking contains(anInt), if return is true
   if (contains(anInt))
   {
      for (int i=0; i < used; i++)
      {
         if(data[i] == anInt)
         {
            flag = i;
```

not the pre → will silently not add if pre not met

← lines cut off

```
      while (flag < MAX_SIZE)
      {
         data[flag] = data[flag+1];
         flag++;
      }
      used--;
   }
   else erased = false;
   return erased;
}

bool equal(const IntSet& is1, const IntSet& is2)
{
   bool equal = false;
   if (is1.isSubsetOf(is2) && is2.isSubsetOf(is1))
   {
      equal = true;
   }
   return equal;
}
```

data[flag] & data[flag+1] are out of bound in general