Examination 1

Name: **CS 3358, Spring 2026** **Grade:**

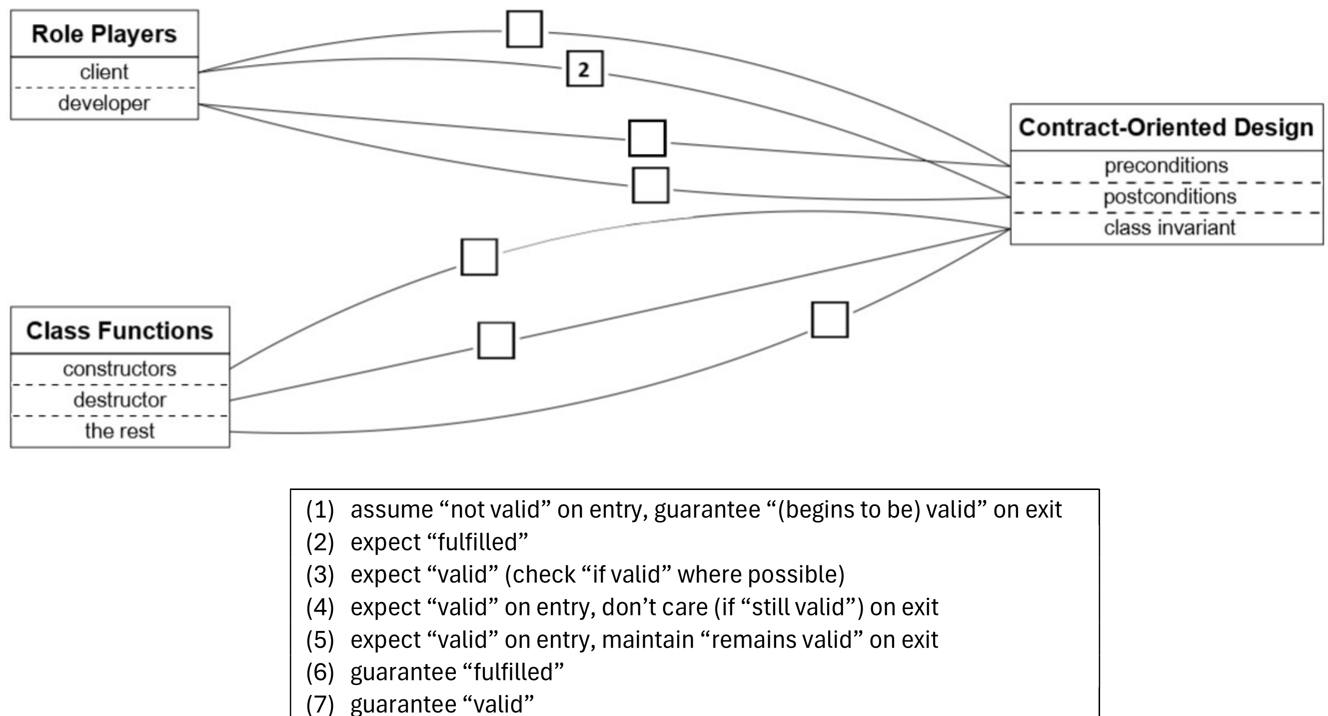
Instructions/directions/notes:

(1) You should attempt to answer all questions.
(2) Closed book and closed notes.
(3) Total points: 80.
   - *Italicized* number in parentheses preceding a question indicates earnable points for the question.
(4) Time allowed (in class): 80 minutes.
(5) **`IntSet.h`** and top of **`IntSet.cpp`** from ***Assignment 2*** provided (last 2 sheets).
(6) If needing more writing space for a question, use the blank page *opposite the page the question is on*.

1. *(6)* C++ supports both the *separation of a component's interface from the component's implementation* (by allowing them to be placed in different files) and *separate compilation* of the component (by allowing the component to be compiled in isolation) in order to facilitate *information hiding* and *modular programming*. In the list below, circle the ***two*** most important benefits of information hiding and modular programming (HINT: Must be those indicated/discussed in class).

Maintainability	Reliability	Reusability
Portability	Scalability	Usability

2. *(12)* Fill each box in the following diagram (that depicts *some logical associations related to object-based data type development using C++* and *contract-oriented approach*) discussed in class with a number indicating the best-matching entry on the accompanying boxed list. ***Each number should appear only once.*** One box is already filled in for you.



3. *(3)* Which of the following best describes how *abstraction* is useful to us in problem solving? (***Check only one.***)

- ☐ It enables us to never have to deal with the complexities of the problem.
- ☐ It promotes the use of bottom-up (not top-down) approach.
- ☐ It promotes the use of top-down (not bottom-up) approach.
- ☐ It helps us in managing the complexities of the problem.
- ☐ It promotes the use of pseudocode and thus frees us from language-dependent details.

4. *(3)* Which of the following best characterizes an ADT (*Abstract Data Type*)? (***Check only one.***)

- ☐ It specifies what it is and how it is to be implemented in an abstract language.
- ☐ It specifies what it is in a header file and how it is to be implemented in an implementation file.
- ☐ It specifies what it is and how it is to be implemented, not the language that is to be used.
- ☐ It specifies what it is, not how it is to be implemented.

5. *(3)* (***Check only one.***) A *member function* of a class…

   ☐ can directly access the private data members of only the invoking object.

   ☐ can directly access the private data members of all objects of the class visible to the function except the invoking object.

   ☐ can directly access the private data members of all objects of the class visible to the function.

   ☐ cannot directly access the private data members of an object of the class that is passed to the function by **`const`** reference.

   <u>NOTE</u>: Access means read for non-modifying functions and read/write for modifying functions.

6. *(8 points)* Indicate which item on the right best matches each of those on the left by filling each of the blanks on the right with the corresponding number. (*Each blank should have a different number*.)

Example Situation		Principle Violated
including **`using namespace std;`** *in header file*	①	___ *least privilege*
unnecessarily passing parameter by reference	②	___ *don't baffle the client*
inappropriately using **`friend`** *function*	③	___ *don't make soup too salty*
inappropriately exposing implementation detail	④	___ *avoid breaking encapsulation*

7. *(4)* If the *procedural* paradigm is adopted to solve a problem, the programming approach would be _________-oriented; if the *object* paradigm is adopted instead, the programming approach would be _________-oriented.

   (answer for <u>first</u> blank - ***check only one***)

   ☐ action

   ☐ data

   (answer for <u>second</u> blank - ***check only one***)

   ☐ action

   ☐ data

8. *(10)* For each statement below, circle **T** or **F** to indicate whether it is **T**rue or **F**alse:

- **[T F]** The statement **`SomeClass s2 = s1;`** will lead to the compiler calling the *copy constructor* of **`SomeClass`**.
- **[T F]** The **`const`** keyword *appearing immediately after the closing parenthesis (*`')'`*) of the parameter list of a (non-static) member function* protects *only the invoking object*.
- **[T F]** A non-member function can always get *friendship* privilege by simply inserting a "**`friend...`**" declaration somewhere in the function's scope (or the global scope).
- **[T F]** When overloading a binary operator (such as the **`operator==`**) in ***Assgnment 2***, the invoking object is implicitly the *left-hand-side* operand.
- **[T F]** Order of items is significant for a *sequence* (or *list*) but not for a *set* or a *multiset*.

9. *(6)* Suppose you want to pass variable **`x`** to a function as parameter. For each of the four situations regarding **`x`** shown on the 1st column of the table below, indicate the parameter passing mechanism you'd use (either because it is required or conventionally preferred as discussed in class) by checking one of the boxes in the 2nd, 3rd and 4th columns. (***Check only <u>1 box</u> for <u>each</u> of the bottom 4 rows.***)

# of bytes used by **`x`**	*intent to have side effect on* **`x`**	*pass* **`x`** *by value*	*pass* **`x`** *by reference*	*pass* **`x`** *by* **`const`** *reference*
small	no	☐	☐	☐
small	yes	☐	☐	☐
big	no	☐	☐	☐
big	yes	☐	☐	☐

10. *(3)* (You may want to refer to the **`IntSet.h`** file provided.) In ***Assignment 2***, the default capacity (**`DEFAULT_CAPACITY`**) of a **`IntSet`** is declared as a **`static`** member constant. In this context, the meaning of the word **`static`** includes which of the following? (***check all that apply***)

☐ *Only one copy* of the constant will be created even if many **`IntSet`** objects are created.
☐ The constant is created even if no **`IntSet`** object has been created yet.
☐ Client code using the **`IntSet`** class can refer to the constant via **`IntSet::DEFAULT_CAPACITY`**.

*(Remainder of this page not used so **Question 11** can appear in its entirety on a single page.)*

11. *(16)* Suppose you are the *developer* for the **`IntSet`** data type of ***Assignment 2***, *i.e.*, you are able to change the header and implementation files (**`IntSet.h`** and **`IntSet.cpp`**). At client's order, you want to provide a new member function named **`fCover`** for finding the *fraction* amount which the elements of another **`IntSet`** object are covered by (*i.e.*, can be found in) the invoking object. By definition, coverage any set (an empty one included) has over an *empty set* is 1.0 (= full coverage).

Shown below are example test results using a modified version of the ***Assignment 2*** tester program (note for this case: `is1` has 8 elements and `is2` has 6, and `is1` and `is2` have `2`, `4` and `5` in common):

```
is1: 8  1  2  5  0  7  4  -1
is2: 4  9  3  5  2  6
is3: (empty)
```

Call	Result	Explanation
`is1.fCover(is1)`	`returns 1`	◄— any set (empty one included) will fully (100%) cover *itself*
`is1.fCover(is2)`	`returns 0.5`	◄— 3 of 6 `is2` elements can be found in `is1` (3/6 = 0.5 = 50%)
`is1.fCover(is3)`	`returns 1`	◄— *empty* set is (by definition) fully (100%) covered by any other
`is2.fCover(is1)`	`returns 0.375`	◄— 3 of 8 `is1` elements can be found in `is2` (3/8 = 0.375 = 37.5%)
`is2.fCover(is2)`	`returns 1`	◄— any set (empty one included) will fully (100%) cover *itself*
`is2.fCover(is3)`	`returns 1`	◄— *empty* set is (by definition) fully (100%) covered by any other
`is3.fCover(is1)`	`returns 0`	◄— 0 of 8 `is1` elements can be found in `is3` (0/8 = 0 = 0%)
`is3.fCover(is2)`	`returns 0`	◄— 0 of 6 `is2` elements can be found in `is3` (0/6 = 0 = 0%)
`is3.fCover(is3)`	`returns 1`	◄— any set (empty one included) will fully (100%) cover *itself*

Write the code for the function as it would appear in the implementation file (**`IntSet.cpp`**).

you won't earn much credit if you don't observe this

NOTE: You are to write code for *both the function header and the function body*.

IMPORTANT:

- For the purpose of this exam, **DO NOT** use/call any of the functions of `IntSet` (including `add`, `remove`, `size`, `isEmpty`, `contains`, `isSubsetOf`, `unionWith`, `intersect` and `subtract`) when implementing the function. (Implicit use of the *assignment operator* and/or *copy constructor* is OK.)

don't overlook this

On the *last* page (part of "**interface file for *Assignment 2***"), show *how* and *where* you'd add this function to **`IntSet.h`**. (TIP: Write the *C++ statement* somewhere on the page and draw an *arrow* to point at exactly where you'd place the statement – statement is for *how*, arrow is for *where*.)

NOTE: You will earn *no credit* for any descriptive or C++ comments added.

12. *(6)* Say you are a *client* writing an application that uses the **`IntSet`** data type of ***Assignment 2*** and you are *not* able to change the header and implementation files (**`IntSet.h`** and **`IntSet.cpp`**). You want to add a custom (non-member) function named **`ffCover`** that you can use to accomplish the same effect as **`fCover`** of ***Question 11***. Shown below are example test results paralleling those seen in ***Question 11***:

```
                    is1: 8  1  2  5  0  7  4  -1
                    is2: 4  9  3  5  2  6
                    is3: (empty)
ffCover(is1, is1) returns 1
ffCover(is1, is2) returns 0.5
ffCover(is1, is3) returns 1
ffCover(is2, is1) returns 0.375
ffCover(is2, is2) returns 1
ffCover(is2, is3) returns 1
ffCover(is3, is1) returns 0
ffCover(is3, is2) returns 0
ffCover(is3, is3) returns 1
```

Write code to implement **`ffCover`** (*i.e.*, write the full function definition for **`ffCover`**, as it would appear in your *application* file).
NOTE: You are to write code for *both the function header and the function body*.

HINT: Risking telling the obvious, use available member functions of **`IntSet`** (not including the **`fCover`** of ***Question 11***) to get it done.

(interface file for *Assignment 2*)

```
// FILE: IntSet.h - header file for IntSet class
// CLASS PROVIDED: IntSet (a container class for a set of int values)
//
// ... ("Pre:  (none)" line for each function suppressed to save space)
//
// CONSTANT
//   static const int DEFAULT_CAPACITY = ____
//     IntSet::DEFAULT_CAPACITY is the initial capacity of an IntSet that is created by the
//     default constructor (i.e., IntSet::DEFAULT_CAPACITY is the highest # of distinct
//     values "an IntSet created by the default constructor" can accommodate).
//
// CONSTRUCTOR
//   IntSet(int initial_capacity = DEFAULT_CAPACITY)
//     Post: The invoking IntSet is initialized to an empty IntSet (i.e., one containing no
//           relevant values); the initial capacity is given by initial_capacity if
//           initial_capacity is >= 1, otherwise it is given by IntSet:DEFAULT_CAPACITY.
//     Note: When the IntSet is put to use after construction, its capacity will be resized
//           as necessary.
//
// CONSTANT MEMBER FUNCTIONS (ACCESSORS)
//   int size() const
//     Post: Number of elements in the invoking IntSet is returned.
//   bool isEmpty() const
//     Post: True is returned if the invoking IntSet has no elements, otherwise false is
//           returned.
//   bool contains(int anInt) const
//     Post: true is returned if the invoking IntSet has anInt as an element, otherwise false
//           is returned.
//   bool isSubsetOf(const IntSet& otherIntSet) const
//     Post: True is returned if all elements of the invoking IntSet are also elements of
//           otherIntSet, otherwise false is returned.
//           By definition, true is returned if the invoking IntSet is empty (i.e., an empty
//           IntSet is always isSubsetOf another IntSet, even if the other IntSet is empty).
//   void DumpData(std::ostream& out) const
//     Post: Contents of the invoking IntSet have been inserted into out with 2 spaces
//           separating one item from another if if there are 2 or more items.
//   IntSet unionWith(const IntSet& otherIntSet) const
//     Post: An IntSet representing the union of the invoking IntSet and otherIntSet is
//           returned.
//     Note: Equivalently (see postcondition of add), the IntSet returned is one that
//           initially is an exact copy of the invoking IntSet but subsequently has all
//           elements of otherIntSet added.
//   IntSet intersect(const IntSet& otherIntSet) const
//     Post: An IntSet representing the intersection of the invoking IntSet and otherIntSet
//           is returned.
//     Note: Equivalently (see postcondition of remove), the IntSet returned is one that
//           initially is an exact copy of the invoking IntSet but subsequently has all of
//           its elements that are not also elements of otherIntSet removed.
//   IntSet subtract(const IntSet& otherIntSet) const
//     Post: An IntSet representing the difference between the invoking IntSet and
//           otherIntSet is returned.
//     Note: Equivalently (see postcondition of remove), the IntSet returned is one that
//           initially is an exact copy of the invoking IntSet but subsequently has all
//           elements of otherIntSet removed.
//
// MODIFICATION MEMBER FUNCTIONS (MUTATORS)
//   void reset()
//     Post: The invoking IntSet is reset to become an empty IntSet.
//   bool add(int anInt)
//     Post: If contains(anInt) returns 0, anInt has been added to the invoking IntSet as a
//           new element and true is returned, otherwise the invoking IntSet is unchanged and
//           false is returned.
//   bool remove(int anInt)
//     Post: If contains(anInt) returns 1, anInt has been removed from the invoking IntSet
//           and true is returned, otherwise the invoking IntSet is unchanged and false is
//           returned.
//
// NON-MEMBER FUNCTIONS
//   bool operator==(const IntSet& is1, const IntSet& is2)
//     Post: True is returned if is1 and is2 have the same elements, otherwise false is
//           returned.
//           By definition, two empty IntSet's are equal.
//
// VALUE SEMANTICS
//   Assignment and the copy constructor may be used with IntSet objects.
```

```
#ifndef INT_SET_H
#define INT_SET_H

#include <iostream>

class IntSet
{
public:
   static const int DEFAULT_CAPACITY = 1;
   IntSet(int initial_capacity = DEFAULT_CAPACITY);
   IntSet(const IntSet& src);
   ~IntSet();
   IntSet& operator=(const IntSet& rhs);
   int size() const;
   bool isEmpty() const;
   bool contains(int anInt) const;
   bool isSubsetOf(const IntSet& otherIntSet) const;
   void DumpData(std::ostream& out) const;
   IntSet unionWith(const IntSet& otherIntSet) const;
   IntSet intersect(const IntSet& otherIntSet) const;
   IntSet subtract(const IntSet& otherIntSet) const;
   void reset();
   bool add(int anInt);
   bool remove(int anInt);

private:
   int* data;
   int capacity;
   int used;
   void resize(int new_capacity);
};

bool operator==(const IntSet& is1, const IntSet& is2);

#endif
```

(top of implementation file for *Assignment 2*)

```
// FILE: 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, dynamic array whose size is
//     stored in member variable capacity; 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 thru
//           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[capacity - 1].
//
//     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's 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.
```