

# ECE2800J

## Programming and Introductory Data Structures

### Interfaces; Invariants

#### Learning Objectives:

Understand what interfaces are and how to implement them in C++

Understand better what invariants are and how to use them to prevent some bugs

# Outline

- Interfaces
- Invariants

# ADTs

## Recall

- Recall the two main advantages of an ADT:
  1. Information hiding: we don't need to know the details of **how** the object is represented, nor do we need to know how the operations on those objects are implemented.
  2. Encapsulation: the objects and their operations are defined in the same place; the ADT combines both data and operation in one entity.

# ADTs

Recall

- To the caller, an ADT is only an **interface**.
  - **Interface**: the contract for using things of this type.
- Once you have an interface, you can pick from among many possible implementations as long as you satisfy the contract.

```
class IntSet { // a mutable set of integers
public:
    void insert(int v); // this + {v}
    void remove(int v); // this - {v}
    bool query(int v); // does v exist in this?
    int size(); // return |this|
};
```

# Interfaces

Separating out the details

- The class mechanism, as we've used it so far, has one shortcoming:
  - It mixes details of the **implementation** with the definition of the **interface**.

# Interfaces

Separating out the details

- Recall that the implementation of a class includes:
  1. Data members
  2. Method implementations
- The method implementations can be written separately from the class definition and are usually in a separate file.
  - Class definition in .h file; method implementation in .cpp file.
- Unfortunately, the **data members** still must be part of the class definition (in .h file).
  - Since any programmer using an `IntSet` must see that definition, those programmers know something about the implementation.

# Interfaces

Separating out the details

- Having data objects in the definition has two undesirable effects:
  1. It complicates the class definition, making it harder to read and understand.
  2. It communicates information to the programmer that s/he shouldn't know.
- The second problem can have very drastic consequences.
  - If a programmer using your class (mistakenly) makes an assumption about a "guarantee" that your implementation provides, but the interface doesn't promise, he is in trouble when you change the implementation.

# Interfaces

Separating out the details

- **Question:** How can you provide a class definition that carries no implementation details (i.e., data members) to the client programmer, yet still has interface information?
- **Answer:** Create an "interface-only" class as a **base class**, from which an implementation can be **derived**.
  - Note: classes **must** contain their data members, so this **base class cannot** have a real implementation!
  - Such a base class is called an **Abstract Base Class**, or sometimes a **Virtual Base Class**, because we're going to leverage virtual methods to do it.

# Interfaces

## Creating an abstract base class

- To create an abstract base class, we first provide an "interface-only" definition of `IntSet`.
- Because there will be no implementation, we need to declare its methods in a special way:
  - Declare each method as a virtual function.
  - “Assign” a zero to each of these virtual functions.

```
class IntSetFull { };

class IntSet {
    // OVERVIEW: mutable set of ints with bounded size
public:
    virtual void insert(int v) = 0;
        // MODIFIES: this
        // EFFECTS: set=set+{v}, throws IntSetFull if full
    virtual void remove(int v) = 0;
        // MODIFIES: this
        // EFFECTS: set=set-{v}
    virtual bool query(int v) = 0;
        // EFFECTS: returns true if v is in set,
        //           false otherwise
    virtual int size() = 0;
        // EFFECTS: returns |set|
};
```

# Interfaces

Creating an abstract base class

```
class IntSetFull { };
class IntSet {
public:
    virtual void insert(int v) = 0;
    virtual void remove(int v) = 0;
    virtual bool query(int v) = 0;
    virtual int size() = 0;
};
```

- These functions are called **pure virtual functions** and are declared not to exist.
- Think about them as a set of **function pointers**, all of which point to NULL.

# Interfaces

Creating an abstract base class

```
class IntSetFull { } ;  
class IntSet {  
    public:  
        virtual void insert(int v) = 0;  
        // MODIFIES: this  
        // EFFECTS: set=set+{v}, throws  
        //           IntSetFull if full  
        virtual void remove(int v) = 0;  
        virtual bool query(int v) = 0;  
        virtual int size() = 0;  
};
```

Note the use of `IntSetFull` as an “exception type”. It is used as something convenient to `throw` instead of some random `int`.

# Interfaces

## Abstract base classes

- A class with one or more Pure Virtual Functions is an **abstract** class.
- You **cannot** create **any instances** of an abstract class, because there are no implementation.
- For example, the following fails:

```
IntSet s;
```

- However, you can always define **references** and **pointers** to an abstract class, so these are both legal:

```
IntSet &r = <something>;
```

```
IntSet *p;
```

# Interfaces

## Abstract base classes

- Abstract base classes aren't very interesting without some derivative of `IntSet` to actually provide an implementation.
- This is done with a simple derived class:

```
const int MAXELTS = 100;  
class IntSetImpl : public IntSet {  
    int elts[MAXELTS];  
    int numElts;  
public:  
    IntSetImpl();  
    void insert(int v);  
    void remove(int v);  
    bool query(int v);  
    int size();  
};
```

**Note:** The implementation has data members.

In general, besides new function members, a derived class can also have new data members.

# Interfaces

## Abstract base classes

- Abstract base classes aren't very interesting without some derivative of IntSet to actually provide an implementation.
- This is done with a simple derived class:

```
const int MAXELTS = 100;
class IntSetImpl : public IntSet {
    int elts[MAXELTS];
    int numElts;
public:
    IntSetImpl();
    void insert(int v);
    void remove(int v);
    bool query(int v);
    int size();
};
```

**Note:** This implementation could be **either** the sorted or unsorted versions.

# Interfaces

## Abstract base classes

- Abstract base classes aren't very interesting without some derivative of IntSet to actually provide an implementation.
- This is done with a simple derived class:

```
const int MAXELTS = 100;
class IntSetImpl : public IntSet {
    int elts[MAXELTS];
    int numElts;
public:
    IntSetImpl();
    void insert(int v);
    void remove(int v);
    bool query(int v);
    int size();
};
```

**Note:** the derived class has to implement the constructor. In the past, it was always in the base class.

It can't be there, because the base class has no implementation to construct!



# In principle, should the implementation code of the derived class of an abstract class be provided to its user?

Select all the correct answers.

- **A.** Yes, so the user understands how the abstract class is implemented.
- **B.** Yes, so the constructor can be called by the user.
- **C.** No, it would go against the spirit of an ADT.
- **D.** No, no file related to the implementation is needed by the user.



# Interfaces

## Abstract base classes

- The interface (the abstract base class) is typically defined in a public header (\*.h) file
  - Users of the **interface** include the \*.h file.
- The implementation (**the derived class**) is defined in a source (\*.cpp) file
  - Users of the interface only *link* against (i.e., compile the file into object code and link with other object codes)
- So, a user of the IntSet abstraction **never sees** the definition for class IntSetImpl.
- The only thing that remains is to give users the means to create a new IntSet:
  - However, they can't do it in the normal way: IntSet s;
  - Also, they can't create objects of the derived class, because its definition is **not visible** to them.

# Interfaces

## Abstract base classes

- If only one instance of the class is needed, the **\*.h** file typically includes the following declaration of an access function:

```
// header file
IntSet *getIntSet();
    // EFFECTS: returns a pointer
    //           to the IntSet
```

- The **\*.cpp** file defines a single, **static instance** (only visible to the \*.cpp file) of the implementation and body of the access function:

```
// source file
static IntSetImpl impl;
IntSet *getIntSet() {
    return & impl;
}
```

# Interfaces

## Abstract base classes

- If only one instance of the class is needed, the **\*.h** file typically includes the following declaration of an access function:

```
// header file  
IntSet *getIntSet();  
// EFFECTS: returns a pointer  
// to the IntSet
```

- The **\*.cpp** file defines a single, **static instance** (only visible to the \*.cpp file) of the implementation and body of the access function:

```
// source file  
static IntSetImpl impl  
IntSet *getIntSet() {  
    return & impl;  
}
```

**Note:** Now the user can do the following and it will be valid:

```
IntSet *s = getIntSet();
```

# Interfaces

## Abstract base classes

- If more than one instance of the class is needed, we need to provide a function that creates them **dynamically**...
  - You will see how to do this later.

# Outline

- Interfaces
- Invariants

# Invariants

- An invariant is a set of conditions that must always evaluate to true at certain well-defined points; otherwise, the program is incorrect.
- For ADT, there is so called **representation invariant**.

# Invariants

- A **representation invariant** applies to the data members of ADT.
- It describes the conditions that must hold on those members for the representation to correctly implement the abstraction.
- It must hold immediately before exiting each method of that implementation – including the constructor.
  - Example: insert() member of IntSet.
  - This is called **establishing the invariant**.

# Invariants

## Representation Invariants

- Each method in the class can assume that the invariant is true **on entry if:**
  - The representation invariant holds immediately before exiting each method (including the constructor), **and**
  - Each data element is truly private.
- This is true because the only code that can change the data members belongs to the methods of that class, and those methods always establish the invariant.

# Invariants

## Representation Invariants

- We've seen two examples of representation invariants, both applied to the private data members of an `IntSet` representation:

```
int elts[MAXELTS];  
int numElts;
```

- For the unsorted version, the invariant is:
  - The first `numElts` members of `elts` contain the integers comprising the set, with no duplicates.
- For the sorted version, the invariant is:
  - The first `numElts` members of `elts` contain the integers comprising the set, from lowest to highest, with no duplicates.

# Invariants

## Representation Invariants

- We used these invariants to write the methods of each implementation.
- For example:

```
insert(int v)          // unsorted version
    if v not in elts // don't allow duplicates
        elts[numElts] = v // this breaks invariant
        numElts++         // this restores it
```

```
insert(int v)          // sorted version
    if v not in elts // don't allow duplicates
        make gap in array // this breaks invariant
        elts[gap] = v // restore elts invariant
        numElts++         // restore numElts invariant
```

# Invariants

## Representation Invariants

- The representation invariant plays a crucial role in implementing an abstract data type.
- Before writing a **single** line of code, write down the rep invariant!
- That tells you **how** to write each method.
- Essentially, for each method, you should:
  - Do the work of the method (i.e. insert)
  - Repair the invariants you broke

# Invariants

## Checking for Representation Invariants

- Invariants can also be coded, to check the sanity of the structure.
- For even moderately complicated data structures, it is worth writing a function to check for invariants.
- In the IntSet case, we **can** check to see if the array satisfies the respective invariants such as there is no duplication or the array is sorted.

# Invariants

## Checking for Representation Invariants

- Use sorted representation for example. We will write the following function to check the invariants:

```
bool strictSorted(int a[], int size)
    // REQUIRES: a has size elements
    // EFFECTS: returns true if a is sorted
    //           with no duplicates
```

- How can we tell if an array is sorted with no duplicates?
  - If  $\text{size} \leq 1$ , the array is sorted with no duplicates.
  - If  $\text{size} > 1$ , then the array must satisfy  
 $a[0] < a[1] < \dots < a[\text{size}-1]$

# Invariants

Checking for Representation Invariants

```
bool strictSorted(int a[], int size)  {
    // REQUIRES: a has size elements
    // EFFECTS: returns true if a is sorted
    //           with no duplicates

    if (size <= 1) return true;

    for (i=0; i<size-1; i++) {
        if (a[i] >= a[i+1]) {
            return false;
        }
    }
    return true;
}
```

# Invariants

## Checking for Representation Invariants

- Writing these “checker” functions is very useful – you can use them for **defensive programming**.
- So, you can write a private method to check whether all invariants are true (**before exiting**, or after entering, each method):

```
bool repOK();  
// EFFECTS: returns true if the  
// rep. invariants hold
```

- For the sorted version, repOK would be:

```
bool repOK() {  
    return strictSorted(elts, numElts);  
}
```

# Invariants

## Checking for Representation Invariants

- Next, add the following code **right before** returning from any function that modifies any of the representation:

```
assert(repOK());
```

- If you are truly paranoid, you can write the same line at the **beginning** of every method, too; this checks that the assumption the method relies on is true.

# References

- **Problem Solving with C++ (8<sup>th</sup> Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 10.4 Introduction to Inheritance
  - Chapter 15.1 Inheritance Basics
  - Chapter 15.3 Virtual Functions in C++