

# Ve 280

## Programming and Introductory Data Structures

### **Abstract Data Types**

#### **Learning Objectives:**

Understand what is an abstract data type (ADT)

Understand the usefulness of an ADT

Know how to define an ADT in C++

# Outline

- Introduction to Abstract Data Types
- Class in C++: A Trivial Example
- More Details on Class
- Another Class Example: a Mutable Set of Integers (IntSet)
- Improve the Efficiency of IntSet

# Types

- The role of a type:
  - The set of values that can be represented by items of the type
  - The set of operations that can be performed on items of the type.
- Example
  - C++ string                  values:
  - operations:

# Struct Types

- Struct types have the following feature:
  - **Every detail** of the type is known to all users of that type.
  - This is sometimes called the **concrete implementation**.
- Example: the `struct Grades` talked before.

```
struct Grades {  
    char name[9];  
    int  midterm;  
    int  final;  
};
```

# Struct Types

```
struct Grades {  
    char name[9];  
    int  midterm;  
    int  final;  
};
```

- Every function knows the details of exactly how Grades are represented.
- A change to the Grades definition (for example, change C-string for name to a C++-String) requires that we **make changes throughout the program** and recompile everything using this struct.

# Abstract Data Types

## Introduction

- Contrast the property of struct types with that of the functions
  - A function written by others shows **what** the function does, but not **how** it does it
- For function, if we find a faster way to implement, we can just replace the old implementation with the new one
  - No other components of the program calling the function need to change

# Abstract Data Types

## Introduction

- To solve the problem for struct type, we'll define **abstract data types**, or **ADTs**.
- An ADT provides an **abstract description** of **values** and **operations**.
- The definition of an ADT must combine **both** some notion of **what** values that type represents, and **what** operations on values it supports.
  - However, we can leave off the details of **how**.
- Example: mobile phone
  - Type: a portable telephone that can make and receive calls
  - Operations: turn on/off, make/receive call, text message

# Abstract Data Types

## Introduction

- Abstract data types provide the following two advantages:
  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.



# Abstract Data Types

## Example

- `list_t`:
  - Information Hiding: In the `list_t` data type, you never knew the precise implementation of the `list_t` structure (except by looking in `recursive.cpp`).
  - Encapsulation: The definitions of the operations on lists (`list_print`, `list_make`, etc.) were found in the same header file as the type definition of `list_t`.

# Abstract Data Types

## Benefits

- Abstract data types have several benefits like we had with functional abstraction:
  - ADTs are **local**: the implementation of other components of the program does not depend on the **implementation** of ADT.
    - To realize other components, you only need to focus **locally**.
  - ADTs are **substitutable**: you can change the implementation and no users of that type can tell.

# Abstract Data Types

## Introduction

- Someone still needs to know/access the details of how the type is implemented.
  - I.e., how the values are represented and how the operations are implemented
  - This is referred to as the “**concrete representation**” or just the “**representation**”
- Question: Who can access the representation?
- Answer: **only** the operations defined for that type should have access to the representation.
  - Everyone else may access/modify this state only **through** operations.

# Abstract Data Types

## On to Classes

- C++ “class” provides a mechanism to give **true** encapsulation.
- The basic idea behind a class is to provide **a single entity** that both defines:
  - The **value** of an object.
  - The **operations** available on that object. These operations are sometimes also called **member functions** or **methods**.

# Outline

- Introduction to Abstract Data Types
- **Class in C++: A Trivial Example**
- More Details on Class
- Another Class Example: a Mutable Set of Integers (IntSet)
- Improve the Efficiency of IntSet

# Abstract Data Types

## Classes – A trivial example

```
class anInt {  
    // OVERVIEW: a trivial class to get/set a  
    //             single integer value  
    int      v;  
public:  
    int      get_value();  
            // EFFECTS: returns the current  
            //             value  
    void     set_value(int newValue);  
            // MODIFIES: this  
            // EFFECTS: sets the current value  
            // equal to newValue  
};
```

# Abstract Data Types

## Classes – A trivial example

```
class anInt {  
    // OVERVIEW: a trivial class to get/set a  
    //           single integer value  
    int      v;  
public:  
    int      get_value();  
            // EFFECTS: returns the current  
            //           value  
    void      set_value(int newValue);  
            // RME: Omitted for space  
};
```

- There are a few things to notice about this definition:
  - There is a single OVERVIEW specification that describes the class as a whole.

# Abstract Data Types

## Classes – A trivial example

```
class anInt {  
    // OVERVIEW: Omitted for space  
    int v;  
    public:  
    int get_value();  
    // EFFECTS: returns the current value  
    void set_value(int newValue);  
    // RME: Omitted for space  
};
```

- There are a few things to notice about this definition:
  - The declaration includes both data elements (`int v`) and member functions/methods (`get_value` and `set_value`).



# Abstract Data Types

## Classes – A trivial example

```
class anInt {  
    // OVERVIEW: Omitted for space  
    int    v;  
    public:  
    int    get value();  
           // EFFECTS: returns the current  
           //          value  
    void    set value(int newValue);  
           // MODIFIES: this  
           // EFFECTS: sets the current value  
           // equal to arg  
};
```

- There are a few things to notice about this definition:
  - Each function that is declared must have a corresponding specification.

# Abstract Data Types

## Classes – A trivial example

```
class anInt {  
    // OVERVIEW: Omitted for space  
    int    v;  
    public:  
    int    get_value();  
           // EFFECTS: returns the current value  
    void    set_value(int newValue);  
           // MODIFIES: this  
           // EFFECTS: sets the current value  
           // equal to arg  
};
```

- There are a few things to notice about this definition:
  - `set_value` says it MODIFIES `this`. This is the generic name for "**this object**".

# Outline

- Introduction to Abstract Data Types
- Class in C++: A Trivial Example
- **More Details on Class**
- Another Class Example: a Mutable Set of Integers (IntSet)
- Improve the Efficiency of IntSet

# Abstract Data Types

## Classes – More Details

- By default, every member of a class is **private**.
  - Members = data members + function members
- A private member is visible only to **other members** of this class.
  - `int v` was a private member in the class `anInt`.
  - “Private” hides the implementation of the type from the user.

# Abstract Data Types

## Classes – More Details

- However, if everything were private, the class wouldn't be particularly useful!
- So, the **public** keyword is used to signify that some members are **visible** to anyone who sees the class declaration, not only visible to other members of this class.
  - Everything **after** the **public** keyword is **visible** to others.

# Abstract Data Types

## Classes – A trivial example

```
class anInt {  
    // OVERVIEW: a trivial class to get/set a  
    //           single integer value  
    int      v;  
    public:  
    int      get_value();  
            // EFFECTS: returns the current  
            //           value  
    void     set_value(int newValue);  
            // MODIFIES: this  
            // EFFECTS: sets the current value  
            // equal to arg  
};
```

# Abstract Data Types

## Classes – A trivial example

**This declaration, as it is, is incomplete. We have not yet defined the bodies of the member functions.**

```
class anInt {  
    // OVERVIEW: a trivial class to get/set a  
    //           single integer value  
    int      v;  
public:  
    int      get_value();  
            // EFFECTS: returns the current  
            //           value  
    void     set_value(int newValue);  
            // MODIFIES: this  
            // EFFECTS: sets the current value  
            // equal to arg  
};
```

# Abstract Data Types

Classes – Defining member functions

```
class anInt {  
    // OVERVIEW: a trivial class to get/set a  
    //           single integer value
```

**Note: You can actually define the functions within the class definition, but this “exposes” information, which is best left hidden!**

```
int anInt::get_value() {  
    return v;  
}  
void anInt::set_value(int newValue) {  
    v = newValue;  
}
```

The definitions of member functions are usually put in the .cpp file;

**You should include .h in the .cpp!**



# Abstract Data Types

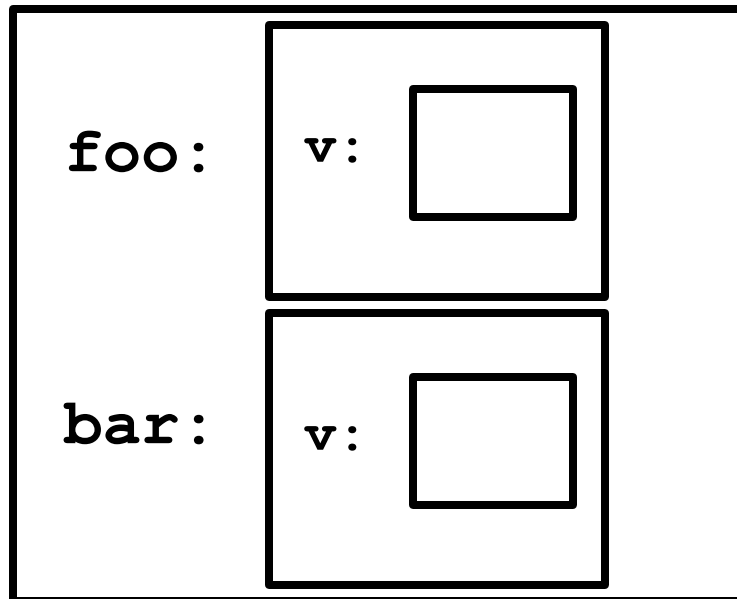
## Classes – Declaring class objects

- We can declare **objects** of type `anInt` as you would expect:

```
anInt    foo;
```

```
anInt    bar;
```

- This produces an environment with two objects:



These values are still undefined (i.e. there is no initial value). We'll see several ways to set an **initial** value for data members later.

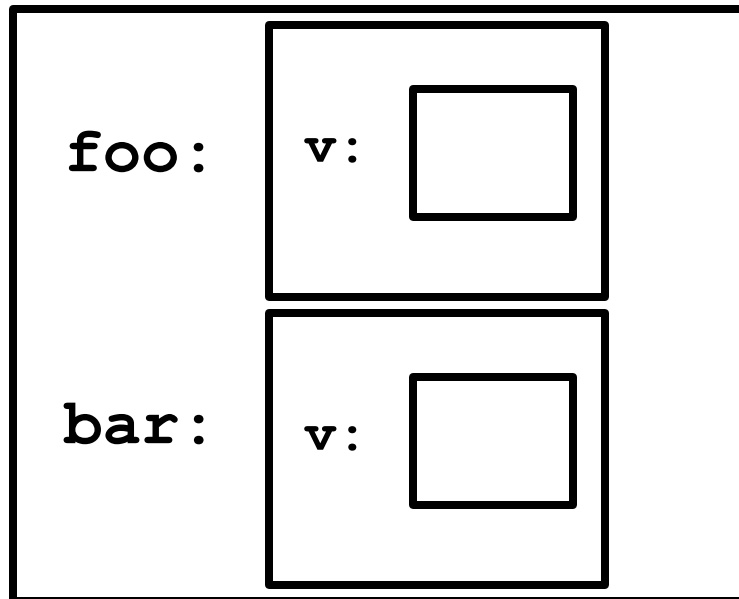
# Abstract Data Types

Classes – Establishing data member values

- We can call the `set_value` member function to establish a value:

```
foo.set_value(1);
```

This calls `foo`'s `set_value()` method.



# Abstract Data Types

Classes – Establishing data member values

- There is one very important difference between normal function calls and **member** function calls:
  - The **other** members of the object are **also visible** to the function members!
  - For example, `v` is visible to the function `set_value()`

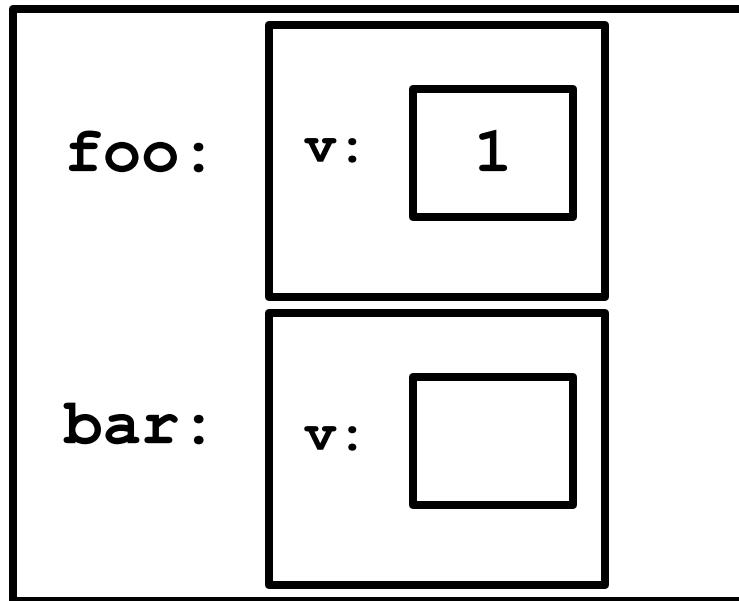
```
void anInt::set_value(int newValue) {  
    v = newValue;  
}
```

# Abstract Data Types

Classes – Establishing data member values

- So, set value changes **foo**'s  $v$ :

```
foo.set_value(1);
```



# Abstract Data Types

## Classes – Accessing data member values

- We can't access `v` directly:

```
cout << foo.v; // Compile-time error  
because v is private!
```

- However, we can use the `get_value()` method to do so for us:

```
cout << foo.get_value(); // OK.  
because get_value() is public!
```

- Finally, class objects can be passed just like anything else.
- Like everything else (except arrays), they are passed by value.

# Abstract Data Types

## Class Example: Classes

- What is the result of the following?

```
void add_one(anInt i) {  
    i.set_value(i.get_value()+1);  
}  
  
int main() {  
    anInt foo;  
    foo.set_value(0);  
    add_one(foo);  
    cout << foo.get_value() << endl;  
    return 0;  
}
```

# Abstract Data Types

## Classes – Passing by reference

- To pass a class object by reference, you use either a pointer argument or a reference argument, i.e.:

```
void add_one(anInt *ip) {  
    ip->set_value(ip->get_value() + 1);  
}
```

- This version would change the class object passed to it!



# Which Statements Are Correct?

- **A.** A C++ class can define a type.
- **B.** The information stored in an object of a class is accessible to any one.
- **C.** A class defines all the basic operations that are possible on objects of that class.
- **D.** All member functions of a class are accessible to any one.





# Outline

- Introduction to Abstract Data Types
- Class in C++: A Trivial Example
- More Details on Class
- Another Class Example: a Mutable Set of Integers (IntSet)
- Improve the Efficiency of IntSet

# Abstract Data Types

## Using Classes

- Suppose we wanted to build an abstraction that held a **mutable** set of integers.
- This is a set in the mathematical sense:
  - A collection of zero or more integers, with **no duplicates**.
- The set is “mutable” because we can insert values into and remove objects from the set.

# Abstract Data Types

## Using Classes

- Suppose we wanted to build an abstraction that held a **mutable** set of integers.
- There are four **operations** on this set that we will define:
  1. Insert a value into the set.
  2. Remove a value from the set.
  3. Query to see if a value is in the set.
  4. Count the number of elements in the set.

# Abstract Data Types

## Using Classes

- Here is an **incomplete** definition of a class implementing such an ADT:

```
class IntSet {  
    // OVERVIEW: a mutable set of integers  
  
    public:  
        void insert(int v);  
            // MODIFIES: this  
            // EFFECTS: this = this + {v}  
        void remove(int v);  
            // MODIFIES: this  
            // EFFECTS: this = this - {v}  
        bool query(int v);  
            // EFFECTS: returns true if v is in this,  
            //           false otherwise  
        int  size();  
            // EFFECTS: returns |this|.   
};
```

# Abstract Data Types

## Using Classes

```
class IntSet { // omitted OVERVIEW for space
public:
    void insert(int v); // omitted RME for space
    void remove(int v); // omitted RME for space
    bool query(int v); // omitted RME for space
    int  size(); // omitted RME for space
};
```

- The class is incomplete because we haven't chosen a **representation** for sets.
- Choosing a representation involves two things:
  - Deciding what **concrete** data elements will be used to **represent the values** of the set.
  - Providing an **implementation** for each **method**.

# Abstract Data Types

## Using Classes

```
class IntSet { // omitted OVERVIEW for space
    public:
        void insert(int v); // omitted RME for space
        void remove(int v); // omitted RME for space
        bool query(int v); // omitted RME for space
        int  size(); // omitted RME for space
};
```

- Despite not having a representation for a set, the (incomplete) definition above is all that a **customer** of the `IntSet` abstraction needs to know since it has:
  - The general overview of the ADT.
  - The specification of each method.

# Abstract Data Types

## Using Classes

- Start with a representation for the set itself:
  - Use an array.
  - Represent a set of size  $N$  as an **unordered** array of integers with no duplicates, stored in the first  $N$  slots of the array.
  - `int numElts`: maintains the number of elements currently in the array.
- These last two statements are called **representation invariants** or **rep invariants** (more on this later).
- This invariant is a rule that the representation must obey both **immediately before** and **immediately after** any method's execution.

rep  
invariant

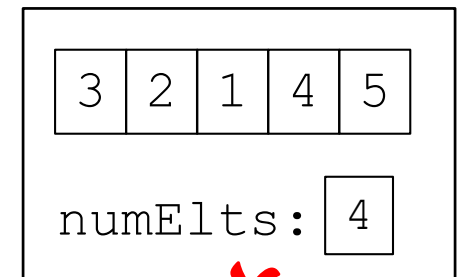
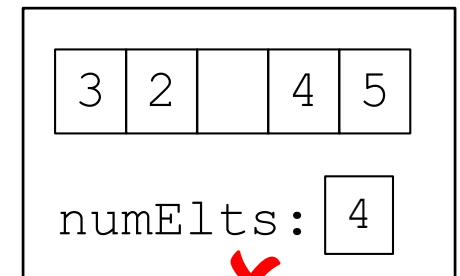
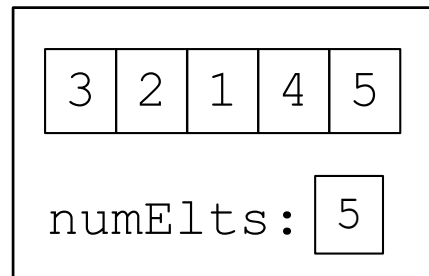
# Abstract Data Types

## Using Classes

- Start with a representation for the set itself:
  - Use an array.
  - Represent a set of size N as an **unordered** array of integers with no duplicates, stored in the first N slots of the array.
  - `int numElts`: maintains the number of elements currently in the array.

rep  
invariant

```
class IntSet {  
    int elts[100];  
    int numElts;  
    ...  
};
```





# Abstract Data Types

## Using Classes

- Since this is an array, and arrays have maximum sizes, we have to choose a maximum size and modify the OVERVIEW:

```
// OVERVIEW: a mutable set of  
//             integers, |set| <= 100
```

- We also have to change the EFFECTS clause of insert:

```
// EFFECTS: this = this + {v} if  
//             room available, throws int  
//             100 otherwise
```

# Abstract Data Types

## Using Classes

```
const int MAXELTS = 100;
class IntSet {
    // OVERVIEW: a mutable set of integers, |set| <= MAXELTS
    int      elts[MAXELTS];
    int      numElts;
public:
    void insert(int v);
        // MODIFIES: this
        // EFFECTS: this = this + {v} if room,
        //           throws int MAXELTS otherwise
    void remove(int v);
        // MODIFIES: this
        // EFFECTS: this = this - {v}
    bool query(int v); // RME omitted for space
    int  size();       // RME omitted for space
};
```

Use a global constant like we have talked about.

# Abstract Data Types

## Using Classes

Given this representation, and the representation invariants, we can write the methods.

```
const int MAXELTS = 100;

class IntSet { // OVERVIEW omitted for space
    int      elts[MAXELTS];
    int      numElts;
public:
    void insert(int v); // RME omitted for space
    void remove(int v); // RME omitted for space
    bool query(int v);  // RME omitted for space
    int  size();        // RME omitted for space
};
```

```
int IntSet::size() {
    return numElts;
}
```

Because our rep invariant says that `numElts` is always the size of the set, we can return it directly.

# Abstract Data Types

## Using Classes

- Next, consider the three final routines:
  - query: search the array looking for a specific number.
  - remove: search the array for a number; if it exists, remove it.
  - insert: search the array for a number; if it doesn't exist, add it.
- All three of these have "search" in common.
- One might be tempted to just write `insert` and `remove` in terms of `query`, will this work?
  - Hint: think about `remove`.
- `query` only tells us **whether** the element exists, not **where** – we need one more method...

# Abstract Data Types

## Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
    int      elts[MAXELTS];
    int      numElts;
```

```
    int indexOf(int v);
        // EFFECTS: returns the index of
        //              v if it exists in the
        //              array, MAXELTS otherwise.
```

```
public:
    void insert(int v);
    void remove(int v);
    bool query(int v);
    int  size();
};
```

**Note:** This member function must be **private**. This is because it exposes details about the concrete representation. It is inappropriate to expose these details to a user of this class.

# Abstract Data Types

## Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
    int      elts[MAXELTS];
    int      numElts;
    int indexOf(int v); // RME omitted for space
public:
    void insert(int v); void remove(int v); // RME omitted
    bool query(int v);  int  size();        // RME omitted
};
```

```
int IntSet::indexOf(int v) {
    for (int i = 0; i < numElts; i++) {
        if (elts[i] == v) return i;
    }
    return MAXELTS;
}
```

# Abstract Data Types

## Using Classes

```
const int MAXELTS = 100;

class IntSet { // OVERVIEW omitted for space
    int      elts[MAXELTS];
    int      numElts;
    int indexOf(int v); // RME omitted for space
public:
    void insert(int v); void remove(int v); // RME omitted
    bool query(int v);  int  size();        // RME omitted
};
```

With `indexOf`, `query` is trivial...

```
bool IntSet::query(int v)    {
    return (indexOf(v) != MAXELTS);
}
```



# How to Implement `insert(v)`?

Select all the correct answers.

- **A.** We can first search `v` to check if it is already there with `indexOf(v)`
- **B.** If `v` is not present, we then add `v`
- **C.** If we add `v`, it should be added as `elts[numEelts-1]` before we increment `numEelts`
- **D.** If `v` is added, we need to increment `numEelts`





# Abstract Data Types

## Using Classes

- The code for `insert` is not much more difficult than query:
  - First look for the `indexOf` the element to insert.
  - If it doesn't exist, we need to add this element to the **end** of the array.
  - What is the index of the current “end” ?



- Place the element in the next slot and **update** `numElts`.
- The only exception to this is if `numElts` already equals `MAXELTS`.

# Abstract Data Types

## Using Classes

```
const int MAXELTS = 100;
class IntSet { // OVERVIEW omitted for space
    int      elts[MAXELTS];
    int      numElts;
    int indexOf(int v); // RME omitted for space
public:
    void insert(int v); void remove(int v); // RME omitted
    bool query(int v); int size();          // RME omitted
};
```

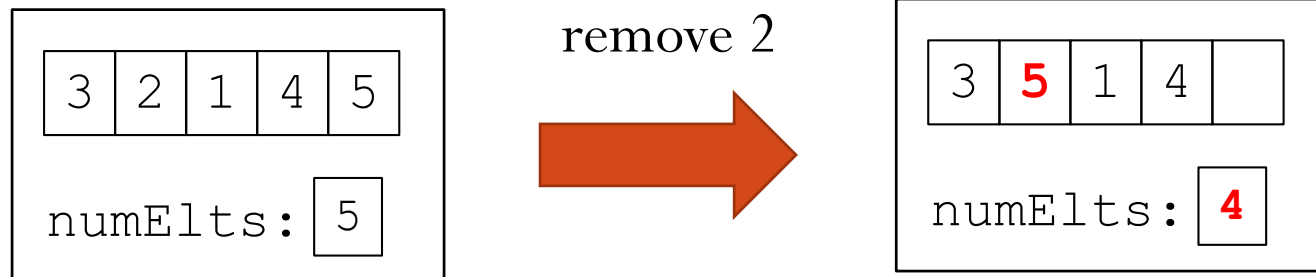
```
void IntSet::insert(int v) {
    if (indexOf(v) == MAXELTS) {
        if (numElts == MAXELTS) throw MAXELTS;
        elts[numElts++] = v;
    }
}
```

# How about Remove?

- If the element (its index is called the `victim`) is in the array, we have to remove it leaving a "hole" in the array.
- What representation invariants are violated?
  - How can we fix them?

# How about Remove?

- Instead of moving each element after the victim to the left by one position, pick up the current "last" element and move it to the hole.
- This also breaks the invariant on `numElems`, so we must fix it.



# Abstract Data Types

## Using Classes

```
void IntSet::remove(int v) {  
    int victim = indexOf(v);  
    if (victim != MAXELTS) {  
        elts[victim] = elts[numElts-1];  
        numElts--;  
    }  
}
```

# Abstract Data Types

## Using Classes

- Question: There is one problem with our implementation. What is it?
- Hint: Consider the newly-created set:

```
IntSet s;
```

What does the computer actually create when we declare `s`?

# Abstract Data Types

## Using Classes

- Question: There is one problem with our implementation. What is it?
- Answer: On creation, *s*'s data members are **uninitialized**!
- This means that the value of `numElements` could be a random value, but our representational invariant says it must be zero!
- How can we fix this?

# Abstract Data Types

## Automatically Initializing Classes

- Using **constructor**!
- The constructor (really, the **default** constructor) has the following type signature:

```
class IntSet { // OVERVIEW omitted for space
    ...
    public:
        IntSet();
        // EFFECTS: creates an empty IntSet
    ...
};
```



# Abstract Data Types

## Automatically Initializing Classes

```
IntSet();  
// EFFECTS: creates an empty IntSet
```

- The name of the function is the same as the name of the class.
- This function doesn't have a return type.
- It also does not take an argument in this case.
- It is guaranteed to be the **first** function called immediately after an object is created.
- It builds a “blank” uninitialized `IntSet` and makes it satisfy the rep invariant.

# Abstract Data Types

Automatically Initializing Classes

```
IntSet();  
    // EFFECTS: creates an empty IntSet
```

- Here's how it's written:

```
IntSet::IntSet() : numElts(0)  
{  
}
```

# Abstract Data Types

## Automatically Initializing Classes

```
IntSet::IntSet()  
    : numElts(0)  
{  
}
```

```
Class_T::Class_T() : anInt(0),  
                    aDouble(1.2),  
                    aString("Yes")  
{  
}
```

- This syntax is called "initialization syntax".
- Each data member is initialized this way.
- **Note**: The order in which elements are initialized is the order they **appear in the definition**, NOT the order in the initialization list. It is a good practice to keep them in the same order to avoid confusion.

# Abstract Data Types

## Automatically Initializing Classes

- Alternatively, we could write this function as follows, but this is not considered as a good way!

```
IntSet::IntSet()  
{  
    numElts = 0;  
}
```



Not Recommended

# Abstract Data Types

## A Benefit of Classes

- Now, instead of writing this:

```
void add_one (int a[], int elts);
```

and having to worry about the number of elements in the array. All we have to write is this:

```
void add_one (IntSet& set);
```

and we no longer have to worry about the array and its count being separated.

# Const Member Functions

- A slight change to the class definition:

```
const int MAXELTS = 100;
class IntSet {
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;

public:
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int size() const;
};
```

# Const Member Functions

```
int size() const;
```

- Each member function of a class has an extra, implicit parameter named **this**.
  - “**this**” is a pointer to the current instance on which the function is invoked.
- **const** keyword modifies the implicit **this** pointer: **this** is now a pointer to a **const instance**.
  - Means: the member function **size()** cannot change the object on which **size()** is called.
  - By its definition, **size()** shouldn't change the object! Adding **const** keyword prevents any accidental change.
  - It is a good practice to add **const** keyword when possible!

# Const Member Functions

- Implement **size()**

```
int IntSet::size() const {  
    return numElts;  
}
```

The function body is the same as before.

- A **const** object can only call its **const** member functions!

```
const IntSet is;  
cout << is.size(); ✓  
is.insert(2); ✗
```



# Const Member Functions

- If a const member function calls other **member** functions, they must be **const** too!

```
void A::g() const { f(); }
```

```
void A::f() {...}
```



```
void A::f() const {...}
```



# Outline

- Introduction to Abstract Data Types
- Class in C++: A Trivial Example
- More Details on Class
- Another Class Example: a Mutable Set of Integers (IntSet)
- Improve the Efficiency of IntSet



How many elements of the array must `indexOf` examine?

```
int IntSet::indexOf(int v) {  
    for (int i = 0; i < numElts; i++) {  
        if (elts[i] == v) return i;  
    }  
    return MAXELTS;  
}
```

- A. In the best case, 1 element
- B. In the worst case, `numElts` elements
- C. In the worst case, `MAXELTS` elements
- D. None of the above



# Abstract Data Types

## Improving Efficiency

- We say the time for `indexOf` grows **linearly** with the size of the set.
- If there are  $N$  elements in the set, we have to examine all  $N$  of them **in the worst case**. For large sets that perform lots of queries, this is too expensive!
- Luckily, we can replace this implementation with a different one that can be more efficient. The only change we need to make is to the **representation (implementation)** – the abstraction can stay precisely the same.

# Abstract Data Types

## Improving Efficiency

- Still use an array to store the elements of the set and the values will still occupy the first `numElements` slots.
- However, now we'll keep the elements in **sorted** order.

# Question: Which Member Functions of the Class Should Be Changed?

```
const int MAXELTS = 100;
class IntSet {
    // OVERVIEW: a mutable set of integers
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;
public:
    IntSet();
    void insert(int v);
    void remove(int v);
    bool query(int v) const;
    int size() const;
};
```

# Abstract Data Types

## Improving Efficiency

- The constructor and size methods don't need to change at all since they just use the `numElts` field.

- `query` also doesn't need to change.

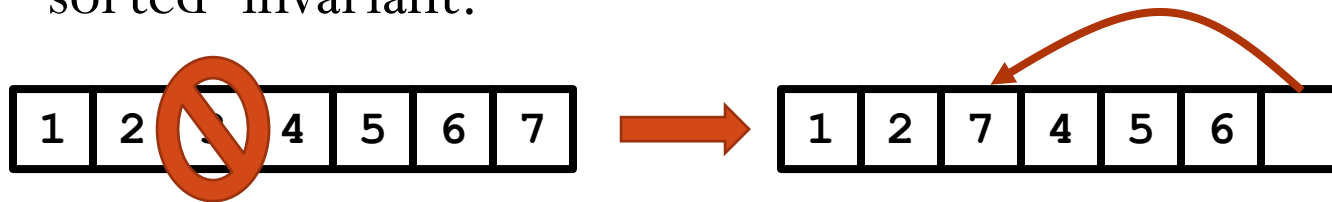
```
bool IntSet::query(int v)    {  
    return (indexOf(v) != MAXELTS);  
}
```

- `indexOf` also doesn't need to change.
- However, `insert` and `remove` do need to change.

# Abstract Data Types

## Improving Efficiency

- We'll start with the easiest one: `remove`.
- Recall the old version that moved the last element from the end to somewhere in the middle, this will break the new “sorted” invariant.



- Instead of doing a swap, we have to "squish" the array together to cover up the hole.





# Abstract Data Types

## Improving Efficiency

- How are we going to do the “squish”?
  - Move the element next to the hole to the left leaving a new hole.
  - Keep moving elements until the hole is “off the end” of the elements.

1	2	4	6	7	
---	---	---	---	---	--

remove 4

1	2		6	7	
---	---	--	---	---	--

1	2	6		7	
---	---	---	--	---	--

1	2	6	7		
---	---	---	---	--	--

- We'll reuse the variable `victim` as a loop variable.
- `victim`'s invariant is that it always points at the hole in the array.

# Abstract Data Types

## Improving Efficiency

```
void IntSet::remove(int v) {  
    int victim = indexOf(v);  
    if (victim != MAXELTS) {  
        // victim points at hole  
        numElts--; // one less element  
        while (victim < numElts) {  
            // ..hole still in the array  
            elts[victim] = elts[victim+1];  
            victim++;  
        }  
    }  
}
```

# Abstract Data Types

## Improving Efficiency

- We also have to change `insert` since it currently just places the new element at the end of the array. This will also break the new “sorted” invariant.



# Abstract Data Types

## Improving Efficiency

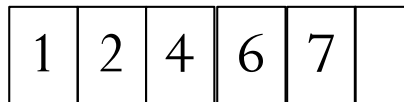
- How are we going to do the insert?
  - Start by moving the last element to the right by one position.
  - Repeat this process until the correct location is found to insert the new element.
  - Stop if the start of the array is reached or the element is sorted.
  - We'll need a new loop variable called `candidate` to track this movement.
  - Its invariant is that *it always points to the next element that might have to move to the right.*

# Abstract Data Types

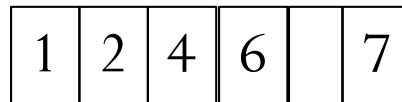
## Improving Efficiency

```
void IntSet::insert(int v) {  
    if (indexOf(v) == MAXELTS) { // duplicate not found  
        if (numElts == MAXELTS) throw MAXELTS; // no room  
        int cand = numElts-1; // last element  
        while ((cand >= 0) && elts[cand] > v) {  
            elts[cand+1] = elts[cand];  
            cand--;  
        }  
        // Now, cand points to the left of the "gap".  
        elts[cand+1] = v;  
        numElts++; // repair invariant  
    }  
}
```

**insert 5**



↑  
cand



↑  
cand



↑  
cand

# Abstract Data Types

## Improving Efficiency

```
void IntSet::insert(int v) {  
    if (indexOf(v) == MAXELTS) { // duplicate not found  
        if (numElts == MAXELTS) throw MAXELTS; // no room  
        int cand = numElts-1; // last element  
        while ((cand >= 0) && elts[cand] > v) {  
            elts[cand+1] = elts[cand];  
            cand--;  
        }  
        // Now, cand points to the  
        elts[cand+1] = v;  
        numElts++; // repair invariant  
    }  
}
```

Note: We are using the "short-circuit" property of &&. If cand is not greater than or equal to zero, we never evaluate the right-hand clause.

# Abstract Data Types

## Improving Efficiency

```
void IntSet::insert(int v) {  
    if (indexOf(v) == MAXELTS) { // duplicate not found  
        if (numElts == MAXELTS) throw MAXELTS; // no room  
        int cand = numElts-1; // largest (last) element  
        while ((cand >= 0) && elts[cand] > v) {  
            elts[cand+1] = elts[cand];  
            cand--;  
        }  
        // Now, cand points to the left of the "gap".  
        elts[cand+1] = v;  
        numElts++; // repair invariant  
    }  
}
```

Question: What is the situation when the loop terminates due to `cand < 0`? Is our implementation correct?

# Abstract Data Types

## Improving Efficiency

- **Question**: Do we have to change `indexOf`?

```
int IntSet::indexOf(int v) {  
    for (int i = 0; i < numElts; i++) {  
        if (elts[i] == v) return i;  
    }  
    return MAXELTS;  
}
```



# Abstract Data Types

## Improving Efficiency

- **Question**: Do we have to change `indexOf`?
- **Answer**: No, but it can be made more efficient with the new representation.
- **How?** Using **binary search**! (The array is sorted)

```
int IntSet::indexOf(int v) {  
    for (int i = 0; i < numElts; i++) {  
        if (elts[i] == v) return i;  
    }  
    return MAXELTS;  
}
```

# Abstract Data Types

Complexity

	<u>Unsorted</u>	<u>Sorted</u>
query	$O(N)$	?
insert	?	?
remove	?	?

# Abstract Data Types

## Complexity

	<u>Unsorted</u>	<u>Sorted</u>
query	$O(N)$	$O(\log N)$
insert	$O(N)$	$O(N)$
remove	$O(N)$	$O(N)$

insert and remove are still **linear**, because they may have to "swap" an element to the beginning/end of the array.

# Abstract Data Types

## Complexity

	<u>Unsorted</u>	<u>Sorted</u>
<code>query</code>	$O(N)$	$O(\log N)$
<code>insert</code>	$O(N)$	$O(N)$
<code>remove</code>	$O(N)$	$O(N)$

- If you are going to do more searching than inserting/removing, you should use the "sorted array" version, because `query` is faster there.
- However, if `query` is relatively rare, you may as well use the "unsorted" version. It's "about the same as" the sorted version for `insert` and `remove`, but it's MUCH simpler!

# References

- **Problem Solving with C++ (8<sup>th</sup> Edition)**
  - Chapter 10.3 **Abstract Data Types**
  - Chapter 10.2 **Classes and constructors**
- **C++ Primer, 4<sup>th</sup> Edition**
  - Chapter 7.7.1 **const Member Function**