Ve 280

Programming and Introductory Data Structures

Abstract Data Type

Outline

- Class in C++: A Trivial Example
- More Details on Class
- Another Class Example: a **Mutable** Set of Integers

Review: Abstract Data Types

- ADT provides an **abstract description** of **values** and **operations**.
- Advantages: Information hiding and encapsulation
- Realized by class in C++
- The basic idea behind a class is to provide **a single entity** that both defines:
  - The **nature** of an object.
  - The **operations** available on that object. These operations are sometimes also called **member functions** or **methods**.

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

- Class in C++: A Trivial Example
- More Details on Class
- Another Class Example: a **Mutable** Set of Integers

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 just 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;
}
```

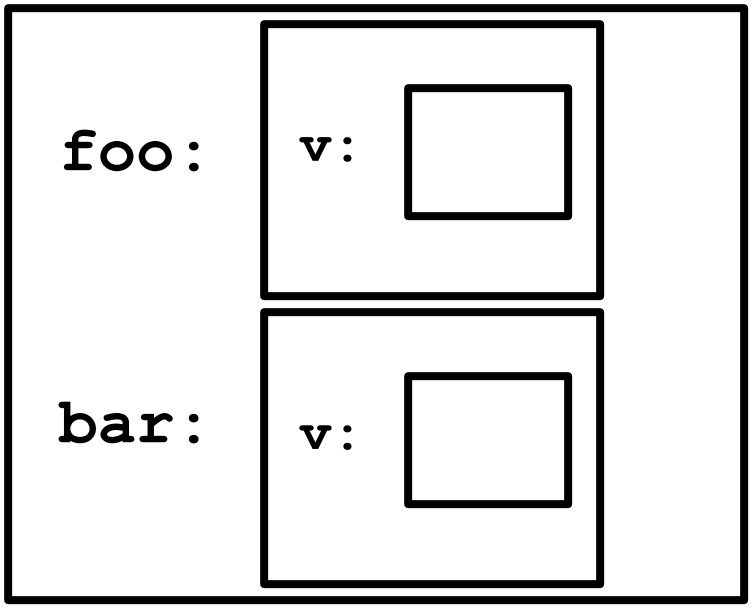
Abstract Data Types

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.

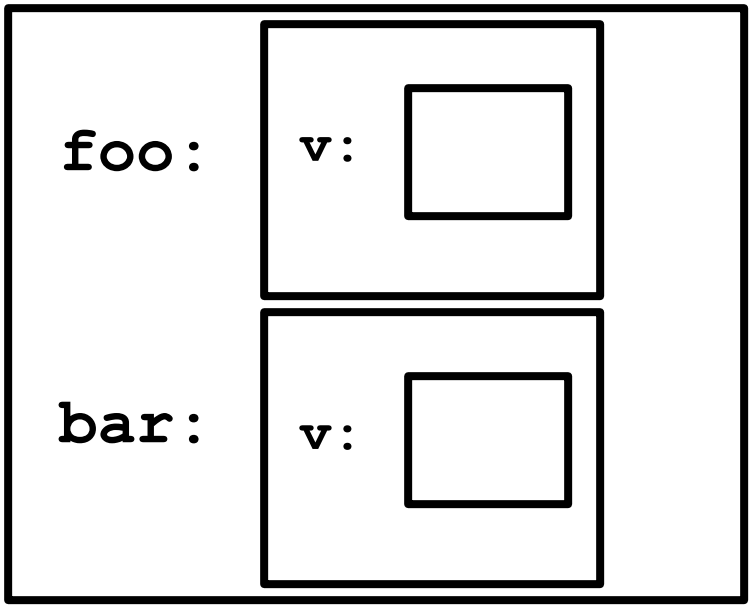
Abstract Data Types

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()`

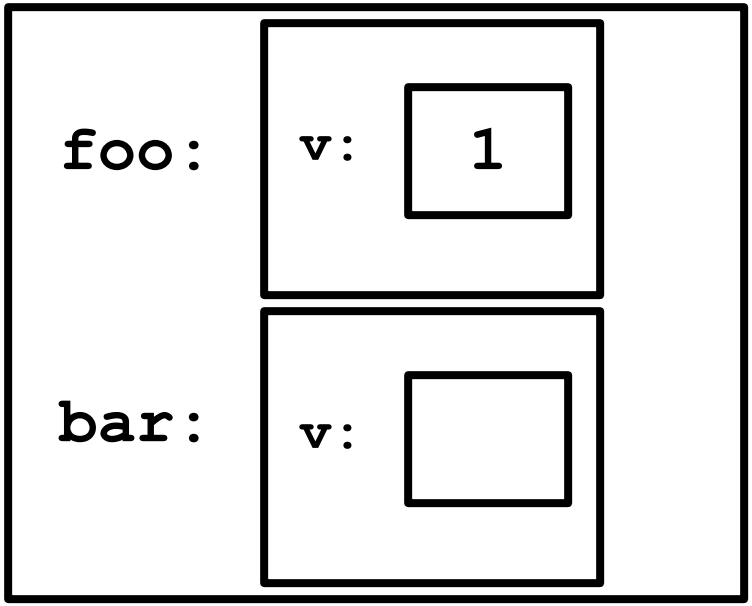
```
void anInt::set_value(int newValue) {
  v = newValue;
}
```

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!

Outline

- Class in C++: A Trivial Example
- More Details on Class
- Another Class Example: a **Mutable** Set of Integers

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.

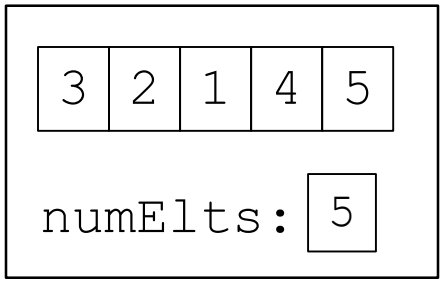
Abstract Data Types

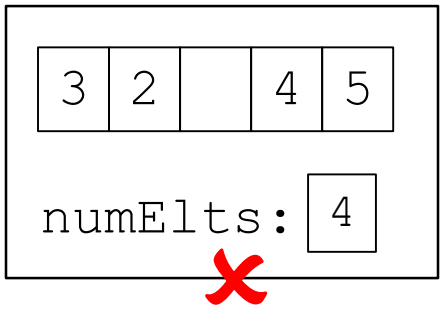
Using Classes

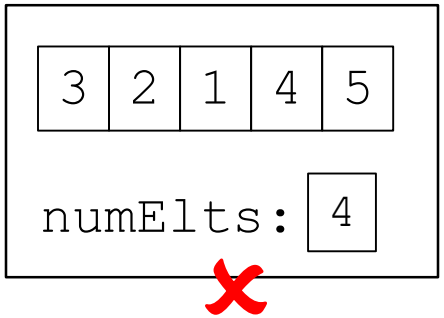
- Start with a representation for the set itself:
  - Use an array.
  - Represent a set of size N as an **unordered** array of integers with no duplicates, stored in the first N slots of the array.
  - `int numElts`: maintains the number of elements currently in the array.

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 both 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
```

100 is arbitrary. What is a better way to specify such size?

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);
}
```

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;
  }
}
```

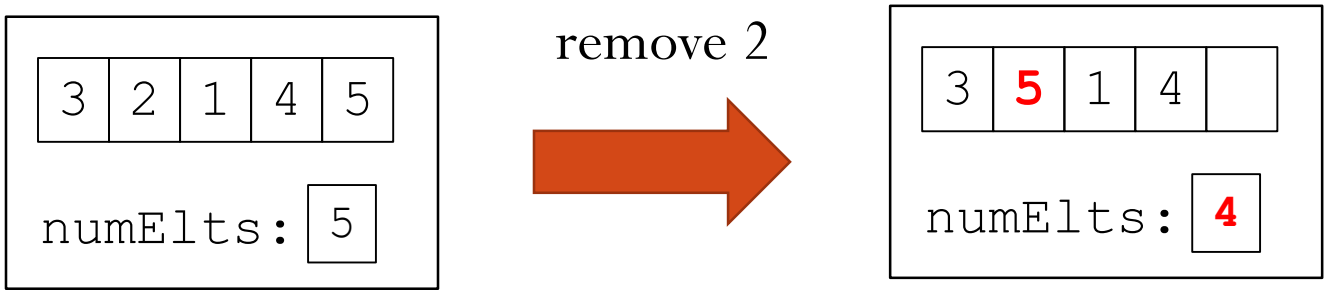
How about Remove?

- If the element (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 again breaks the invariant on `numElts`, 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 `numElts` 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;
}
```

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 {...}
```

References

- Constructor
  - Problem Solving with C++, 8th Edition, Chapter 10.2

- const Member Function
  - C++ Primer, 4th Edition, Chapter 7.7.1