Ve 280

Programming and Introductory Data Structures

Virtual Functions; Interfaces; Invariants

Announcement

- Programming Project Four posted
  - On abstract data types (ADTs), inheritance, and interfaces (abstract base classes)
  - Due time: 11:59 pm, July 17th

Outline

- Virtual Functions
- Interfaces
- Representation Invariants

Review

- Subtype: satisfy the substitution principle

- Creating subtype

1. Add one or more operations.
2. **Strengthen** the **postcondition** of one or more operations.
3. **Weaken** the **precondition** of one or more operations.

Review: Subclasses

- It is possible to create **subclasses** that are **NOT** **subtypes** and don't follow the substitution principle.

```
class PosIntSet : public IntSet {
  // OVERVIEW: a mutable set of positive integers
public:
  void insert(int v);
    // EFFECTS: if v is non-negative,
    //          and s has room to include it,
    //          s = s + {v}.
    //          if v is negative throw int -1
    //          if s is full thrown int MAXELTS
};
void PosIntSet::insert(int v) {
  if (v < 0) throw -1;
  IntSet::insert(v);
}
```

PosIntSet is not a subtype!

Review: Subclasses

- Unfortunately, the rules of C++ allow a **subclass** to be used wherever a **superclass** is expected.
- For example, the following code is perfectly legal:

```
PosIntSet s;
IntSet* p = &s;
IntSet& r = s;
```

- **Apparent type**: the declared type of the reference. (`IntSet`)
- **Actual type**: the real type of the referent. (`PosIntSet`)

Review: Subclasses

```
PosIntSet s;
IntSet* p = &s;
IntSet& r = s;
```

```
try {
  s.insert(-1);
} catch (int i) {
  cout << "Exception thrown\n";
}
```

This will do exactly what you expect: "Exception thrown\n".

```
try {
  r.insert(-1);
} catch (int i) {
  cout << "Except
}
```

Question: What will this do?

Answer: Because in default situation, C++ chooses the method to run based on its apparent type, this code calls `IntSet::insert()`, which happily inserts `-1`.

Virtual Functions

- There is a way to tell C++ to choose the actual type.
- In the class definition, we add the keyword **`virtual`** to the declaration of `insert`:

```
class IntSet {
    ...
public:
  ...
 virtual void insert(int v);
    ...
};
```

- This tells the compiler "someone might override my implementation: always check at run-time to see which version to call."

Virtual Functions

- You don't have to add `virtual` keyword when you **define** the function, i.e., the following is OK

```
void IntSet::insert(int v) { // OK
    ...
}
```

- You don't have to add the `virtual` keyword to `PosIntSet`'s definition, since "virtualness" is inherited just like everything else

```
class PosIntSet: public IntSet {
    ...
public:
    void insert(int v); // OK
    ...
};
```

Virtual Functions

- Now consider:

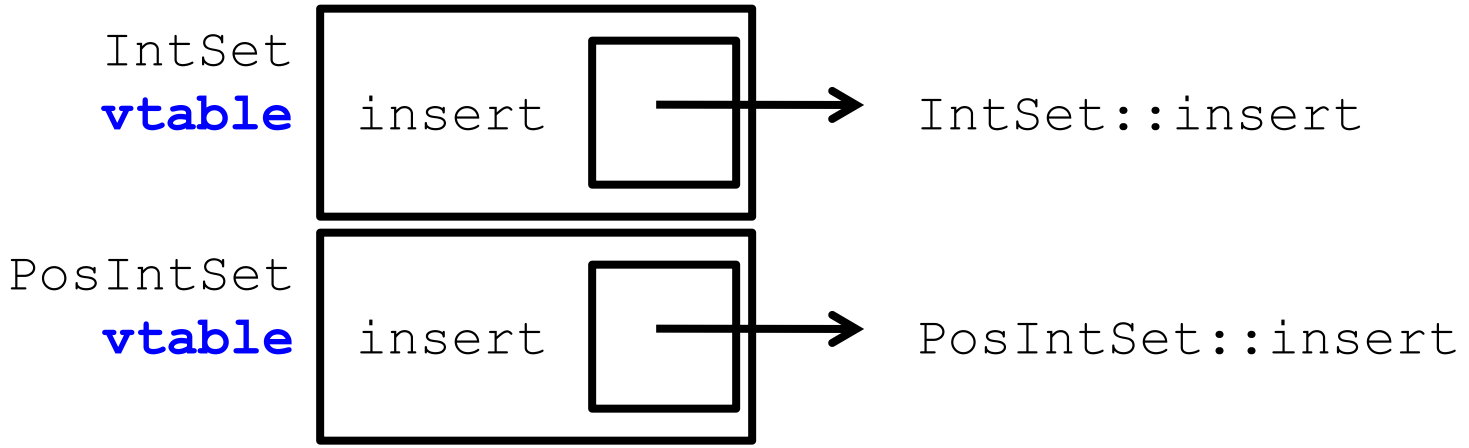
```
posIntSet s;
IntSet* p = &s;
IntSet& r = s;

p->insert(-1);
```

- `p` is declared as a pointer-to-IntSet, but it might really be pointing at some **derived class** type.
- The compiler will create code that checks the actual type of the object and calls the **right** function **at runtime**.

Virtual Functions

- Classes with virtual functions include information that allows you to figure out what type it is.
  - First, for each class with virtual functions, the compiler creates a **vtable** (or **virtual table**) with one function pointer for each virtual function initialized to the appropriate implementation.
  - Then, each instance of a class **with virtual methods** has both the class' state, **plus** a pointer to the appropriate `vtable`.



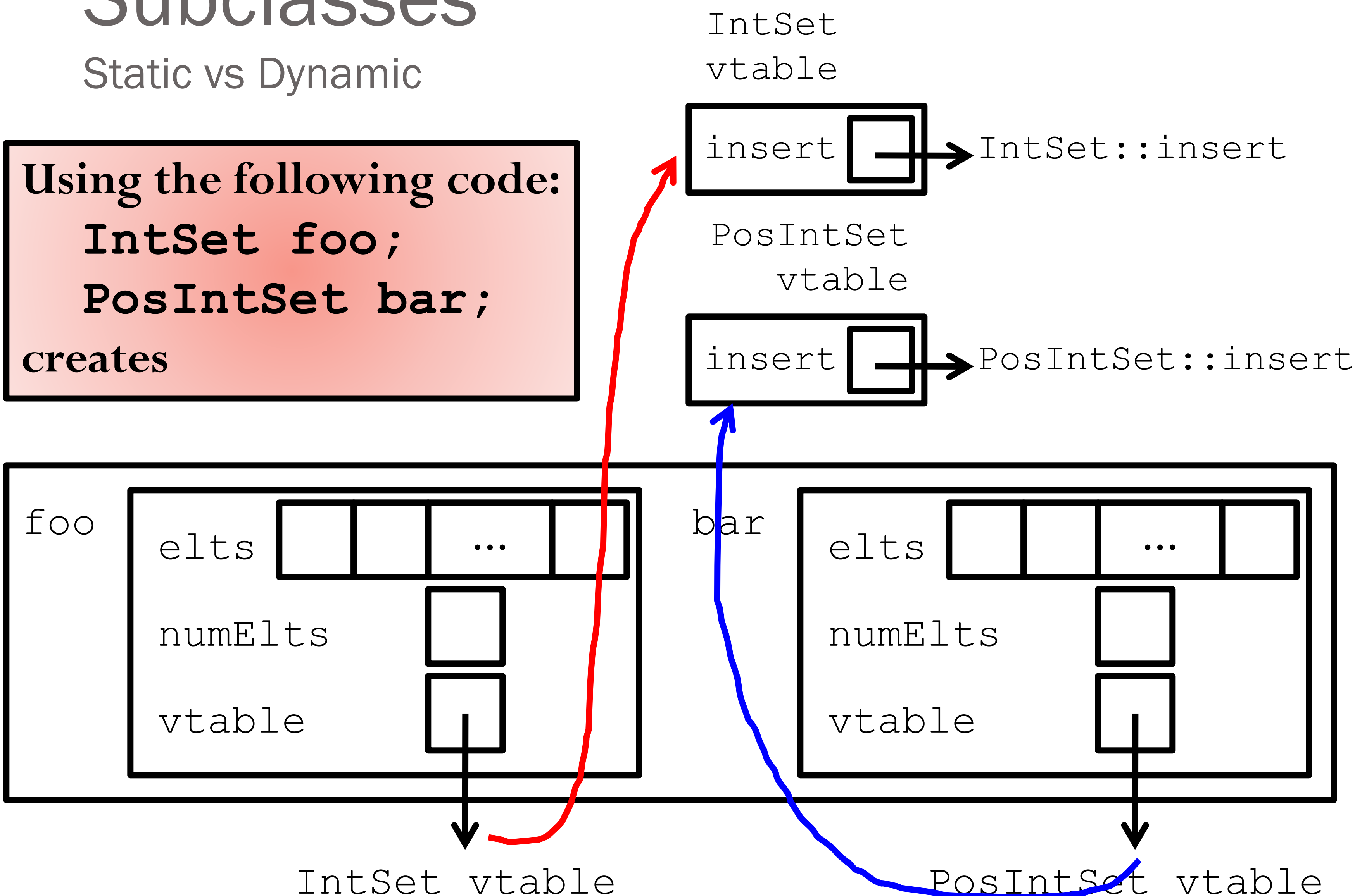
Subclasses

Static vs Dynamic

Using the following code:

```
IntSet foo;
PosIntSet bar;
```

creates



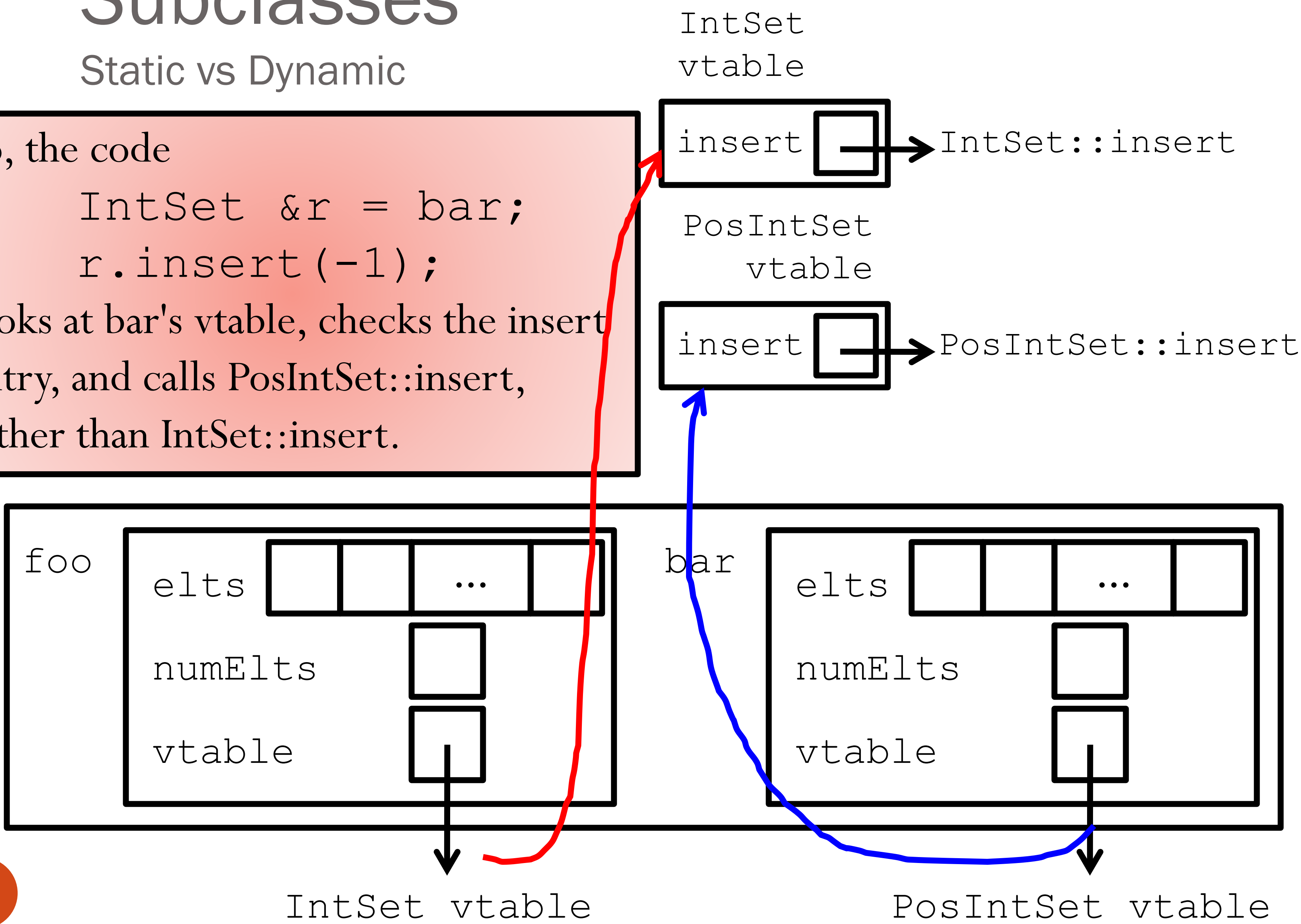
Subclasses

Static vs Dynamic

So, the code

```
IntSet &r = bar;
r.insert(-1);
```

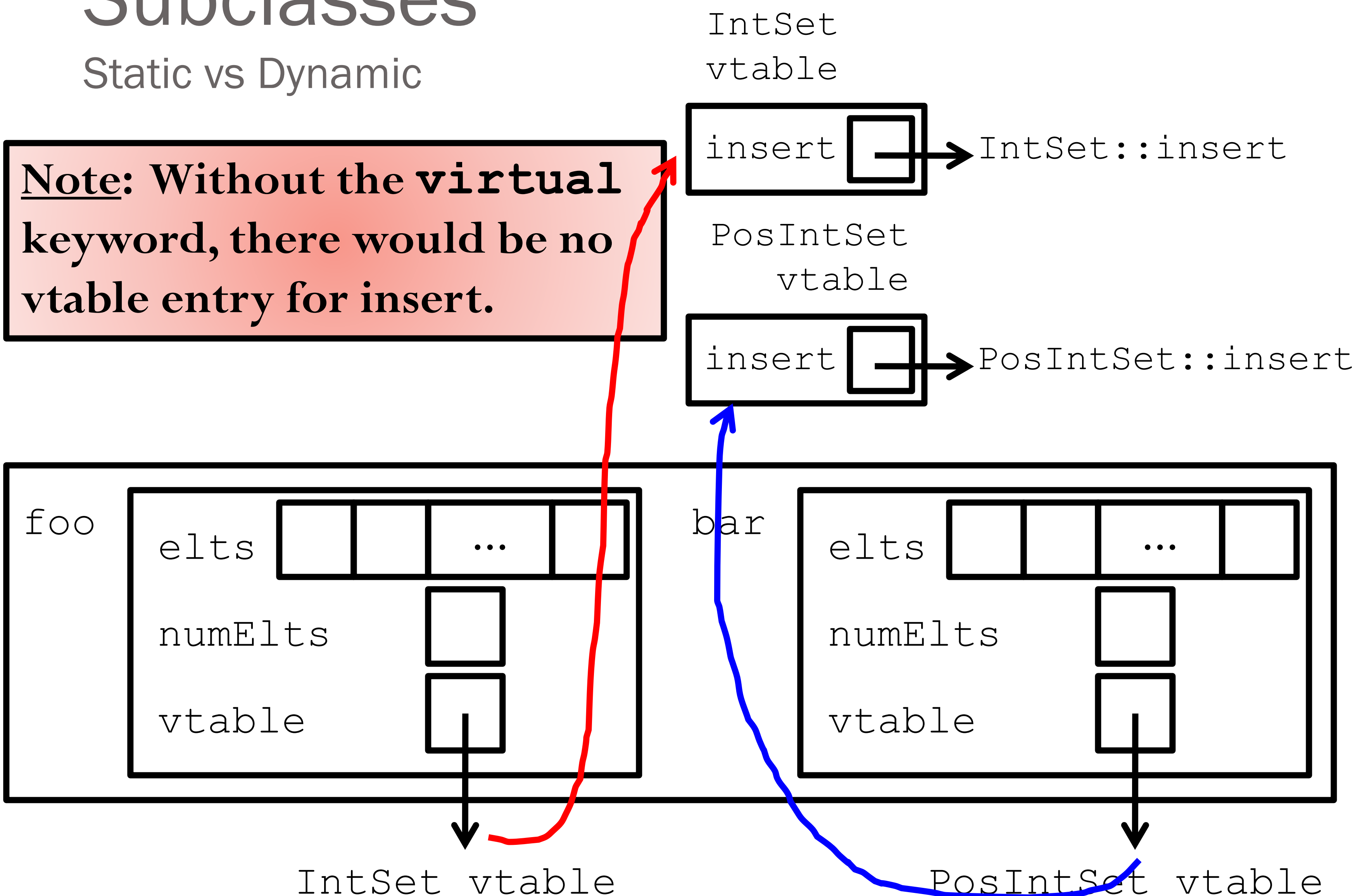
looks at bar's vtable, checks the insert entry, and calls PosIntSet::insert, rather than IntSet::insert.



Subclasses

Static vs Dynamic

<u>Note</u>: Without the `virtual` keyword, there would be no vtable entry for insert.



Virtual Function Exercises

- Suppose we have

```
Foo *fp;
Bar bar;
Baz baz;

baz.f();
fp=&baz;
fp->f();
fp->g();
```

- What's the output?

```
class Foo {
public:
  void f()
  { cout<<"Foo::f"<<endl; }
  virtual void g()
  { cout<<"Foo::g"<<endl; }
};
class Bar: public Foo {
public:
  void g()
  { cout<<"Bar::g"<<endl; }
};
class Baz: public Bar{
public:
  void f()
  { cout<<"Baz::f"<<endl; }
};
```

Outline

- Virtual Functions
- Interfaces
- Representation 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 two separate files.
  - 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 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 class **cannot** have an 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 declaring to not 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: 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!

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 prototype for 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 prototype for 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  in
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**…
- …but we don't know how to do that yet.
- We will know it soon!

Outline

- Virtual Functions
- Interfaces
- Representation 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 Invariant

- 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

References

- **Problem Solving with C++ (8th 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++