Ve 280

Programming and Introductory Data Structures

Subtypes; Inheritance; Virtual Functions

Learning Objectives:

Understand what is a subtype and why they are useful

Know how to create subtype via inheritance

Understand what is a virtual function and know how to use it

Outline

- Introduction to Subtypes
- Creating Subtypes
- Creating Subtypes using C++ Inheritance Mechanism
- Virtual Functions

Subtypes

Introduction

- Suppose we have two types, S and T.
- S is a **subtype** of T, written "S <: T", if:
  - For any instance where an object of type T is expected, an object of type S can be supplied without changing the correctness of the **original** computation.
  - In other words, code written to correctly use T is still correct if it uses S.

- This is called the "substitution principle".
- If S <: T, then we also say that "T is a **supertype** of S".

Subtypes

Example

```
void add(istream &source) {
   int n1, n2;
   source >> n1 >> n2;
   cout << n1+n2;
}
add(inFile); //inFile has been declared
             //as an ifstream and opened
```

- The function call `add(inFile)` is valid and works because `ifstream` is a **subtype** of `istream`
- `ifstream` can be supplied (substituted for `istream`) without changing the correctness.

Subtypes

Introduction

- Subtypes are different from the notion of "type-convertible".
  - For example, in any computation that expects a `double`, you can use an `int`.
  - However, the object isn't an `int` when it is used.
  - It is first "converted" to a `double` and its physical representation changes!

- However, if you use a subtype where a supertype is expected, it is **not converted** to the supertype
- Instead, it is used as-is.

Benefits of Subtyping

- Code reuse

```
void add(istream &source) {
  int n1, n2;
  source >> n1 >> n2;
  cout << n1+n2;
}
```

Outline

- Introduction to Subtypes
- Creating Subtypes
- Creating Subtypes using C++ Inheritance Mechanism
- Virtual Functions

Subtypes

Creating

- In an Abstract Data Type, there are three ways to create a subtype from a supertype:

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

Subtypes

Creating by Adding New Methods

- The first way of creating a subtype is to **add some new method** to the subtype.
- Any code using the original supertype expects only the "old" methods, which are still available.
- The "new" method makes no difference.

Subtypes

Creating by Adding New Methods

- Example: we can create a subtype of `IntSet`, called `MaxIntSet`, by adding an operation `max()` that returns the maximum element in the set. The other part remains the same as IntSet.
  - Any code using `IntSet` can have `IntSet` be replaced with `MaxIntSet`. It won't call the `max()` method of `MaxIntSet`!

```
void foo(IntSet& is)
{
   ...
}
```

```
void main()
{
   MaxIntSet ms;
   foo(ms);
   ...
}
```

Subtypes

Creating by Strengthening Postcondition

- Second method: **Strengthen** the **postcondition** of one or more operations.

- The postconditions of a method are formed by two things:
  - The EFFECTS clause
  - Its return type

- One way of strengthening the postcondition is to strengthen the EFFECTS clause by promising everything you used to, plus extra.

Subtypes

Creating by Strengthening Postcondition

```
int A::f(int arg);
  // REQUIRES: arg is positive
  // and even.
  // EFFECTS: returns arg/2.
```

- We can create a subtype of `A`, called `B`, by only changing function `f` of `A`.
- We strengthen the effects clause of `A::f()` by printing a message to the screen for each invocation, in addition to computing `arg/2`.

Subtypes

Creating by Strengthening Postcondition

- When `A::f()` is expected, we can also replace it with `B::f()`, since `B::f()` does all the things `A::f()` does with an extra message printing.
  - User's expectation is satisfied.
  - So B can substitute A.

```
int g(A& a)
{
   int arg = 2;
   return a.f(arg);
}
```

B::f(arg)

```
void main()
{
   B b;
   int c = g(b);
   ...
}
```

Subtypes
Creating

- Third method: **Weaken** the **precondition** of one or more operations.

- The preconditions of a method are formed by two things:
  - The REQUIRES clause
  - Its argument type

- One way of weakening the precondition is to weaken the REQUIRES clause.

Subtypes

Creating

```
int A::f(int arg);
  // REQUIRES: arg is positive and even.
  // EFFECTS: returns arg/2.
```

- You could weaken this REQUIRES clause by allowing:
  - Negative, even integers
  - Positive integers
  - All integers
  - …

Why Weaken REQUIRES Create Subtype?

```
int A::f(int arg);
  // REQUIRES: arg is positive and even.
  // EFFECTS: returns arg/2.
```

- We can create a subtype of A, called B, which allows **all integers** for the function f of A.

```
int g(A& a) {
  int arg = 2;
  return a.f(arg);
}
```

B::f(arg)

```
void main() {
  B b;
  int c = g(b);
}
```

✓

Why Weaken REQUIRES Create Subtype?

```
int A::f(int arg);
  // REQUIRES: arg is positive and even.
  // EFFECTS: returns arg/2.
```

- It is fine to call function `g` on an object `b` of subtype `B`.
  - When `A::f()` is expected, the argument (e.g., positive even integers) to `A::f()` is a subset of that to `B::f()` (e.g., integers). Then, that argument for `A::f()` works perfectly for `B::f()`!

```
int g(A& a){
  int arg = 2;
  return a.f(arg);
}
```

B::f(arg)

```
void main() {
  B b;
  int c = g(b);
}
```

Outline

- Introduction to Subtypes
- Creating Subtypes
- Creating Subtypes using C++ Inheritance Mechanism
- Virtual Functions

Subclasses

Creating Subclasses using inheritance

- C++ has a mechanism to enable subtyping, called "**subclassing**", or sometimes **inheritance**.

- So, if we have some ADT class `foo`, and want to make a subtype class `bar`, we do so by saying:

```
class bar : public foo {
    ...
};
```

- This says: "`bar` is a `foo`, possibly with extra state, and possibly with new or redefined member functions.“
- We say that `bar` is a **derived class**, and it is **derived from** `foo`.

Subclasses

Creating Subclasses using inheritance

```
class bar : public foo {
   ...
};
```

- `public` means **public inheritance**. All public members of the base are also public in the derived class; all private members of the base are also private in the derived class
- We can also have **private inheritance**

  ```
  class bar : private foo
  ```

  - Then, all members of the base class are **private** in the derived class
- We normally use **public inheritance**. All the previous public member functions are still public

Subclasses

Creating Subclasses using inheritance

```
class MaxIntSet : public IntSet {
  // OVERVIEW: a set of integers,
  //           where |set| <= 100
public:
  int max();
    // REQUIRES: set is non-empty
    // EFFECTS: returns largest element in
 set.
};
```

- This creates a new type that has all of the behavior of `IntSet`, **plus** one new operation.
- `MaxIntSet` **automatically inherits** all of the `IntSet` methods and data elements.

Subclasses

Creating Subclasses using inheritance

- Conceptually, we have

MaxIntSet Object

IntSet members: elts[], numElts, insert(), remove(),…
New member of MaxIntSet: max()

Subclasses

Creating Subclasses using inheritance

- Unfortunately, MaxIntSet::max() is **not** a member of IntSet.

```
class IntSet {
   // data members plus indexOf
public:
   // the public interface to the class.
};
```

- All the data members are (by default) private. This means "they can be seen **only** by other members of **this** class".
- All of the data members of `IntSet` are inaccessible to the additional members of the derived class `MaxIntSet`, and specifically `MaxIntSet::max()`.

Subclasses

Creating Subclasses using inheritance

- Thankfully, we still have access functions that are public and could write it this way:

```
int MaxIntSet::max() {
  int i;
  for (i=INT_MAX; i>=INT_MIN; i--) {
    if (query(i)) return i;
  }
}
```

- However, this function is **inefficient**!
- We'll have to query 2^{31} (i.e. ½ of 2^{32}) numbers on average to find the maximum element in a randomly-constructed set!

Subclasses

```
int MaxIntSet::max() {
  int i;
  for (i=INT_MAX; i>=INT_MIN; i--) {
    if (query(i)) return i;
  }
}
```

- C++ has a mechanism that allows us to get around this problem
  - The "**protected**" storage class.
- If a member is "**protected**", it means "can be seen by all members of this class and **any derived** classes".

Subclasses

```
class IntSet {
protected:
  // all of the data members plus indexOf
public:
  // the public interface to the class.
};
```

- Since `MaxIntSet` is derived from `IntSet`, the protected members of `IntSet` are visible to `MaxIntSet`.
- Other users of the IntSet class still **cannot** see the members.
- With this new structure, we can write `max` much more efficiently.

Subclasses

```
int MaxIntSet::max() {
  int so_far = elts[0];
  for (int i = 1; i<numElts; i++) {
    if (elts[i] > so_far)
      so_far = elts[i];
  }
  return so_far;
}
```

Unsorted Array

```
int MaxIntSet::max() {
return elts[numElts-1];
}
```

Sorted Array

Subclasses

Consequences of `protected`

- If we expose `IntSet`'s implementation to `MaxIntSet`, changing that implementation will break `MaxIntSet`.
- For example, if we switch `IntSet` from a sorted implementation to an unsorted one, `MaxIntSet::max()` will return the wrong value.
- Worse, it will compile correctly---you'll never know!

This shows the bad consequence of exposing detailed implementation

- **Protected** data members make derived classes extremely fragile, and it is a matter of taste as to whether it's worth doing.

Subclasses

- You can "extend" functionality in ways other than just adding methods – you can also **change** an individual method.
- In order to create **subtype**, you can't change it arbitrarily though; your subtype must still adhere to the **substitution principle**.
  - The new method must do everything the old method did, but it is allowed to do more as well. strengthen the postconditions
  - It must require no more of the caller than the old method did, but it can require less. weaken the preconditions

Subclasses

```
class SafeMaxIntSet : public MaxIntSet {
  // OVERVIEW: a mutable set of integers,
  //           where |set| <= 100
public:
  int max();
  // EFFECTS: if set is non-empty, returns largest
  //          element in set
  //          otherwise, returns INT_MIN.
};
```

- In this method, we've both **weakened** the preconditions AND **strengthened** the postconditions of "old" `max`.

Subclasses

```
class SafeMaxIntSet : public MaxIntSet {
  // OVERVIEW: a mutable set of integers,
  //            where |set| <= 100
public:
  int max();
  // EFFECTS: if set is non-empty, returns largest
  //           element in set
  //           otherwise, returns INT_MIN.
};
```

- Preconditions: Old `max` required the set to be non-empty, new `max` doesn't.
- Postconditions: Old `max` returned the largest element of a non-empty set, new max does that, **plus** it returns `INT_MIN` for an empty set.

Subclasses

This new subtype correctly satisfies the **substitution principle**: Code that was correctly written to use a `MaxIntSet` will work unchanged if using a `SafeMaxIntSet`.

```
void foo()
{
   MaxIntSet a;
   a.insert(1);
   ...
   cout << a.max();
   ...
}
```

```
void foo()
{
   SafeMaxIntSet a;
   a.insert(1);
   ...
   cout << a.max();
   ...
}
```

✓

Subclasses

```
class SafeMaxIntSet : public MaxIntSet {
  // OVERVIEW: a mutable set of integers,
  //            where |set| <= 100
public:
  int max();
  // EFFECTS: if set is non-empty, returns largest
  //           element in set
  //           otherwise, returns INT_MIN.
};
```

- This defines a new class that is exactly like a `MaxIntSet`, except that it **replaces** or "**overrides**" the method `max`.
- The compiler decides which `max` to call by the type of the object to which we call `max`.

Subclasses

- So, if we declared one object of each type, calling the `max` method would give us the "right" one:

```
MaxIntSet     ms;
SafeMaxIntSet ss;

ss.max(); // calls SafeMaxIntSet::max()
          // returns INT_MIN on empty set
ms.max(); // calls MaxIntSet::max()
          // is undefined on empty set
```

Subclasses

- The implementation of this new `max` is surprisingly simple:

```
int SafeMaxIntSet::max() {
  if (size())
    return MaxIntSet::max();
  else
    return INT_MIN;
}
```

- Most of the hard work is done by the "old" implementation of `max` (called `MaxIntSet::max`).
- This just covers the case that the set is empty.

Select All Correct Statements

- **A.** An ADT can be implemented as a class in C++.
- **B.** A class in C++ is always an ADT.
- **C.** A subtype can be implemented as a subclass in C++.
- **D.** A subclass in C++ is always an ADT subtype.

Outline

- Introduction to Subtypes
- Creating Subtypes
- Creating Subtypes using C++ Inheritance Mechanism
- Virtual Functions

Subclasses

Leaving behind subtypes

- Finally, 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 positive
    //          and s has room to include it,
    //          s = s + {v}.
    //          if v <= 0, throw int -1
    //          if s is full, thrown int MAXELTS
};
void PosIntSet::insert(int v) {
  if (v <= 0) throw -1;
  IntSet::insert(v);
}
```

Subclasses

Leaving behind subtypes

- Finally, 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 positiv
    //          and s has room
    //          s = s + {v}.
    //          if v <= 0, thro
    //          if s is full, t
};
void PosIntSet::insert(int v) {
  if (v <= 0) throw -1;
  IntSet::insert(v);
}
```

Why is PosIntSet not a subtype?

Because code that is correctly written to use an `IntSet` could fail when using a `PosIntSet`, e.g., when inserting a negative number. It does not pass the substitution principle!

Subclasses

Leaving behind subtypes

- 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;
```

- Because `PosIntSet` is a subclass of `IntSet`, it is perfectly legal to make these assignments.
- We have three variables: `s` is a `PosIntSet`, `p` is a pointer that points to precisely this `PosIntSet`, and `r` is a reference to this `PosIntSet`.

Subclasses

Leaving behind subtypes

```
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 << "Exception thrown\n";
}
```

<u>Question</u>: What will this do?

Subclasses

Leaving behind subtypes

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

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

- The type of `r` is declared to be "reference to an IntSet", but it refers to a `PosIntSet`.
- **Apparent type**: the declared type of the reference. (`IntSet`)
- **Actual type**: the real type of the referent. (`PosIntSet`)
  - In this example, the **apparent type** and the **actual type** differ.
- In default situation, C++ chooses the method to run based on its apparent type.

Subclasses

Leaving behind subtypes

```
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
}
```

Answer: Because `r`'s apparent type is "reference-to-IntSet", this code calls `IntSet::insert()`, which happily inserts `-1`.
The same thing happens if we use the pointer `p`.

Subclasses

Leaving behind subtypes

```
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 << "Exception thrown\n";
}
```

This breaks the abstraction of the set `s`, which is **Very Bad**.

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:

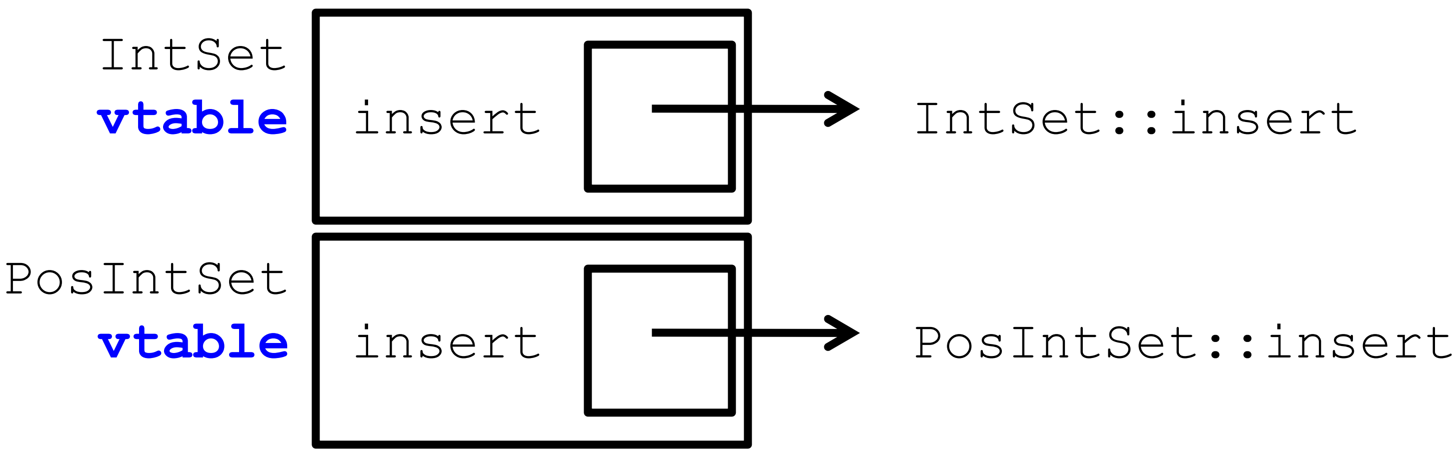
```
posIntSet s;
IntSet* p = &s;
IntSet& r = s;

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`.



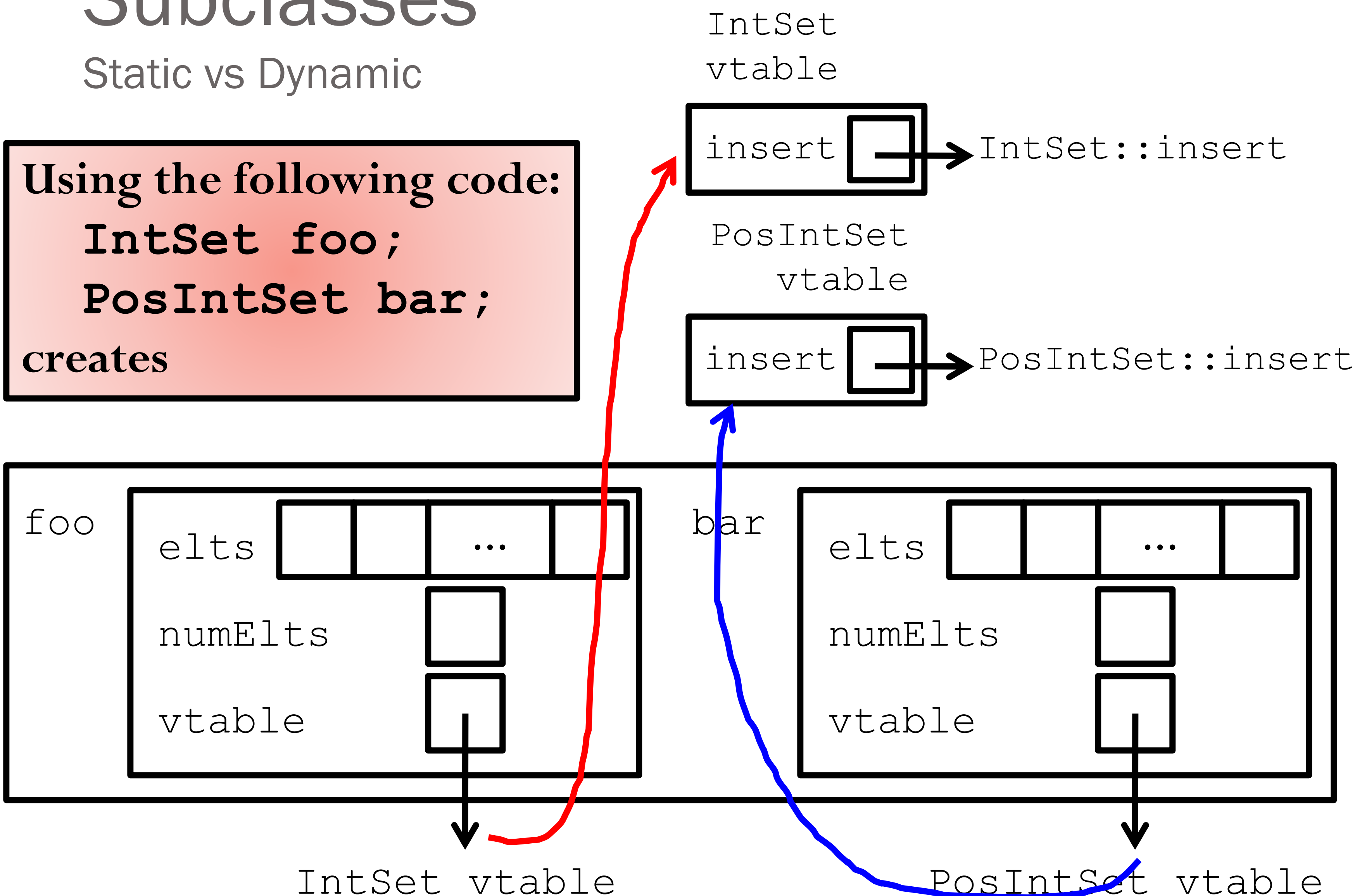
Subclasses

Static vs Dynamic

Using the following code:

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

creates



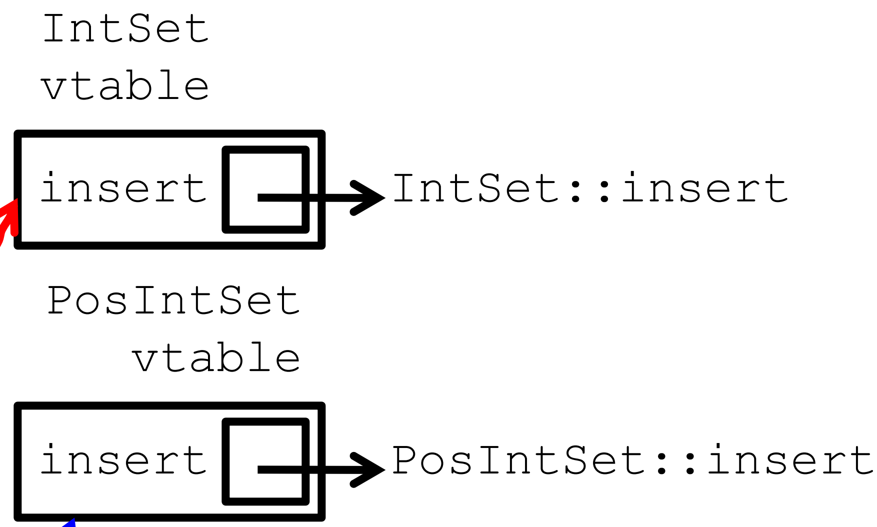
Subclasses

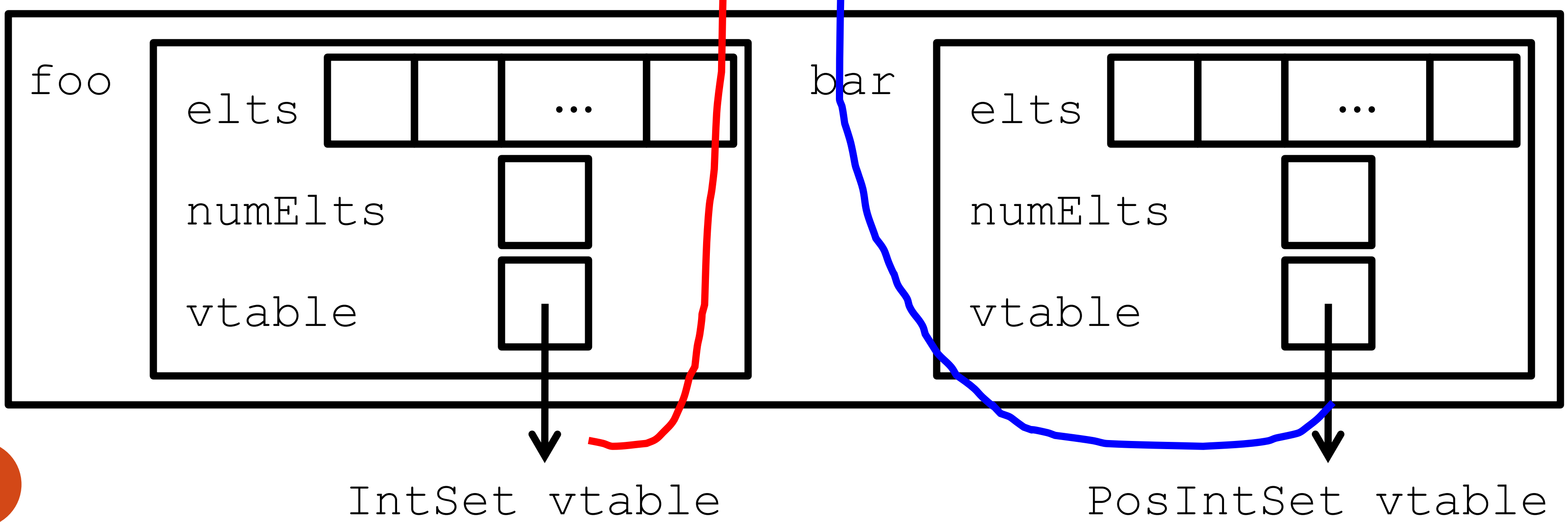
Static vs Dynamic

So, the code

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

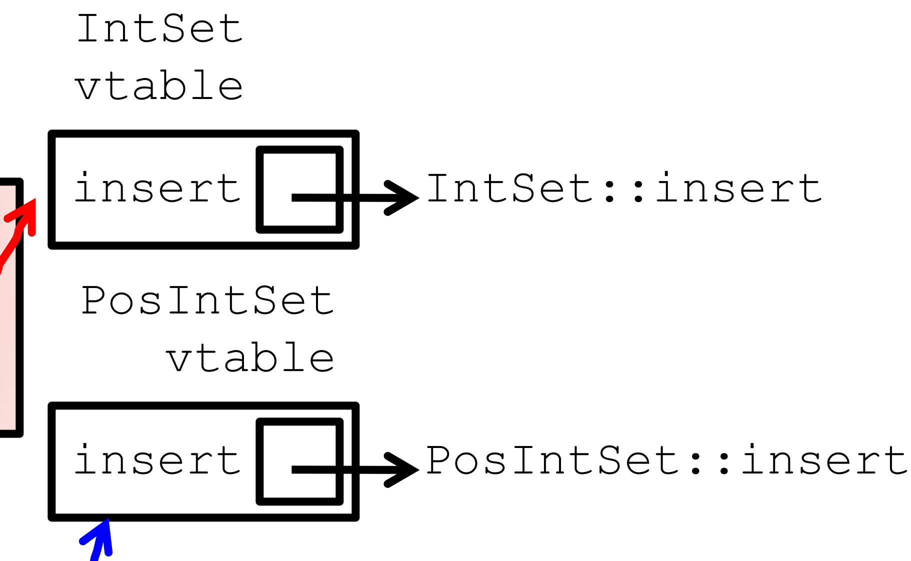
looks at bar's vtable, checks the insert entry, and calls PosIntSet::insert, rather than IntSet::insert.



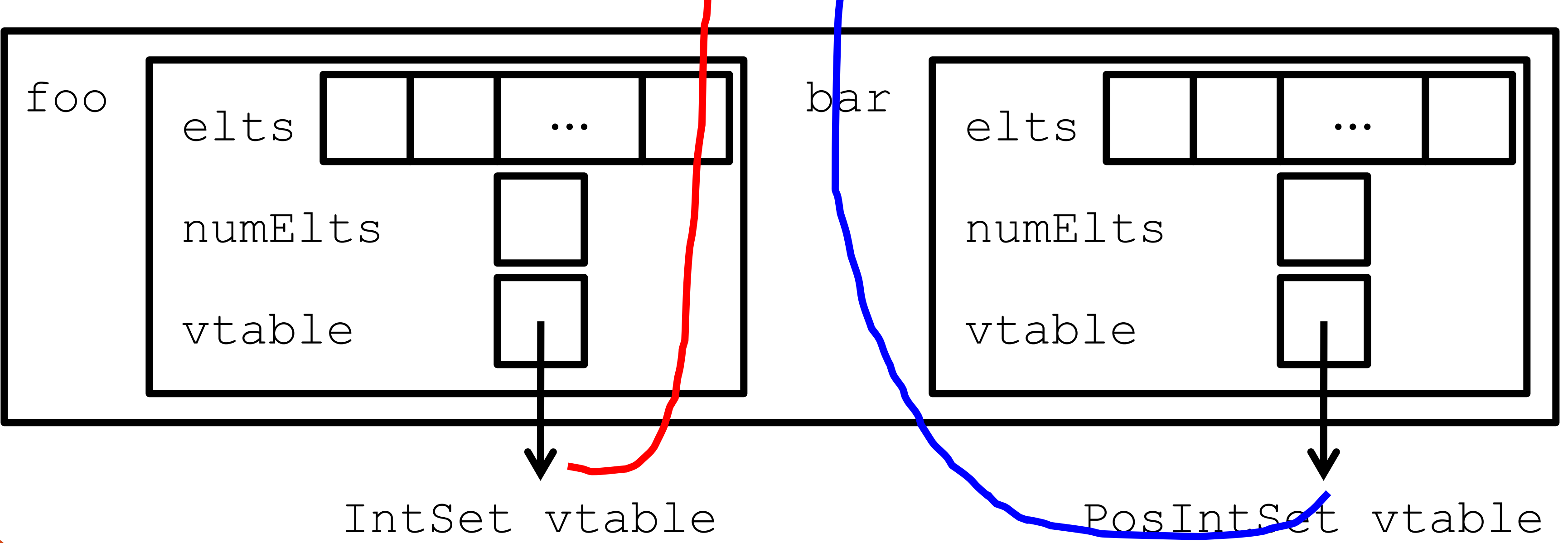


Subclasses

Static vs Dynamic



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



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++