More on Inheritance

Outline

- How to use private and protected variables
- How to use private and protected class derivations
- How to use call-by-reference class parameters
- How to call destructors in class hierarchy

Box Class w/ Private Variables & Public Readers

- Move `height,width,length` to private with public readers

```
class box : public container {
  public:  box () { }
           box (double h, double w, double l) {
               height = h; width = w; length = l;
           }
           double read_height () {return height;}
           double read_width () {return width;}
           double read_length () {return length;}
           double volume () {return height * width * length;}
  private: double height, width, length;
};
class box_car : public railroad_car, public box {
  public: // Default constructor:
          box_car () : box (10.5, 9.2, 40.0) { }
          // Displayers:
          virtual void display_short_name () {cout << "box";}
          virtual void display_capacity () {cout << volume ();}
};
```


Box Class w/ Protected Member Variables

- Alternatively, move them into protected part of class
  - Now, they are accessible in the same class or subclasses

```
class box : public container {
  public:  box () { }
           box (double h, double w, double l) {
               height = h; width = w; length = l;
           }
           double volume () {return height * width * length;}
  protected: double height, width, length;
};
class box_car : public railroad_car, public box {
  public: // Default constructor:
          box_car () : box (10.5, 9.2, 40.0) { }
          // Displayers:
          virtual void display_short_name () {cout << "box";}
          virtual void display_capacity () {cout << volume ();}
          virtual void display_height () {cout << height;}
};
```


Box Class w/ Protected Readers & Private Variables

- If you do not want them to be modifiable outside of box
  - But they still can be accessible in box_car class
  - They are not accessible at all outside of box and box_car

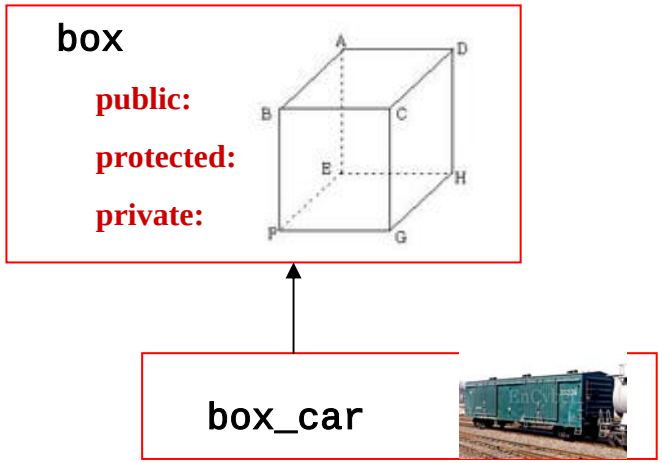
```
class box : public container {
  public:   box () { }
             box (double h, double w, double l) {
                 height = h; width = w; length = l;
            }
            double volume () {return height * width * length;}

 protected: double read_height () {return height;}
            double read_width () {return width;}
            double read_length () {return length;}

private:    double height, width, length;
};
```


Box Class's Protected, Private, Public Members

- Box's private member variables and functions
  - Available only to member functions defined in box
- Box's protected member variables and functions
  - Available only to member functions defined in box, box_car
- Box's public member variables and functions
  - Available to ordinary and member functions everywhere



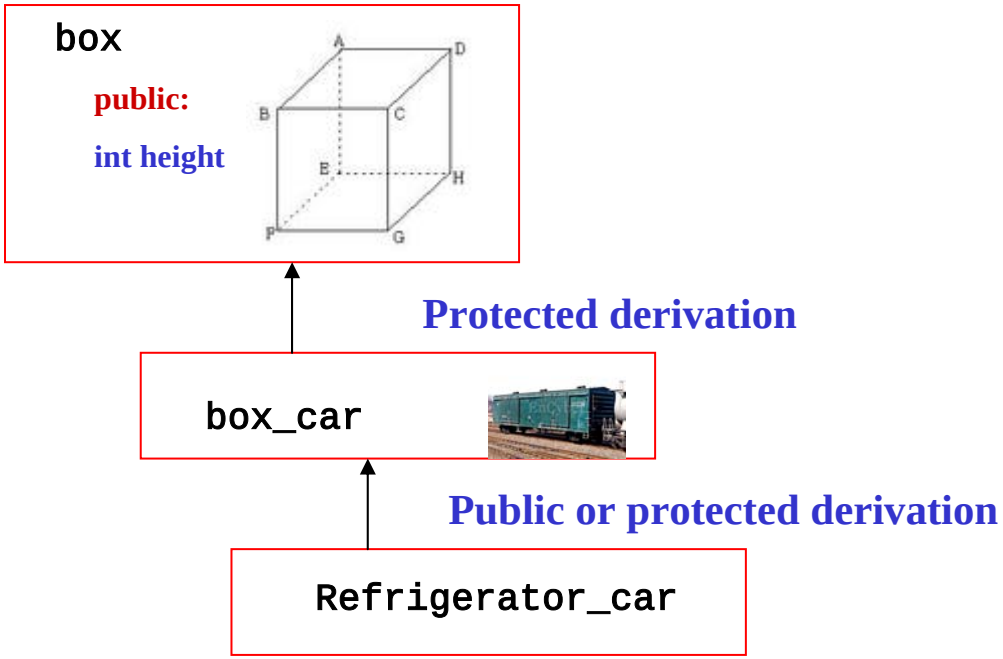
Protected Derivation

- Protected derivation of `box_car` from `box`
  - All public member variables and functions in `box` act as if they are protected member functions and variables in `box_car`

```
class box : public container {
  public:  box () { }
           box (double h, double w, double l) {
               height = h; width = w; length = l;
           }
           double volume () {return height * width * length;}
           double height, width, length;
};
class box_car : public railroad_car, protected box {
  public: // Default constructor:
          box_car () : box (10.5, 9.2, 40.0) { }
          // Displayers:
          virtual void display_short_name () {cout << "box";}
          virtual void display_capacity () {cout << volume ();}
          virtual void display_height () {cout << height;}
};
```


Effect of Protected Derivation

- If we have `refrigerator_car`, a public subclass of `box_car`
  - It can access the `height` variable defined in `box`, but with a `box_car` or a `refrigerator_car` object, cannot access height
  - `box_car x;`
    `cout << x.height;// Error!`
  - `refrigerator_car y;`
    `cout << y.height;// Error!`
  - `box z;`
  - `cout << z.height;// OK!`



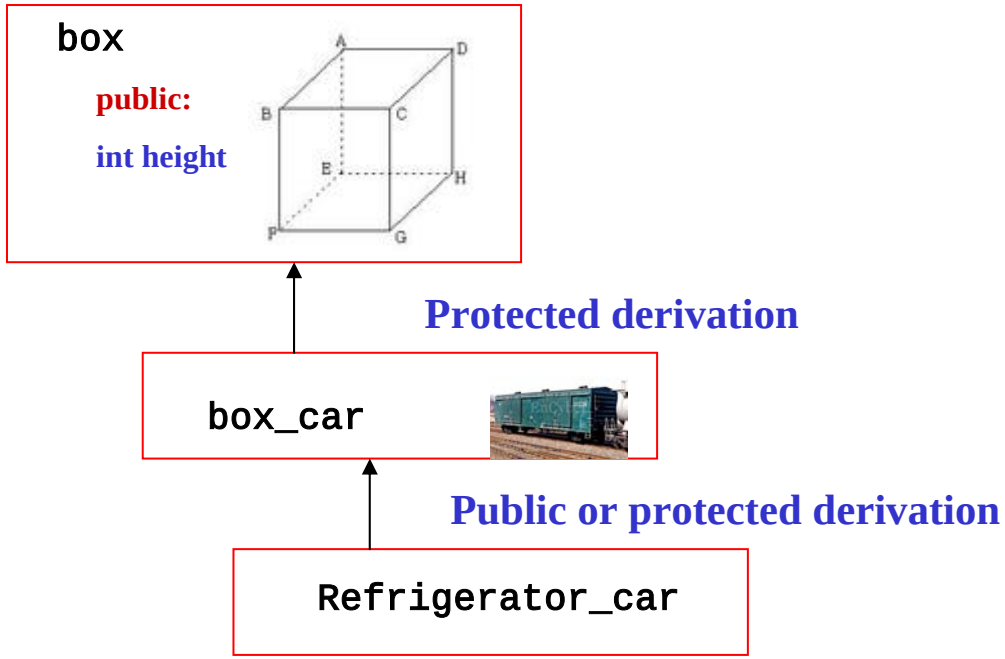
Private Derivation

- Private derivation of `box_car` from `box`
  - All public and protected member variables and functions in `box` act as if they are private member functions and variables in `box_car`

```
class box : public container {
  public:  box () { }
           box (double h, double w, double l) {
               height = h; width = w; length = l;
           }
           double volume () {return height * width * length;}
           double height, width, length;
};
class box_car : public railroad_car, private box {
  public: // Default constructor:
          box_car () : box (10.5, 9.2, 40.0) { }
          // Displayers:
          virtual void display_short_name () {cout << "box";}
          virtual void display_capacity () {cout << volume ();}
          virtual void display_height () {cout << height;}
};
```


Effect of Private Derivation

- All public members in `box` are accessible only in `box_car`
  - Not in any member functions of `box_car`'s subclasses
- If we have `refrigerator_car`, a public subclass of `box_car`
  - It cannot access the `height` variable defined in `box`



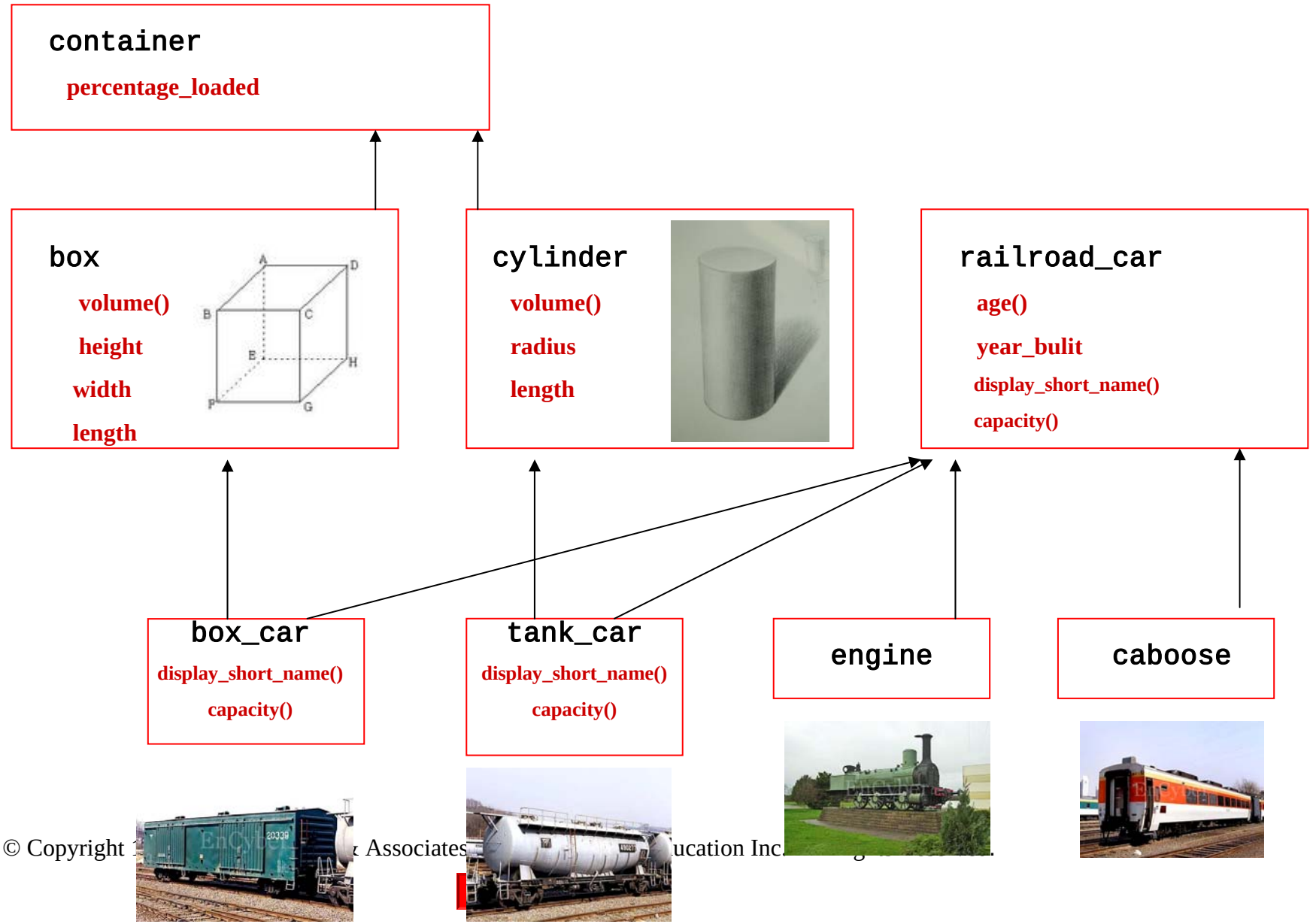
Effect of Protected and Private Derivation Summary

	Public Derivation	Protected Derivation	Private Derivation
Public Members	Remains public	Becomes Protected	Becomes Private
Protected Members	Remains Protected	Remains Protected	Becomes Private
Private Members	Remains Private	Remains Private	Remains Private

Slightly Updated Class Definitions

```
class railroad_car {
  public: railroad_car () { }
          virtual void display_short_name () { }
          virtual double capacity () { return 0.0 }
};
class box_car : public railroad_car, public box {
  public: // Default constructor:
          box_car () : box (10.5, 9.2, 40.0) { }
          // Displayers:
          virtual void display_short_name () {cout << "box";}
          virtual double capacity () {return volume ();}
};
class tank_car : public railroad_car, public cylinder {
  public: // Default constructor:
          tank_car () : cylinder (3.5, 40.0) { }
          // Displayers:
          virtual void display_short_name () {cout << "tnk";}
          virtual double capacity () { return volume ();}
};
```


Revisit Our Full Class Hierarchy



Call-by-Value Class Parameter

- Let's define an ordinary function with class parameter
  - Takes an ordinary railroad_car object and computes its volume()

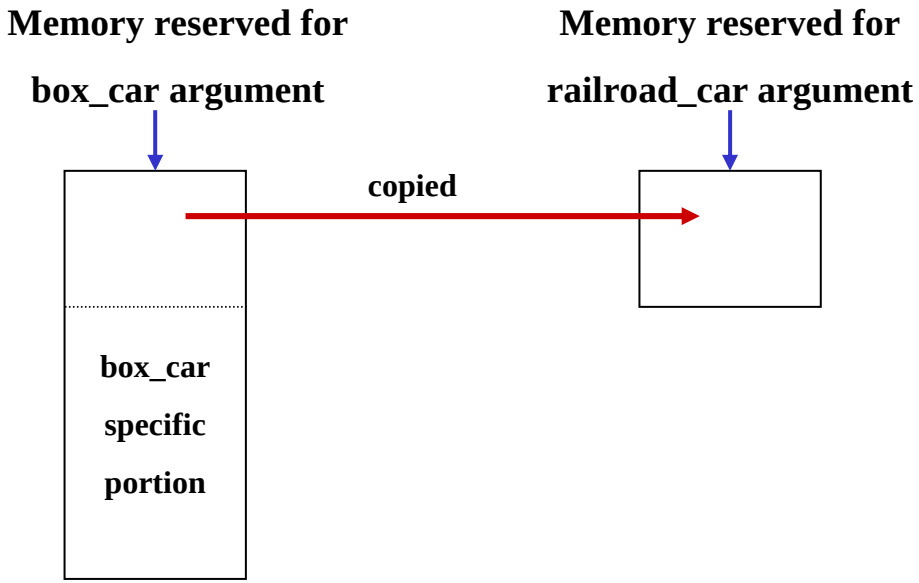
```
double ordinary_capacity_function ( railroad_car r ) {
  return r.capacity ();
}
for (n = 0; n < car_count; ++n) {
  // Display short name and capacity and terminate the line:
  cout << train[n]->short_name ( )
       << "     "
       << ordinary_capacity_function( *train[n] )
       << endl;
}
```

  - Output is not what we want

```
Output:  eng 0
         box 0
         box 0
         …
```


What is the Problem?

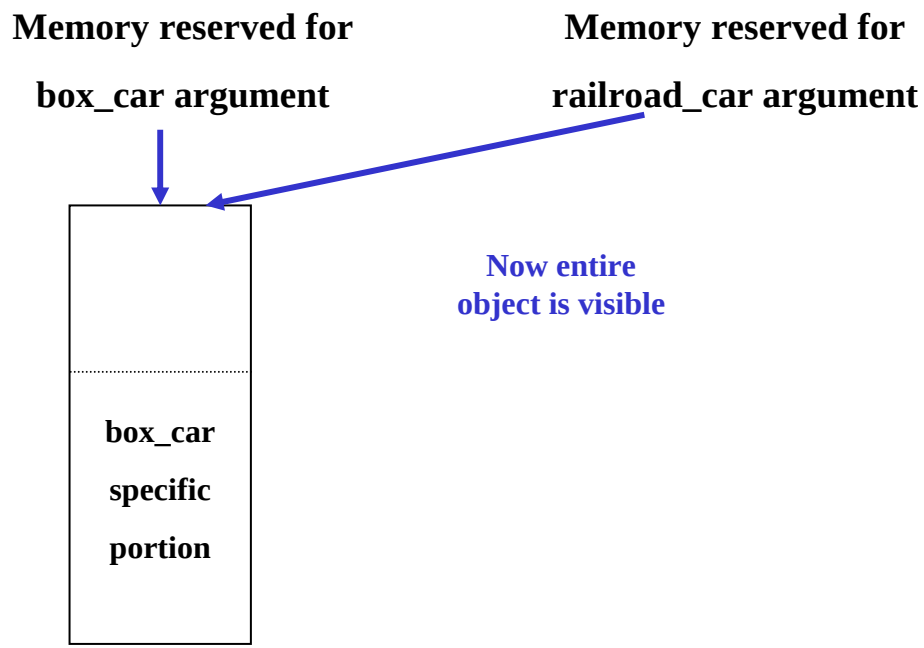
- Call-by-value really makes a copy of the object
  - C++ reserves a space for a `railroad_car` formal parameter
  - However, the actual parameter is an `box_car` object
  - So, only the railroad_car portion is copied
  - C++ calls `capacity()` defined in railroad_car class



Solution: Call-by-Reference

Replace the call-by-value argument by call-by-reference argument

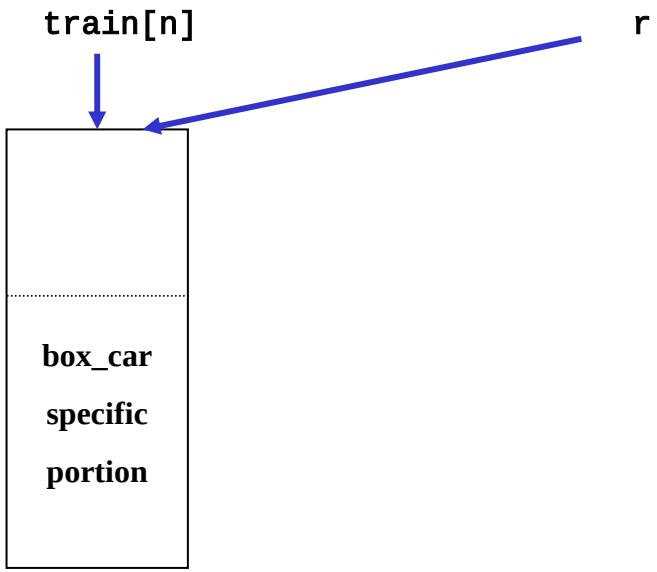
```
double ordinary_capacity_function ( railroad_car& r ) {
  return r.capacity ();
}
```



Another Solution: Call-by-Value Pointer

- Based on polymorphism

```
double ordinary_capacity_function ( railroad_car* r ) {
  return r->capacity ();
}
cout << train[n]->short_name ( )
       << "     "
       << ordinary_capacity_function( train[n] )
```



Benefit of Call-by-Reference

- Obviate object copying
- Allows modification of arguments

```
void loading_function (box_car& b) { // OK
  b.percentage_loaded = 100;
  return;
}


void loading_function (box_car b) {  // DEFECTIVE!
  b.percentage_loaded = 100;
  return;
}
```


Revisit Polymorphism & Virtual Function

What if we save a class object to its superclass variable

- Would virtual function call & polymorphism work as before?

```
Class A {
  public:
    virtual void foo() { cout << “foo() for A” << endl;}
}
Class B {
  public:
    virtual void foo() { cout << “foo() for B” << endl;}
}
int main() {
  A a1, *a2;
  B b;
  a1 = b; a1.foo();   // What is printed? foo() for A or B?
  a2 = &b; a2->foo(); // What is printed? foo() for A or B?
}
```


Delete and Destructors

- `delete` reclaims memory of previously created object

```
delete train[car_count];
```

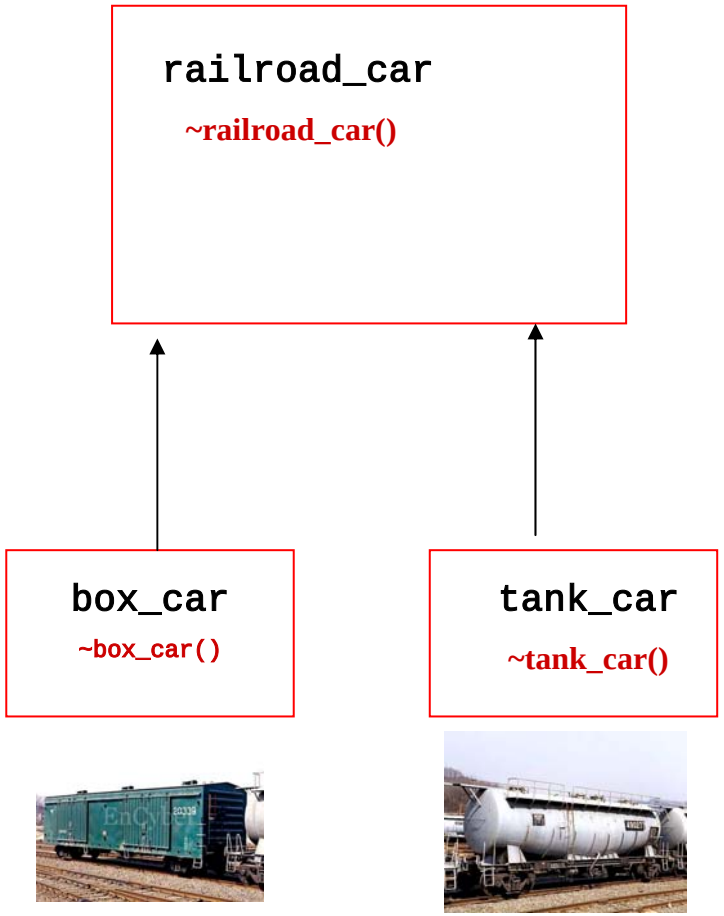
- What if the object also has a previously created object in it?
- Resort to the destructor defined in the class (~class-name())
  - Supposed to be called when an object is de-allocated via
    - function call return (local variables), delete (dynamic objects), program exit
  - All destructors in the class hierarchy are called (from bottom to top)

Destructor Hierarchy

```
Void foo() {
  box_car x;
  tank_car *y;
  y = new tank_car();
  …
  delete y;
  return;
}
```

Call Sequences:

```
~tank_car()
~railroad_car()
~box_car()
~railroad_car()
```



What Happens when using Polymorphism?

- When a pointer variable points to a subclass object, what destructor(s) are called when delete the pointer?

```
railroad_car *x;

x = new (box_car);
..
delete x;
..
```

- Calls Only destructor for the superclass (~railroad_car())
- We need to declare the destructor virtual as well
  - Unlike other virtual functions, they have different names

```
class railroad_car {
  public: char *serial_number;
          // Constructors:
          railroad_car ( ) { }
          railroad_car (char *input_buffer) {
            // Create new array just long enough:
            serial_number = new char[strlen(input_buffer) + 1];
            // Copy string into new array:
            strcpy (serial_number, input_buffer);
          }
          // Destructor:
          virtual ~railroad_car ( ) {
            cout << "Deleting a railroad serial number" << endl;
            delete [ ] serial_number;
          }
          // Other:
          virtual char* short_name ( ) {return "rrc";}
          virtual double capacity ( ) {return 0.0;}
};
```