[368] Inheritance

Tyler Caraza-Harter

Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

Object State

Dynamic Cast

Demos

C++ Surprises, a Preview...

```
class Animal {
public:
  void speak() {
    cout << "TODO\n";
  };
};

class Dog : public Animal {
public:
  void speak() {
    cout << "bark!\n";
  }
};

int main() {
  Dog* d = new Dog;
  d->speak();
  Animal* a = d;
  a->speak();
}
```

what does it print?

what does it print?

What will you learn today?

Learning objectives

- write classes that inherit from other classes
- describe how function overriding is implemented internally with the help of vtables
- decide when a function should be virtual
- avoid common C++ OOP pitfalls, such as lack of virtual destructor, vectors of object values of different types, etc.

Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

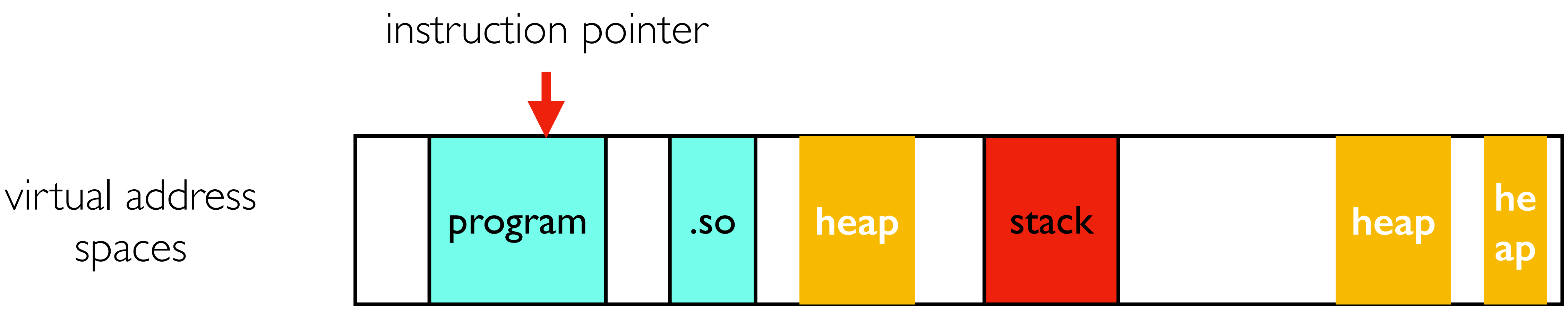
Object State

Dynamic Cast

Demos

Review: Address Space

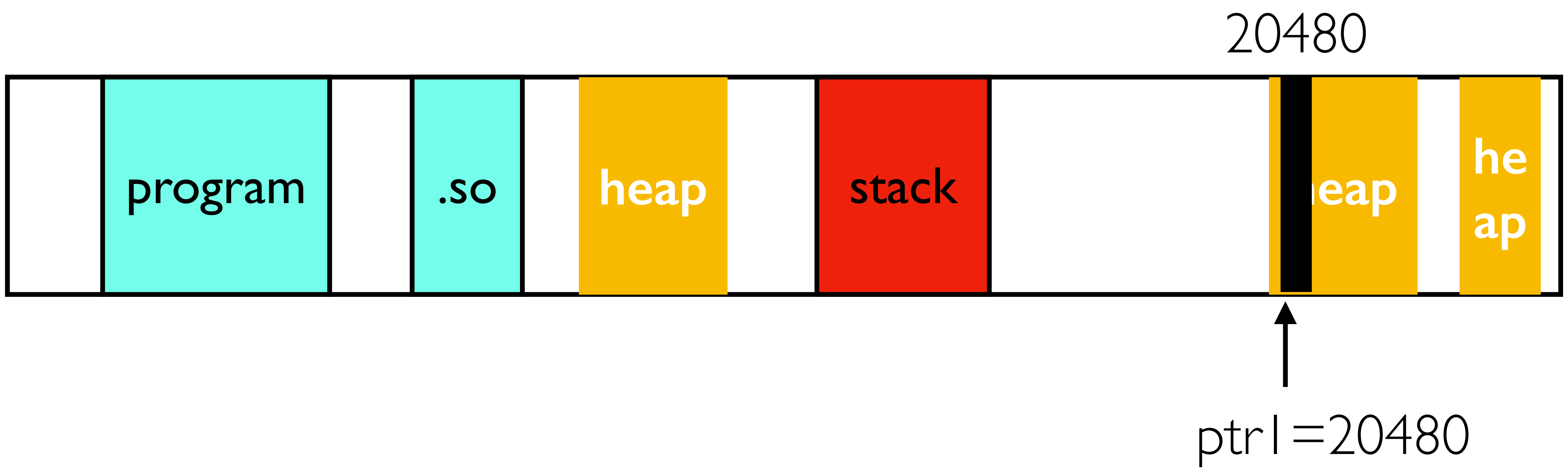
- our code (functions live in a program and possibly shared libraries)
- each thread has a stack pointer (to code) and a contiguous stack (for local variables)
- non-contiguous heap is shared between threads



Review: Address Space

- our code (functions live in a program and possibly shared libraries)
- each thread has a stack pointer (to code) and a contiguous stack (for local variables)
- non-contiguous heap is shared between threads

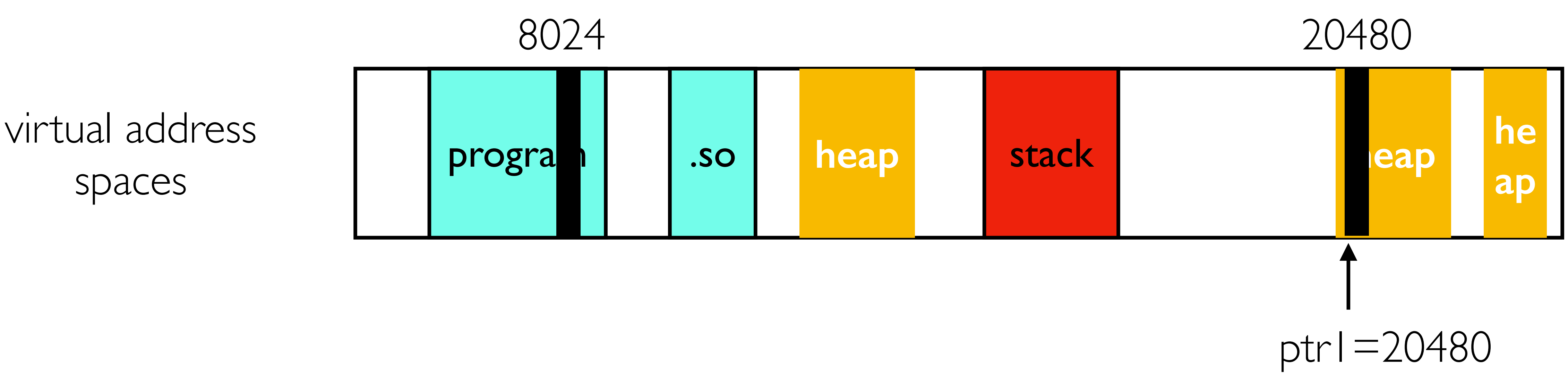
virtual address spaces




```
auto ptr1 = new MyClass;
```

Function Code Lives in Memory Too

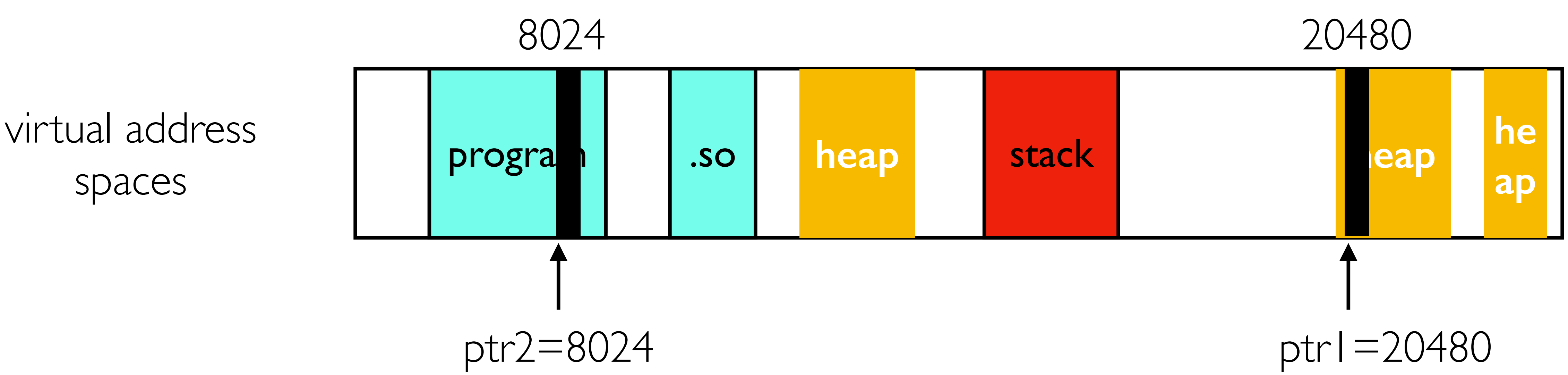
- an offset into the address space (i.e., "address") corresponds to function code
- that address can be stored in a pointer (a function pointer)
- function pointers can be used to call functions




```
void f(int x) {...}        auto ptr1 = new MyClass;
```

Function Code Lives in Memory Too

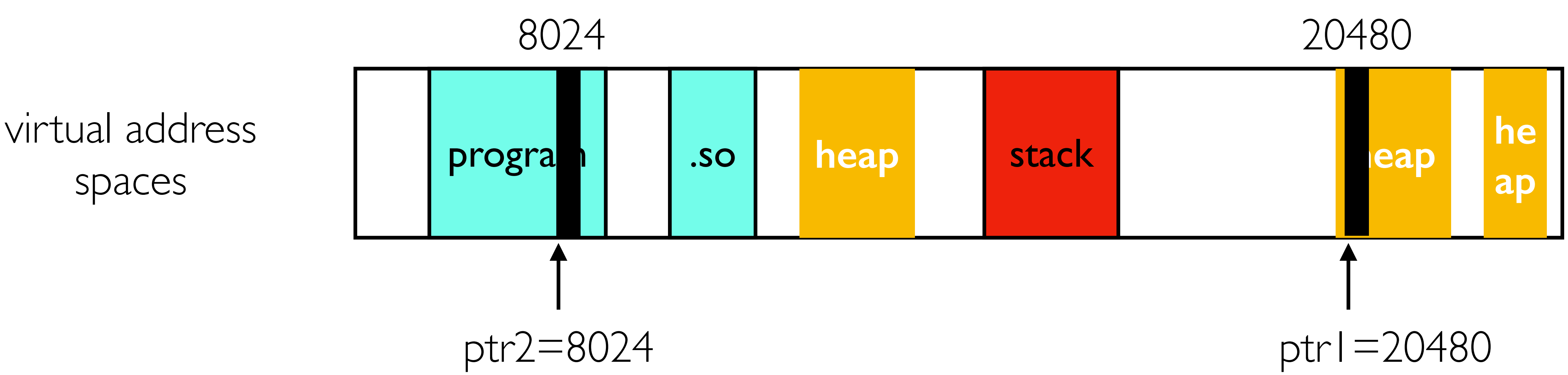
- an offset into the address space (i.e., "address") corresponds to function code
- that address can be stored in a pointer (a function pointer)
- function pointers can be used to call functions




```
void f(int x) {...}        auto ptr1 = new MyClass;

auto ptr2 = f;
```

Function Code Lives in Memory Too

- an offset into the address space (i.e., "address") corresponds to function code
- that address can be stored in a pointer (a function pointer)
- function pointers can be used to call functions

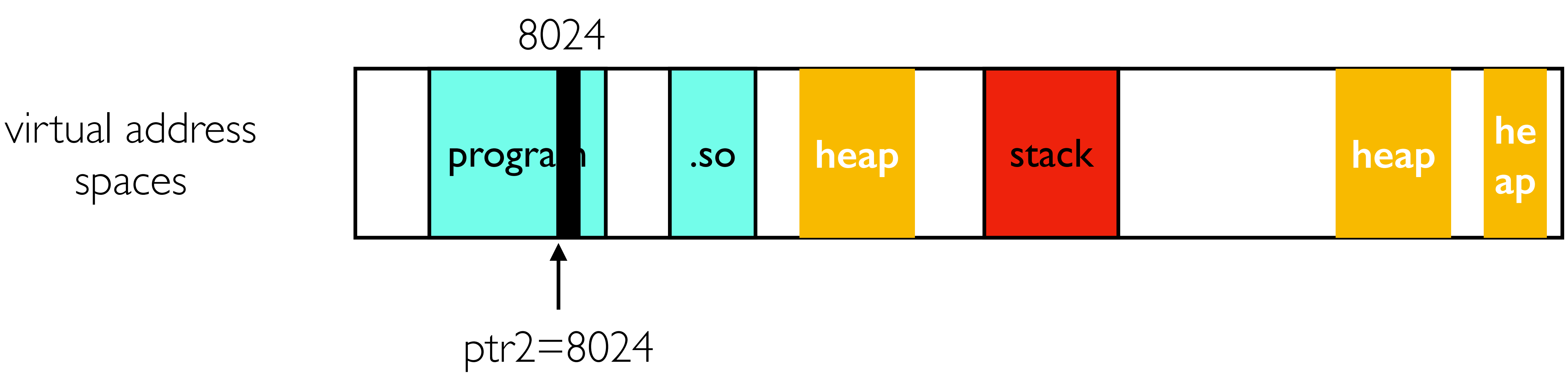



```
void f(int x) {...}        auto ptr1 = new MyClass;

auto ptr2 = f;
```

calls: `f(3)` OR `ptr2(3)`

Function Pointer Syntax

- auto is helpful because the syntax is ugly (and unnecessarily confusing)
- param types and return type ARE part of the function type
- function name and param names ARE NOT part of the function type




```
void f(int x) {...}

auto ptr2 = f;
```

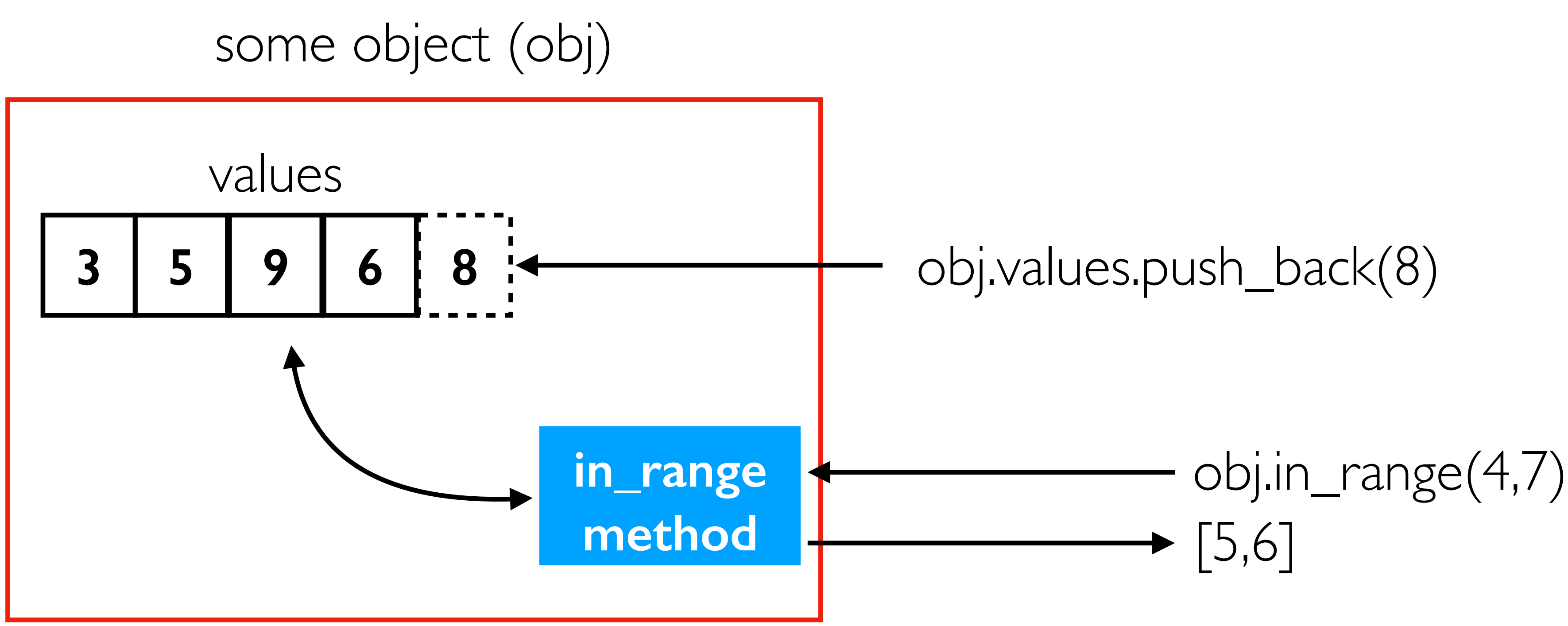
```
// without auto
void (*ptr2)(int) = f;
```

calls: `f(3)` OR `ptr2(3)`

Passing Func Pointers to Funcs Enables Customizable Behavior

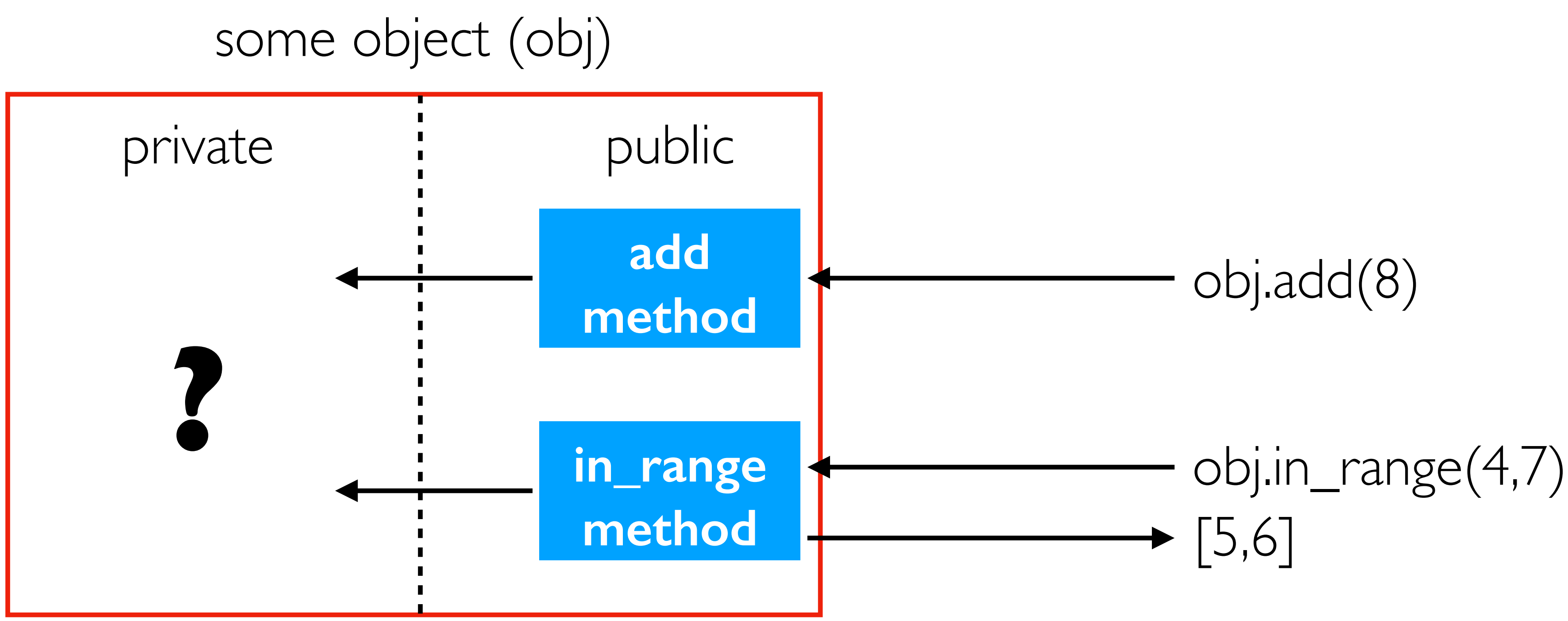
```
bool CompareAlpha(string x, string y) {
  return x < y;
}

bool CompareLen(string x, string y) {
  return x.size() < y.size();
}

using CompareFn = bool (*)(string, string);

void PrintFirst(string a, string b, CompareFn fn) {
  if (fn(a, b))
    cout << a << "\n";
  else
    cout << b << "\n";
}

int main() {
  PrintFirst("Apple", "Pie", CompareAlpha);
  PrintFirst("Apple", "Pie", CompareLen);
}
```

Review: Motivation for Encapsulation



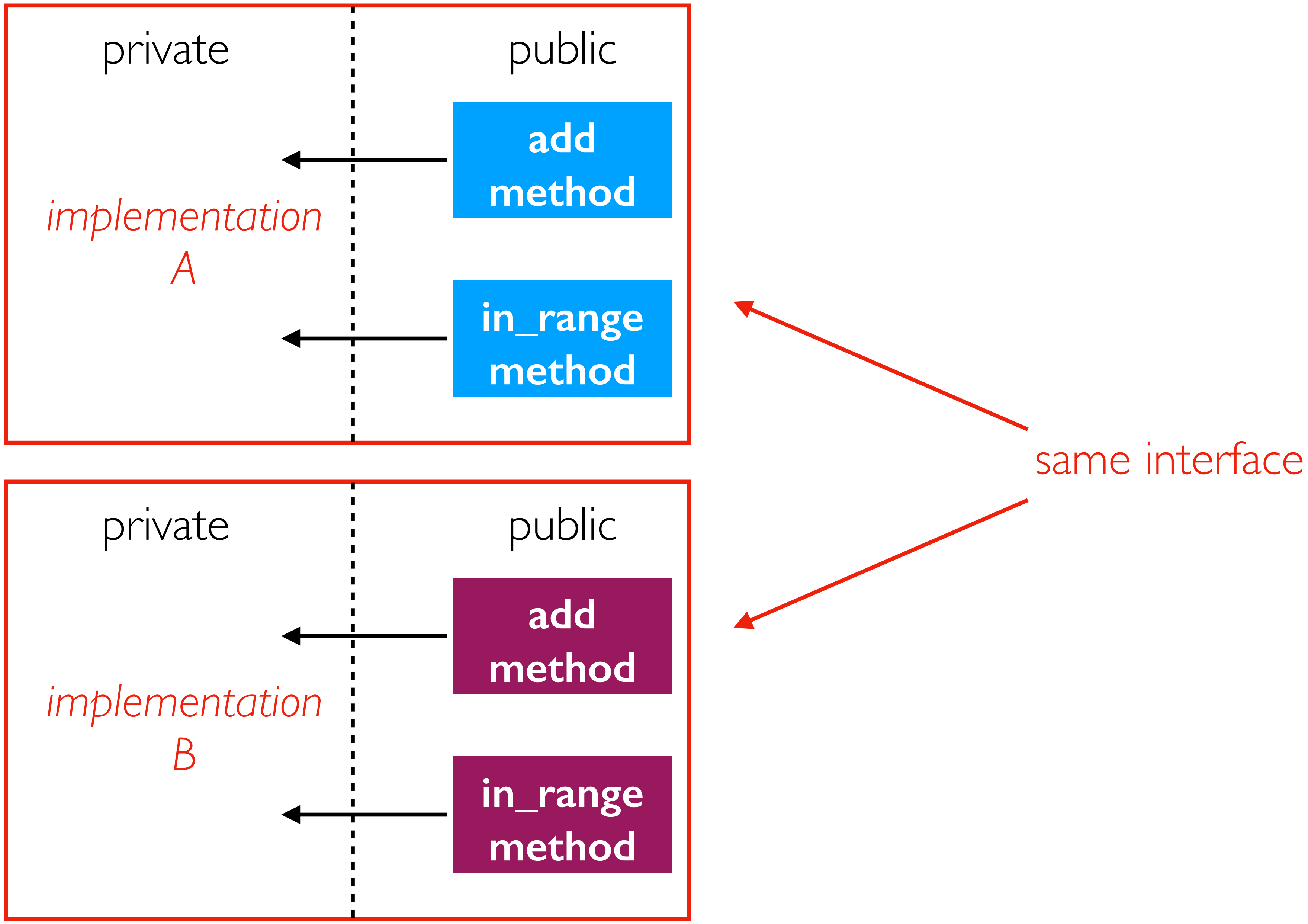
- if we add frequently and call in_range rarely, this implementation is good
- what if we call in_range frequently? Can we improve the library without breaking all the programs that use the library?

Review: Motivation for Encapsulation



- encapsulation lets us modify internal implementation without breaking code that uses our libraries

Encapsulation and Interfaces



- encapsulation lets us modify internal implementation without breaking code that uses our libraries
- interfaces further let us have multiple implementations of the same interface, designed for different scenarios!

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}

void duck_speak() {
  cout << "quack!\n";
}

bool duck_can_fly() {
  return true;
}

...
```

Step 1:

- decide what types (dog, duck, etc)
- decide what "methods" (speak, can_fly, etc)
- write regular functions for each combo

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

using SpeakFn = void (*)();
using CanFlyFn = bool (*)();

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};
```

Step 2:

- define function pointers for each "method"
- create a struct of function pointers

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};

struct Animal {
  AnimalFuncTable vtable;
  void *data;
};
```

Step 3: pair "table" of function ptrs with some data

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};

struct Animal {
  AnimalFuncTable vtable;
  void *data;
};
```

```
Animal* make_dog() {
  return new Animal{
    .vtable = AnimalFuncTable{
        .speak=dog_speak,
        .can_fly=dog_can_fly
    },
    .data = nullptr // TODO
  };
}
```

Step 4: write functions that initialize func table alongside corresponding data (for each type)

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};

struct Animal {
  AnimalFuncTable vtable;
  void *data;
};
```

```
Animal* make_dog() {
  return new Animal{
    .vtable = AnimalFuncTable{
        .speak=dog_speak,
        .can_fly=dog_can_fly
    },
    .data = nullptr // TODO
  };
}

int main() {
  Animal* dog = make_dog();
  dog->vtable.speak();
  cout << dog->vtable.can_fly();
}
```

Step 5: use vtable to determine what function we should call for a specific type

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};

struct Animal {
  AnimalFuncTable vtable;
  void *data;
};
```

```
Animal* make_dog() {
  return new Animal{
    .vtable = AnimalFuncTable{
        .speak=dog_speak,
        .can_fly=dog_can_fly
    },
    .data = nullptr // TODO
  };
}

int main() {
  vector<Animal*> farm{
    make_dog(),
    make_duck(),
    make_cat(),
    ...
  };
  for (auto animal : farm)
    animal->vtable.speak();
}
```

different types implementing the same interface can be used together!

Doing OOP without Classes

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};

struct Animal {
  AnimalFuncTable vtable;
  void *data;
};
```

```
Animal* make_dog() {
  return new Animal{
    .vtable = AnimalFuncTable{
        .speak=dog_speak,
        .can_fly=animal_can_fly
    },    vtables suppot inheritance patterns
    .data = nullptr // TODO
  };
}

int main() {
  vector<Animal*> farm{
    make_dog(),
    make_duck(),
    make_cat(),
    ...
  };
  for (auto animal : farm)
    animal->vtable.speak();
}
```

Language Support for OOP

```
void dog_speak() {
  cout << "bark!\n";
}

bool dog_can_fly() {
  return false;
}
...

struct AnimalFuncTable {
  SpeakFn speak;
  CanFlyFn can_fly;
};

struct Animal {
  AnimalFuncTable vtable;
  void *data;
};
```

```
Animal* make_dog() {
  return new Animal{
    .vtable = AnimalFuncTable{
        .speak=dog_speak,
        .can_fly=dog_can_fly
    },
    .data = nullptr // TODO
  };
}

int main() {
  vector<Animal*> farm{
    make_dog(),
    make_duck(),
    make_cat(),
    ...
  };
  for (auto animal : farm)
    animal->vtable.speak();
}
```

- OOP languages usually have a vtable, but hide it from you
- extra lookup adds function call overhead
- C++ lets you decide when to use a vtable

```
animal.speak();
```

Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

Object State

Dynamic Cast

Demos

Virtual Functions

```
class MyClass {
public:
    void f() {
        ...
    }

    virtual void g() {
        ...
    }
};
```

- f will NOT go in the vtable. It will be faster, but CANNOT be overriden
- g with go in the vtable, calls will take an extra lookup step
- most languages just use virtual functions for everything, without using that vocabulary
- C++ does not use virtual functions unless you explicitly ask for it

Assembly Code

```
// try in https://godbolt.org/
#include <iostream>

class Animal {
public:
    // TODO: make it virtual
    void speak() {
        std::cout << "TODO\n";
    }
};

int main() {
    Animal* a = new Animal;
    a->speak();
}
```

Virtual vs. Non-Virtual

```
class Animal {
public:
  void speak() {
    cout << "TODO\n";
  };
};

class Dog : public Animal {
public:
  void speak() {
    cout << "bark!\n";
  }
};

int main() {
  Dog* d = new Dog;
  d->speak();
  Animal* a = d;
  a->speak();
}
```

bark!

TODO

Virtual vs. Non-Virtual

```
class Animal {
public:
  virtual void speak() {
    cout << "TODO\n";
  };
};

class Dog : public Animal {
public:
  void speak() {
    cout << "bark!\n";
  }
};

int main() {
  Dog* d = new Dog;
  d->speak();
  Animal* a = d;
  a->speak();
}
```

bark!

bark!

Virtual vs. Non-Virtual

```
class Animal {
public:
  virtual void speak() {
    cout << "TODO\n";
  };
};

class Dog : public Animal {
public:
  void speak() override {      "override" is an optional safety check
    cout << "bark!\n";
  }
};

int main() {
  Dog* d = new Dog;
  d->speak();                  bark!
  Animal* a = d;
  a->speak();                  bark!
}
```

Virtual vs. Non-Virtual

```
class Animal {
public:
  void speak() {
    cout << "TODO\n";
  };
};

class Dog : public Animal {
public:
  void speak() override {
    cout << "bark!\n";
  }
};

int main() {
  Dog* d = new Dog;
  d->speak();
  Animal* a = d;
  a->speak();
}
```

```
error: 'void Dog::speak()' marked
'override', but does not override
```

"override" is an optional safety check

Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

Object State

Dynamic Cast

Demos

Classes vs. Interfaces in Java

```
class Property {
    private String address;

    public Property(String address) {
        this.address = address;
    }
}

interface Sortable {
    int compareTo(Property other);
}

public class House extends Property implements Sortable {
    private int numberOfRooms;

    public House(String address, int numberOfRooms) {
        super(address);
        this.numberOfRooms = numberOfRooms;
    }

    public int compareTo(Property other) {
        ...
    }
    ...
}
```

classes have declarations and possible definitions
(abstract classes may not have all definitions)

interfaces have method declarations

single inheritance, but can implement multiple interfaces

C++ differences

- can inherit from multiple classes
- classes may be abstract
- a class with just declarations acts like an interface

Multiple Inheritance

```
class Property {
private:
    std::string address;

public:
    Property(const std::string& address) : address(address) {}
};

class Sortable {
public:
    virtual int compareTo(const Property& other) = 0;
};

class House : public Property, public Sortable {
private:
    int numberOfRooms;

public:
    House(const std::string& address, int numberOfRooms)
        : Property(address), numberOfRooms(numberOfRooms) {}

    int compareTo(const Property& other) override {
        ...
    }
    ...
};
```

Pure Virtual Functions, Abstract, Interfaces

```
class Property {
private:
    std::string address;

public:
    Property(const std::string& address) : address(address) {}
};

class Sortable {
public:
    virtual int compareTo(const Property& other) = 0;
};

class House : public Property, public Sortable {
private:
    int numberOfRooms;

public:
    House(const std::string& address, int numberOfRooms)
        : Property(address), numberOfRooms(numberOfRooms) {}

    int compareTo(const Property& other) override {
        ...
    }
    ...
};
```

virtual: MAY be overriden

this means "pure virtual": it MUST be overridden

you cannot create objects from an abstract class, just from it's child classes

- "abstract" class: <u>at least one</u> pure virtual function
- "interface" class: <u>all</u> the functions are pure virtual

Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

Object State

Dynamic Cast

Demos

Object State: Initialization

```
class Property {
private:
    std::string address;

public:
    Property(const std::string& address) : address(address) {}
};

class Sortable {
public:
    virtual int compareTo(const Property& other) = 0;
};

class House : public Property, public Sortable {
private:
    int numberOfRooms;

public:
    House(const std::string& address, int numberOfRooms)
        : Property(address), numberOfRooms(numberOfRooms) {}
    int compareTo(const Property& other) override {
        ...
    }
    ...
};
```

parent constructor — child state

Object State: Visibility

```
class MyClass {
public:
  int x;
private:
  int y;
protected:
  int z;
}
```

visibility: MyClass, friends, child class, others

visibility: MyClass, friends

visibility: MyClass, friends, child class

Object State: Size

```
class Coord2D {
public:
  int x, y;
};

class Coord3D : public Coord2D {
public:
  int z;
};
```

- `sizeof(Coord2D) > sizeof(Coord3D)`
- `sizeof(Coord2D*) = sizeof(Coord3D*)`
- if you want a vector of mixed types, need to use pointers, not values!

Object State: Size

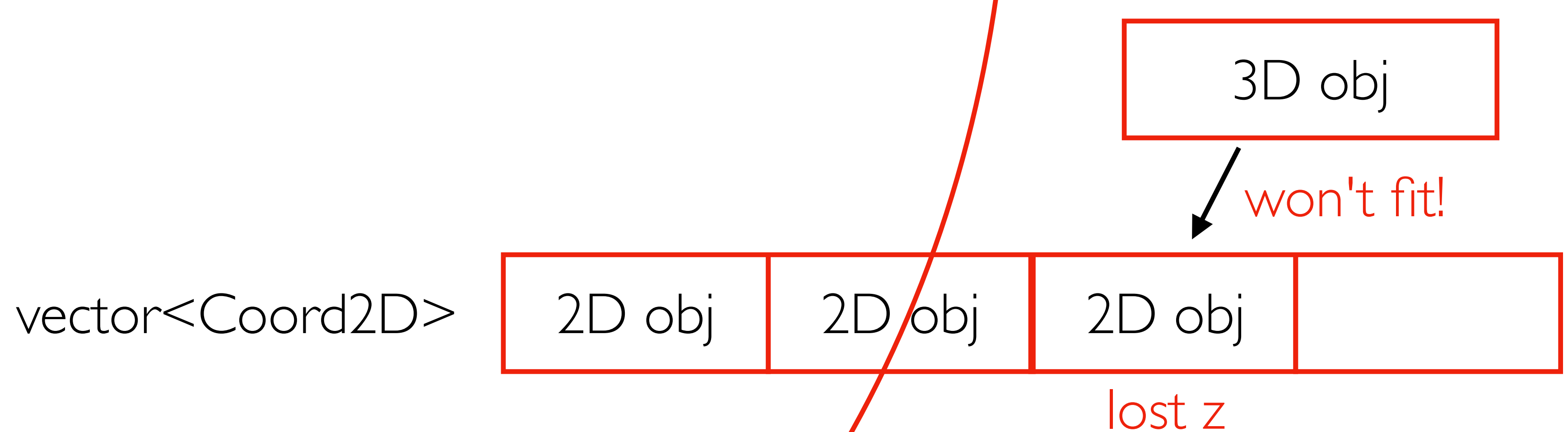
```
class Coord2D {
public:
  int x, y;
};
```

implicit copy constructor can take a Coord3D

```
Coord2D(const Coord2D& other)
: x(other.x), y(other.y) {}
```

```
class Coord3D : public Coord2D {
public:
  int z;
};
```

BAD




```
Coord2D c2{1,2};
Coord3D c3{1,2,3};
vector<Coord2D> vec;
vec.push_back(c2);
vec.push_back(c3);
```

Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

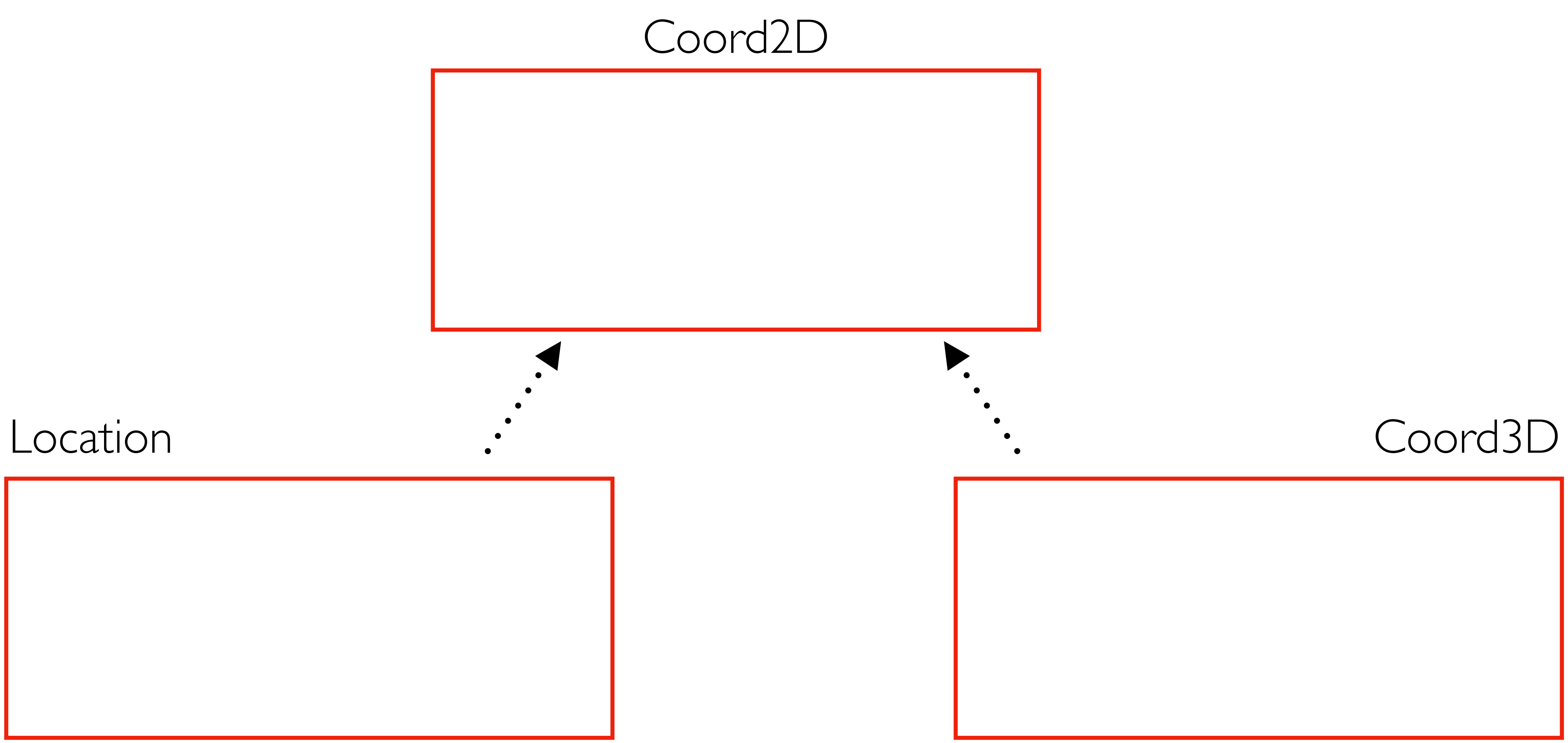
Object State

Dynamic Cast

Demos

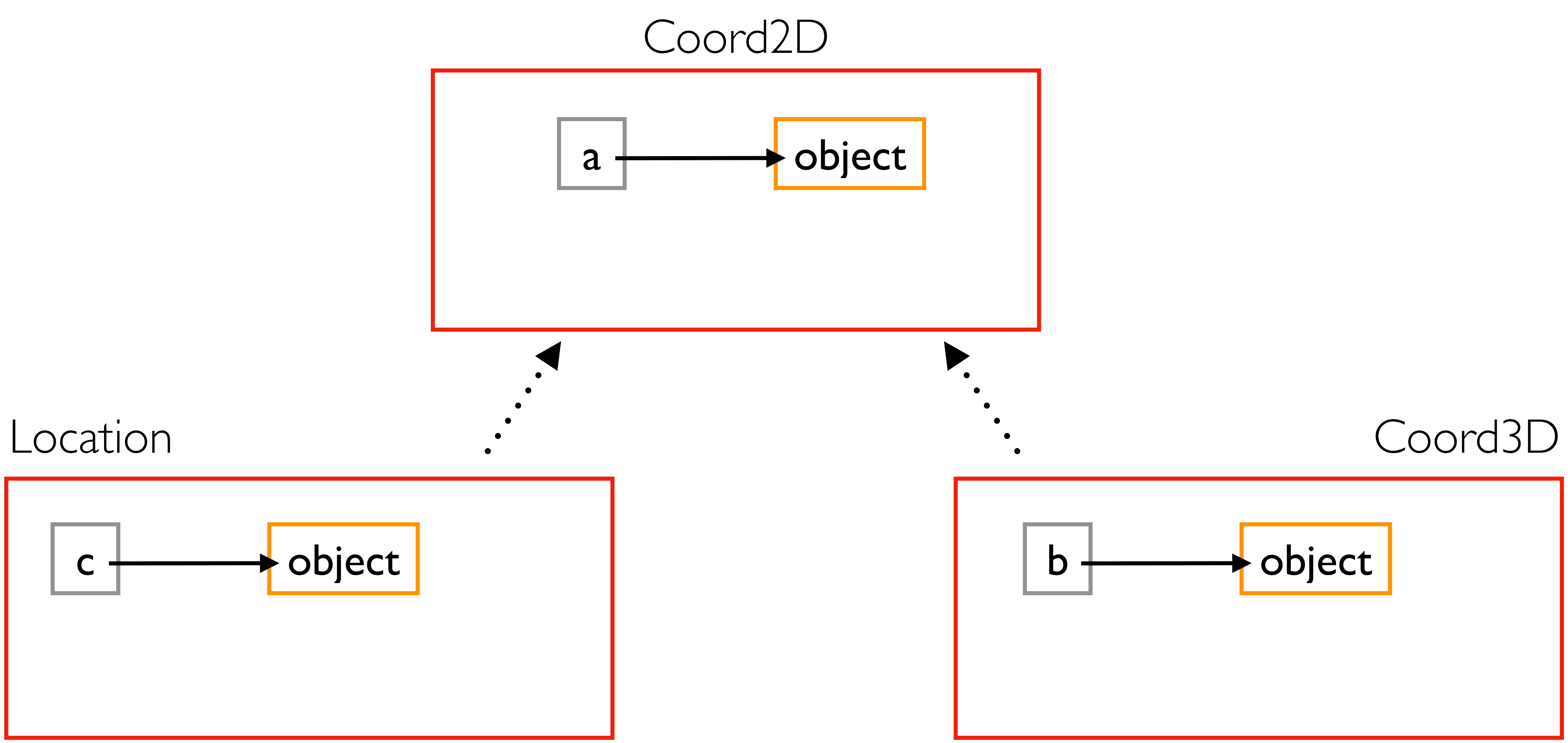
Class Hierarchy

```
class Coord2D {int x; int y; ...};
class Coord3D : public Coord2D {int z; ...};
class Location : public Coord2D {string name; ...};
```



Class Hierarchy

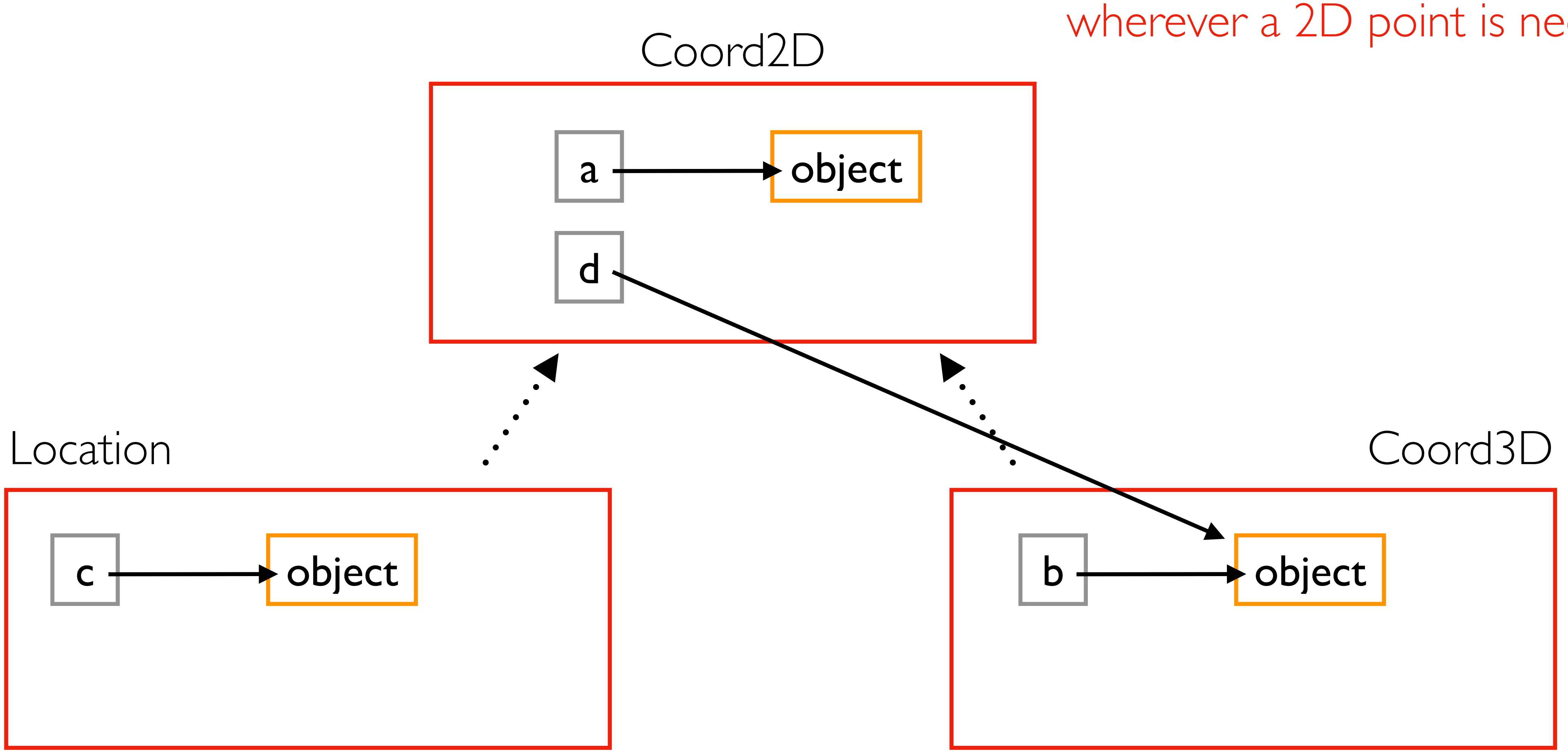
```
class Coord2D {int x; int y; ...};
class Coord3D : public Coord2D {int z; ...};
class Location : public Coord2D {string name; ...};

auto a = new Coord2D{8,9};
auto b = new Coord3D{8,9,3};
auto c = new Location{8,9,"capitol"};
```



static_cast

```
class Coord2D {int x; int y; ...};
class Coord3D : public Coord2D {int z; ...};
class Location : public Coord2D {string name; ...};

auto a = new Coord2D{8,9};
auto b = new Coord3D{8,9,3};
auto c = new Location{8,9,"capitol"};

auto d = static_cast<Coord2D*>(b);
```

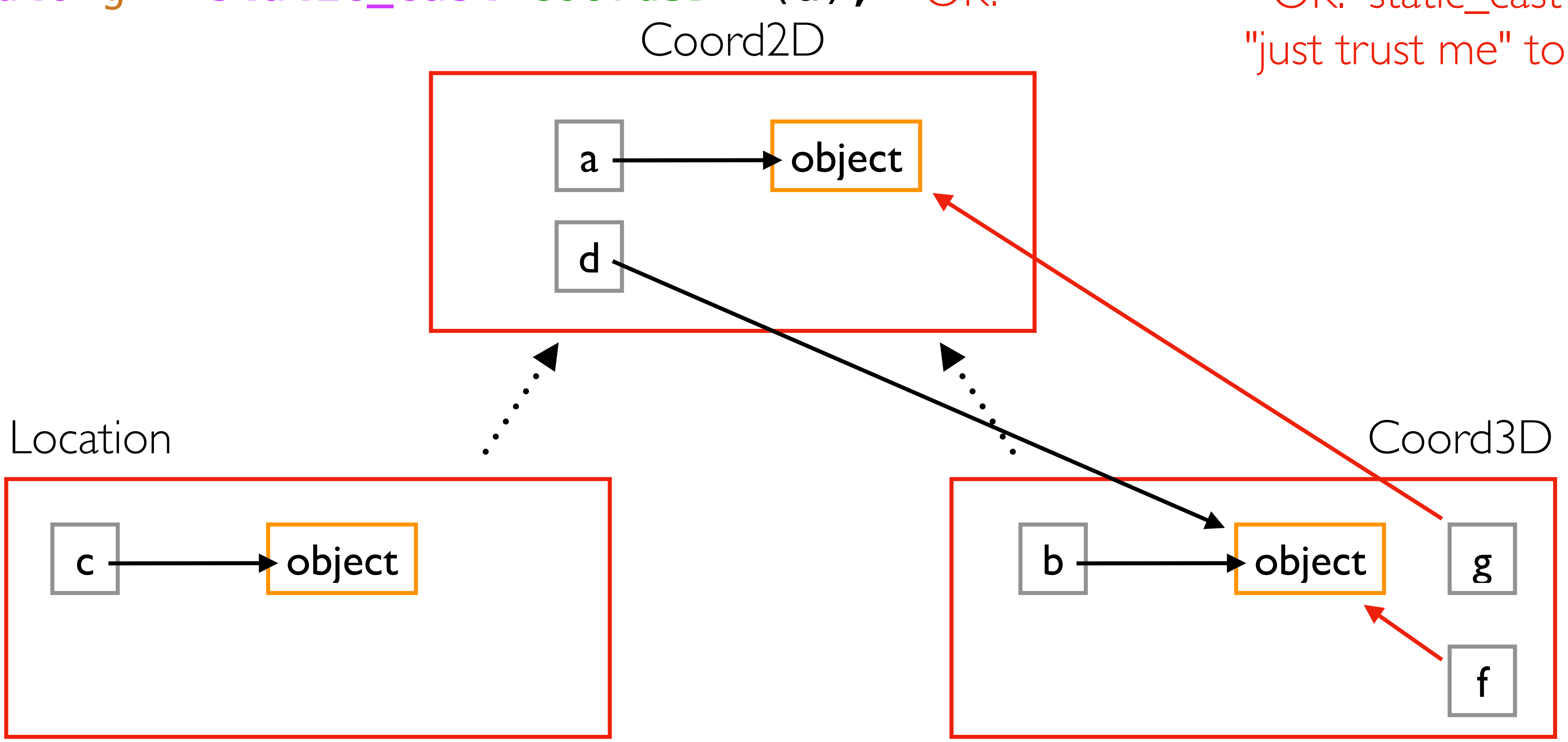
casting up the hierarchy is clearly OK. A 3D point has everything a 2D point has and more, so it can be used wherever a 2D point is needed.



static_cast

```
class Coord2D {int x; int y; ...};
class Coord3D : public Coord2D {int z; ...};
class Location : public Coord2D {string name; ...};

auto a = new Coord2D{8,9};
auto b = new Coord3D{8,9,3};
auto c = new Location{8,9,"capitol"};

auto d = static_cast<Coord2D*>(b);
auto f = static_cast<Coord3D*>(d);   OK?
auto g = static_cast<Coord3D*>(a);   OK?
```

casting down the hierarchy is sometimes OK. static_cast says "just trust me" to C++



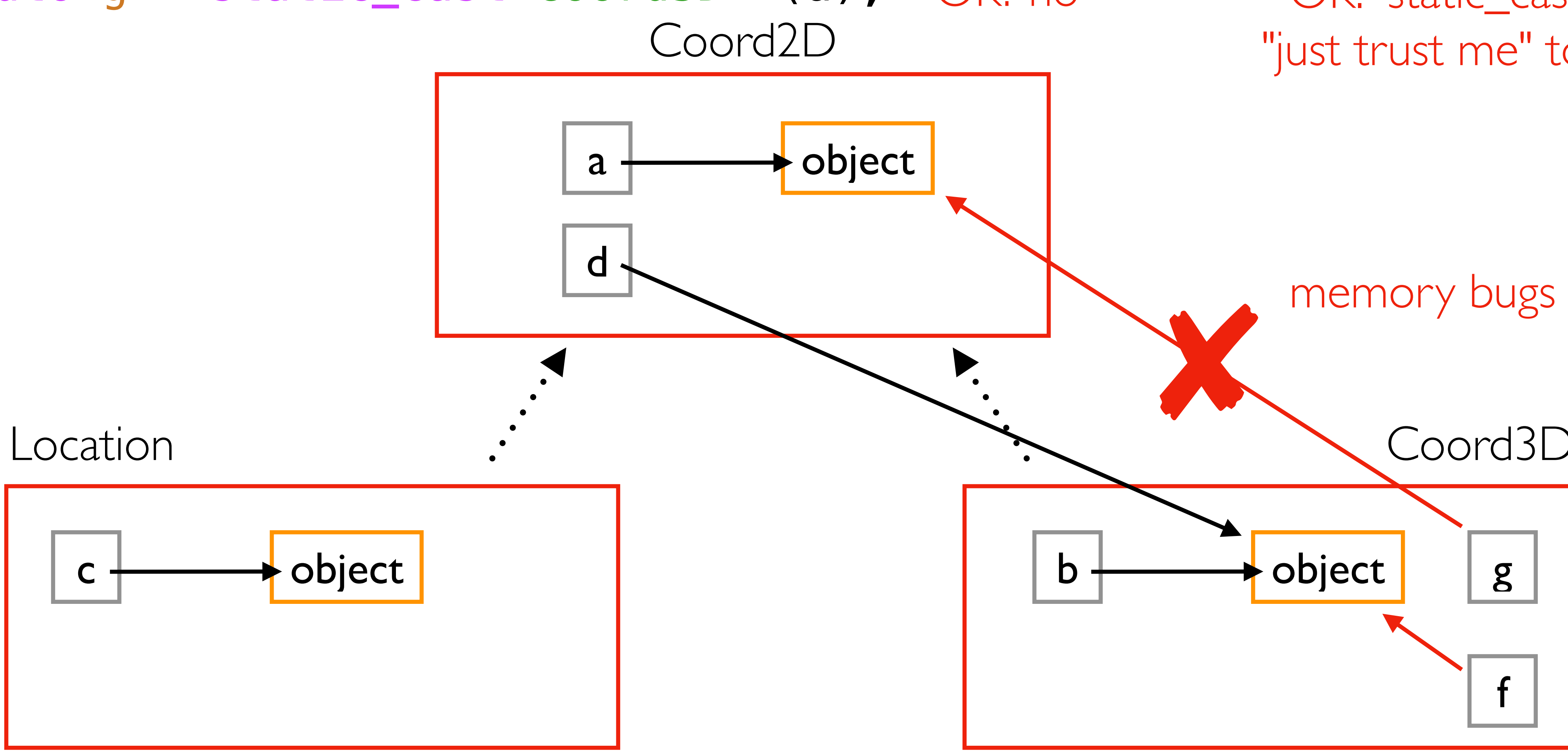
static_cast

```
class Coord2D {int x; int y; ...};
class Coord3D : public Coord2D {int z; ...};
class Location : public Coord2D {string name; ...};

auto a = new Coord2D{8,9};
auto b = new Coord3D{8,9,3};
auto c = new Location{8,9,"capitol"};

auto d = static_cast<Coord2D*>(b);
auto f = static_cast<Coord3D*>(d);
auto g = static_cast<Coord3D*>(a);
```

OK? yes

OK? no

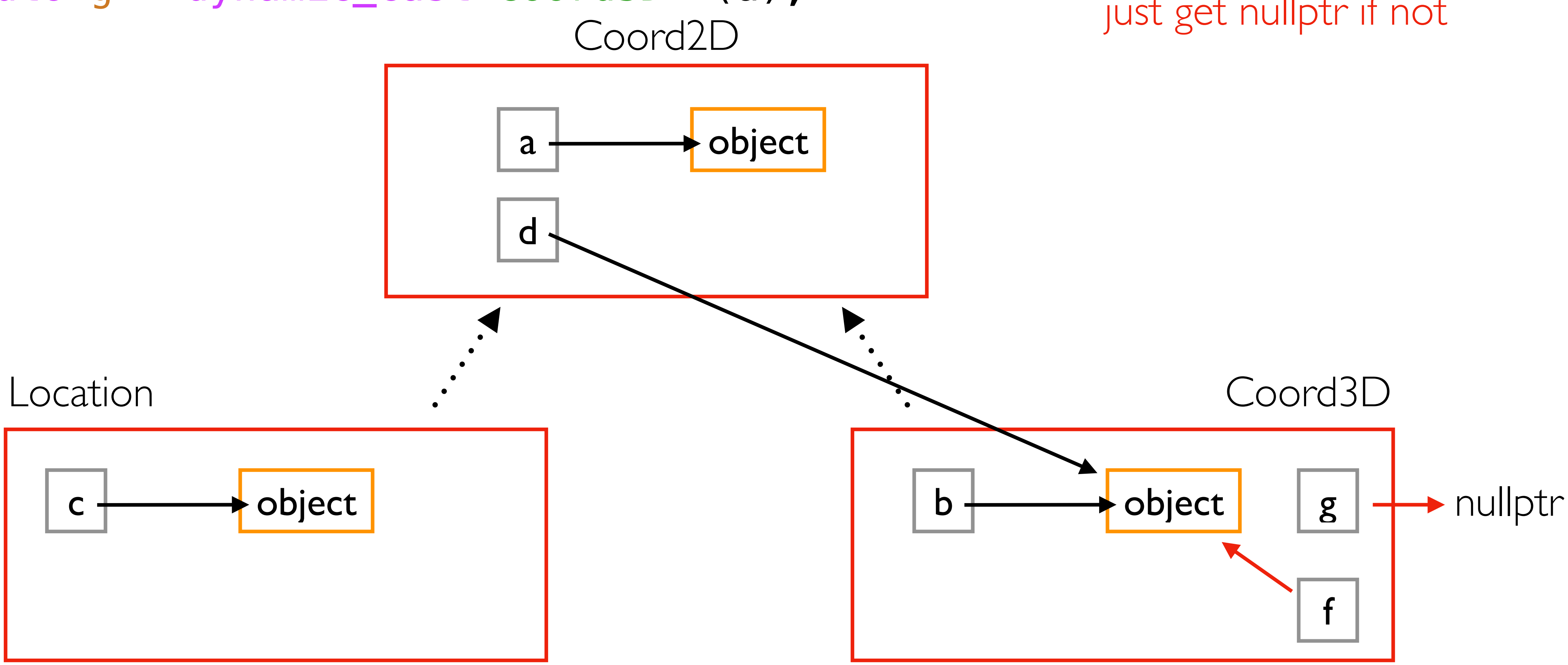
casting down the hierarchy is sometimes OK. static_cast says "just trust me" to C++



dynamic_cast

```
class Coord2D {int x; int y; ...};
class Coord3D : public Coord2D {int z; ...};
class Location : public Coord2D {string name; ...};

auto a = new Coord2D{8,9};
auto b = new Coord3D{8,9,3};
auto c = new Location{8,9,"capitol"};

auto d = dynamic_cast<Coord2D*>(b);
auto f = dynamic_cast<Coord3D*>(d);
auto g = dynamic_cast<Coord3D*>(a);
```

dynamic_cast checks conversion is OK. We just get nullptr if not



Outline

TopHat and Worksheet

Function Pointers, C-Style Interfaces

Virtual Functions

Pure Virtual

Object State

Dynamic Cast

Demos