[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

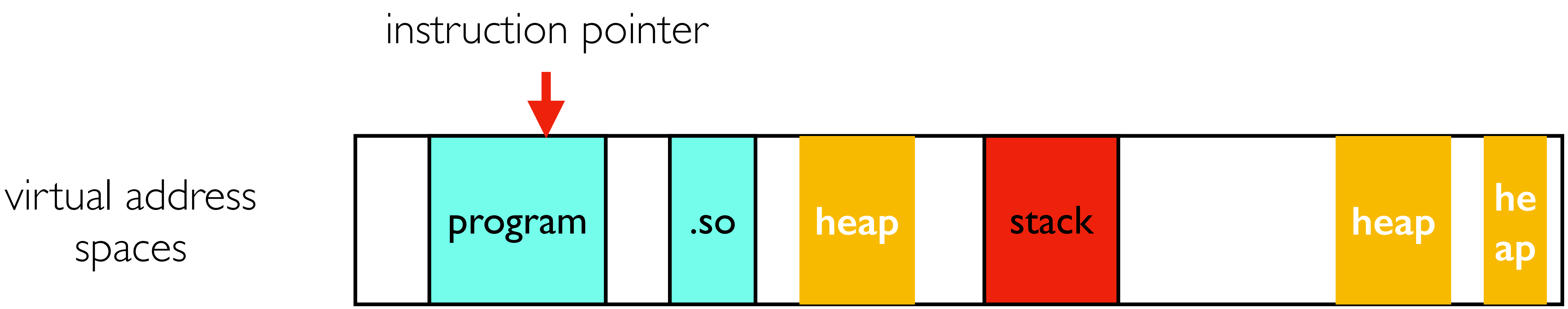
Object State

Dynamic Cast

Demos

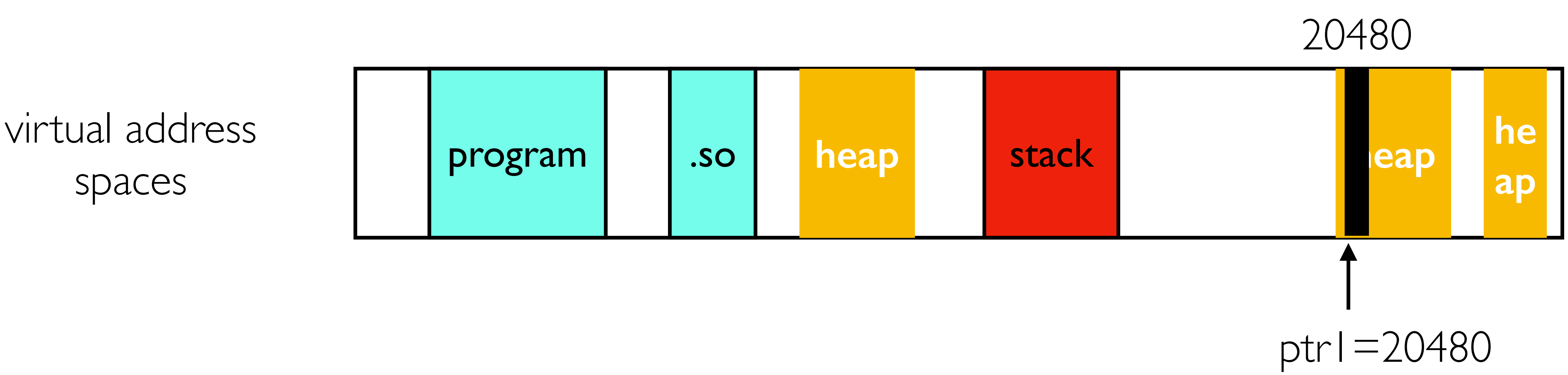
Review: Address Space

- 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

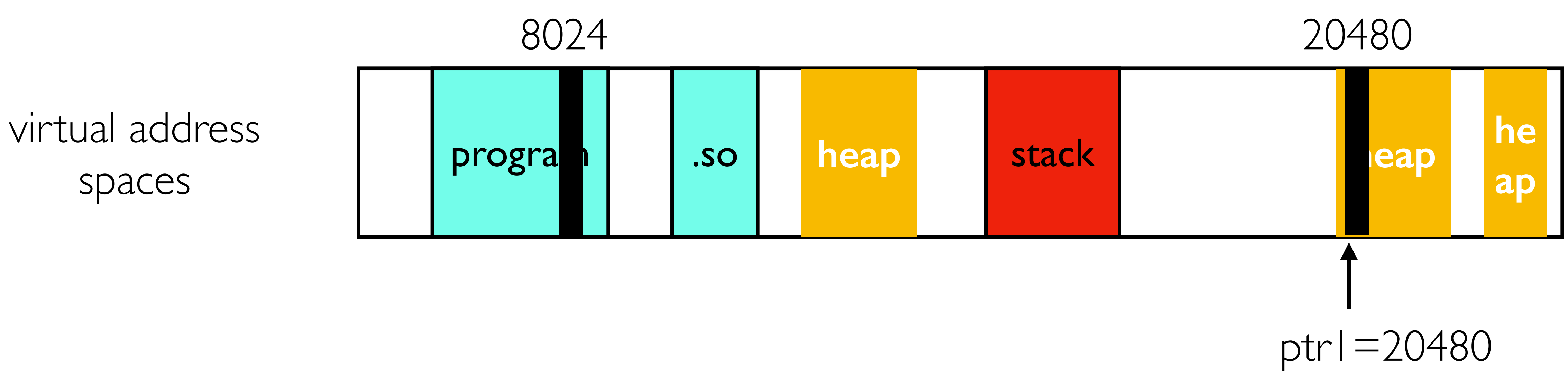
- our code (functions live in a program and possibly shared libraries)
- each thread has a stack pointer (to code) and a contiguous stack (for local variables)
- non-contiguous heap is shared between threads




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

Function Code Lives in Memory Too

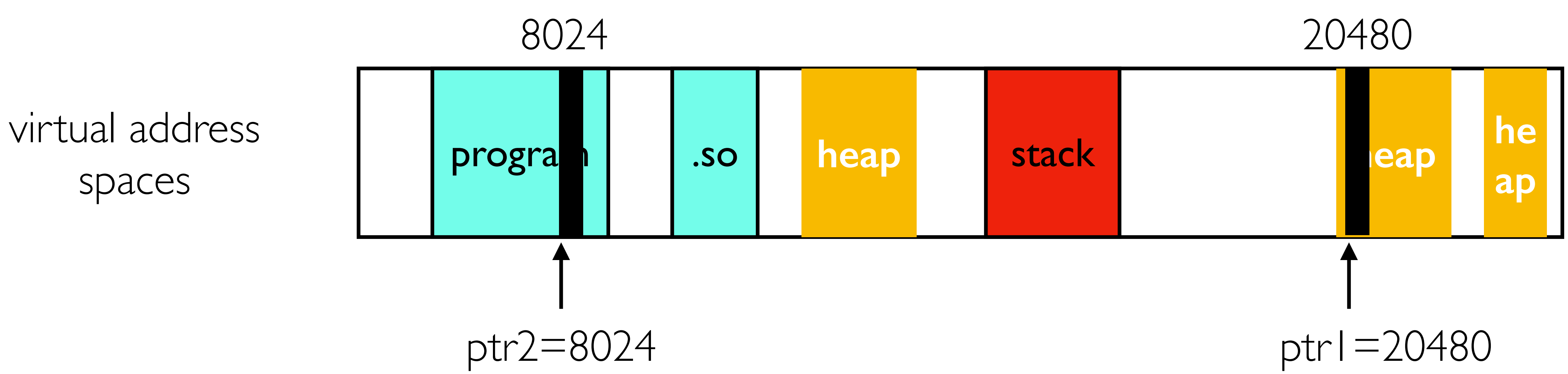
- an offset into the address space (i.e., "address") corresponds to function code
- that address can be stored in a pointer (a function pointer)
- function pointers can be used to call functions




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

Function Code Lives in Memory Too

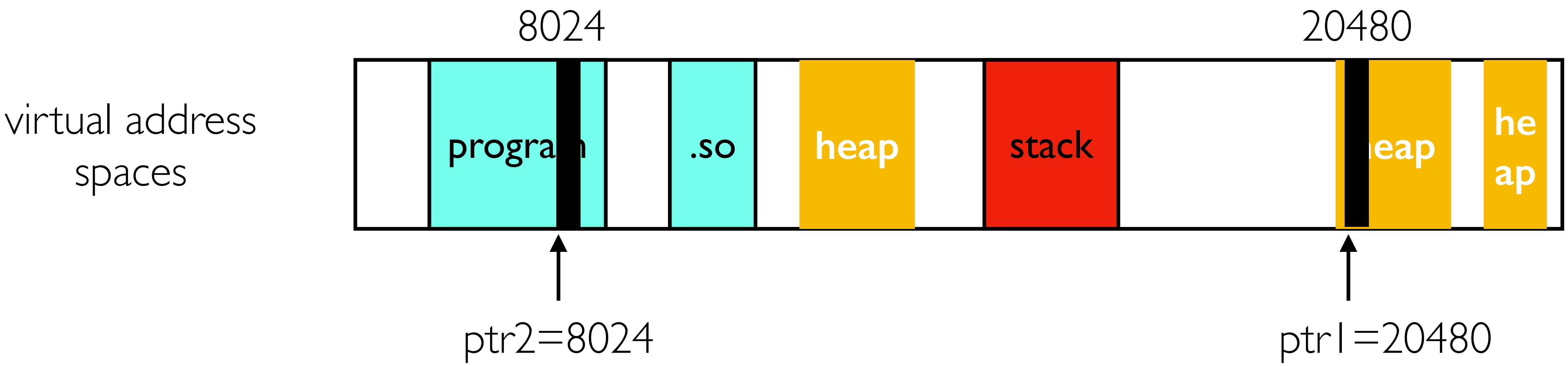
- an offset into the address space (i.e., "address") corresponds to function code
- that address can be stored in a pointer (a function pointer)
- function pointers can be used to call functions




```
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) {...}
```

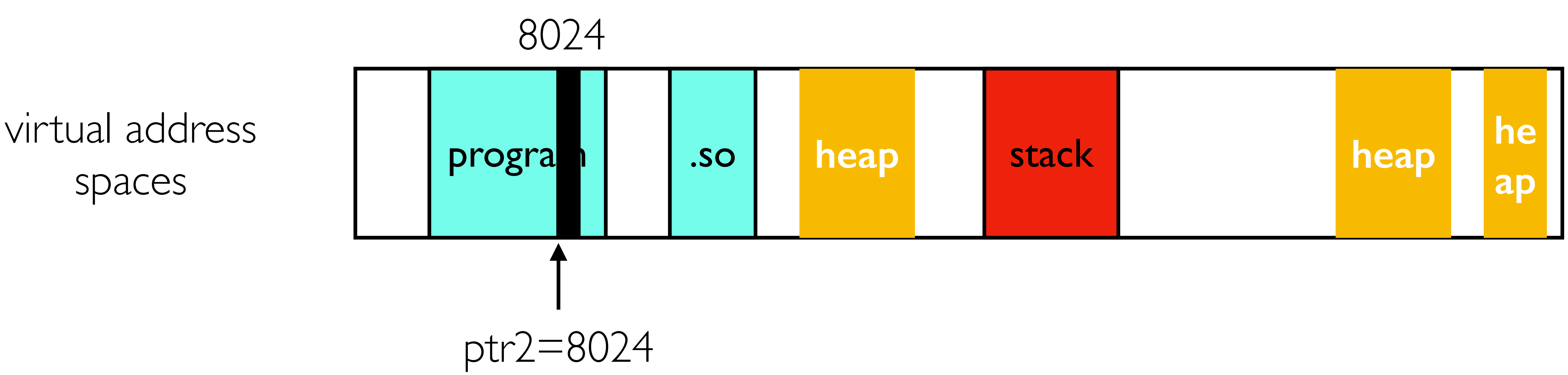
```
auto ptr1 = new MyClass;
```

```
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;
```

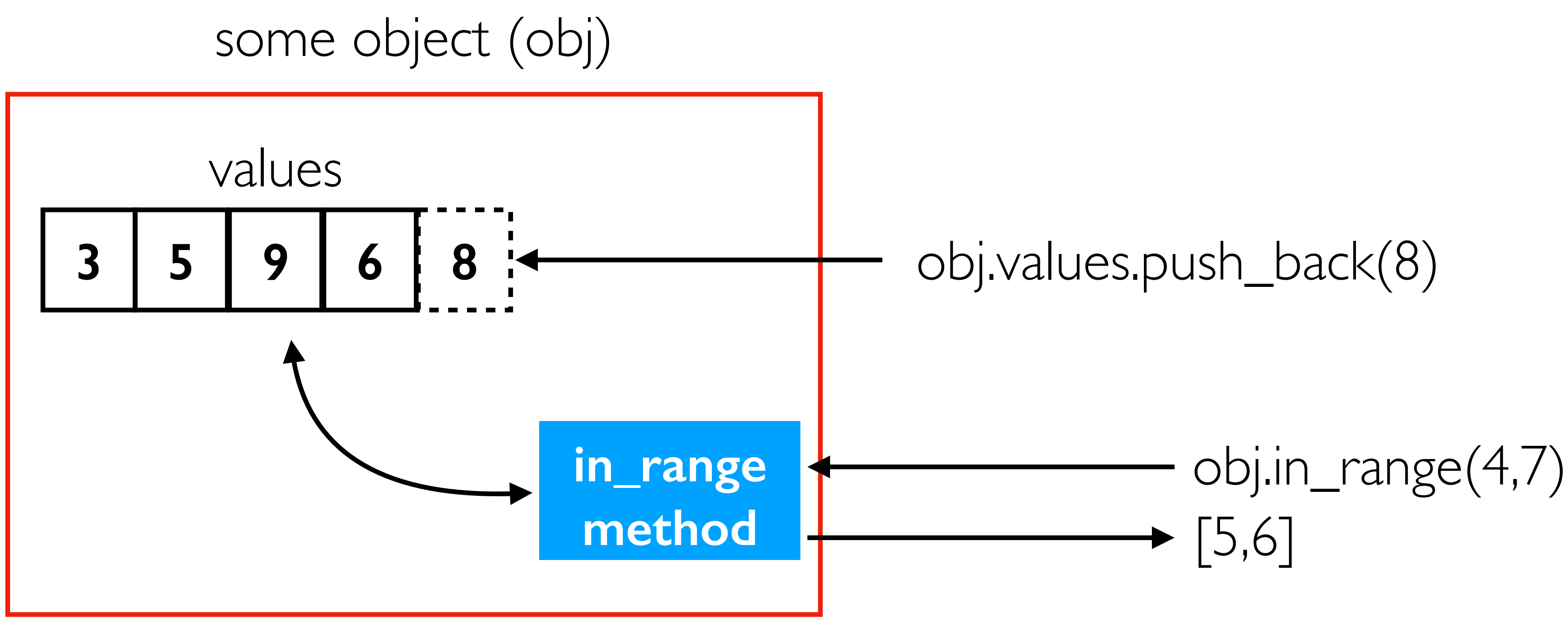
```
// without auto
void (*ptr2)(int) = f;
```

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

Passing Func Pointers to Funcs Enables Customizable Behavior

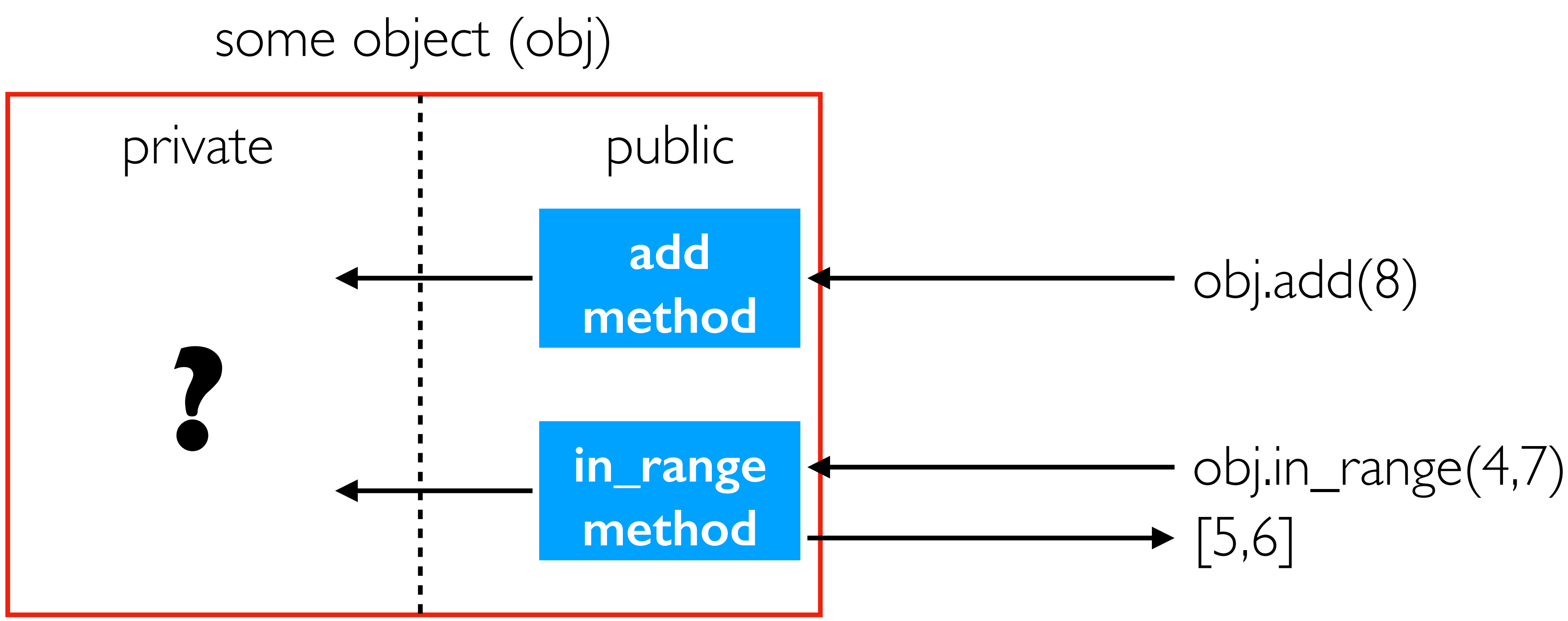
```
bool CompareAlpha(string x, string y) {
  return x < y;
}

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



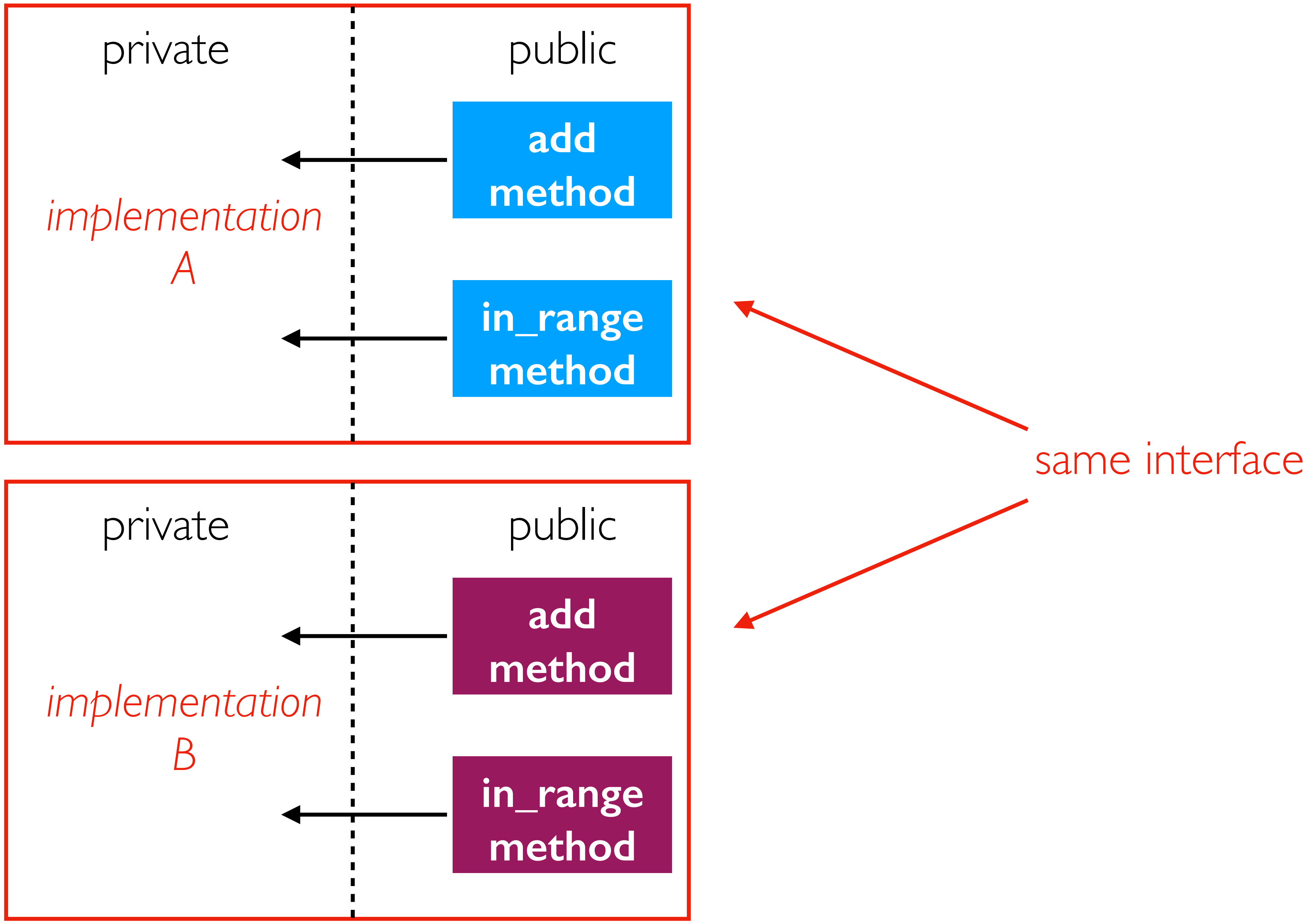
- if we add frequently and call in_range rarely, this implementation is good
- what if we call in_range frequently? Can we improve the library without breaking all the programs that use the library?

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();
}
```

```
animal.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