CSE 333
Section 7

Smart Pointers, C++, and Inheritance

Ever have a moment like this when programming?

UNIVERSITY of WASHINGTON

Logistics

- **Exercise 9**
  - Due **Wednesday** (11/15) @ 10 pm
- **HW3**
  - Partner matching form due **11/16** @ 10 pm
  - Due **Thursday** (11/23) @ 10 pm
    - Relatively long HW, so please get started if you haven’t already

Smart Pointers!

Review: Smart Pointers

- **`std::unique_ptr`** (Documentation) – Uniquely manages a raw pointer
  - Used when you want to declare unique ownership of a pointer
  - Disabled `cctor` and `op=`
- **`std::shared_ptr`** (Documentation) – Uses reference counting to determine when to delete a managed raw pointer
  - **`std::weak_ptr`** (Documentation) – Used in conjunction with `shared_ptr` but does **not** contribute to reference count

Using Smart Pointers

- Treat a smart pointer like a **normal (raw) pointer**, except now you **won't** have to use **`delete`** to deallocate memory!
  - You can use `*`, `->`, `[]` as you would with a raw pointer!

- **Initialize** a smart pointer by passing in a pointer to **heap memory**:

```
unique_ptr<int[]> u_ptr(new int[3]);
```

  - For **`shared_ptr`** and **`weak_ptr`**, you can use cctor and op= to get a copy

```
shared_ptr<int[]> s_ptr(another_shared_ptr);
```

Using Smart Pointers cont.

- Want to transfer ownership from one `unique_ptr` to another ?

```
unique_ptr<T> V = std::move(unique_ptr<T> U);
```

- Want to get the reference count of a `shared_ptr`?

```
int count = s.use_count();
```

- Want to convert your `weak_ptr` to a `shared_ptr`?

```
std::shared_ptr s = w.lock();
```

Exercise 1

Exercise 1

Change the following code to use smart pointers. Should each field be a unique, shared or weak pointer?

```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node): value(val), next(node) {}

  ~IntNode() { delete value; }

  int* value;
  IntNode* next;
};
```

Exercise 1

```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node) :
   value(unique_ptr<int>(val)), next(shared_ptr<IntNode>(node)) {}

  ~IntNode() { delete value; }

  unique_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

Exercise 1

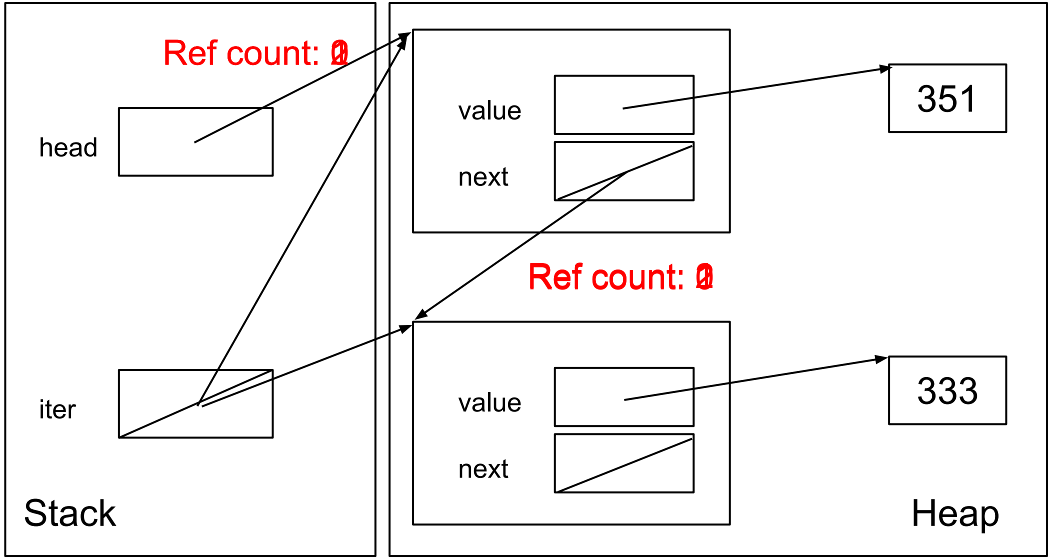
```
#include <memory>
using std::shared_ptr;
using std::unique_ptr;
using std::weak_ptr;

struct IntNode {
  IntNode(int* val, IntNode* node) :
   value(unique_ptr<int>(val)), next(shared_ptr<IntNode>(node)) {}

  ~IntNode() { delete value; }

  unique_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

Example: Client Code




```cpp
#include <iostream>

using std::cout;
using std::endl;
using std::shared_ptr;

int main() {
  shared_ptr<IntNode> head(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333), nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Example: Client Code

Nothing left on the heap!

```
#include <iostream>

using std::cout;
using std::endl;
using std::shared_ptr;

int main() {
  shared_ptr<IntNode> head(new IntNode(new int(351), nullptr));
  head->next = shared_ptr<IntNode>(new IntNode(new int(333), nullptr));
  shared_ptr<IntNode> iter = head;
  while (iter != nullptr) {
    cout << *(iter->value) << endl;
    iter = iter->next;
  }
}
```

Inheritance

Inheritance

- Motivation: Better modularize our code for similar classes!

- The public interface of a derived class inherits all **non-private** member variables and functions (**except** for `ctor`, `cctor`, `dtor`, `op=`) from its base class
  - *Java analogue*: A subclass inherits from a superclass

- Aside: We will be only using **public, single** inheritance in CSE 333

Polymorphism

- **Polymorphism** allows for you to access objects of related types
  - Allows interface usage instead of class implementation

- **Dynamic dispatch**: Implementation is determined *at* **runtime** via lookup
  - Allows you to call the **most-derived** version of a function
  - Generally want to use this when you have a derived class

- `virtual` replaces the class's default **static dispatch** with **dynamic dispatch**
  - Static dispatch determines implementation at compile time

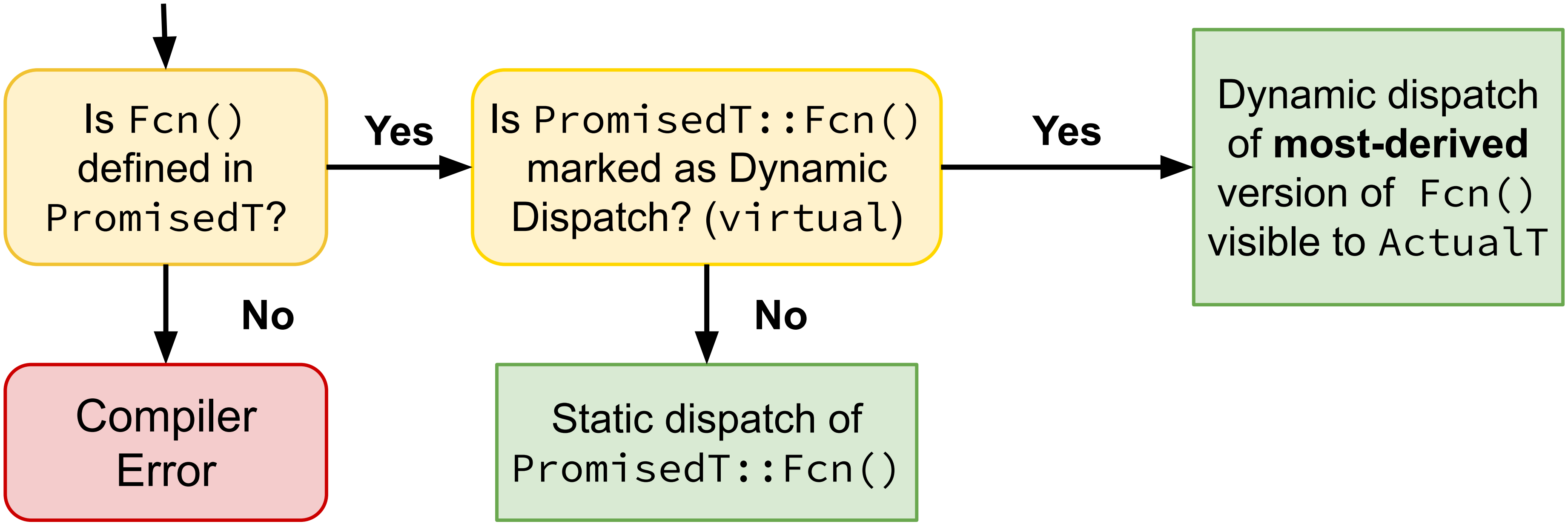
Dynamic Dispatch: Style Considerations

- Defining Dynamic Dispatch in your code base
  - Use `virtual` **only once** when first declared in the base class
    - Although in older code bases you may see it repeated on functions in subclasses
  - All derived classes of a base class should use `override` to get the compiler to check that a function overrides a virtual function from a base class

- Use `virtual` for destructors of a base class – Guarantees all derived classes will use dynamic dispatch to ensure use of appropriate destructors

Dispatch Decision Tree

```
PromisedT* ptr = new ActualT();
ptr->Fcn();  // which version is called?
```



Exercise 2

Exercise 2: static, dynamic, or error?

```cpp
class Base {
  void Foo();              // static dispatch
  void Bar();              // static dispatch
  virtual void Baz();      // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();      // dynamic (for more derived)
  void Bar() override;     // compiler error
  void Baz();              // still dynamic (sticky!)
};
```

Exercise 2: static, dynamic, or error?

```
class Base {
  void Foo();               // static dispatch
  void Bar();               // static dispatch
  virtual void Baz();       // dynamic dispatch
};

class Derived : public Base {
  virtual void Foo();       // dynamic (for more derived)
  void Bar();               // static dispatch
  void Baz() override;      // still dynamic (sticky!)
};
```

Abstract Classes

Abstract Classes

- Pure `virtual` Functions – Functions without any implementation
  - Declaration Example: `virtual int foo() = 0;`
  - Used for creating an interface of a function

- **Abstract Classes** are those with one or more pure `virtual` functions
  - Creates an interface for the client to use without knowing its details
  - **Requires** a derived class to implement its functionality (cannot itself be instantiated)

- Often used like an interface!

  Usage Example: `AbstractClass* a = new DerivedClass(params);`

Example Abstract Class/Derived Class

```
using std::string;

class Fruit {
 public:
  Fruit() = default;
  virtual ~Fruit() {}

  // A fun fact
  virtual string FunFact() = 0;

};
```

```
using std::string;

class Banana : public Fruit {
 public:
  Banana() = default;
  virtual ~Banana() = default;

  string FunFact() override {
    return “It’s a berry”;
  }
};
```

Exercise 3

Exercise 3A: Abstract Animals

Create an `Animal` Abstract class. It should have a `protected` member `legs` variable and a `public num_legs` pure `virtual` function. Try to use good style!

Exercise 3A: Abstract Animals

Create an `Animal` Abstract class. It should have a `protected` member `legs` variable and a `public num_legs` pure `virtual` function. Try to use good style!

```
class Animal {
 public:
  Animal() = default;
  virtual ~Animal() {}
  virtual int num_legs() const = 0;
 protected:
  int     legs;
};
```

Exercise 3B: Create an Animal Derived class

Now that you have made an abstract `Animal` class, try to make a implementation with a derived class of `Animal`.

This is an open-ended question, so you are free to be imaginative with your implementation of the abstract `Animal` class!

Exercise 3B: Create an Animal Derived class

```
class Dog : public Animal {
 public:
  Dog(int legs, string breed) : Animal(), legs(legs), breed(breed) {}
  virtual ~Dog() {}
  int num_legs() const override {
    return legs;
  }
  virtual int get_breed() const {
    return breed;
  }
 protected:
  string breed;
};
```

Casting

Different Flavors of Casting

- ```
  static_cast<type_to>(expression);
  ```
  Casting between related types, checked at compile time.

- ```
  dynamic_cast<type_to>(expression);
  ```
  Casting pointers of similar types (only used with inheritance), checked at runtime.

- ```
  const_cast<type_to>(expression);
  ```
  Adding or removing **const**-ness of a type

- ```
  reinterpret_cast<type_to>(expression);
  ```
  Casting between incompatible types of the **same size** (doesn’t do float conversion)

Tips with Casting

- Style: Use C++ style casting in C++
  - Tradeoff: Extra programming overhead, but provides **clarity** to your programs
  - Be **explicit as possible** with your casting! This means if you notice multiple operations in an implicit cast, you should explicitly write out each cast!

- Read documentation of casting on which casting to use
  - Documentation: https://www.cplusplus.com/articles/iG3hAqkS/
  - The purpose of C++ casting is to be less ambiguous with what casts you're using

Thanks for coming to section!