333 Section SOLUTIONS 7 - C++ Casting and Inheritance

C++ Smart Pointers

std::shared_ptr (Documentation) – Uses reference counting to determine when to delete a managed raw pointer

- Most commonly used type of smart pointer in practice
- **std::weak_ptr** (Documentation) – Used in conjunction with shared_ptr but does **not** contribute to reference count

std::unique_ptr (Documentation) – Uniquely manages a raw pointer

- Used when you want to declare unique ownership of a pointer
- Disabled cctor and op=

Exercise 2

"Smart" LinkedList

Consider the `IntNode` struct below. Convert the `IntNode` struct to be "smart" by using `shared_ptr`.

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

template <typename T>
struct IntNode {
  IntNode(int* val, IntNode* node): value(shared_ptr<int>(val)),
                                    next(shared_ptr<IntNode>(node)){}
  ~IntNode() {delete value;}
  shared_ptr<int> value;
  shared_ptr<IntNode> next;
};
```

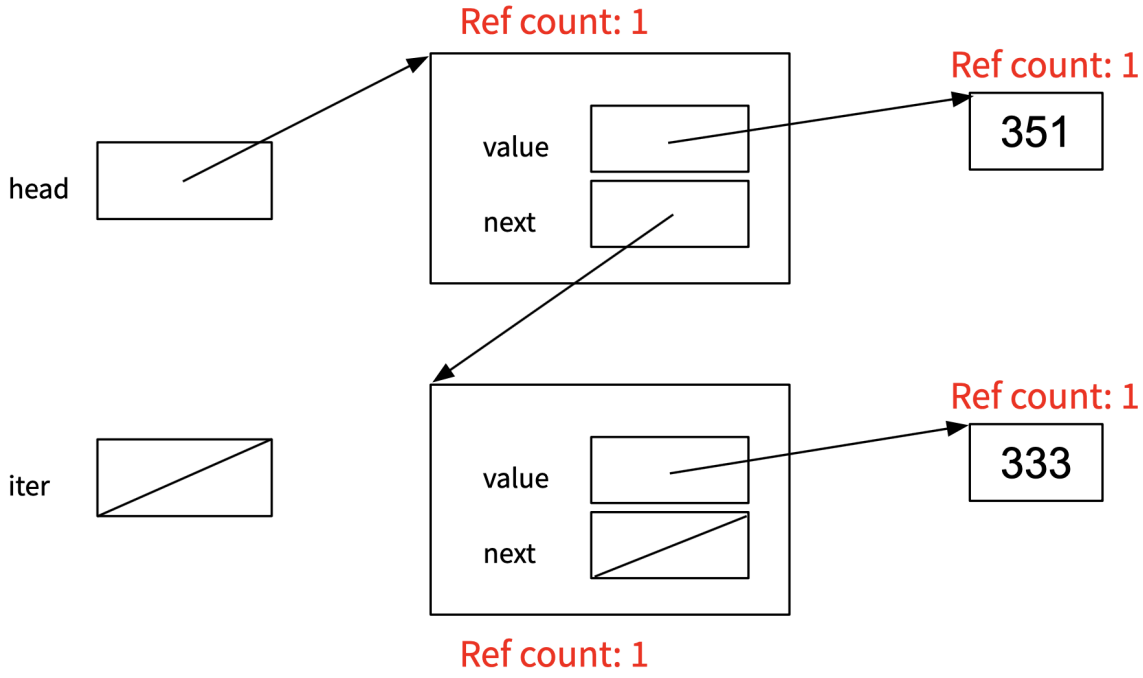
After the conversion, draw a memory diagram with the reference count for blocks of memory.

```
#include <iostream>

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

int main() {
  shared_ptr<IntNode> head =
      shared_ptr<IntNode>(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;
  }
}
```



Casting in C++

While in C++, we want to use casts that are more explicit in their behavior. This gives us a better understanding of what happens when we read our code, because C-style casts can do many (sometimes unwanted) things. There are four types of casts we will use in C++:

- `static_cast<type_to>(expression);`
  Casting between related types
- `dynamic_cast<type_to>(expression);`
  Casting pointers of similar types (only used with inheritance)
- `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)

Exercise 2

For each of the following snippets of code, fill in the blank with the most appropriate C++ style cast. Assume that we have the following classes defined:

`class Base {`
` public:`
`  int x;`
`};`
	`class Derived : public Base {`
	` public:`
	`  int y;`
	`};`

```
int64_t x = 0x7ffffffffe870;
char* str = reinterpret_cast<char*>(x);

void foo (Base* b) {
   Derived* d = dynamic_cast<Derived*>(b);
   // additional omitted code
}

Derived* d = new Derived;
Base* b = static_cast<Base*>(d);

double x = 64.382;
int64_t y = static_cast<int64_t>(x);
```

Inheritance in C++

Inheritance

A **Derived** class inherits from a **base** class (*Similar to*: A subclass inherits from a superclass)

- A derived class Inherits all **non-private** member variables and functions (**except** for ctor, cctor, dtor, op=)
- Aside: We will be only using **public** inheritance in CSE 333

Inheritance in HW3

Base Class: HashTableReader	Derived Classes
• `list<IndexFileOffset_t> LookupElementPositions( HTKey_t hash_val) const;` • `FILE* file_;` • `IndexFileOffset_t offset_;` • `BucketListHader header_;`	• `IndexTableReader` – Reads index table • `DocIDTableReader` – Reads DocID Table • `DocTableReader` – Reads DocTable • `FileIndexReader` – Reads File's Index

Abstract Class Examples

Fruit Abstract Class

```
#include <string>
using std::string;

class Fruit {
 public:
  Fruit() = default;
  virtual ~Fruit() {}
  // A fun fact
  virtual string FunFact() = 0;

};
```

Banana Derived Class

```
#include <string>
using std::string;

class Banana : public Fruit {
 public:
  Banana() = default;
  virtual ~Banana() = default;
  string FunFact() override {
    return "It's a berry";
  }
};
```

Style Considerations

- Use virtual **only once** when first defined in the base class
- All derived classes of a base class should use **override** to check at compile time that a function uses dynamic dispatch
- Call dtors of a base class as `virtual` – Guarantees all derived classes will use dynamic dispatch for their destructors

Exercise 3

Exercise 3A – Create an Animal Abstract class. It should have a protected member legs variable and a public num_legs member function. Try to use good style!

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

Exercise 3B

Now that you have made an abstract Animal class, try to make an 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!

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