333 Section SOLUTIONS 6 - C++ Casting and Inheritance

C++ Smart Pointers

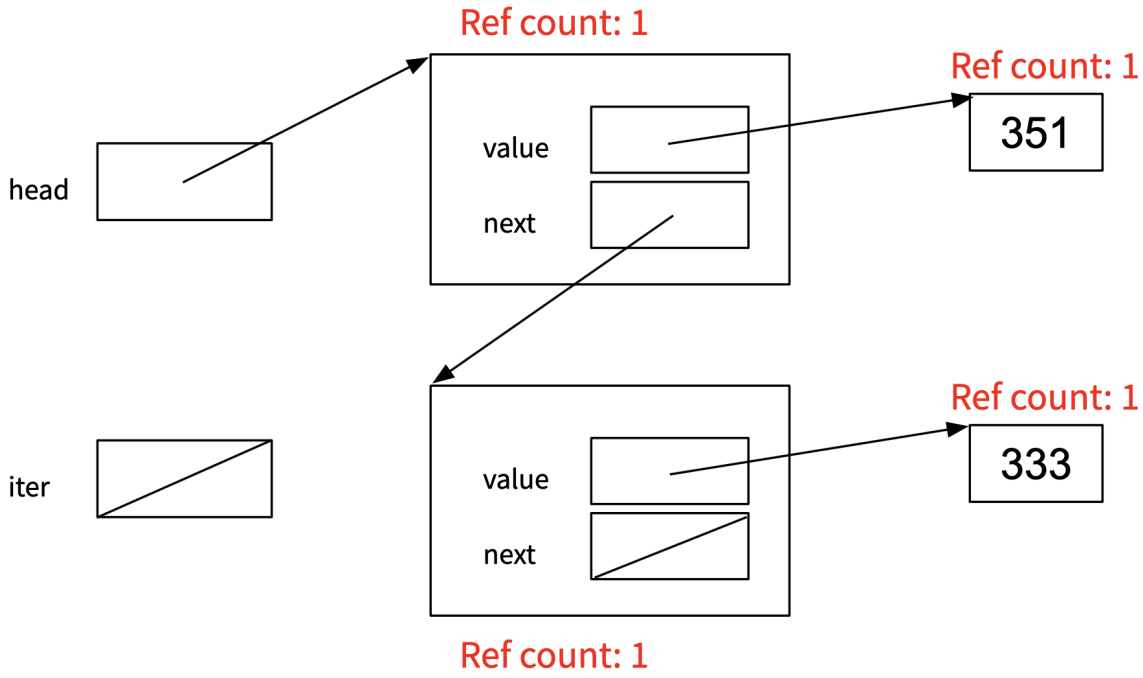
Exercise 1 - **"Smart" LinkedList**

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;
  }
}
```


Ref count: 1
Ref count: 1
351
head
value
next
iter
Ref count: 1
333
value
next
Ref count: 1

Casting in C++

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++

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;
};
```