CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Paper Exercise 4
Due Tuesday, March 10th at 4:30 P.M. P.S.T.
Name: Chih-Hsuan (Carolyn) Kao
SID: 06366163; chkao831@stanford.edu

Question 0: Lecture 10 Before move semantics were added in C++11 there were a lot of issues around using pointers. One way to solve this was to use RAII and wrap every pointer in a class to ensure that it would be properly cleaned up when the pointer went out of scope and prevent memory leaks. The code below implements one of these **smart pointers**.

```
template<class T>
class WrapPtr {
  T* ptr_;

  public:
  //Assign the T pointer to the member object.
  WrapPtr(T* ptr):ptr_(ptr) {

  }
  // Call the destructor and clean up the pointer resource.
  ~WrapPtr() {
    delete ptr_;
  }

  // Define the regular pointer operations.
  T& operator*() const { return *ptr_; }
  T* operator->() const { return ptr_; }
};
```

However, this class comes with certain difficulties. What goes wrong in the following code snippets?

a

```
void add_one(WrapPtr<int> obj) {
  *obj += 1;
}

int main() {
  // Create an int on the heap and wrap it in a smart pointer.
  WrapPtr<int> A(new int {212});
  add_one(A);
  std::cout << *A << std::endl;
  return 0;
}
```

Solution:

When we do `WrapPtr<int> A(new int 212);` on line 7, we have initialized a pointer on local scope within `main()` with corresponding memory allocation on heap. Then, when we do `add_one(A);` on line 8, the problem with the original code is that we actually call the pointer's default copy constructor, which performs copy-by-reference to the underlying object on heap and fails to carry out proper memory allocation as we leave the function `add_one()` by the end of function call. Without code revision, the original output with my added debug message is

```
Original Address of A: 0x7ffeefbff4b0
Copied (Inside Func) Address of A: 0x7ffeefbff498
Destructing...0x7ffeefbff498
```

//Note: Here, we have left the function `add_one()` with the `int` object on heap already deleted, so dereferencing the original smart pointer back in `main()` would result in an undefined behavior, since it's not a valid approach. Hence, the below message (of dereferenced value) from here is a result of an undefined behavior...

```
Dereferenced Value: 213
Destructing...0x7ffeefbff4b0
malloc: *** error for object 0x10070b980: pointer being freed was not allocated
```

In this case, to resolve such an issue, a revision would then be passing in the original pointer to function by reference like `void add_one(WrapPtr<int>& obj)` such that we only have one pointer throughout the process. This is verified by

```
Original Address of A: 0x7ffeefbff4c0
Copied (Inside Func) Address of A: 0x7ffeefbff4c0
Dereferenced Value: 213
Destructing...0x7ffeefbff4c0
```

On the other hand, although error-less, such an approach might not be a good idea as it does not align with the idea of a shared pointer, which shares an ownership of another pointer as the following,

```cpp
#include <iostream>

void add_one_shared(std::shared_ptr<int> obj) {
    std::cout << "Copied Address of S: " << &obj << std::endl;
    *obj += 1;
}

int main() {
    std::shared_ptr<int> S(new int {212});
    std::cout << "Original Address of S: " << &S << std::endl;
    add_one_shared(S);
    std::cout << "Final Address of S in main(): " << &S << std::endl;
    std::cout << "Dereferenced Value: " << *S << std::endl;
    return 0;
}
```

with the output of

```
Original Address of S: 0x7ffeefbff4b0
```

`Copied Address of S: 0x7ffeefbff488` //leaving func, copy implicitly destroyed
`Final Address of S in main(): 0x7ffeefbff4b0`
`Dereferenced Value: 213` //obj implicitly destroyed when terminating main()

b

```cpp
WrapPtr<std::string> create_message(bool greet)
{
  if (greet) {
    WrapPtr<std::string> message {new std::string {"hello"}};
    return message;
  } else {
    WrapPtr<std::string> message {new std::string {"goodbye"}};
    return message;
  }
}

int main()
{
  // Use a function to create a message
  WrapPtr<std::string> message = create_message(true);
  std::cout << *message << std::endl;
  return 0;
}
```

Solution:

Here, my compiler gives me the same error message as `malloc: *** error for object 0x1006344c0: pointer being freed was not allocated`. Then I added some debug messages in `main()`, `create_message()`, and destructor to see what's going on. The following messages are shown
`In-func Address: 0x7ffeefbff430`
`destructing...0x7ffeefbff430`
`destructing...0x7ffeefbff4a8` //error!
Here, I notice that a potential double-delete occurred, meaning that my program has somehow invoked delete when the pointed underlying object is no longer there. This happens when I end the function call from `create_message()` function with copied pointer pointing to underlying object `std::string "hello"` on heap deleted. Back in `main()`, when the destructor got called on the original pointer, the underlying object is no longer there on heap, so such an error occurred. This is also due to memory allocation, which could be further resolved by move semantics in question 1 below.

Question 1: Lecture 10 Move semantics have resolved the issues in Question 0(b). A move constructor or assignment allows you to transfer resources instead of having to copy them. This prevents two pointers from pointing to the same resource, so we can actually move `WrapPtr`s between functions without always passing references.

Improve the `WrapPtr` class so it uses move semantics.

Solution:

From lecture 10, we learned that in this case, we could define `WrapPtr`'s move constructor to "steal" the representation from its source.

```cpp
#include <iostream>

template <class T>
class WrapPtr {
    T* ptr_;

    public:
    WrapPtr(T* ptr)
        : ptr_(ptr){
            std::cout << "Ctor..." << &ptr_ << std::endl;
        }

    WrapPtr(WrapPtr&& a)
        : ptr_{a.ptr_}{
            a.ptr_ = nullptr;
            std::cout << "MCtor..." << &ptr_ << std::endl;
        }

    ~WrapPtr(){
        std::cout << "Dtor..." << &ptr_ << std::endl;
        delete ptr_;
    }

    T& operator*() const { return *ptr_; }
    T* operator->() const { return ptr_; }

};

WrapPtr<std::string> create_message(bool greet){
    if (greet) {
        WrapPtr<std::string> message {new std::string {"hello"}};
        std::cout << "create_message() Address: " << &message << std::endl;
        return message;
    } else {
        WrapPtr<std::string> message {new std::string {"goodbye"}};
        return message;
    }
}

int main() {
    WrapPtr<std::string> message = create_message(true);
    std::cout << "main() Final Address: " << &message << std::endl;
    std::cout << "main() Final Value: " << *message << std::endl;
44      return 0;
45  }
```

```
Ctor...0x7ffeefbff440
create_message() Address: 0x7ffeefbff440
MCtor...0x7ffeefbff4b8
Dtor...0x7ffeefbff440
MCtor...0x7ffeefbff4c0
Dtor...0x7ffeefbff4b8
main() Final Address: 0x7ffeefbff4c0
main() Final Value: hello
Dtor...0x7ffeefbff4c0
```

Here, I would like to anatomize the process as the following.
On line 41, when the `create_message(true)` gets called, we jump into line 31 with new pointer constructed with original constructor on line 8. Then as we `return message;` on line 33, just before this pointer got destroyed by the destructor as function call ends, we capture its representation using move semantics with move constructor on line 13. This explains the first 4 lines in the output above.
Then, back in `main()`, on line 41, we have
`WrapPtr<std::string> message = create_message(true);` again using moving constructor, we capture the value with a new pointer `message` on the LHS before the carrier pointer on the RHS got destroyed. Finally, the message is printed out correctly as shown. Eventually, as `main()` terminates by returning 0, the pointer `message` is destroyed.