```cpp
#include "Integer.h"
/// Lecture 54-55: Move Semantics - Implementation (C++11)
//////////////////////
// main.cpp (NO MOVE SEMANTICS)
#include <iostream>
// A function which adds two integer objects and returns the result by value.
// So we create a temporary object
Integer Add(const Integer &a, const Integer &b){
    Integer temp ; // create temporary object
    temp.SetValue(a.GetValue() + b.GetValue()) ; // compute the sum of a and b objects
    return temp; // Return temp by value.
}
int main() {
    Integer a(1), b(3) ;
    // Obtain the number from the temporary object returned by the function.
    // We store the sum in the variable a
    a.SetValue(Add(a,b).GetValue());  // Without the move semantics, A COPY of this temp object gets created, so that it can be accessed here. in some cases, this COPY of the temp object may not get created, and this is due to the compiler performing copy or move elision (Lecture 57).

    return 0;
}




//// Implementing the Move constructor
//////////////////////
// Integer.h
#pragma once
#include <iostream>
class Integer {
    int *m_pInt;
public:
```

```
33          //Default constructor
34          Integer();
35          //Parameterized constructor
36          Integer(int value);
37          //Copy constructor
38          Integer(const Integer &obj);
39          //MOVE CONSTRUCTOR!
40          Integer(Integer &&obj);
41          int GetValue()const;
42          void SetValue(int value);
43          ~Integer();
44
45      };
46      ////////////////////
47
48      // Integer.cpp
49      Integer::Integer() {
50          std::cout << "Integer()" << std::endl;
51          m_pInt = new int(0);
52      }
53
54      Integer::Integer(int value) {
55          std::cout << "Integer(int)" << std::endl;
56          m_pInt = new int(value);
57      }
58
59      Integer::Integer(const Integer & obj) {
60          std::cout << "Integer(const Integer&)" <<
•             std::endl;
61          m_pInt = new int(*obj.m_pInt);
62      }
63
64      // MOVE CONSTRUCTOR
65      Integer::Integer(Integer && obj) {
66          std::cout << "Integer(int&&)" << std::endl; //
•             move constructor invoked!
67          m_pInt = obj.m_pInt; // shallow copy.
68          obj.m_pInt = nullptr; // to let the other object
•             know that we have stolen the resources, we will
•             assign null to the pointer. So this way when
•             the other object is destroyed, our own
•             destructor will not crash. The call to delete
```

```
        in the destructor will be ignored, because that
        would be delete on a null pointer.
}

int Integer::GetValue() const {
    return *m_pInt;
}

void Integer::SetValue(int value) {
    *m_pInt = value;
}

Integer::~Integer() {
    std::cout << "~Integer()" << std::endl;
    delete m_pInt;
}
//////////////////
// UPDATED main.cpp (with move semantics)
#include <iostream>
// A function which adds two integer objects and
  returns the result by value.
// So we create a temporary object
Integer Add(const Integer &a, const Integer &b){
    Integer temp ; // create temporary object
    temp.SetValue(a.GetValue() + b.GetValue()) ; //
      compute the sum of a and b objects
    return temp; // Return temp by value.
}
int main() {
    Integer a(1), b(3) ;
    // Obtain the number from the temporary object
      returned by the function.
    // We store the sum in the variable a
    a.SetValue(Add(a,b).GetValue());
    // When we do this now, the move constructor is
      invoked. This means that the state of the temp
      object WAS NOT copied into the temporary that
      was accessed at the end of the function call.
      Instead, the STATE of temp was MOVED into that
      temporary.

    // Again, in some cases, you may not see the call
```

```
        to this move constructor, and that is because
        of move elision (lecture 57).
    // So, this is going to be a MUCH FASTER
        OPERATION than copying (deep copying), so in
        copy semantics we have to allocate new memory
        and then copy the data. Here, we don't have to
        allocate new resources - we just copy the data
        from the source object INTO the target object.
        We'll also have to implement the MOVE
        ASSIGNMENT OPERATOR, but we'll explore more of
        that in the section on Operator Overloading.

    // Hence, the rule of 3 that we talked about
        earlier now becomes a rule of 5.
    return 0;
}

```