```
1   ////////////////////////////////////////
2   // Lecture 56: Rule of 5 and 0
3   ////////////////////
4   // Integer.h
5   #pragma once
6   #include <iostream>
7   class Integer {
8       int *m_pInt;
9   public:
10      //Default constructor
11      Integer();
12
13      //Parameterized constructor
14      Integer(int value);
15
16      //Copy constructor
17      Integer(const Integer &obj);
18
19      //Move constructor
20      Integer(Integer &&obj);
21
22      //Copy assignment Operator (Section 6 - Operator
        Overloading)
23      Integer & operator=(const Integer &obj) ;
24
25      //Move assignment (Section 6 - Operator
        Overloading)
26      Integer & operator=(Integer && obj) ;
27
28      int GetValue()const;
29      void SetValue(int value);
30      ~Integer();
31
32  };
33
34  ////////////////////
35  // Integer.cpp
36  #include "Integer.h"
37  #include <iostream>
38  Integer::Integer() {
39      std::cout << "Integer()" << std::endl;
40      m_pInt = new int(0);
```

```
41    }
42
43    Integer::Integer(int value) {
44        std::cout << "Integer(int)" << std::endl;
45        m_pInt = new int(value);
46    }
47
48    Integer::Integer(const Integer & obj) {
49        std::cout << "Integer(const Integer&)" <<
          std::endl;
50        m_pInt = new int(*obj.m_pInt);
51    }
52
53    Integer::Integer(Integer && obj) {
54        std::cout << "Integer(Integer&&)" << std::endl;
55        m_pInt = obj.m_pInt;
56        obj.m_pInt = nullptr;
57    }
58     // Copy Assignment operator
59    Integer& Integer::operator=(const Integer& obj) //
          (Section 6 - Operator Overloading)
60    {
61        std::cout << "operator=(const Integer& obj)" <<
          std::endl;
62        if(this == &obj){
63            return *this ;
64        }
65        delete m_pInt ;
66        m_pInt = new int(*obj.m_pInt);
67        return *this ;
68    }
69     // Move Assignment operator
70    Integer& Integer::operator=(Integer&& obj) //
          (Section 6 - Operator Overloading)
71    {
72        std::cout << "operator=(Integer&& obj)" <<
          std::endl;
73        if(this == &obj){
74            return *this ;
75        }
76        delete m_pInt ;
77        m_pInt = obj.m_pInt;
```

```
78          obj.m_pInt = nullptr;
79          return *this ;
80      }
81
82      int Integer::GetValue() const {
83          return *m_pInt;
84      }
85
86      void Integer::SetValue(int value) {
87          *m_pInt = value;
88      }
89
90      Integer::~Integer() {
91          std::cout << "~Integer()" << std::endl;
92          delete m_pInt;
93      }
94
95      ////////////////////
96      // main.cpp
97      // To demonstrate how the compiler synthesises these
          functions, we create
98      // 1 more class.
99      // We create a class called 'Number' and add Integer
          as a member of this Number class.
100     // Then add a parameterised constructor
101     #include "Integer.h"
102     class Number{
103         Integer m_Value{} ;
104     public:
105         Number()=default ;
106         Number(int value):m_Value{value}{
107         }
108         // Since the integer is a  member of the number
              class, when we use the instances of Number in
              expressions that require copying or moving, the
              compiler will automatically synthesise the
              corresponding copy and move operations for the
              Number class.
109         // Since we are logging the calls to the
              functions in the Integer class, we will be able
              to know what functions are synthesized for the
              Number class because any function that is
```

```
        synthesised will internally call any function
        of the integer class.

    // It is not possible otherwise to see if the
        compiler synthesised the copy and move
        operations.
};
int main(){
    Number n1 ;
    auto n2{n1} ; // When we run this, we should be
        able to see the call to the copy constructor of
        Integer, and that's because the compiler
        synthesises the copy constructor in Number that
        will internally invoke the copy constructor of
        Integer.
    n2 = n1 ; // In the same way, if we use the
        assignment, the compiler will automatically
        synthesise the assignment operator (or the COPY
        assignment operator)for the Number class.

}
// When we run this, we should be able to see the
    call to the copy constructor of Integer, and that's
    because the compiler synthesises the copy
    constructor in Number that will internally invoke
    the copy constructor of Integer.

////////////////////////////////////
// main.cpp (V2)
// Now let's see how Move operations are synthesised:

#include "Integer.h"
class Number{
    Integer m_Value{} ;
public:
    Number()=default ;
    Number(int value):m_Value{value}{
    }
};
Number CreateNumber(int num){
    Number n {num};
    return n;
```

```
136     } // Number here is returned BY VALUE.
137
138     int main(){
139         auto n3 {CreateNumber(3)}; // In the first case,
            CreateNumber returns an r-value, so the
            compiler will choose MOVE constructor for the
            Number, but the Number class DOES NOT contain a
            move constructor! However, the Integer class
            contains a move constructor, so the compiler
            will automatically synthesise the move
            constructor for the Number class that WILL
            INTERNALLY INVOKE the move constructor of the
            Integer class.
140         n3 = CreateNumber(3); // same here for the move
            assignment operator.
141         // The Integer(Integer &&) move constructor is
            invoked.
142     }
143     // From this example, we can see that the compiler
        created the 5 functions for us, as long as there
        are no custom implementations of ANY of the five.
        What happens when we provide a custom
        implementation of the copy constructor?
144     ////////////////////////////
145     // main.cpp (V3)
146
147     class Number{
148         Integer m_Value{} ;
149     public:
150         Number()=default ;
151         Number(int value):m_Value{value}{
152         }
153         // User DefinedCopy Constructor
154         Number(const Number &n): m_Value{n.m_Value}{
155             // You may want to do this cuz perhaps you
                would like to log the call to the copy
                constructor,
156         }
157         // As a result, this will cause a CHANGE in this
            class because you have provided a copy
            operation, it doesn't matter whether it is a
            copy constructor or a copy assignment, the move
```

```
        operations will be deleted, so they will NOT be
        synthesised by the compiler.
};
int main(){
    auto n3 {CreateNumber(3)};
    n3 = CreateNumber(3);
}
// So when we run this, you wil not see a call to any
  move operation - in fact you will see calls to the
  copy constructor of the Integer class instead. So
  the move operations have become deleted! The same
  will happen if you define the Destructor - and
  there would be no calls to move operations.
////////////////////////////
// main.cpp (V4)
// What if we provide a custom implementation to a
  move operation ONLY?
class Number{
    Integer m_Value{} ;
public:
    Number()=default ;
    Number(int value):m_Value{value}{
    }
    // UDF Move function
    Number(Number &&n): m_Value{std::move(n.m_Value)}{

    }
};
int main(){
    auto n3 {CreateNumber(3)};
    n3 = CreateNumber(3);
}
// If we try to build this now, there will be a bunch
  of compiler errors. This is because the move
  assignment is also deleted.
// In addition, all the copy constructors and
  assignment also become deleted.

////////////////////////////
// main.cpp (V5)

class Number{
```

```
189         Integer m_Value{} ;
190     public:
191         Number()=default ;
192         Number(int value):m_Value{value}{
193         }
194         // UDF Move function
195         Number(Number &&n): m_Value{std::move(n.m_Value)}{
196         }
197         // What should we do in this case - suppose we
              have to provide this because there is some
              custom  implementation for the move()
              Constructor, and you do want to provide the
              other move operation. Should you manually
              implement the other move operation?
198         // You don't have to - instead you can simply use
              the 'default' specifier.
199         Number & operator=(Number&&)= default; // And
              this will cause the compiler to synthesis a
              default implementation of the move assignment.
              This implementation will also internally call
              the move assignment of the Integer class, even
              though we are using the default specifier, this
              has been 'defined' by the user. That is why
              this is still considered a custom
              implementation of the function.
200         // So now we have the move() assignments and the
              move operator for our class, but there is no
              support for copying.
201         // You will find many classes in STL that have
              support for move() and not for copying - one
              example is the UNIQUE pointer, whose objects
              can only be moved but not copied. So the
              default specifier is useful in these cases, and
              it is possible to use the default specifier on
              line 195.
202
203         // To finish up the rule of 5, we can use the
              default specifier to implement the copy
              operations:
204         Number(const Number &n) = default; // copy
              constructor.
205         Number & operator=(const Number&) = default; //
```

```
    Number& operator=(const Number&) = default; //
      copy assignment


};
int main(){
    auto n3 {CreateNumber(3)};
    n3 = CreateNumber(3);
}
// If we try to build this now, there will be a bunch
  of compiler errors. This is because the move
  assignment is also deleted.
// In addition, all the copy constructors and
  assignment also become deleted.

```