```

///////// Lecture 61: Operator Overloading II - Assignment Operator
// main.cpp
int main(){
    Integer a(1), b(3);
    Integer c;
    c = a; // this will create a shallow copy if we did not overload the assignment operator.
    // we have already seen examples of this in Section 5.
    // Create a copy assignment operator overload, to prevent shallow copying.
}

// Copy Assignment operator overload
// integer.h
class Integer{
public:
    ...
    Integer & operator =(const Integer &a);
}
//Integer.cpp
Integer & operator =(const Integer &a){
    delete m_pInt; // First delete the memmory for the existing object. otherwise this will cause a memory leak.
    m_pInt = new int (*a.m_pInt); // Then we allocate new memory and we assign the value from the other object.
    return *this; // Finally, return the current object, because *this is not local we can return this by reference.
}

// There is a small bug though - what if the variable is assigned to itself?
int main()
{
    Integer a(1);
    a = a;
    ...
```

```
}
// So if you do this, you get some garbage value - why? Because the first thing you do is that you delete the pointer. consequently you find yourself with undefined values.
// So the first thing you need to do in an assignment operator is to check for self assignment:
Integer & operator =(const Integer &a)
{
    if (this != &a) // only do the following if the addresses are different.
    {
        delete m_pInt;
        m_pInt = new int...;
    }
    // otherwise simply return the current object.
}

// Let's finish this video with the Rule of 5.
// Overloading move assignment
// integer.h
Integer & operator =(Integer && a);

// integer.cpp
Integer & Integer::operator=(Integer && a){
    if (this != &a){
        delete m_pInt;
        m_pInt = a.m_pInt;
        // Assign null to the pointer of the other object - DON'T FORGET!
        a.m_pInt = nullptr;
    }
    return *this;
}

///////// Lecture 62: Operator Overloading III - Global Overloads

// okay so this works because of type conversion. Just understand that this works for now.
int main(){
    Integer a(1), b(3);
    // It is possible to replace the second operand
      with a primitive type.
    Intger sum  = a + 5; // This still works because
      this integer gets converted into an Integer
      object through a process called type conversion
      (KIV - Lecture 67).

    Integer sum2 = 5 + a; // This doesn't work :( Why?
    // As a member function, the operator overload
      will be invoked by the object on the left hand
      side. For the second expression who will invoke
      this plus(+) operator, because the object on
      the LHS is NOT an integer object, so the
      program cannot run.

    // So, if the first operand of an overloaded
      operator HAS to be a primitive type then the
      operator should be overloaded as a global
      function. Overload this as a global function -
      overload this as an integer, and the second
      would be an integer object.
    // Something like this --> see below
}
Integer operator+(int x, const Integer &y){
    Integer temp;
    temp.SetValue(x + y.GetValue());
    return temp;
}
// Overloading the bitshift (<<) operator:
// The insertion operator is found in the ostream
  class and we cannot touch that class to overload
  the operator for our type.  That's why you'll have
  to overload this as a global function:
std::ostream & operator << (std::ostream &out, const
  Integer &a) // should not be constant because we
  have to insert the value of the integer into this
  object.
{
    out << a.GetValue();
    return out;
}
// this expression is resolved as, the compiler
```

```
invokes the call to operator insertion, cout is passed as an argument and then sum is passed as an argument. this entire expression returns an ostream object and that invokes in operator insertion and the endl manipulator is passed as well.

//We caan alsoverload the extraction operator for our class, so that we can directly write an expression like this:

int main(){
    Integer a;
    cin >> a; // extraction operator overloaded.
}

// let's implement the extraction operator overload:
// Return type would be istream by reference.
std::ifstream& operator >> (std::istream &input, Integer &a){
    int x;
    input >> x;
    return input; // Read and write directly to a.
}

/// Overloading the Function Call Operator:
// Very useful operator, used extensively in STL.
// We can overload this operator to perform any operation that we want on the object.

// In integer.h, define the overload.
void operator () (); // this operator can accept any number of arguments.

// In main.cpp, implement:
void Integer::operator() (){
    std::cout  << *m_pInt << std:: endl;
    return 0;
}

// In main.cpp, this is how you invoke functon calls - It can be used with templates to implement callbacks. (KIV - Section on Templates)
```

```
  callbacks. (RIP - section on templates)
int main(){
    Integer a(1), b(3);
    a(); // --> this prints the object!
    std:: cin >> a;
    std::cout << a << std::endl;
}

```