```cpp
// Lecture 49: Copy Constructor II
#include <iostream>
// So the problem is due to the copying of pointers.
// Why does this occur? If we allocate memory for a
  pointer and we want to create a copy of the pointer,
class Integer {
    int *m_pInt; // pointer as member
public:
    Integer(); // default
    Integer(int value); //parameterised
    // Copy constructor
    Integer(const Integer &obj); // you should pass
      this object by reference and NOT by value. If
      you pass this obj by value, then again the copy
      of this object will get created and that would
      invoke the copy constructor again.
    // So this call will go in a loop, and to avoid
      that, we have to pass this by reference.
    // Because we are passing this object by
      reference, to avoid the modification of the
      original object in this function, we should
      qualify the integer object with the const
      keyword.

    // In most of the cases, when objects are passed
      by reference into functions they are always
      qualified with const.

    //
    int GetValue()const; // gets
    void SetValue(int value); // set
    ~Integer(); // destructor - free the memory
      allocated for the integer pointer.

};

// Implementation
Integer::Integer() {
    std::cout << "Integer()" << std::endl;
    m_pInt = new int(0); // default
}

```

```
30  Integer::Integer(int value) {
31      std::cout << "Integer(int)" << std::endl;
32      m_pInt = new int(value); /
33  }
34
35  // Copy Constructor!!
36  Integer::Integer(const Integer & obj) {
37      std::cout << "Integer(const Integer&)" <<
      •   std::endl;
38      // Store the integer in the new address
39      m_pInt = new int(*obj.m_pInt);
40  }
41
42
43  int Integer::GetValue() const {
44      return *m_pInt;
45  }
46
47  void Integer::SetValue(int value) {
48      *m_pInt = value;
49  }
50
51  Integer::~Integer() {
52      std::cout << "~Integer()" << std::endl;
53      delete m_pInt;
54  }
55  // Case 2: Copy of the object is created because we
      •   are passing by value.
56  void Print(Integer i){}
57
58  // Case 3: copy of the object is created because we
      •   are returning by value.
59  Integer Add(int x, int y){ return Integer(x+y);}
60  // Driver code
61  // So the problem is due to the copying of pointers.
62  // Why does this occur? If we allocate memory for a
      •   pointer and we want to create a copy of the pointer,
63  int main(void)
64  {
65      int *p1 = new int(5);
66      // this is called a shallow copy
67      // If you create a copy like this, it will only
```

```
    copy the ADDRESS, so any change that we make to
    p1 or p2 is going to reflect in all other
    pointers that hold the same address.
// This is called a SHALLOW COPY.
int *p2 = p1;

// To create a DEEP COPY, allocate new memory and
    then copy the value at the address
int*p3 = new int(*p1);

Integer i (5); // creates an integer i
// Creating a DEEP COPY is what we'll have to do
    in the integer class and this means that we have
    to define a copy constructor.
// The copy of the object is created when you
    invoke the copy constructor like this
Integer i2(i);
// or when the object is passed into a function by
    value, or when a function returns an object by
    value.

// A copy also gets created when one object is
    assigned to another object - so if I assigned i2
    to i - like so:
i = i2; // this also creates a copy through copy
    assignment even though we have not provided any
    assignment operator in our class, the compiler
    synthesised one for us, just like it did  for
    the copy constructor and the default
    implementation of this assignment operator will
    perform a shallow copy of the state of i2 into
    i. We'll see in subsequent lectures how to
    implement this assignment operator and perform a
    deep copy. In all these cases, the copy of the
    object is created.

// In C++, we try to avoid creating copies of an
    object but in some cases we cannot avoid.  So to
    avoid the problems due to shallow copy we have
    to implement a user defined copy constructor.

```