```cpp
///////// Lecture 64 - Operator Overloading V (Smart Pointers)

#include "Integer.h"
#include <iostream>
#include <memory>
// In main.cpp
void CreateInteger(){
    Integer *p = new Integer; // create an integer object on heap
    p->SetValue(3); // set some value in the integer obect
    std::cout << p->GetValue() << std::endl; // display it in the console.
    delete p;
}
int main(){
    // Invoke function on main()
    CreateInteger();
    return 0;
}
// When you have large programs, it is not always possible to remember to call delete on the pointers, so that will lead to massive memory leaks in the program. There are a lot of tools out there to detect memory leaks in your program - and visual studio has a built in library that helps you detect a memory leak. Instead of using those tools, we can prevent memory leaks outright, and that is possible using an idiom called as
//                  RESOURCE ACQUISITION IS INITIALISATION
// this idiom was created by Bjarne Stroustrup and is also popular in some other languages . With this idiom, the lifetime of a resource is bound to a local object - so twhen the local object is destroyed, in its destructor it'll automatically release the resource. We can apply this idiom for the memory that we have allocated on the heap.

// We can create a class called IntPtr in main.cpp and we'll put a pointer of Integer object into it,
```

```
which will be initialised through the constructor of IntPtr and the destructor will free the memory.
class IntPtr{
    Integer *m_p;
public:
    IntPtr (Integer *p) : m_p(p){
    }
    ~IntPtr(){
        delete m_p;
    }
};

void CreateInteger()
{
    IntPtr p = new Integer;
    // We can replace the Integer * with IntPtr and this doesnt have to be a pointer, in fact it is going to be a local object. Obviously we cannot invoke SetValue() on this IntPtr because it is not a member of IntPtr.
    // So we can remove it for a moment, and if you run this now, you will see the constructor and destructor calls. That means the memory that we allocated on the heap using new is properly destroyed.
}

// But what use is this IntPtr if we cannot access the members of our Integer object? So we'll set up this IntPtr class in such a way that when its object is created the object will behave like a Pointer. That means we'll be able to access the member functions of the underlying resource using the arrow operator, and we'll do that by overloading the arrow operator.
// So the return type of arrow operator would be Integer pointer.

class IntPtr{
    Integer *m_p;
public:
    IntPtr (Integer *p) : m_p(p){
```

```
    }
    ~IntPtr(){
        delete m_p;
    }
    Integer* operator ->(){ // overloaded the arrow operator!
        return m_p;
    }
};
// Now in our CreateInteger() function we can set the value for the Integer:
void CreateInteger(){
    IntPtr p = new Integer;
    p->SetValue(3);
}
int main(){
    CreateInteger();
}

// If you run this, it builds fine and you can see, it also calls the destructor of Integer because at the end of the scope, the p object is destroyed, and the destructor calls delete on the underlying pointer. So using RAII, we have bound the resource. Our resource is the memory allocation that is bound to the lifetime of the local object and this local object is automatically destroyed at the end of the scope.

// We can also overload one more operator that would give the behaviour of a pointer to this object and that operator is the asterisk(*) operator.

Integer & operator *(){
    return *m_p;
}
// And we can even dereference Pointers
void CreateInteger(){
    IntPtr p = new Integer;
    (*p).SetValue(3);
}
// So all the places where we use a pointer can be
```

```
replaced with this object that uses the RAII idiom. This object p is a local object but it behaves like a pointer, and it ALSO automatically deletes the underlying memory, so we can say this object is a SMART POINTER.


// DEFINITION OF SMART Pointer
/*
A SMART POINTER behaves like a pointer but it automatically frees the memory. Unfortunately, this smart pointer can be used only with the Integer objects but C++ STL provides us with smart pointers that can be used to manage any kind of pointer. We'll look at those in the next video.
*/

////////// Lecture 65: Operator Overloading VI - Smart Pointers (C++11)
/*
C++ provides different kinds of smart pointers and they are defined in the STD header file: memory.

SMART PTR #1: STD UNIQUE_PTR.
The first smart pointer I want to talk about is the unique_ptr. This is a class template and it requires the type of the pointer that you want to manage.
*/
class Integer {...};
void CreateInteger(){
    std::unique_ptr<Integer> p (new Integer); // Specify the Integer type within the < > and then allocate memory for the integer.
    // When we run this NOW, we can see the memory was allocated and freed. this is because we used a smart pointer.

    // Unique pointer is used when you do not want to share the underlying resource. That means we CANNOT CREATE A COPY of the unique pointer. If you try to do so, the compiler will not allow
```

```
        you try to do so, the compiler will not allow it because its copy constructor is a DELETED FUNCTION.
    auto p2(p); // COMPILER ERROR!!! - attempting to reference a deleted function.
    // Even though you cannot create a copy of the unique_ptr, you can move it. It has move semantics only and does not support copy semantics.
}
// Even though you cannot create a copy of the unique_ptr, you can move it. It has move semantics only and does not support copy semantics.
void Process(std::unique_ptr<Integer> ptr){
    std::cout << ptr->GetValue() << std::endl;
}

void CreateInteger()
{
    std::unique_ptr<Integer> p (new Integer);
    // If we try to invoke the process function like this,
    *p.SetValue(3);
    Process(p); // it will not compile, because you're trying to pass the unique_ptr by value and THAT will create a copy.

    // Do this instead - this will invoke the move constructor and move the underlying resource into the Process() function.
    Process(std::move(p));
    // After this, p will no longer hold the resource.
    // That is why you should not use it after move, so the ownership will be transferred to ptr, and p will be set to a nullptr.
}
// SMART PTR #2: STD SHARED POINTER
// If you want to use this smart pointer in CreateInteger() after Process(), then that means you are SHARING THE UNDERLYING RESOURCE.
// Let's change our above code accordingly.
void Process(std::shared_ptr<Integer> ptr){
    std::cout << ptr->GetValue() << std::endl;
```

```
    std::cout << ptr->GetValue() << std::endl;
}

void CreateInteger()
{
    std::shared_ptr<Integer> p (new Integer);
    // If we try to invoke the process function like this,
    *p.SetValue(3);

    // Do this instead - this will invoke the move constructor and move the underlying resource into the Process() function.
    Process(p);
    // After this, p will no longer hold the resource.
    // That is why you should not use it after move, so the ownership will be transferred to ptr, and p will be set to a nullptr.
    // A shared pointer is used when you want to share the underlying resource with other parts of the code.
    // It internally implements some kind of reference counting and each time a COPY of a shared pointer is created, the reference count is incremented by 1. When a shared pointer is destroyed, the reference count is decremented and if the reference count becomes zero, then it releases the underlying resource, so we can still use the shared pointer p after the Process() function.

    // in this function, the reference count of the shared pointer is 2, and when the shared pointer ptr is destroyed at the end of the scope, that is decremented to 1, and the underlying resource will be released only at the end of the scope of this CreateInteger() function. When the object p gets destroyed, it will decrement the reference count by 1, which will go to zero and the memory will be released.

    // MODERN C++ emphasises use of smart pointers rather than raw pointers. That is why, it is
```

```
rather than raw pointers. That is why, it is
recommended that you should always use smart
pointers rather than raw pointers. This way you
can prevent memory leaks for your programs.

    // SMART POINTERS ARE DISCUSSED IN GREATER DEPTH
      IN THE NEXT SECTION.
}
```