```cpp
//////// Lecture 72: Unique Pointer

// // We will use the smart pointer: Unique Pointer.
// This pointer is used when this pointer is used when the underlying pointer does not have to be shared with other parts of the code,
//  nd it seems like we are not sharing out pointer with different parts of the code, so that's why we'll use the unique pointer here.

class Integer {
...
};


// SOURCE.CPP
// first we have to include the memory header file.
#include <memory>

Integer* GetPointer(){
    Integer* ptr = new Integer {};
    return p;
}
void Display(Integer *p)
{
    if (!p){return;}
    std::cout << p->Getvalue() << std::endl;
}
void Operate(int value)
{
    // Unique pointer is in the standard namespace
    // and it is also a class template.
    std::unique_ptr<Integer*> p(GetPointer(value)); // unique pointer has an explicit constructor, that is why we cannot use this assignment to initialise. Instead, we will use direct initialisation - like that ^
    // That's it.
    // Because GetPointer() may return a null pointer sometimes, although we have not written that implementation here, let's assume it can also return nullptr. in that case the underlying
    pointer inside the unique pointer will be null as well.
    // And before we use the underlying pointer, we have to compare it with null.
    // And, we had already written this code and this code will still work with the unique pointer, because the comparison operator == is overloaded for nullptr as the second parameter
    if (p == nullptr) {
        //p = new Integer{};
        p.reset(new Integer(value)); // reset will do 2 things - if the smart pointer object holds an existing pointer, that will be deleted first, and then it will take ownership of the new pointer.
    }
    p->SetValue(100); // Then you can see there is no change required here.


    // We are able to use this smart pointer just like a pointer, except it is NOT just a pointer, it is actually an object.
    // At the end of the scope, it will be automatically destroyed.
     // now we need to pass the smart pointer into the display function, but Display() accepts an integer pointer.
    // There are in some cases  where you will want to access the underlying pointer inside the unique_pointer, and in this case, you may use a function (method) called get().
    // get () will return the underlying pointer.
    Display(p.get());


    // delete p; // we are NOT ALLOWED TO CALL DELETE on this pointer, cuz remember its not a pointer, its an object that behaves like a pointer.
    // so remove this call to delete.
    p = nullptr; // this statement is VALID! Unique
    pointer has provided an overload of assignment operator that accepts nullptr as a parameter, that implementation of the assignment operator simply deletes the underlying pointer and makes the pointer variable null.
    //So this statement is just like calling delete on the raw pointer, and makes the pointer variable null.

    // Okay so now we need to assign a new pointer to the smart pointer, but remember you can't initialise the smart pointer using the assignment operator, but you can use the reset() method.
    p.reset(new Integer{});
    // reset will automatically delete the underlying Pointer, and take ownership of the new pointer.

    *p = __LINE__; // C-macro that expands into a line number.
    // There is no need of this statement now.
    // The dereferencing operator is overloaded by the unique pointer, and it returns the value at the address of the underlying pointer.
    // So in this case, the current line number is assigned to the integer object.


    // In the next line we will have to make a small change, where we use the .get() method.
    Display(p.get());
    // delete p; - we don't have to invoke delete.
    // Notice - that we don't have to deal with memory management at all!
}
// note that when you run this, the number of constructors and the number of destructors STILL MATCHES, though we don't have to use delete anymore.


// Add a new user defined function:
void Store(std::unique_ptr<Integer> p)
```

```cpp
{
    std::cout << "Storing data into file: " << p->getValue() << std::endl;
}
void Operate(int value)
{
    std::unique_ptr<Integer> p(GetPointer(value));
    ...
    ...
    Display(p.get());
    Store(p); // THIS DOESN'T WORK!
    // Error: Attempting to reference a deleted function. The copy constructor
    // of the unique_ptr is deleted.
    // Therefore, you cannot create a copy of the unique_ptr object.
}

// So how do we pass the unique pointer into this function? notice that we don't use this smart pointer after the Store() function anyway,
// it's at the end of the scope, this p is going to be destroyed.

// So we don't need it after the Store() function, so we can use the library function Move().
void Store(std::unique_ptr<Integer> p){}

void Operate(int value)
{
    ...
    Store (std::move(p)); // this works, and the underlying pointer is shifted to the function argument.
    // The function argument ptr will be destroyed when the function finishes executes,
    // and our pointer p gives up ownership of the underlying pointer.
    // After this statement, you should not try to access the pointer within the smart pointer.
    // Remember all pointers can be reassigned, and we can store a new pointer inside it by simply
```

```
      we can store a new pointer inside it by simply
      calling Reset().
    p.reset(new Integer{}); // remember?
}

// There is 1 more alternative - which is to pass the
  unique pointer by
// REFERENCE.
void Store(std::unique_ptr<Integer>& p){}
void Operate(int value)
{
    ...
    Store(p); // passed by reference and not value.
    // We no longer need to use std::move()
    // The advantage of that is that you can still
      use this pointer
    // after the store function
    Display(p); // this will work too! - if we move
      the smartptr this will not work.
}

```