```cpp
//////// Lecture 76: Weak Ownership
#include <iostream>
class Printer{
    int *m_pValue{};
public:
    void SetValue(int *p){
        m_pValue = p;
    }
    void Print(){
        std::cout << "Value i*" << *m_pValue << std::endl;
    }
};

int main(){
    Printer prn;
    int num{};
    std::cin >> num; // get number from user
    int *p = new int{num};
    prn.SetValue(p);
    prn.Print();
    delete p;
}
// Nothing wrong with this code here and it does work fine.


// Let's say there is more complex code now...

int main(){
    Printer prn;
    int num{};
    std::cin >> num; // get number from user
    int *p = new int{num};
    prn.SetValue(p);
    // Let's say we want to implement more complex logic, and there are a few conditions.
    // Let's assume we want to compare the value if it's greater than 10, then we no longer need this pointer (so we are going to delete it)
    if (*p > 10) {
        delete p;
```

```
            p = nullptr;
        }
        prn.Print();
        delete p; // This is to ensure that if delete is called again on the same pointer it will not do anything.
        // If we do not assign nullptr, this will lead to a double delete situation, and that may crash the program.
    }
    // So now, if we write a value that is smaller than 10, it works.
    // If you enter a value larger than 10, there is a runtime error as the printer will print out some garbage value.
    // This is because the printer is being assigned this pointer value over here
    prn.SetValue(p);
    // and afterwards, the pointer may be deleted over here:
    if (*p > 10) {
        delete p;
        p = nullptr;
    }
    // The memory address that this m_pValue has may be deleted.
    // So this m_pValue will point to invalid memory - that is memory that has been released.
    // Therefore within the class Printer, we need something within the print() function to check if the pointer is still valid or not.
    // Can we compare against null? Obviously that doesn't work, because when you assign nullptr to the p variable, the m_pValue isn't assigned to null!
    // That pointer variable m_pValue will NOT know about it because it's a different pointer variable, so there is NO WAY of checking whether the pointer is still valid or not.
    // We need some kind of communication between these pointers and manually it is extremely difficult to implement.
    // So what should we do here? let's try one thing -
```

```cpp
instead of using raw pointers, let's use smart pointers,
// and since we are sharing this pointer with the Printer class, we should use the shared_ptr.
#include <iostream>
#include <memory>
class Printer{
    std::shared_ptr<int> m_pValue{};
public:
    void SetValue(std::shared_ptr<int>p){
        m_pValue = p;
    }
    void Print(){
        std::cout << "Value i*" << *m_pValue << std::endl;
    }
};

int main(){
    Printer prn;
    int num{};
    std::cin >> num; // get number from user
    std::shared_ptr<int> p {new int{num}}; // use direct initialisation.
    prn.SetValue(p);
    if (*p > 10) {
        // delete p; // don't need to call delete on this pointer (Cuz its not a pointer)
        // instead, we can either assign nullptr to the shared_ptr that will automatically decrement the reference count or we should call reset.
        p = nullptr; // or you can call reset () on this.
    }
    prn.Print();
}
// The code works now and if you key in a value > 10, this will print you the value you entered. However there is a slight issue - the problem is when this shared_ptr p is passed as a parameter inside SetValue(), pass by value is used, and a copy of
```

```
// the shared_ptr is created, and the reference count becomes 2.
// So even if you assign nullptr to the smart pointer shared_ptr object, the reference count will only be DECREMENTED BY 1.
// so the underlying memory is NOT released.
// And maybe it is important for us to release the underlying memory here, because if the reference count does not become zero, the memory is NOT released.
// It will be released only at the end of the scope, or when the Printer object is destroyed.
// What if this Printer object is destroyed much much later? Therefore, until it has been destroyed, the memory that was allocated here will still remain in use and it will NOT be deleted on this line
    p = nullptr;
// when this statement is executed. So what should we do here?
// how do we know that the memory should be destroyed here? We need a mechanism so that after that smart ptr is destroyed, then this
// m_pValue pointer  should somehow know that the underlying memory has been released.
// And obviously, shared_ptr cannot be used here because it blocks this shared_ptr from releasing the underlying memory.
// we need  a way through which this Printer object can access the underlying pointer that this shared_ptr has. If the underlying pointer is destroyed, we should know that the underlying pointer is destroyed, and we should not use it.

// in this situation, we can use the Weak Pointer.

//////// Lecture 77: std::weak_ptr internals
// Change all shared_ptr to weak_ptr
#include <iostream>
#include <memory>
class Printer{
    std::weak_ptr<int> m_pValue{};
public:
```

```
public:
    void SetValue(std::weak_ptr<int>p){
        m_pValue = p;
    }
    void Print(){
        std::cout << "Reference count: " << m_pValue.use_count() << std::endl;
        if (m_pValue.expired()) {
            std::cout<< "Resources no longer available!" << std::endl;
        }
        auto sp = m_pValue.lock(); // returns a shared_ptr, and increments the reference count by 1.
        std::cout << "New Reference count:" << sp.use_count() << std::endl;
        std::cout << "Value i*" << *sp << std::endl;
    }
};

int main(){
    Printer prn;
    int num{};
    std::cin >> num; // get number from user
    std::shared_ptr<int> p {new int{num}}; // NOTICE this is a shared_ptr.
    // Not a weak_ptr!
    // only the functions accept a weak_ptr by VALUE.
    prn.SetValue(p);
    if (*p > 10) {
        p = nullptr;
    }
    prn.Print();
}
// There is another problem that weak_ptrs can solve for us, but we'll discuss that in the next video.
```