```cpp
////// Lecture 58: std::move() Function (C++11)

/////// main.cpp
/*
 std:: move() always used with L-values (these have a name) - it forces the compiler to use the move function instead of the copy
 It forces the compiler to use move semantics instead of copy semantics.

 */
#include "Integer.h"
#include <iostream>
main(){
    Integer a{1};
    // Create a copy of the object a in b
    Integer b{a}; // because a is an l-value,the function overload resolution will choose the copy constructor, because the parameter type is an L-value.
    // In some cases, you may not want to create a copy of this object, instead you want to MOVE it into b, but by default the compiler will call the COPY CTOR.
    //// Applying TYPECAST
    // If you would like the latter, we can apply a TYPECAST to an R-value reference, when we do this you'll see that the compiler invokes the move constructor instead of the copy.
;
    // So that looks like this:
    auto c{static_cast<Integer&&>(a)}; // And you will see the move constructor is invoked.
    // This code is not very readable and does not communicate the intent clearly because what we're doing here is 'just perfoming a cast', and that does not indicate that we want to move the state from A into B.
    // To avoid ambiguity and to increase readability, the std:: lirary provides a function called Move() that internally performs the same cast, but it communicates the intent of the code:
```

```cpp
    auto d{std::move(a)};
    // The reader will immediately know that the programmer intends to move the state from A into B, std::move is defined in the utility header file. Even though we did not explicitly include it, it is indirectly included in the iostream header. If we run this, we get the same output and the move constructor has been invoked.
}
///// Why would you want to do this anyway?

main(){
    // A possible reason is that you create an object and initialise it here:
    Integer a{1};
    // and you perform some operations on it, and you no longer need its state, but the state that it contains is required in some other object.
    // So in this case, you would apply std move on it, this is commonly required if you use a function like this: (SEE FUNCTION BELOW)
    Print(std::move(a));

    // After moving from A, you cannot read from this object because we set it to a nullptr, so that's why if you try reading from A, the program may crash.
    // so when you move FROM  an object, the object is an unspecified state, but the object is still a valid object and you can reinitialise and reuse it.
    /// reinitialising the object:
    a.SetValue(5); // We need to adjust more things if you would like to do reinitialisation
}
////// Integer.cpp
int Integer::GetValue() const {
    return *m_pInt;
}
// This is insufficient, because dereferencing a moved object is just like dereferencing a nullptr.

```

```
// For reinitialisation, adjust the setValue function:
int Integer::GetValue() const {
    if (m_pInt == nullptr) m_pInt = new int{};
    return *m_pInt;
}
// Remember after moving an object, DO NOT read from it.
void Print(Integer val){
    // If you don't want the state of the Integer object after you invoke the print function, instead of letting the compiler copy the object into val, you can move it. the advantage is that, when you move the state of a, after the Print() function finishes execution, the object will be destoryed, and it will release the underlying resources. If you simply pass the Int by value a copy is created, and when the function val finishes it will release its own resources, but the underlying resources of A will only be released at the end of the function. And we don't want to utilise these reasources after the Print() function. So thats's why we will implement std::move() here.

}

///// 2ND USE OF STD::MOVE() => Objects which are non-copyable,
/* A non-copyable object does NOT have copy operations, it only has move() operations. The class may contain members that cannot be copied.
 For instance, you cannot create a copy of a file stream. So imagine Integer class contains a member that cannot be copied, we can simulate this by removing the copy constructor and copy assignments.
*/
void Print(Integer val){}
main(){
    Integer a(1);
    Print(std::move(a)); // This works!
    // Print(a); ERROR CODE!
}
```

```
///// other common areas of implementation: UNIQUE POINTER, FILES, THREADS
// This is also a common pattern with unique pointer, it's a smart pointer and used extensively on C++, and you will see heavy usage of std::move() in the unique pointers chapter.

// Conclusion:
/*
std::move() basically performs a type cast to an R-value reference, it hence only applies to l-values. The typecast causes the compiler to choose the move functions over the copy functions. When an object is moved, it goes in unspecified state, but you can still reuse it by reinitialising it.

What will happen if we apply std::move on a primitive type? Since primitive type do not have any underlying resoruces, applying std::move() does not accomplish anything. So its usage on primitive types is redundant.
*/
```