```cpp
//////// Lecture 79: Deleter
// We have seen that we can use smart pointers to automatically manage dynamic memory, but in some cases you may want to manage a pointer that points to a resource which cannot be released using delete.

// I may have some legacy code.
#include<cstdlib>
#include <iostream>
int main(){
    int *p = (int*) malloc(4);
    *p = 100;
    std::cout << *p << "\n";
    free(p);
}
// can we use a smart pointer here? Using a smart pointer here will cause
// undefined behaviour.
//Try using the smart pointer here - it could either be a unique_pointer or a shared_ptr.

//// Using smart pointers:
#include <memory>
int main()
{
    std::unique_ptr<int> p{(int*)malloc(4)};
    *p = 100;
    std::cout << *p << std::endl;
}
// It works fine, but there is no guarantee that it will always work.
// Because by default, the deleter of the smart ptr that is unique_ptr will call delete
// and we know that for malloc we have to invoke free(), while for new we have to call delete().

// So this code will cause undefined behaviour.
// In the same way , I may want to manage a file stream pointer or a file handle.
// If we want to manage these with smart pointers, then the resources will not be released properly because the smart pointer by default will call
```

```
delete and not the corresponding function that closes the stream.

// Thankfully, smart ptrs support management of resources that may not have been acquired using new. I said earlier, by default, the smart pointer of destructors invoke delete. They don't invoke delete directly, instead they invoke a DELETER, a DELETER is a callback that releases the resource.

// Both smart pointers - unique_ptr and shared_ptr use default deleters that default call delete. So if we want to release a different kind of resource, then we can specify our own deleter.

// A deleter can be any kind of callable.

// So for example, we want to manage the malloc() pointer resource and i want to create a custom deleter for it. I can either create a deleter as
 // 1)a global function
 // 2)a function object or
 // 3)a lambda expression
 // or any other kind of callable.
 // We have not discussed function objects yet - but a function object is simply a function that has a state, and in C++, we can create function objects by overloading the function call 'operator'.

// He is using a struct here because he wants everything to be public.
// It is more convenient, you can even use a class and this should work too!
 struct Free
 {
     // Overloading the function call operator()
     // argument should be the type of pointer you want to release.
     void operator()(int *p) {
         // Inside, call free on this pointer.
         free(p);
         std::cout << "Pointer freed!" << std::endl;
     }
 };
 // Driver code
 int main()
 {
     // How to make a custom deleter: 2 things
     // In this case we specify the type of deleter to the template definition here.
     // and then specify an object of this class
     // so we can either create a temporary object
     std::unique_ptr<int, Free> p{(int*) malloc(4), Free{}};
     // or create an object and pass it in.
     Free free{};
     std::unique_ptr<int, Free> p{(int*) malloc(4), free};
 }
 // When you run it out, you'll see that the smart pointer has invoked the custom deleter.

/// We can also use a function as a custom deleter.
void MallocFree(int *p){
    free(p);
    std::cout << "Pointer freed!" << std::endl;
}
int main(){
    // The return type of the function is void
    std::unique_ptr<int,void (*)(int*) > malloc(4, MallocFree); // number of bytes, and the function address.
    // Must specify the TYPE of the function pointer within the template declaration.
}
// this will also work. But it is recommended to use Function Objects as deleters, especially when they dont contain any attributes.
// Using a function pointer will increase the size of the unique_ptr object.

////// Whhat if its a shared_ptr instead of unique_ptr?
// For function pointers
int main(){
    std::shared_ptr<int> p {(int*) malloc(4), MallocFree}; //this will work - for the shared_ptr there is no need to specify the function pointer as a template argument.
}
// For Function Objects
int main(){
    std::shared_ptr<int> p{(int*) malloc(4), Free{}}; // there is no need to specify the type of the Function object.
}
// Now we are able to use smart pointers to manage any kind of resource that cannot be released, using simple delete calls. Just specify the custom deleter and the smart pointer will automatically invoke it and the underlying resource will be released.

```