# Improving `std::shared\_ptr` Usability with Classes Having Protected or Private Destructors

Author: Davit Kalantaryan  
GitHub: https://github.com/davitkalantaryan

## Abstract

This paper proposes a revision to the internal deletion mechanism of `std::shared\_ptr` to make it compatible with classes whose destructors are declared as protected or private, as long as `std::shared\_ptr<T>` is explicitly declared a friend. The goal is to improve safety, encapsulation, and prevent accidental misuse, particularly in large codebases. Additionally, it suggests a secondary extension: optional notification hooks on reference count changes.

## 1. Motivation

In current C++ standard implementations, `std::shared\_ptr` cannot manage objects of a class `T` if `T` has a protected or private destructor, even if `std::shared\_ptr<T>` is declared as a friend. The compiler reports an error due to access control violations during destruction. This limitation breaks the encapsulation pattern where the lifetime of objects is intentionally managed only via smart pointers.

Example:

class MyClass {  
 friend class ::std::shared\_ptr<MyClass>;  
protected:  
 virtual ~MyClass() {}  
};  
  
std::shared\_ptr<MyClass> ptr(new MyClass()); // Error on all major compilers

This error occurs because internal deletion is done by helper classes or lambdas that are not friends of `MyClass`, even though `std::shared\_ptr<MyClass>` is.

## 2. Real-World Experience

In a large system I developed, all objects were managed by `std::shared\_ptr`. After a brief pause in development, I mistakenly deleted a managed object manually, having forgotten it was supposed to be owned by `shared\_ptr`. This led to undefined behavior and difficult-to-debug crashes.

To solve this, I implemented my own version of `shared\_ptr` that supports usage with protected/private destructors via friend declaration.

Demo repo with failing compilation: https://github.com/davitkalantaryan/demo-cpputils  
Working solution (with fixes and reference count callback):  
https://github.com/davitkalantaryan/cpputils/blob/master/include/cpputils/sharedptr.hpp  
https://github.com/davitkalantaryan/cpputils/blob/master/include/cpputils/sharedptr.impl.hpp

## 3. Current Behavior (Confirmed on GCC, Clang, and MSVC)

All major standard library implementations (libstdc++, libc++, MSVC STL) use internal deleter classes or lambdas that do not have access to protected/private destructors, even if `std::shared\_ptr<T>` is declared a friend. This leads to compilation errors on GCC, Clang, and MSVC, examples of which are shown in Appendix A.

## 4. Proposal

Modify the standard (and/or standard library guidance) to require that:  
  
- `std::shared\_ptr<T>` must delete the managed object directly, not through non-friend internal helpers.  
- This ensures that if `T` declares `friend class std::shared\_ptr<T>`, its destructor will be accessible.

## 5. Optional Extension: Reference Count Change Callbacks

My custom smart pointer implementation also supports a hook mechanism to monitor reference count changes, not just when it drops to zero.

Signature:

using TypeClbk = std::function<void(void\* clbkData, PtrType\* pData, size\_t refBefore, size\_t refAfter)>;

## 6. Impact and Compatibility

- Backward compatibility: Existing code will continue to work as-is.  
- ABI impact: Minimal, if any, since this affects internal mechanics of `shared\_ptr`.  
- Performance: No change unless optional features are used.  
- Implementation complexity: Low; concept already proven in my implementation.

## 7. Conclusion

Allowing `std::shared\_ptr<T>` to safely manage classes with non-public destructors (when declared as a friend) strengthens C++'s commitment to safety, ownership clarity, and RAII. Adding optional hooks for reference count changes further enhances observability. These improvements can reduce bugs in large systems and enable developers to better express intent and control.

# Appendix A: Compiler Error Examples

## A.1 GCC (g++ 9)

error: ‘TestClass::~TestClass()’ is private within this context  
note: declared private here

## A.2 MSVC

error C2248: 'TestClass::~TestClass': cannot access private member declared in class 'TestClass'  
note: declared private here

## A.3 Clang (libc++ on macOS)

error: no matching constructor for initialization of '::std::shared\_ptr<TestClass>'  
note: candidate template ignored: substitution failure [...] no member named '\_EnableIfImpl' in 'std::\_MetaBase<false>'

# Appendix B: Clarification After Initial Feedback

After reading the initial feedback from the community, I decided to include the following clarification in the proposal:  
  
Many of you are absolutely right — if a smart pointer manages lifetime, manual deletion should be avoided. But in large, evolving codebases, it’s not uncommon for someone to forget or misunderstand ownership. That’s exactly why C++ provides access control like private and protected: not because developers are careless, but to let the compiler help enforce intended usage.  
  
Saying “just don’t delete manually” is similar to saying “we don’t need private or protected; developers should just remember not to call internal functions.” But we do use those specifiers — to make misuse harder, not just discouraged.  
  
This proposal aims to bring similar safety to destructors. If shared\_ptr<T> is a friend, it should be allowed to delete T, just like any other friend.