```
#ifndef INCLUDED_LIBPTR_UNIQUE_PTR_H
#define INCLUDED_LIBPTR_UNIQUE_PTR_H

// You can use sax::DefaultDeleter from deleter.hpp if you are implementing the
optional part
#include "deleter.hpp"
#include <utility> //for std::exchange, std::swap, std::move

namespace sax
{
    template <typename T, typename Deleter = sax::DefaultDeleter<T>>
    // If the user does not provide a custom deleter:
    // UniquePtr<int> => uses DefaultDeleter<int> (calls delete)
    // If the user does provide a custom deleter:
    // auto filePtr = UniquePtr<FILE, decltype(&fclose)>(fopen("file.txt", "r"),
&fclose); => uses fclose instead of delete

    class UniquePtr
    {
    private:
        T* ptr_ = nullptr;
        Deleter deleter_; //customer deleter

    public:
        //default constructor from raw pointer
        explicit UniquePtr(T* ptr = nullptr) noexcept : ptr_(ptr),
deleter_(Deleter{}) {}

        //constructor to pass custome deleter => ex: &fclose
        UniquePtr(T* ptr, Deleter deleter) noexcept : ptr_(ptr),
deleter_(std::move(deleter)) {} //move new deleter (&fclose) to deleter_, make
default deleter empty

        //move constructor
        UniquePtr(UniquePtr&& other) noexcept : ptr_(other.release()),
deleter_(std::move(other.deleter_)) {}

        //move assignment
        UniquePtr& operator=(UniquePtr&& other) noexcept {
            if(this != &other){
                reset(other.release());
                deleter_ = std::move(other.deleter_); //when moving an object
also need to move it's deleter so it knows how to delete it
            }
            return *this;
        }

        //Delete copy constructor/ copy assignment
        UniquePtr(UniquePtr const&) = delete;
        UniquePtr& operator=(UniquePtr const&) = delete;

        //Destructor
        ~UniquePtr() noexcept { reset(); }
```

```
        //Modifiers
        T* release() noexcept { //release() to release ownership of the pointer
without deleting it
            return std::exchange(ptr_, nullptr);  //set ptr_ = nullptr, return
the old value of ptr_
        }

        void reset(T* newPtr = nullptr) noexcept{
            if(ptr_ != newPtr){
                if(ptr_) deleter_(ptr_); //use custome deleter instead (like
fclose(ptr) of assigned)
                ptr_ = newPtr;                          //!!! delete ptr_ only
works when the resource was created with new
            }
        }
        //UniquePtr<int> p(new int(10));
        //p.reset(new int(20)); => delete the old int(10) => now owns int(20)

        void swap(UniquePtr& other) noexcept{
            std::swap(ptr_, other.ptr_);
            std::swap(deleter_, other.deleter_);
        }

        //Observers
        T* get() const noexcept { return ptr_; } //get() to access the raw
pointer

        Deleter& get_deleter() noexcept { return deleter_; }
        const Deleter& get_deleter() const noexcept { return deleter_; }

        //operator bool() to check if the pointer is non-null
        explicit operator bool() const noexcept { return ptr_ != nullptr; }

        //operator* and operator-> for dereferencing
        T& operator*() const noexcept { return *ptr_; }
        T* operator->() const noexcept { return ptr_; }

    };
    //non-member swap for ADL
    template <typename T, typename Deleter>
    void swap (UniquePtr<T, Deleter>& lhs, UniquePtr<T, Deleter>& rhs) noexcept{
        lhs.swap(rhs);
    } //std::swap performs 3 move operations (slower),
    // while member swap just swaps two pointers (faster), so ADL is needed to
make sure the fast version is used


} //namespace sax

#endif /* INCLUDED_LIBPTR_UNIQUE_PTR_H */
```