

- Solution

The solution to the exercise of the previous lesson.

WE'LL COVER THE FOLLOWING ^

- Solution
- Explanation

Solution

```
// templatesCRTPShareMe.cpp

#include <iostream>
#include <memory>

class ShareMe: public std::enable_shared_from_this<ShareMe>{
public:
    std::shared_ptr<ShareMe> getShared(){
        return shared_from_this();
    }
};

int main(){

    std::cout << std::endl;

    // share the same ShareMe object
    std::shared_ptr<ShareMe> shareMe(new ShareMe);
    std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();

    // both resources have the same address
    std::cout << "Address of resource of shareMe "<< (void*)shareMe.get() << " " << std::endl;
    std::cout << "Address of resource of shareMe1 "<< (void*)shareMe1.get() << " " << std::endl;

    // the use_count is 2
    std::cout << "shareMe.use_count(): "<< shareMe.use_count() << std::endl;
    std::cout << std::endl;

}
```



Explanation

- With the class `std::enable_shared_from_this`, we can create objects which return a `std::shared_ptr` on itself. For that, we must derive the class public from `std::enable_shared_from_this`.
- The smart pointer `shareMe` (line 18) is copied by `shareMe1` (line 19). The call `shareMe->getShared()` in line 19 creates a new smart pointer.
- `getShared()` (line 8) internally uses the function `shared_from_this`.
- In lines 22 and 23, `shareMe.get()` returns a pointer to the resource. In line 26, the `shareMe.use_count()` returns the value of the reference counter.

For further information, see:

- [std::unique_ptr](#)
- [std::shared_ptr](#)
- [std::weak_ptr](#)

Now, let's compare the performance of smart pointers in the next lesson.