Søren Hansen <sha@ase.au.dk>
V1.7

Resource Handling

Introduction

In embedded system and systems in general one of the most challenging tasks is to keep track of resources and relinquish them when they are not needed anymore. This is especially the case with dynamically allocated memory

Prerequisites

Before you can start on this exercise, you have to have:

- Basic understanding of classes, constructors, destructors and operator overloading.
- “Rule of 3”
- Know what the different terms *RAII*, *Smart Pointer* and *Counted Smart Pointer* signify.

Exercise 1 Ensuring garbage collection on a dynamically allocated std::string

In this basic exercise we revisit the *RAII* idiom once more, however this time in an enhanced form, where our goal is to extend it with the *SmartPointer* idiom.

Recap: If we inspect the code from the `ScopedLock` class (or idiom) - see below - it becomes apparent that we have no way of accessing the data residing inside the class. These are considered private and, in the specific use case, sacred in order for the idiom to work as intended.

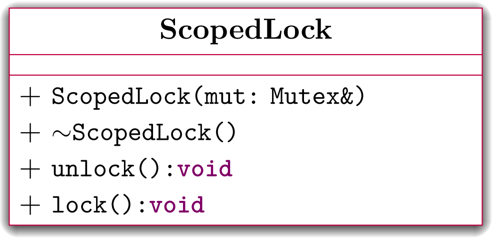


Figure 1.1: The class `ScopedLock`

The questions that comes to mind are:

- What do we do if we need access?
- Under which circumstances is this a necessity?
- What syntax and semantics do we want/need?

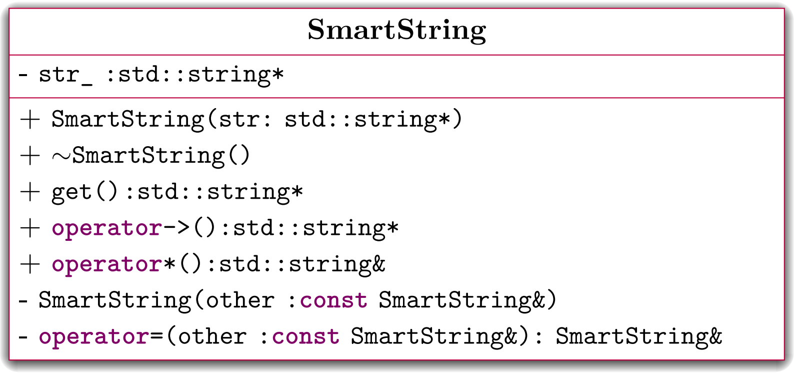
The answer to these in short form is:

> *In a situation where dynamically allocated resources are a necessity, a garbage collected approach which incorporates pointer like semantics is desired!*

This is exactly what is achieved by combining both the *RAII* and the *SmartPointer* idiom.

In this exercise your task is to combine the above two mentioned idioms and write a class that encapsulates a `std::string`, such that garbage collection is enforced when leaving scope *and* that we get pointer like semantic and syntax usage.

Implement the below shown UML class `SmartString`:



```
SmartString
- str_ :std::string*
+ SmartString(str: std::string*)
+ ~SmartString()
+ get():std::string*
+ operator->():std::string*
+ operator*():std::string&
- SmartString(other :const SmartString&)
- operator=(other :const SmartString&): SmartString&
```

Figure 1.2: The class `SmartString`

The code snippet below can be used to verify that your solution works.

Listing 1.1: Example showing `SmartString` in use

```cpp
int main(int argc, char* argv[])
{
  SmartString ss(new std::string("Hello world"));
  std::cout << "String length: " << ss->length() << std::endl;

  if(ss->find("world") != std::string::npos)
    std::cout << "The 'ss' string does contain 'world'" << std::endl;
  else
    std::cout << "The 'ss' string does NOT contain 'world'" << std::endl;

  SmartString ss2 = ss; // <- Deliberately why? What happens?
}
```

Questions to answer:

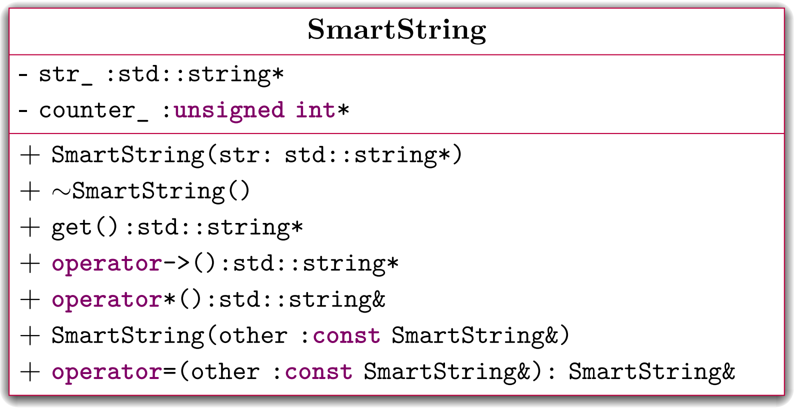
- Why must the copy constructor and assignment operator be private with no implementation? and what is the consequence when these are private?
  Hint: Whats wrong with the default editions of the constructor/assignment operator and destructor methods for this particular class?
- What exactly does the `operator->()` do?
- See code for more question(s)?

Exercise 2 The Counted Pointer

In this exercise we are going to extend our fine `SmartString`, such that multiple parties may have a “reference” to it.

Since we want sharing of the data as opposed to what we did in Exercise 1, the *assignment operator* and *copy constructor* must now be implemented.

This leads to this new interface:



```
SmartString
- str_ :std::string*
- counter_ :unsigned int*
+ SmartString(str: std::string*)
+ ~SmartString()
+ get():std::string*
+ operator->():std::string*
+ operator*():std::string&
+ SmartString(other :const SmartString&)
+ operator=(other :const SmartString&): SmartString&
```

Figure 2.1: The class `SmartString`

Things to be aware and consider:

- Why must the counter be dynamically allocated? Where should this happen?
- How is the *copy constructor* to be implemented?
- How is the *assignment operator* to be implemented?
- What happens in the *destructor* and how should it be implemented?

Create appropriate test harnesses to validate that your solution works. These test harnesses must be part of your documentation on the wiki.

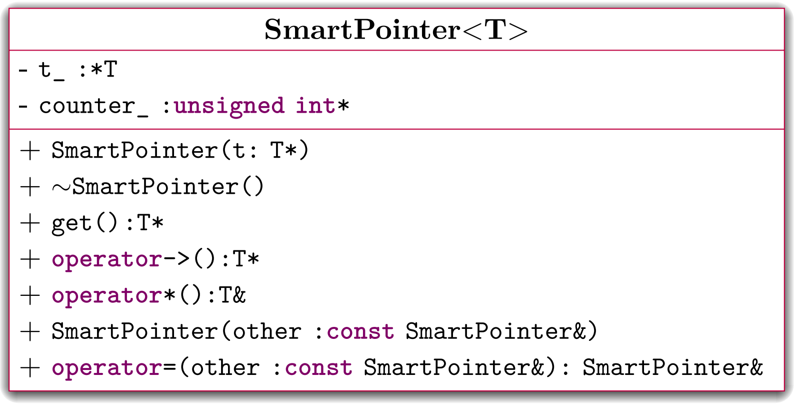
Exercise 3 Templated Counted Smart Pointer (OPTIONAL)

The current implementation is somewhat limited in the sense that it can *only* hold a pointer to an object of type `std::string`. On the other hand, the concept that it implements is rather universal.

Rethinking the proposed solution and taking another stance we may transform the class into a template that provides us with a generic solution. This is *exactly* what is desirable.

The new class would therefore get the following interface:

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING



```
SmartPointer<T>
- t_ :*T
- counter_ :unsigned int*
+ SmartPointer(t: T*)
+ ~SmartPointer()
+ get():T*
+ operator->():T*
+ operator*():T&
+ SmartPointer(other :const SmartPointer&)
+ operator=(other :const SmartPointer&): SmartPointer&
```

Figure 3.1: The class `SmartPointer<T>`

Create appropriate test harnesses to validate that your solution works. Describe the different tests and explain how they verify that your solution works.

Exercise 4 Discarding our solution in favor of boost::shared_ptr<>

During the different exercises we have embarked on a journey where each and every step improves/enhances our previous solution. More over we have gained a better understanding of the language that we utilize.

In Exercise 3 we succeeded in creating a viable solution that in reality would improve our programs by making them less error-prone. However instead of reinventing the wheel, we will make a rather large *jump* and start using the *boost library*, in particular we will be using `boost::shared_ptr<>`[1].

Exercise 4.1 Using boost::shared_ptr<>

The template class `boost::shared_ptr<>` incorporates the different aspects that we have covered and then some.

Use the `boost::shared_ptr<>` and get to know it. Utilize the different test harnesses that you have concocted and create a new one that verifies that `boost::shared_ptr<>` works as intended, and that it can be used in the use-cases discussed above.

As stated earlier the test harnesses must be presented on your wikis.

Exercise 4.2 Messages wrapped in a boost::shared_ptr<> (OPTIONAL)

Describe how `boost::shared_ptr<>` can be used to encapsulate the messages that we send from one thread to another ensuring that proper garbage collection follows. In particular, why will it work assuming the following:

[1] If you have your own personal Linux installation then you probably need to install the boost library. In Ubuntu this can be done by writing `sudo apt-get install libboost-dev`

1. Thread A creates and sends a message to Thread B.
2. Thread B sends the message onward to Thread C, but also retains the message at the same time (although it retains the message it is considered Read-Only for all parties the moment it left Thread A.).
3. Which one of the two Threads (Thread B and C) that releases the message is unknown.

Discuss the needed changes in your concurrent program if all messages were wrapped in a `boost::shared_ptr<>`.

Exercise 5 Resource Handling

Perspective of this particular exercises:

- What do you consider a *resource*?
- In which situations do you foresee challenges with resources and how could they be handled?
- In particular regarding memory, when would you be able eliminate the need for allocations. Or when is it a *must* that something is allocated(on the heap)?