Søren Hansen <sha@ase.au.dk>

Generel repetition using a shared pointer

V1.1

Introduction

In this exercise we will revised the shared pointer (RAII) concept and extend it using templates and further incorporating functors, implicit-, explicit conversions

Prerequisites

- Templates intermediate
- Well experience with inheritance and virtual methods
- Familiar with RAII

Exercise 1 Making a basic shared pointer

In your endevour to create this entity ensure that it at least has the following properties.

- It implements the counted pointer idiom
- Copy construction and assignment is implemented and works as one would expect.
- Need to be able to use it as a *pointer like* entity - E.g. the smart pointer idiom
- Use *delegating constructors* and others where appropriate.

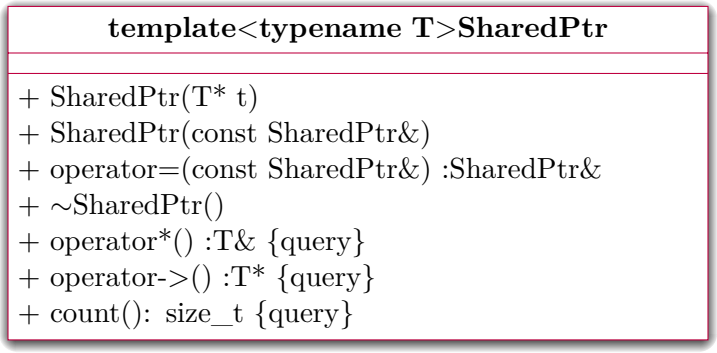

Figure 1.1: Class SharedPtr

Exercise 2 Conversions

Exercise 2.1 The 'explicit' constructor

Inspect your `SharedPtr<>` template class and discuss which of the constructors, if any, should be `explicit` and why/why not. A code snippet for or against should be included in your answer.

Exercise 2.2 Overloading

Exercise 2.2.1 Which overloads do we use?

Which overloads have been implemented and why? Are there any other overloads that might be useful?

Exercise 2.2.2 "Implicit" conversion to bool

If one wants to be able to check whether the shared pointer contains a usable pointer the following code must work:

Listing 2.1: Conversion to bool

```
1 SharedPtr<std::string> sp(new std::string("Hello world"));
2
3 if(sp)
4   std::cout << "SP Contains: " << *sp << std::endl;
```

But on the other hand these statements below should not work!

Listing 2.2: Conversion to bool

```
1 SharedPtr<std::string> sp(new std::string("Hello"));
2 SharedPtr<std::string> sp2(new std::string("world"));
3
4 if(sp == sp2) // No comparison function implemented, thus should fail
5    doStuff();
6
7 int x = test << 1; // bool is not an int and should not be treated as one.
```

Determine exactly what is important to utilise from the language in order to achieve what we want but at the same time inhibit misuse.

Exercise 2.2.3 Comparison overloads

Add the global operator overload for equality - Note that equality in this context the pointer value itself and not whether the contents are the same.

Before you commence do take the time to reflect on the signature of the template function... Hint: Think about inheritance and pointers.

Snippets that verify the overloads do work must be provided.

Exercise 3 Destruction

Enhance your already cool shared pointer with the ability to give it a functor in the constructor. This optional functor is assumed to perform the clean-up in the event that the contained resource is to be relinquished.

For this to work you are to add the following constructor to your `SharedPtr<>`:

Listing 3.1: Additional template constructor to support a custom deleter

```
1 template<typename T, typename D>
2 SharedPtr(T* t, D d)
3 : ... h_(new Helper<T, D>(d)) // Consider this a hint :-)
4 {
5 }
```

Hints regarding implemention:

- Ensure that the `deleter` is accessible by all shared pointers to a given object.
- The `deleter` is a functor that takes one argument being a pointer to the type in question. To keep it simple you can hardcode it to handle your specific type, otherwise you can simply make it a template function to handle most types.
- Implementing the functionality that handles the `deleter` appropriately involves using the "External polymophism" pattern. An alternate source is how `boost::any` is implemented - in particular see classes `placeholder` and `holder` and their relationship[1].
- No the solution is ***NOT*** long - 20 lines including its own namespace...

Exercise 4 Namespace

Finally place your `SharedPtr<>` and associated free function(s) in their own namespace.

Retest your code snippets from 2.2.3 and see if they still work.

Explain your findings...

[1] Another example with source can be found here `https://sourcemaking.com/design_patterns/adapter/cpp/2`