sha@ase.au.dk
V2.0

Boost::Function and Boost::Bind

Introduction

In this exercise we are going to work with `std::function` and `std::bind`. The idea is to create a small system that can send and receive events from a timer object. This means that when events do occur in the timer object the registered callback functions are to be called with an `Event` pointer input parameter. The callback function then does whatever is appropriate depending on the type which is actually passed via the event object.

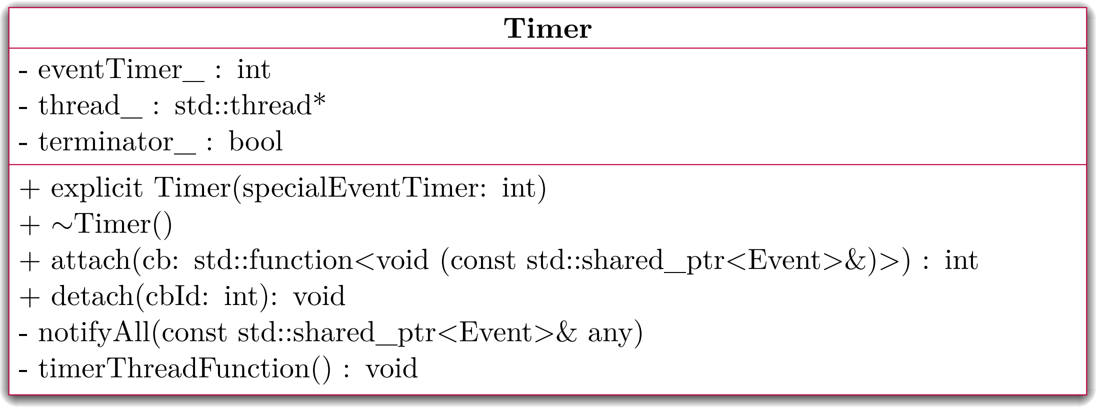
To help you out use the code provided in the directory `TimerCallback`, found in the place as this file.

Exercise 1 Implementing the class `Timer`

Exercise 1.1 The code

Before you start inspect the code through and through to get acquainted.

Exercise 1.2 Design and interface for Timer



Timer

- eventTimer_ : int
- thread_ : std::thread*
- terminator_ : bool

+ explicit Timer(specialEventTimer: int)
+ ~Timer()
+ attach(cb: std::function<void (const std::shared_ptr<Event>&)>) : int
+ detach(cbId: int): void
- notifyAll(const std::shared_ptr<Event>& any)
- timerThreadFunction() : void

Figure 1.1: The class MyArray interface - simplified

For now the implemented methods are the constructor, the destructor as well as the thread function itself. In this first exercise e.g. exercise 1 you will be tasked with implementing the missing methods. Furthermore, in particular, the free function which is used in this first exercise misses some printout. This is to be done as well.

Note that the events being used in this simple example are all generated from within the thread function `timerThreadFunction()`. After an event has been generated in this function, it is delegated to the function `notifyAll()`. As the name implies - it is this function that notifies all.

Exercise 1.3 Implementing method `attach()` and `detach()`

To implement the `attach()` method some additional internal state is required. Obviously we need to keep the callback functions in some container and at the same time be able to uniquely identify them, such that they can be removed at a later timer using `detach()`.

To simplify things we will just choose a simple `int` variable to identify each callback. This means that at every insertion we increment it. Consequently we need a container that has a key value pair property. (Obviously you have to determine which one you want to use...)

Inspecting the class interface it can be seen that `attach()` returns an `int` being the unique identifier and `detach()` takes an `int` again the unique identifier. Thus ensuring that a proper cleanup can be carried out.

Exercise 1.4 Implementing method `notifyAll()`

Is a rather simple task, just iterate through each and every element and call the specific *callback function* with the event as the input parameter.

Exercise 1.5 Missing part of function `freeFunction()`

Note the lock guard, why do you think it is a good idea to have?

Since it is unknown which event the function `freeFunction()` was called with as input, a type checking approach is needed.

Which can be used in conjuction with `std::shared_ptr<>`[1] and discuss how you handle it.

Exercise 1.6 Compile and test

Using the free function readily available verify that you implementation works as expected.

Exercise 2 Using other callbackable entites

Other types of possible callback functions...

Exercise 2.1 Functor

Implement a *functor* and try using it as a callback function in your newly created program.

Exercise 2.2 Using `boost::bind()` when the function signature does not fit

Exercise 2.2.1 Free function with an extra parameter

In file `Bindfunction.cpp` you will find a free function called `void withAnExtra(const std::shared_ptr<Event>&, const std::string&)`. This function does not fit the original requirement regarding having only one input parameter. In this exercise utilize the power of `boost::bind` such that the above requirement is upheld. For the extra `std::string` input parameter choose some text.

Note that the printout code is mising as was the case with function `freeFunction()`. You have to add this...

[1] Are there any limitation at all?

Exercise 2.2.2 A reference object taking multiple parameters

In the same file as in the previous exercise you will find the class `ReferenceObj`. This time you want to accomplish two things regarding the callback entity.

- A pre-instantiated object of this type is to be passed to the timer callback system and it must be the same, e.g. the original and not a copy that is called each time.
- A specific method of this particular object must be called namely `call()`.

Again use `boost::bind()` to accomplish this task. Remember to verify that it is indeed the *same instance* being used.

Exercise 3 The challenge - Timebox it (OPTIONAL CHALLENGE)

In this exercise you will be challenged in the sense that you are to rewrite the method `notifyAll()`.

This exercise is rather difficult so beware.

Instead of using the old school `for(...)` loop, you are to use `std::for_each()` and create a `std::bind()` expression. Obviously the intent is that `notifyAll()` still works as expected.

A couple of hints (this is rather difficult):

- Do make a typedef of `cb` for `std::function<void (const std::shared_ptr<Event>&)>` it actually comes in handy.
- The solution requires two binds that are nested.
- Elements in a container are typedef'ed to `C::value_type`, where C could be a `vector`, `map` etc.