Coding assignment

Problem 1

Create a **Timer** which dispatches time **Events** at regular intervals to a **Worker**. The **Worker** then runs **Jobs** at the time specified in the events.

Timer

The **Timer** class is responsible for generating **Events** at a fixed time interval (100ms). Each **Event** contains a unique event id as well as a *random future* execution time, which is anywhere between 0 and 500ms from the **Event** creation time. Every time the **Timer** triggers an **Event** it must print its event id, current time and future execution time.

Worker

The user can register any number of **Jobs** with the **Worker**. The **Worker** will then execute all its registered **Jobs** for each **Event** it receives from the **Timer**. The **Job** execution must be as close as possible to the execution time specified in the **Event**. When a **Job** is run, it must print the event id, current time as well as the time difference (a.k.a. *lag*) between the desired execution time and current time.

Job

The **Job** represents some work to be done and should be generic enough that it can take any number of arguments and can be tailored to do anything the application chooses. **Jobs** can be registered at any time with the **Worker**. They can also be de-registered with the **Worker** when the application doesn't want this **Job** to execute anymore. Therefore there needs to be a mechanism for adding and removing **Jobs**.

Coding instructions

The chosen design must take system performance into account (memory, threading, CPU usage, etc) and should try to minimize as much as possible the time *lag*. A description of the design choices should also be provided for easier understanding.

Example

Timer generates events at time Tn:

1) **T0 : { event_id: 1, execution_time: T0+323ms } -> worker must execute @ T0+323ms**
2) **T1 : { event_id: 2, execution_time: T1+5ms } -> worker must execute @ T1+5ms (~T0+105ms)**

Note that in this particular case the Worker must execute event 1 before event 0 because the execution time is earlier.

Problem 2

Write a utility (**comparer_builder**) that creates a *functor* that defines arbitrary search order. For example:

```
struct MyStruct {
   int x1;
   int get_x2() const { return x2; }
   const std::string& get_x3() const { return x3; }
private:
   int x2;
   std::string x3;
};

std::vector<MyStruct> v{ … };

std::sort(v.begin(), v.end(),
comparer_builder<MyStruct>().by(&MyStruct::x1).by(&MyStruct::get_x2));

std::sort(v.begin(), v.end(),
comparer_builder<MyStruct>().by(&MyStruct::get_x3).by(&MyStruct::get_x2));
```