COMP 3512

Assignment 2

This assignment uses the MVC (Model-View-Controller) paradigm pioneered by the Smalltalk programming environment. It is used in GUI & web applications.

1 MVC

In the Model-View-Controller paradigm, model objects represent data, view objects are responsible for displaying the data & controller objects handle user actions (e.g., keyboard & mouse input in GUI applications).

An example is a timer that can be started or stopped by clicking on a button. In this case, the model is the timer, the view displays the timer & the controller is the button. When the timer is ticking, the view must update itself; when the button is clicked, the timer must change its behaviour (start or stop ticking).

In general, the 3 components of the MVC triad need to communicate with one another. A common goal is to make the model independent of the view & the controller so that different views & controllers can be used with the same model. However, this may not always be possible.

2 The Observer Pattern

A theme in MVC is that when one object changes, another object needs to change in response. This can be achieved through a general "dependency mechanism" where the object that is changing notifies other objects so that they can perform some actions in response. This notification of changes is the purpose of the Observer pattern.

In the Observer pattern, a subject has one or more observers. When the subject changes, its observers are notified. In C++, we can have 2 classes — `Subject` & `Observer` — to represent the 2 types of objects.

A subject basically needs to keep a list of its observers so that it can notify them when it changes. The `Subject` class has methods that allow an observer to "subscribe" or "unsubscribe" itself with the subject. When the subject changes, it "notifies" all subscribed observers, asking them to "update" themselves.

```
// Subject.h
class Subject {
public:
  virtual ~Subject() {}
  virtual void subscribe(Observer *obs);   // subscribe an observer
  virtual void unsubscribe(Observer *obs);  // unsubscribe an observer
  virtual void notify();  // ask each subscribed observer to update itself
protected:
  Subject() {}
private:
  std::list<Observer *> observers_;         // list of observers
};
```

Each observer must have an `update` method. It's called by the `notify` method of a subject. (The subject notifies the observer, asking it to update itself.) This method is abstract because it's specific to the actual type of the observer. For example, the `update` method may stop or start a ticking timer or increment a counter. Any concrete class derived from the `Observer` class must implement the `update` method.

```
// Observer.h
class Observer {  // abstract base class
public:
  virtual ~Observer() {}
  virtual void update(Subject *) = 0;
protected:
```

```
  Observer() {}
};
```

(Why are the `Subject` & `Observer` default constructors protected?)

3 The Program

Our model is a timer that can tick. The keyboard (standard input) is the controller. The user can start, stop & reset the timer by entering commands through the keyboard. The timer object has several views that are different ways of displaying the timer. These views are updated as the timer changes (ticks or is reset). Note that the timer is an observer of the keyboard controller; the timer itself is observed by the views.

We'll be using C++11 threads in this assignment (see §4). The program uses 2 threads:

- The main thread (the one that executes the `main` function) creates a timer object, a keyboard controller & 3 views of the timer object (one each of the 3 types of views specified in §3.2) & then executes the code for the keyboard controller. This controller reads user commands line by line & uses only the first word of the line as the command. The user can enter a command that begins with the character 's' (for "start") to start the timer, that begins with the character 'h' (for "halt") to stop the timer, or that begins with the character 'r' (for "reset") to reset the timer to 0. Other commands are simply ignored. The program exits when the user presses the end-of-file key at the beginning of a line.
- The timer is executed in another thread. Initially its value is 0 & it is not ticking.

3.1 The Timer

A `Timer` object basically keeps track of the number of seconds that have elapsed. It can be started, stopped & reset to 0. It is both an observer (of the keyboard) & a subject (observed by the views). The members of the `Timer` class are basically as follows:

```
public:
  Timer();                          // ctor (see below)
  void           start();           // start the timer
  void           stop();            // stop the timer
  void           reset();           // reset timer to 0
  unsigned long  get() const;       // returns number of seconds elapsed

  Timer(const Timer&) = delete;             // (*)
  Timer& operator=(const Timer&) = delete;  // (*)

  // ...  (inherited methods not shown)
private:
  std::atomic_ulong  sec_;          // number of seconds elapsed
  std::atomic_bool   ticking_;      // is timer ticking or not?
  void run();                       // function executed by thread (see below)
  // additional members if necessary
```

When a timer is first created, it has the value 0 & is not ticking. Each timer runs in its own thread. The `Timer` constructor needs to create a thread that executes the `run` member function. (See §4.) This `run` function is basically an infinite loop that alternately sleeps for 1 second & increments the counter if it is ticking.

Because of the use of threads, we should not allow the copying or the assignment of timers. This is achieved by the 2 lines labelled (*).

3.2 The Views

We'll use 3 ways to display a timer. The `SecView` simply displays the number of seconds that have elapsed. The `MinSecView` displays the seconds elapsed as minutes & seconds separated by a colon (e.g. `0:25` for 25 seconds, `12:05` for 12 minutes & 5 seconds). Finally, the `TickView` prints an asterisk whenever the timer ticks. All three view classes are derived from the abstract `TimerView` class:

```
class TimerView: public Observer {
public:
  TimerView(Timer *timer);
  virtual void update(Subject *s);  // from Observer class
  virtual void display(std::ostream& os) const = 0;
protected:
  Timer *timer_;
};
```

It's clear that a "timer view" must be associated with a timer (see constructor). Note that the `TimerView` class is itself derived from `Observer` because it's going to observe a `Timer`. (When the observed timer changes, the view needs to change.)

Each class derived from `TimerView` implements its own `display` method to display the timer in a suitable format. For example

```
class SecView: public TimerView {
public:
  SecView(Timer *timer): TimerView(timer) {}
  virtual void display(std::ostream& os) const;
};
```

The other two view classes are similar. Note that you may need to add additional declarations in some or all of these classes. (Do we want to allow the copying of views?)

3.3 The Keyboard Controller

The keyboard controller is the subject of observation of `Timer` objects. When it is started (by the `start` method), it keeps reading commands from the keyboard. When it gets a command, it notifies all its observers.

```
class KeyboardController: public Subject {
public:
  void    start();             // start the loop to get user commands
  string  getCommand() const;  // return the command
private:
  string  cmd_;
};
```

4 Threads

C++-11 introduced support for threads. Threads are basically created to execute functions. This can be achieved by passing in the function to execute followed by the arguments to that function to the constructor of the `thread` class. By default, arguments to the `thread` constructor are copied into memory accessible by the thread.

The following example creates a thread to print `"hello"` 100 times:

```
#include <iostream>
#include <string>
#include <thread>
```

```
void print(const std::string& msg) {
  for (int i = 0; i < 100; i++)
    std::cout << msg << std::endl;
}

int main() {
  std::thread  t(print, "hello");
  t.join();  // wait for thread to finish executing
}
```

In the above, the main thread (the thread executing the `main` function) waits for the new thread to finish by invoking `t.join()`. The alternative is to call `detach()` on a thread which allows the thread to run in the background. However, calling `t.detach()` instead of `t.join()` in the above would be problematic because then `main` would not wait for the thread to finish & when `main` returns, all threads are terminated. Note also that if we don't wait for a thread to finish, we need to ensure that any data accessed by the thread is valid until the thread has finished using it.

We need to call either `join` or `detach` on a thread before its `thread` object is destroyed. Otherwise, the program is terminated when the `thread` object is destroyed.

We can also create a thread to execute a method on an object. The following is an example:

```
#include <iostream>
#include <string>
#include <thread>

class printer {
  int  id_;
public:
  printer(int id): id_(id) {}
  void print(const std::string& msg) const {
    for (size_t i = 0; i < 100; i++)
      std::cout << "printer " << id_ << ": " << msg << std::endl;
  }
};

int main() {
  printer p(1);
  std::thread  t(&printer::print, &p, string("hello"));  // note: 2nd arg can also be p
  t.join();
}
```

In the above, we are passing in the address of `p` to the `thread` constructor. This would make it possible for the `print` method to modify `p` should it choose to do so. (In our example, the method does not change `p`.) Since we pass in `&p`, we need to ensure that the object `p` is valid as long as the thread needs to access it. In the above, we achieve this by calling `t.join()` to wait for the thread to finish. Had we passed in `p` to the constructor, the thread would have got a copy of `p`. In our example, this would also work fine.

In general, if one thread is reading an object while another thread is writing to it, the behaviour is not well-defined. This is because "reads" & "writes" may not be atomic (indivisible) operations. However, the standard does provide some atomic types with certain atomic operations. We need to include `<atomic>` to use them. Some atomic types were shown in the `Timer` class. (See §3.1.)

5 Additional Information

Each class should have its own header file & possibly an implementation file. Note that there are 8 classes — `Subject`, `Observer`, `Timer`, `TimerView`, `TickView`, `SecView`, `MinSecView` & `KeyboardController`. Note also that you may need to use additional switches with `g++`.

6 Submission

This assignment is due at 11pm, Monday, November 23, 2015. Submit your source files in a zip file to `ShareIn` in the directory:

```
\COMP\3512\a2\set<X>\
```

where `<X>` is your set. Your zip file must be named `<name_id>.zip`, where `<name_id>` is your name & student ID separated by an underscore (for example, `SimpsonHomer_a12345678.zip`) Do not use spaces to separate your last & first names. *Your zip file must unzip directly to your source files without creating any directories.* We'll basically compile your files using

```
g++ -std=c++11 -pthread -W -Wall -pedantic *.cpp
```

after unzipping.

Do not put executables in your zip file. Also, do not submit rar files—they will not be marked.

If you need to submit more than one version, name the zip file of each later version with a version number after your ID, e.g., `SimpsonHomer_a12345678_v2.zip`. If more than one version is submitted, we'll only mark the version with the highest version number. (If it is unclear to us from your submissions which version to mark, you may fail to get credit for the assignment.)

This assignment will be marked mainly based on its functionalities. Hence, if your program does not compile, you may receive a score of 0 for the assignment.

Assuming your program works correctly, the grade breakdown is *approximately* as follows:

Design & Code clarity	10%
The `Subject` & `Observer` classes	15%
The `KeyboardController` class	15%
The `Timer` class	25%
The view classes	25%
The main program	10%