CSCI 262 Data Structures

Spring 2019

Lab 10: Classes, Objects, and Operator Overloading

(Back to Labs)

Goals

- Practice overloading operators
- Explore some basic object capabilities

Overview

This lab will walk you through creating a simple class and some accompanying functions. As usual, we ask that you work with a partner or a team. If you end up solo, please raise your hand and we'll pair you with someone else (or make a three-person team if necessary). You are welcome to change partners throughout the semester!

The class you will create will be called `clock_time`, and it will model time as kept on a 24-hour clock. That is, it will represent the hours, minutes, and seconds as displayed on a 24-hour clock. It will allow for adding time to an existing `clock_time`, correctly accounting for increases which would cause the clock to rollover to the next day.

Although this will be a very simple class, it will give you a chance to practice a number of things, including creating header and source files for a class, writing constructors and methods, and overloading operators to work with your new class. Along the way we will also briefly discuss the default assignment operator and copy constructor, and how they work with your new class.

Instructions

Step 0: preliminaries

Download main.cpp, which contains a main() function with code to test your new class. The tests are broken into three parts, corresponding to the steps 1 - 3 below. The tests for steps 2 and 3 are commented out to start with. You should only uncomment the tests for step 2 once the tests for step 1 are running without error, and uncomment the tests for step 3 once the tests for step 2 are running without error. Once all the tests are running without error, congratulations - you've completed the lab successfully!

To kick things off, go ahead and create a new project, and add two files: a header file named `clock_time.h`

and a source file named `clock_time.cpp`. Don't forget to put appropriate block comments at the tops of both files giving your name and the purpose of the files (similar to what you see in main.cpp). After the top comment in the header file, put in some include guards or `#pragma once`.

Step 1: Basic class, setters, getters, constructors

In your header file, declare the class `clock_time`. It should have the following public methods and constructors:

Prototype	Description
`int get_hour()`	Get the hour part of the time. E.g., if the clock_time object represents the time "16:45:07", then this method would return 16.
`int get_minute()`	Get the minute part of the time. E.g., if the clock_time object represents the time "16:45:07", then this method would return 45.
`int get_second()`	Get the second part of the time. E.g., if the clock_time object represents the time "16:45:07", then this method would return 7.
`void set_time(int h, int m, int s)`	Set the time to the time that would exist at h hours, m minutes, and s seconds after midnight. E.g, the call `set_time(16,45,7)` should set the clock_time object to represent the time "16:45:07". For simplicity, `set_time` allows values outside the normal ranges, including negative values. Thus `set_time(99,-44,66)` is valid, and represents a time which is 99 hours and 66 seconds, minus 44 minutes, past midnight (2:17:06, if you were curious).
`clock_time()`	Construct a clock_time object representing midnight.
`clock_time(int h, int m, int s)`	Construct a clock_time object representing the same time as an object after calling `set_time(h, m, s)`.

You also need some kind of private internal representation of the time. At first glance, it seems as if storing hours, minutes, and seconds is the way to go - it certainly suffices for most of the public methods described above. However, it is very difficult to handle out-of-range values (e.g., hours greater than 23, minutes or seconds greater than 59, and negative values) with this representation. Addition of `clock_time` objects is also difficult.

A better solution (or so I will claim) is to simply store the total number of seconds past midnight the clock_time object represents. It is fairly easy to convert back and forth from hh:mm:ss to total seconds, and it is much easier to handle rollovers (using the mod operator)!

Accordingly, while you can make your own implementation choices, I strongly recommend a private `int` member variable named `_seconds` to store the number of seconds past midnight represented. This value should never exceed 24 * 60 * 60 - 1, and should never be negative. (The use of the _ prefix to designate private members is fairly common, but you can name your variable whatever you choose.)

You may add whatever additional private member variables (e.g., constants) or methods that you wish.

Now, implement all of the public methods (and any private methods you add) in your `clock_time.cpp` source file. Because time in lab is limited, and this is a programming lab and not a math lab, let me provide some conversions for you: if it is *s* seconds past midnight, then it is *s* / 3600 hours past midnight, (*s* % 3600) / 60 minutes into the hour, and *s* % 60 seconds into the minute. Going the other way, note that an hour is 3600 seconds and a minute is 60 seconds.

Programming hints:

- Once you've implemented `set_time()`, do you want to copy all of the same code into your constructor? Consider calling set_time() from within your constructor instead. That way, if you have a bug, you only have to fix it one place!
- Note that some of the tests go *backwards* from midnight, rather than forwards. The mod operator (`%`) in C++ has the funny property that a negative number mod some other number is a negative number (if not zero). If you get a negative number after applying the modulo operator, you can simply add the modulus to the negative number to get the positive result you need.

Be sure to run the tests in main() (in test.cpp) before going on to the next step!

Step 2: << ostream operator, to_string function

The << operator is commonly overloaded for outputting human-readable representations of objects to output streams. The `to_string()` function is commonly used to convert objects into a string representation (the <string> standard library defines this method for converting base types to their string representations). You must now implement both of these - the prototypes are below:

Prototype	Description
`ostream & operator<<(ostream & out, clock_time c)`	Output the human-readable time to the output stream `out`.
`string to_string(clock_time c)`	Return the human-readable time as a string.

The human-readable representation/string representation of a clock_time object is the time as it would appear on a 24-hour clock. That is, it would appear as h:mm:ss, where h is the natural representation of the hour (not padded), while mm is the minutes past the hour, left padded with zeroes if necessary to ensure it has two digits, and ss is similarly the nubmer of seconds past the minute, left padded with zeroes.

Declare these functions (and note that they are functions, not methods!) in your header file, after the class declaration. Then write the functions in your source file. You can go about writing these functions various

ways; however, once you have one of them working, the other can easily be written in terms of the first one. If you write the `to_string()` function first, you can simply call it in your implementation of `operator<<`. Alternately, if you write the `<<` operator first, you can take advantage of the setw() and setfill() stream manipulators; the `to_string` function then can be written using an `ostringstream` object.

Uncomment the tests for step 2 and run the tests in main() (in test.cpp) before going on to the next step!

Step 3: operators ==, !=, and +

The equality and inequality comparison operators can be written as functions or as methods in the clock_time class (see the lecture on operator overloading for more info); however, for this lab you should write them as free functions. Note that while you can use the public getter methods of `clock_time` to do the comparisons, it would be easier and faster to simply compare the internal representation (assuming you did it as suggested) of the two clock_time objects. However, to do that, you will first need to declare at least one of these as a *friend* function (the other one can easily be written in terms of the first). To do this, anywhere in your class declaration (at the top, before any visibility modifiers is traditional), simply add the line

```
friend prototype;
```

where *prototype* is the operator's prototype. E.g., for the equality operator, you might have the line:

```
friend bool operator==(clock_time a, clock_time b);
```

Note that both of these operators must return a `bool` value.

Next, write the + operator, again as a free function, taking two clock_time values and returning a clock_time value. Again, this will be significantly cleaner and easier if you declare the operator function as a friend in clock_time and use the internal representation.

There's no extra credit for it, but you might also want to implement operator-, just for the practice.

Extra credit (0.5 points each):

- Create a `to_string` option that allows display of 12-hour time together with AM or PM. Use a defaulted Boolean parameter to control whether 24-hour or 12-hour time is used (24 hour should be the default).
- Read the documentation on the C-library `localtime()` function and the `tm` struct, or the newer C++ `system_clock` class, and add a `set_now()` method to `clock_time` to set the time to the current local time.

Additional Notes

Perhaps surprisingly, C++ does not provide a default implementation of == for your classes; == is undefined for your new class until you overload the operator appropriately (as you did in step 3 above). Furthermore, C++ does not automatically define != once you have defined ==. Essentially all of the comparison operators must be

overloaded individually (although you can often write them in terms of one or two basic operators).

What C++ does provide by default are the *assignment operator* and the *copy constructor*. The assignment operator is just what gets called when you do an assignment of a `clock_time` object to a `clock_time` variable. For simple objects like `clock_time`, the default does the right thing; it simply copies over all instance variable values from one object to the other. In our case, this means copying over the internal time representation, which is exactly what we want to have happen. When we look at classes using dynamically allocated memory, the situation will get more complicated, and we'll want to override C++'s default implementation.

Similar to the assignment operator, the copy constructor is invoked when you do either of the following (these are equivalent):

```
clock_time x(y);   // where y is an existing clock_time object
```

or

```
clock_time x = y;
```

The second one above looks like an assignment, but because it occurs at the same time as the variable declaration, it actually uses the copy constructor. The effect is pretty much the same, though, in the default implementation - the new object gets a copy of all the instance variables of the old object.

You can see examples of the assignment operator and copy constructor used in the test.cpp file.

Fun Diversion

Why dealing with time in software is hard: https://www.youtube.com/watch?v=-5wpm-gesOY

Submission Instructions

Submit a .zip file to Canvas containing your clock_time.h and clock_time.cpp files, together with a README. The README should contain your name, the name of your lab partner(s), and any other information you think we should know.

Grading

Code passes tests	1 point each
Code style and formatting	1 point
README	1 point

Total:	**12 points**
Extra credit	Up to 1 point