CS 2 – Lab 2

Alex Donahue

Introduction

The goal of this lab is to write the implementation of a series of functions defined by the lab instructions, on top of writing sufficient tests to calculate the validity of said implementation. I will structure this report around each of these classes. As a quick aside, every test that I write will be a test that I've checked the validity of, and has indeed passed. This way I don't have to clog the report up with extra pictures that probably aren't necessary.

The lab was pretty fun, if not sneakily time consuming, it has taken me up until the very last day to finish. I have a better understanding of c++ testing now, and also I got taught a lesson about how I should be writing pseudo code beforehand, because this had a lot of moving parts I wasn't considering! Mostly in regards to rolling over time. I believe I came up with a pretty elegant solution that I'm happy to share with you though.

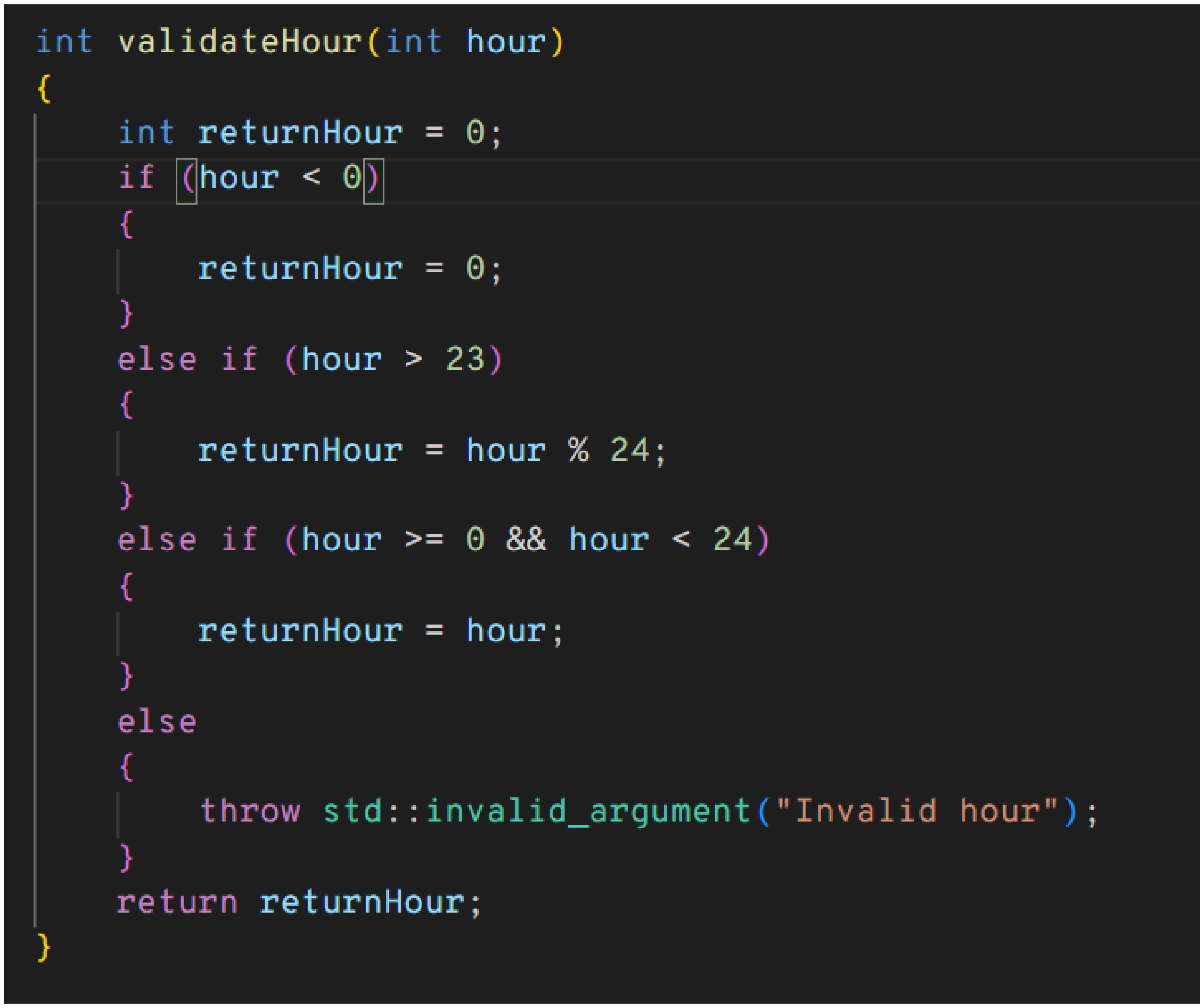
Helper Functions:

The backbone of my program revolves around 4 different helper functions that I defined to aid me in organizing my logic and ensuring consistent results when formatting/rolling over the time. I'll go over these first as understanding them makes understanding the rest of my code much easier. These helper functions are as follows

Int validateHour(int hour)
Int validateMinute(int minute)
Int validateSecond(int second)
TimePart validateTime(int hour, int minute, int second)

Int validateHour(int hour)

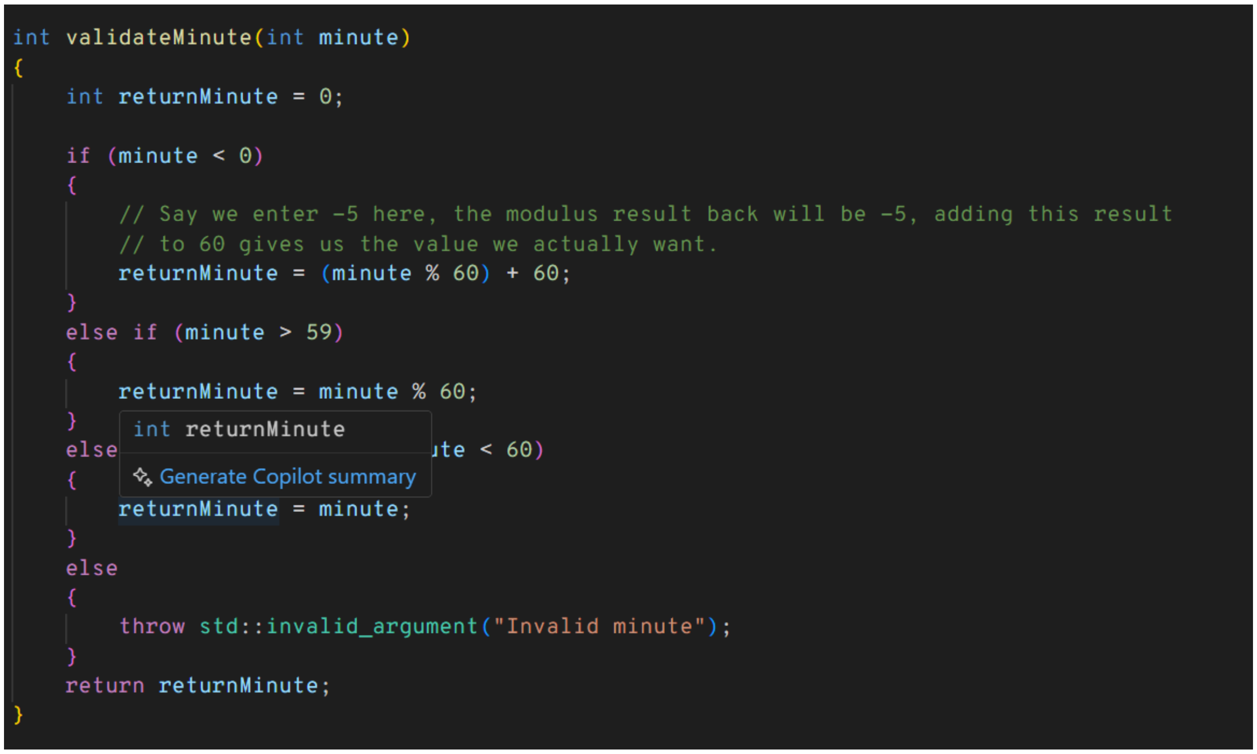
This function receives the proper hour from validateTime and then does some formatting based on how large hour is. I think the logic is pretty self explanatory. I made the decision to go with setting negative hours to 0.



```
int validateHour(int hour)
{
    int returnHour = 0;
    if (hour < 0)
    {
        returnHour = 0;
    }
    else if (hour > 23)
    {
        returnHour = hour % 24;
    }
    else if (hour >= 0 && hour < 24)
    {
        returnHour = hour;
    }
    else
    {
        throw std::invalid_argument("Invalid hour");
    }
    return returnHour;
}
```

Int validateMinute(int hour)

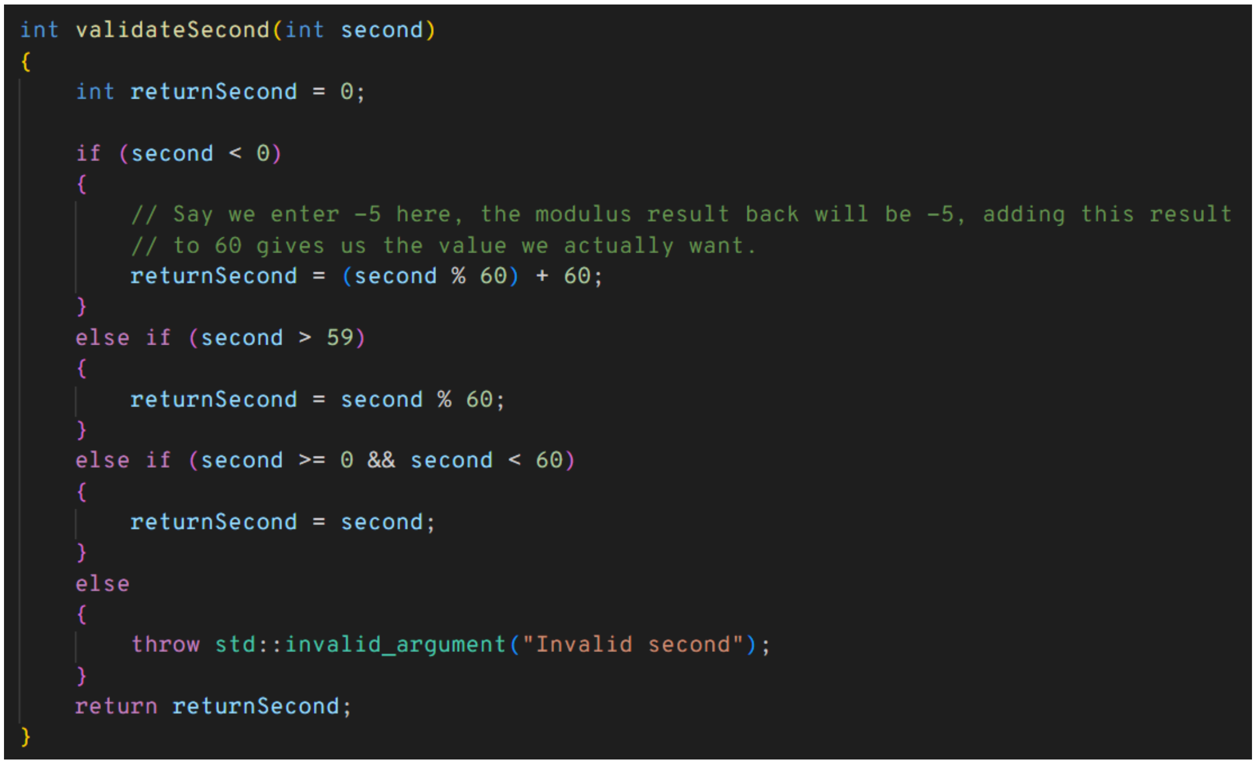
Same as above with a little more nuance in terms of how numbers are handled. This function needs to handle negatives properly, and I figured out that (minute % 60) + 60 is adequate for all negative cases in order to get the proper second. This allows for very large negatives to be formatted properly. The other 3 cases are pretty easy to understand.



```
int validateMinute(int minute)
{
    int returnMinute = 0;

    if (minute < 0)
    {
        // Say we enter -5 here, the modulus result back will be -5, adding this result
        // to 60 gives us the value we actually want.
        returnMinute = (minute % 60) + 60;
    }
    else if (minute > 59)
    {
        returnMinute = minute % 60;
    }
    else                     ute < 60)
    {
        returnMinute = minute;
    }
    else
    {
        throw std::invalid_argument("Invalid minute");
    }
    return returnMinute;
}
```

Int validateSecond(int hour)

The logic is basically identical to validateMinute, so I won’t waste your time explaining it again.

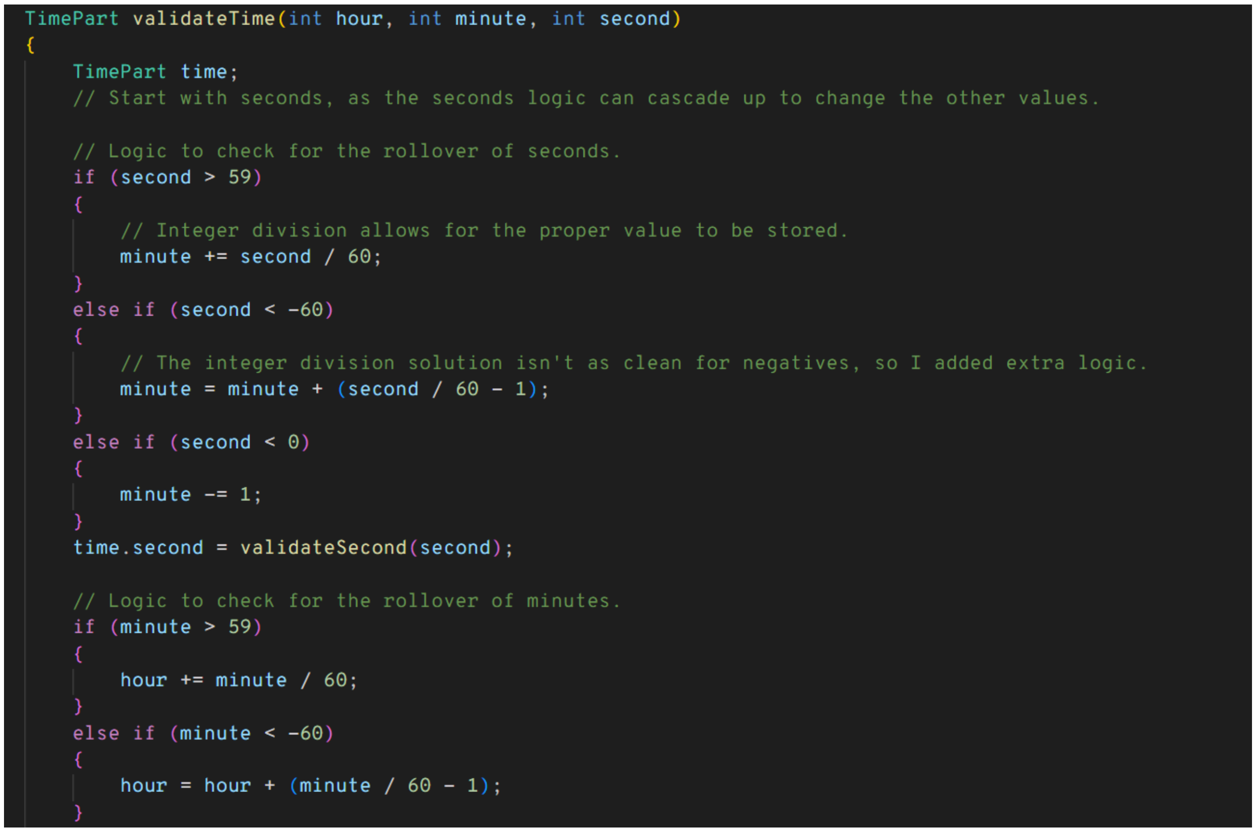


```
int validateSecond(int second)
{
    int returnSecond = 0;

    if (second < 0)
    {
        // Say we enter -5 here, the modulus result back will be -5, adding this result
        // to 60 gives us the value we actually want.
        returnSecond = (second % 60) + 60;
    }
    else if (second > 59)
    {
        returnSecond = second % 60;
    }
    else if (second >= 0 && second < 60)
    {
        returnSecond = second;
    }
    else
    {
        throw std::invalid_argument("Invalid second");
    }
    return returnSecond;
}
```

Timepart validateTime(int hour)

This is the backbone of my entire program. The return value is Timepart so that I can store and return multiple different time values after all of my validations are done. The function starts with seconds and analyzes how much it needs to add to minute based on second's input. After that it validates second with my function, and then sends it to the return TimePart variable called time. I started the function with second so that the logic would cascade into minutes and hours properly.

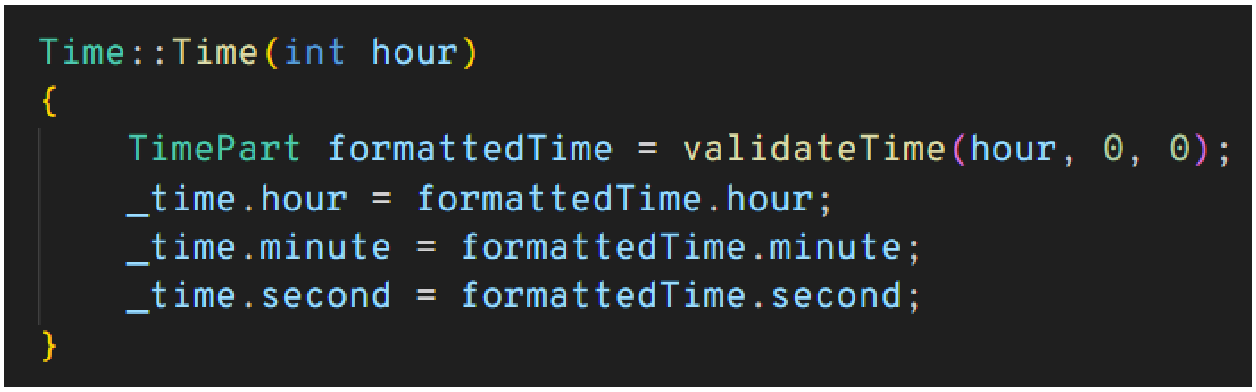
Minute follows largely the same logic except it changes the value of hour. Then hour just launches straight into validateHour, because the hour doesn't roll over into any other values. Storing all this logic into a function proved INCREDIBLY useful for writing the rest of the program, as you'll see later.



```
TimePart validateTime(int hour, int minute, int second)
{
    TimePart time;
    // Start with seconds, as the seconds logic can cascade up to change the other values.

    // Logic to check for the rollover of seconds.
    if (second > 59)
    {
        // Integer division allows for the proper value to be stored.
        minute += second / 60;
    }
    else if (second < -60)
    {
        // The integer division solution isn't as clean for negatives, so I added extra logic.
        minute = minute + (second / 60 - 1);
    }
    else if (second < 0)
    {
        minute -= 1;
    }
    time.second = validateSecond(second);

    // Logic to check for the rollover of minutes.
    if (minute > 59)
    {
        hour += minute / 60;
    }
    else if (minute < -60)
    {
        hour = hour + (minute / 60 - 1);
    }
```

Time(int hour):

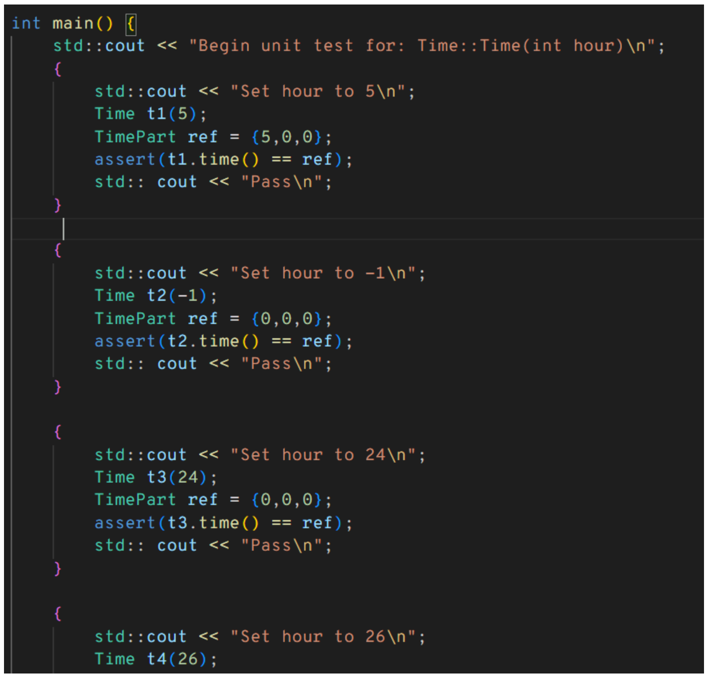
The validateTime function makes this function only 4 lines of code. Hour is passed along to validateTime while manually setting minutes and seconds to 0. Then this result is stored in a TimePart object and then finally stored in whatever Time class calls it.



```
Time::Time(int hour)
{
    TimePart formattedTime = validateTime(hour, 0, 0);
    _time.hour = formattedTime.hour;
    _time.minute = formattedTime.minute;
    _time.second = formattedTime.second;
}
```

Time(int hour) Testing:

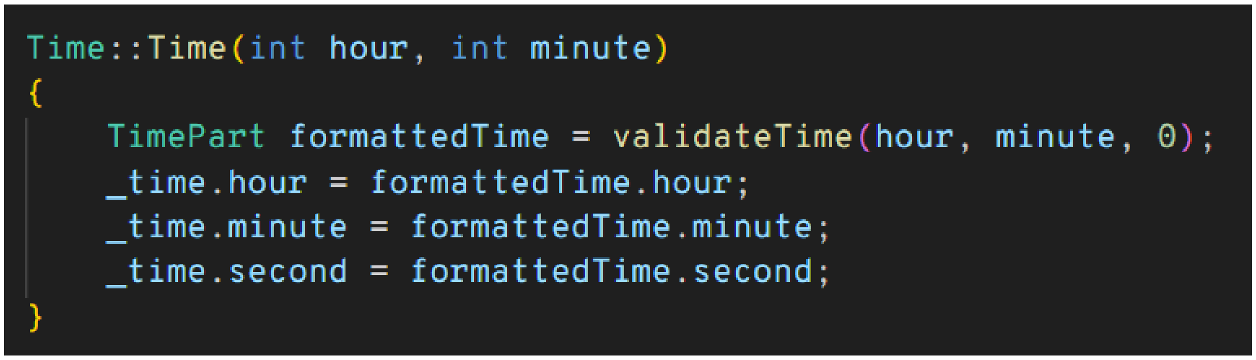
With this testing I just focused on some potential edge cases like negatives and numbers over 26. As an aside, my testing is by no means exhaustive, I did the best with my time allotted on the lab, and tried to cover the widest amount of cases possible with the fewest tests.



```
int main() {
    std::cout << "Begin unit test for: Time::Time(int hour)\n";
    {
        std::cout << "Set hour to 5\n";
        Time t1(5);
        TimePart ref = {5,0,0};
        assert(t1.time() == ref);
        std:: cout << "Pass\n";
    }

    {
        std::cout << "Set hour to -1\n";
        Time t2(-1);
        TimePart ref = {0,0,0};
        assert(t2.time() == ref);
        std:: cout << "Pass\n";
    }

    {
        std::cout << "Set hour to 24\n";
        Time t3(24);
        TimePart ref = {0,0,0};
        assert(t3.time() == ref);
        std:: cout << "Pass\n";
    }

    {
        std::cout << "Set hour to 26\n";
        Time t4(26);
```

Time(int hour, int minute)

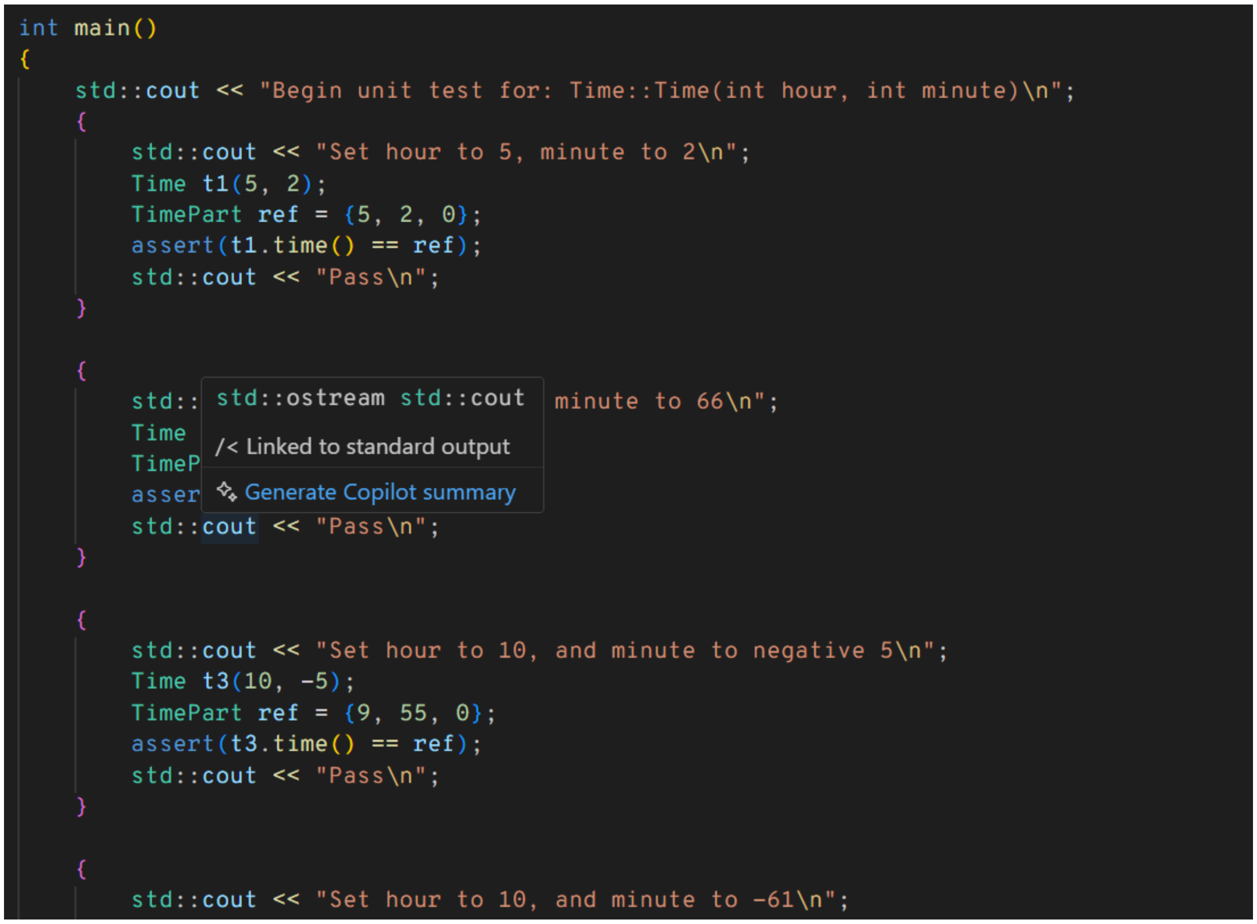
Just like the previous function, this function is the same logic with the exception of a minute variable being passed.



```cpp
Time::Time(int hour, int minute)
{
    TimePart formattedTime = validateTime(hour, minute, 0);
    _time.hour = formattedTime.hour;
    _time.minute = formattedTime.minute;
    _time.second = formattedTime.second;
}
```

Time(int hour, int minutes) Testing:

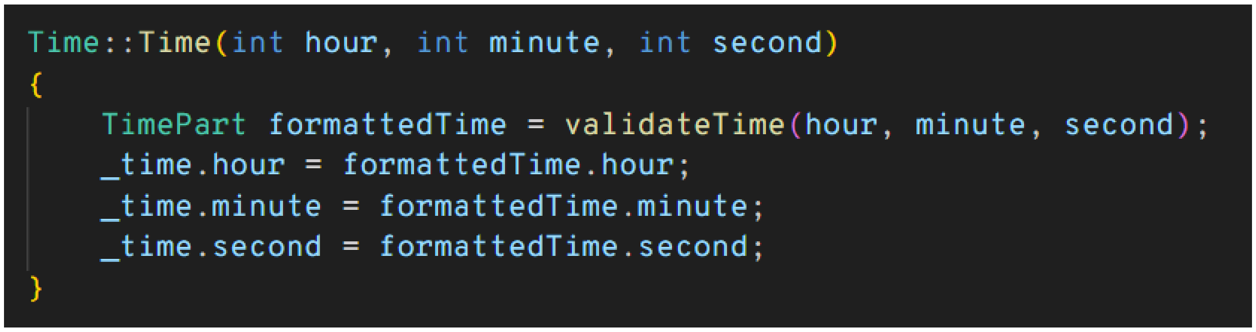
In this testing I added more complexity, mainly revolving around how negatives were calculated, and making sure the rolling over was functional.



```cpp
int main()
{
    std::cout << "Begin unit test for: Time::Time(int hour, int minute)\n";
    {
        std::cout << "Set hour to 5, minute to 2\n";
        Time t1(5, 2);
        TimePart ref = {5, 2, 0};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std:::  std::ostream std::cout   minute to 66\n";
        Time    /< Linked to standard output
        TimeP
        asser   Generate Copilot summary
        std::cout << "Pass\n";
    }

    {
        std::cout << "Set hour to 10, and minute to negative 5\n";
        Time t3(10, -5);
        TimePart ref = {9, 55, 0};
        assert(t3.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Set hour to 10, and minute to -61\n";
```

Time(int hour, int minute, int second):

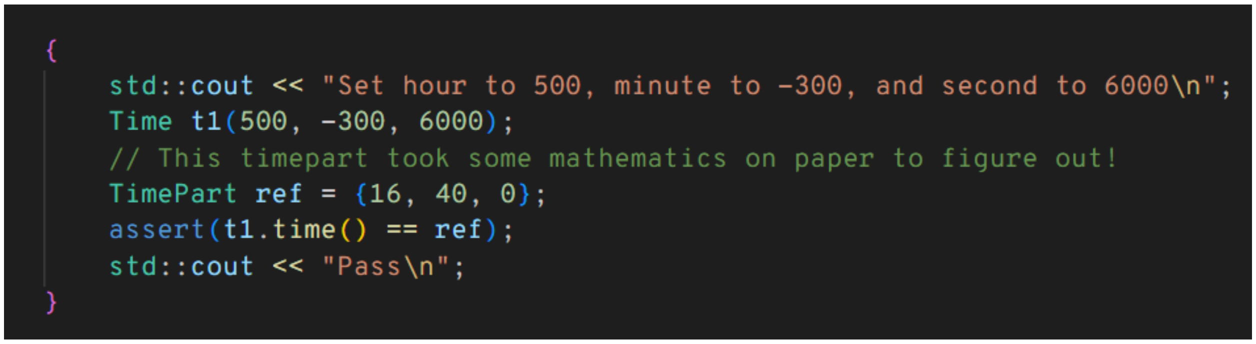
Same as before, except no more hard coded 0’s, every parameter is used.



```
Time::Time(int hour, int minute, int second)
{
    TimePart formattedTime = validateTime(hour, minute, second);
    _time.hour = formattedTime.hour;
    _time.minute = formattedTime.minute;
    _time.second = formattedTime.second;
}
```

Time(int hour, int minute, int second): Testing:

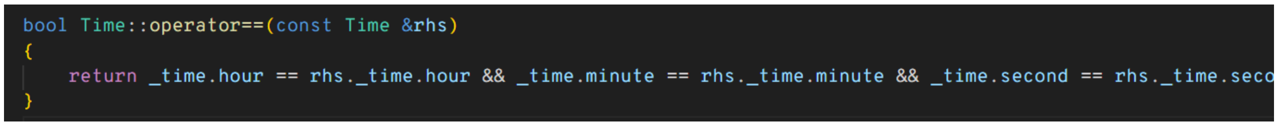
Testing here is largely the same as before, I did have fun throwing in some really large values and calculating them out. So my program is functional even if incredibly unrealistic times are being given!



```
{
    std::cout << "Set hour to 500, minute to -300, and second to 6000\n";
    Time t1(500, -300, 6000);
    // This timepart took some mathematics on paper to figure out!
    TimePart ref = {16, 40, 0};
    assert(t1.time() == ref);
    std::cout << "Pass\n";
}
```

Bool operator==(const Time &rhs):

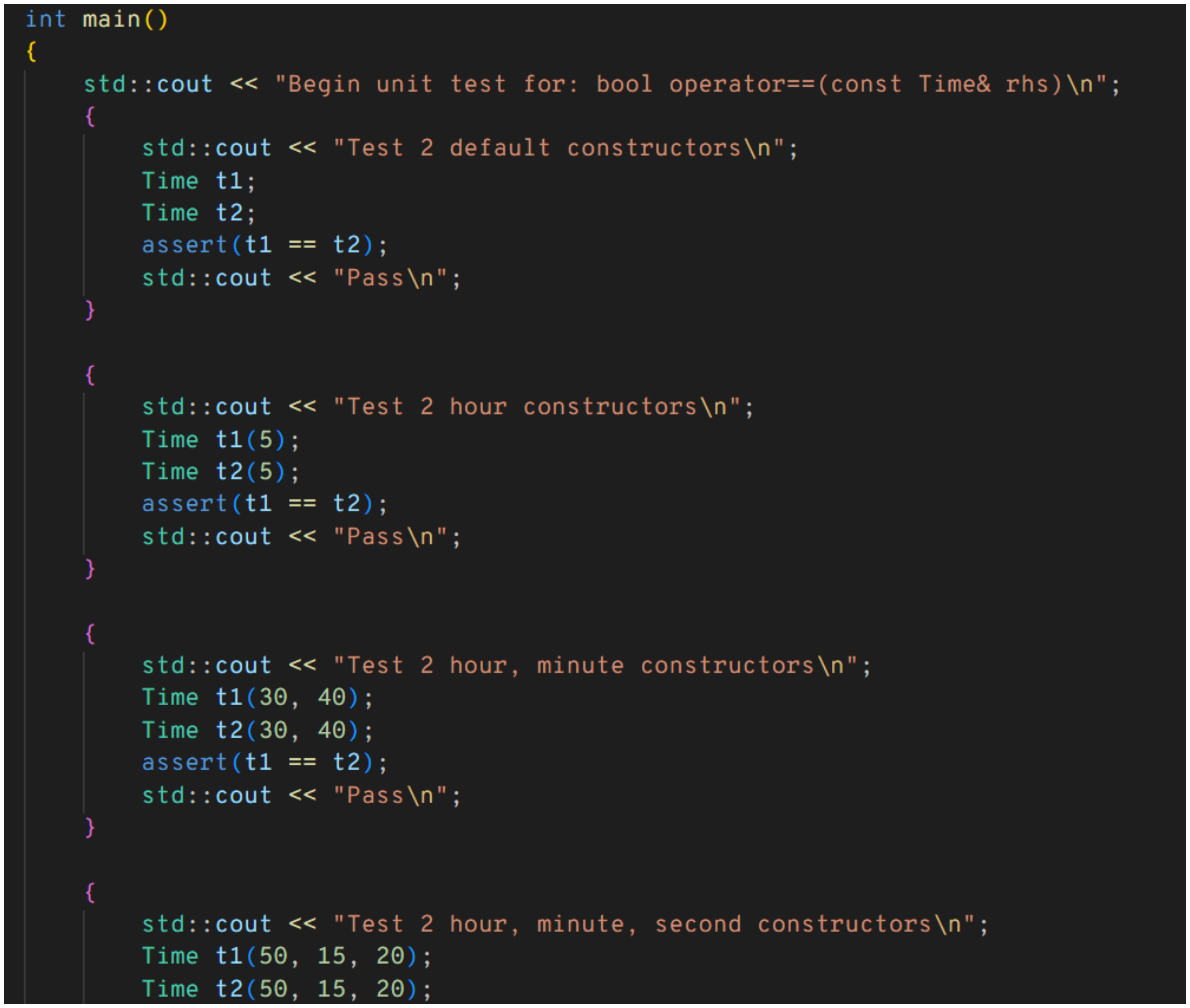
A simple solution. Simply comparing the timepart values between both Time objects.



```
bool Time::operator==(const Time &rhs)
{
    return _time.hour == rhs._time.hour && _time.minute == rhs._time.minute && _time.second == rhs._time.seco
}
```

Bool operator==(const Time &rhs) Testing:

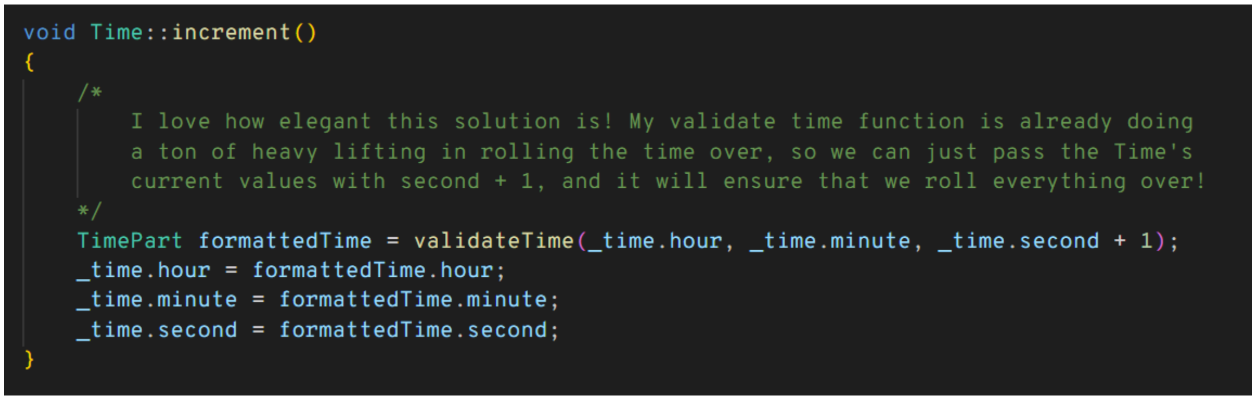
Here I focused on testing different constructors, and also did a test for non-equvalence.



```
int main()
{
    std::cout << "Begin unit test for: bool operator==(const Time& rhs)\n";
    {
        std::cout << "Test 2 default constructors\n";
        Time t1;
        Time t2;
        assert(t1 == t2);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Test 2 hour constructors\n";
        Time t1(5);
        Time t2(5);
        assert(t1 == t2);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Test 2 hour, minute constructors\n";
        Time t1(30, 40);
        Time t2(30, 40);
        assert(t1 == t2);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Test 2 hour, minute, second constructors\n";
        Time t1(50, 15, 20);
        Time t2(50, 15, 20);
```

Void increment():

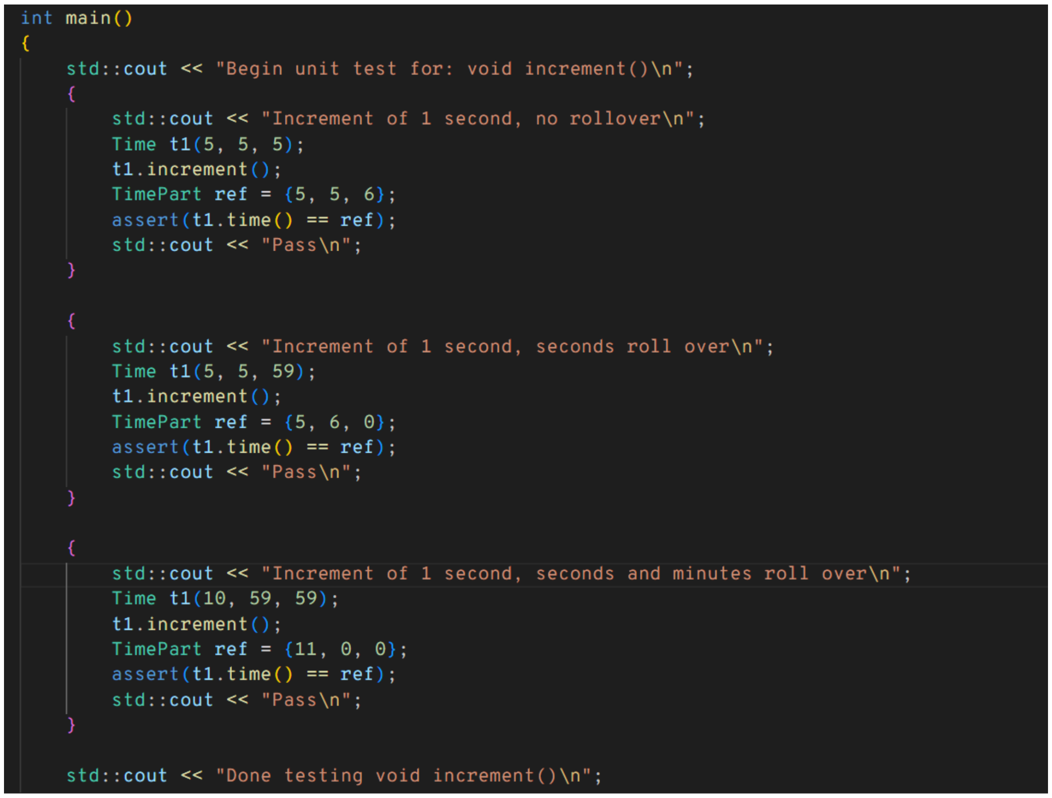
This was my favorite part of the lab. The reason being I had a huge “aha” moment when I realized I could use my already built validateTime function to very elegantly handle incrementation. Simply passing the values to validateTime, except 1 is added to second does the trick!



```cpp
void Time::increment()
{
    /*
        I love how elegant this solution is! My validate time function is already doing
        a ton of heavy lifting in rolling the time over, so we can just pass the Time's
        current values with second + 1, and it will ensure that we roll everything over!
    */
    TimePart formattedTime = validateTime(_time.hour, _time.minute, _time.second + 1);
    _time.hour = formattedTime.hour;
    _time.minute = formattedTime.minute;
    _time.second = formattedTime.second;
}
```

Void increment() Testing:

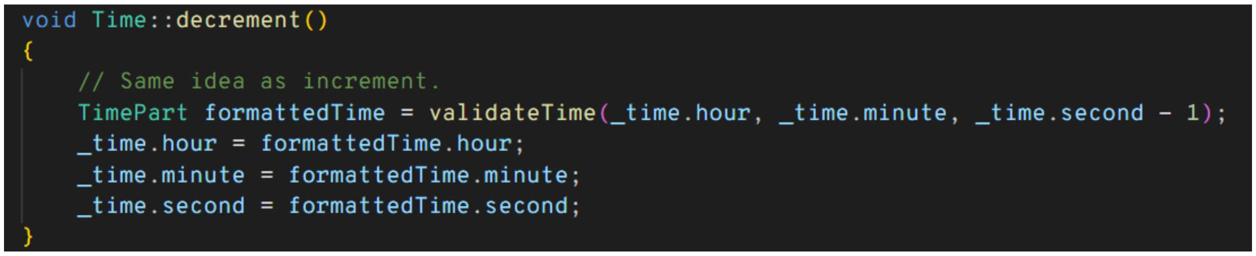
Like before, most of my testing revolves around ensuring that incrementation works with rolling over, but at this point it’s pretty redundant to continue to check that.



```cpp
int main()
{
    std::cout << "Begin unit test for: void increment()\n";
    {
        std::cout << "Increment of 1 second, no rollover\n";
        Time t1(5, 5, 5);
        t1.increment();
        TimePart ref = {5, 5, 6};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Increment of 1 second, seconds roll over\n";
        Time t1(5, 5, 59);
        t1.increment();
        TimePart ref = {5, 6, 0};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Increment of 1 second, seconds and minutes roll over\n";
        Time t1(10, 59, 59);
        t1.increment();
        TimePart ref = {11, 0, 0};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    std::cout << "Done testing void increment()\n";
```

Void decrement():

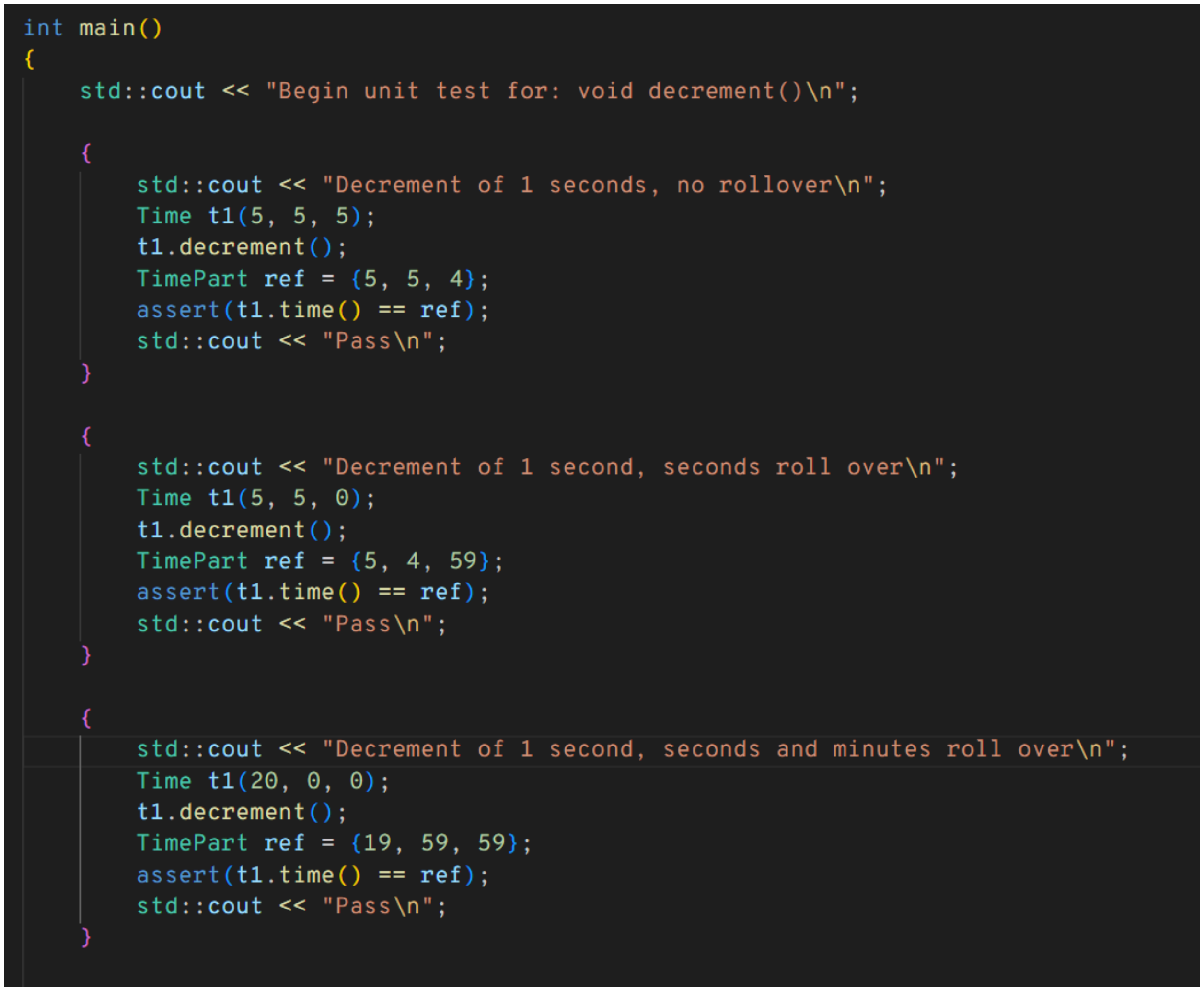
Same idea except we pass second - 1 now.



```cpp
void Time::decrement()
{
    // Same idea as increment.
    TimePart formattedTime = validateTime(_time.hour, _time.minute, _time.second - 1);
    _time.hour = formattedTime.hour;
    _time.minute = formattedTime.minute;
    _time.second = formattedTime.second;
}
```

Void decrement() Testing:

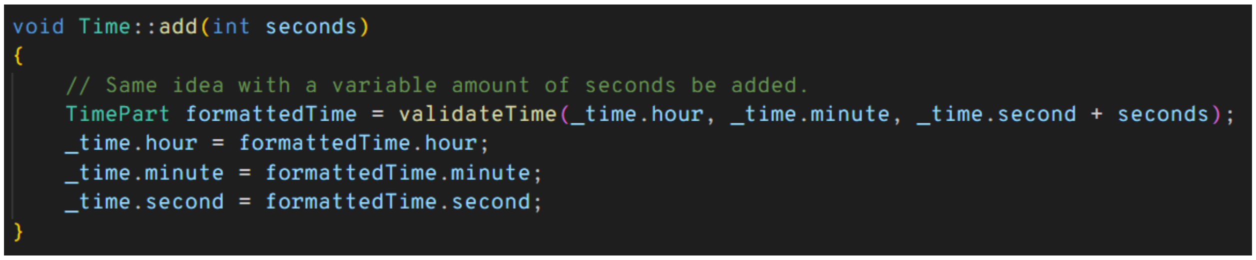
Not much to say, testing that's very similar to increment.



```cpp
int main()
{
    std::cout << "Begin unit test for: void decrement()\n";

    {
        std::cout << "Decrement of 1 seconds, no rollover\n";
        Time t1(5, 5, 5);
        t1.decrement();
        TimePart ref = {5, 5, 4};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Decrement of 1 second, seconds roll over\n";
        Time t1(5, 5, 0);
        t1.decrement();
        TimePart ref = {5, 4, 59};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Decrement of 1 second, seconds and minutes roll over\n";
        Time t1(20, 0, 0);
        t1.decrement();
        TimePart ref = {19, 59, 59};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }
```

Void add(int seconds):

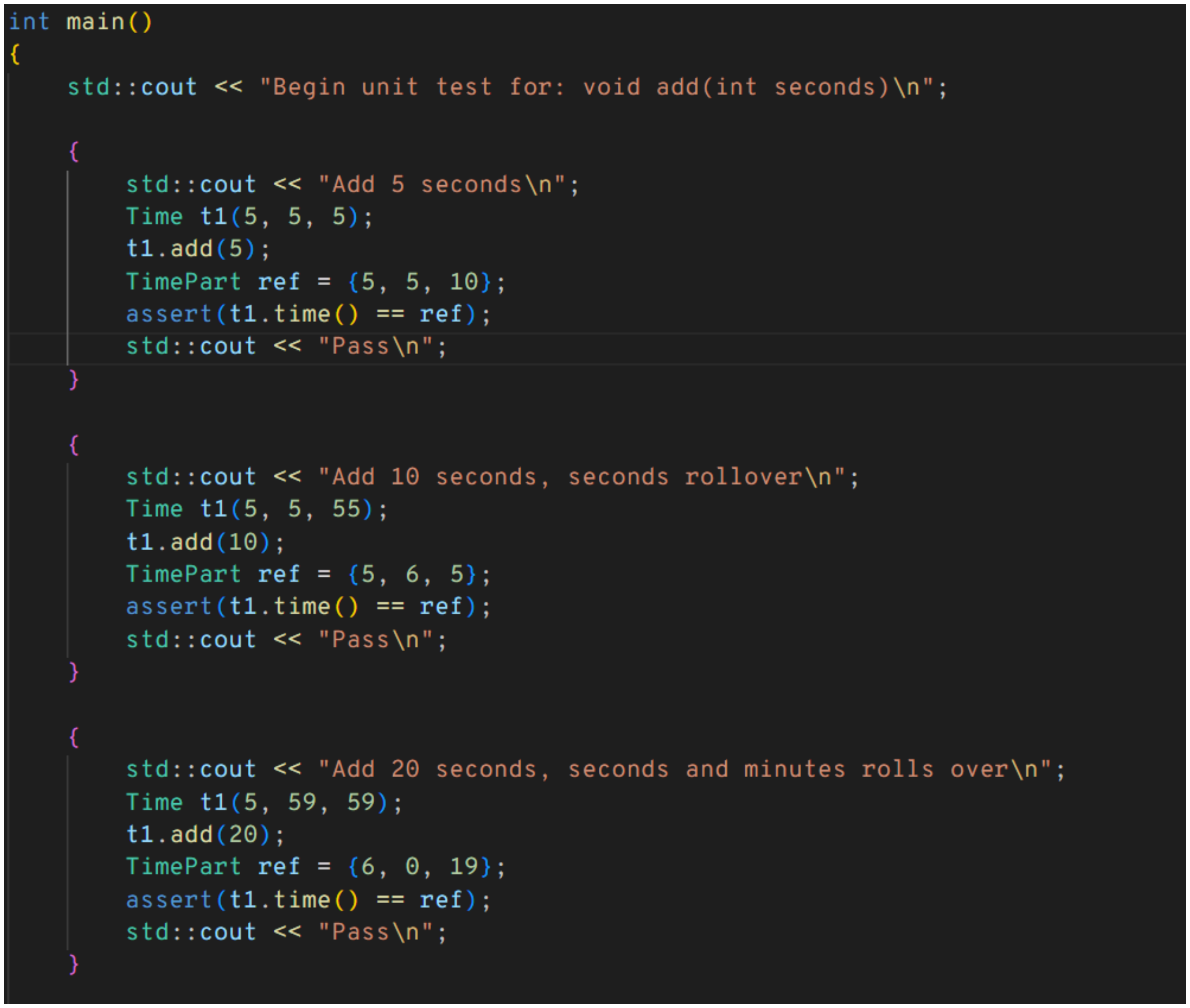
Similar idea to increment/decrement, except we change the hard coded + 1 to a variable we passed instead.



```cpp
void Time::add(int seconds)
{
    // Same idea with a variable amount of seconds be added.
    TimePart formattedTime = validateTime(_time.hour, _time.minute, _time.second + seconds);
    _time.hour = formattedTime.hour;
    _time.minute = formattedTime.minute;
    _time.second = formattedTime.second;
}
```

Void add(int seconds) Testing:

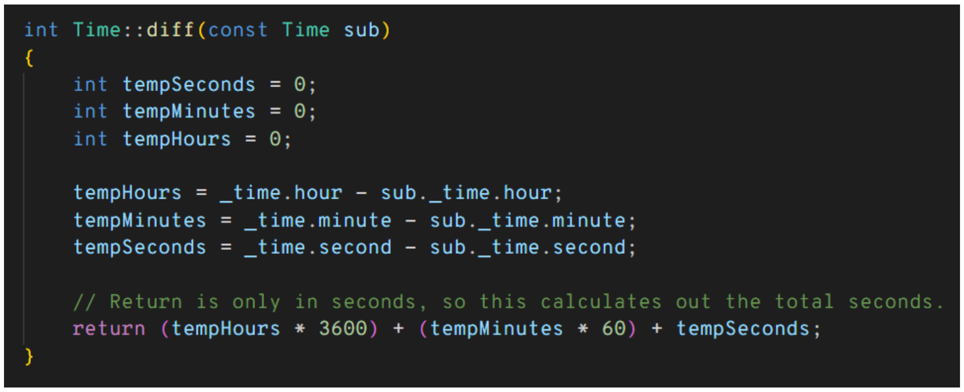
Just like before, focused on the rollover functionality.



```cpp
int main()
{
    std::cout << "Begin unit test for: void add(int seconds)\n";

    {
        std::cout << "Add 5 seconds\n";
        Time t1(5, 5, 5);
        t1.add(5);
        TimePart ref = {5, 5, 10};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Add 10 seconds, seconds rollover\n";
        Time t1(5, 5, 55);
        t1.add(10);
        TimePart ref = {5, 6, 5};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Add 20 seconds, seconds and minutes rolls over\n";
        Time t1(5, 59, 59);
        t1.add(20);
        TimePart ref = {6, 0, 19};
        assert(t1.time() == ref);
        std::cout << "Pass\n";
    }
```

Int diff(const Time sub):

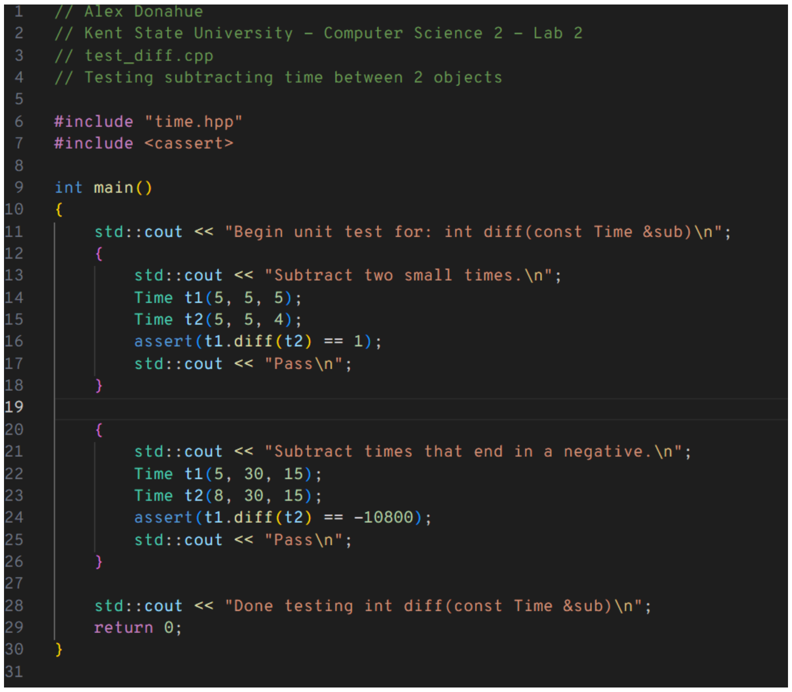
This function is straightforward with the only tricky part being the calculations between hours and minutes. They need to be multiplied so that they are represented as seconds. In hindsight I could have shortened the lines of this function by a few, but I think my implementation is clear enough that it has its own merit.



```cpp
int Time::diff(const Time sub)
{
    int tempSeconds = 0;
    int tempMinutes = 0;
    int tempHours = 0;

    tempHours = _time.hour - sub._time.hour;
    tempMinutes = _time.minute - sub._time.minute;
    tempSeconds = _time.second - sub._time.second;

    // Return is only in seconds, so this calculates out the total seconds.
    return (tempHours * 3600) + (tempMinutes * 60) + tempSeconds;
}
```

Int diff(const Time sub) Testing:

I'll be honest, I only included two tests here because by this point I was a little gassed out, but I made sure that both a positive and negative case were both covered.

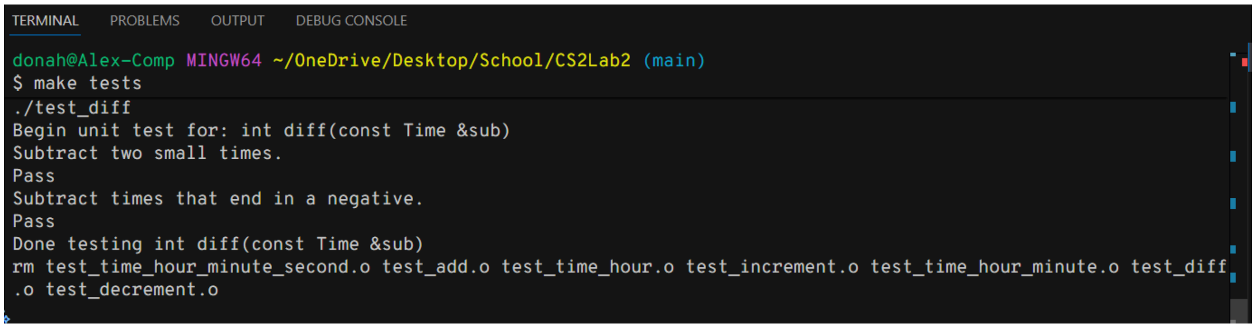


```cpp
// Alex Donahue
// Kent State University - Computer Science 2 - Lab 2
// test_diff.cpp
// Testing subtracting time between 2 objects

#include "time.hpp"
#include <cassert>

int main()
{
    std::cout << "Begin unit test for: int diff(const Time &sub)\n";
    {
        std::cout << "Subtract two small times.\n";
        Time t1(5, 5, 5);
        Time t2(5, 5, 4);
        assert(t1.diff(t2) == 1);
        std::cout << "Pass\n";
    }

    {
        std::cout << "Subtract times that end in a negative.\n";
        Time t1(5, 30, 15);
        Time t2(8, 30, 15);
        assert(t1.diff(t2) == -10800);
        std::cout << "Pass\n";
    }

    std::cout << "Done testing int diff(const Time &sub)\n";
    return 0;
}

```

Conclusion:

An interesting lab that was a bit more dense and time consuming than I expected going in. I'll make sure to budget more time for the upcoming project as this definitely took me close to 15 hours to complete from start to finish. I'm pretty happy with my result but would love to hear any feedback that you have for me, even if it's just about the structure of my program rather than how functional it is.

Also, like I mentioned in the introduction, all my tests passed, I just thought it would be a little excessive to add a bunch of test logs to this already gigantic report. So you know I'm not lying to you, here's the end of my tests finishing.



```
donah@Alex-Comp MINGW64 ~/OneDrive/Desktop/School/CS2Lab2 (main)
$ make tests
./test_diff
Begin unit test for: int diff(const Time &sub)
Subtract two small times.
Pass
Subtract times that end in a negative.
Pass
Done testing int diff(const Time &sub)
rm test_time_hour_minute_second.o test_add.o test_time_hour.o test_increment.o test_time_hour_minute.o test_diff
.o test_decrement.o
```

Let me know if this is in line with what you were expecting from the report. If I'm off or not understanding something I'll happily make adjustments.