

## Time on Your Hands

---

Have you ever had time on your hands? Was it a mess? Let's start this chapter off by writing a program that adds and subtracts time. It should ask the user for two input values—a time (like **3:57**) and duration (such as **1:05**). Then, print the **sum** (here **5:02**) and **difference** (**2:52**)<sup>1</sup>.



This problem is a little more difficult than **H00**. In **H01** you'll learn quite a bit about the C++ Programming Language and dealing with *integers*.

### 1. Planning a Solution

One of the reasons that this program is a little bit more difficult because we there are **several different ways** you can solve it. The first difficulty is figuring out how to **read a value** like **3:57** from the keyboard.

The second question we have to answer is: "how do you want to **represent** the time elements"? You really have three choices:

- As a single **real** (or floating-point) number representing a **fractional hour**. For instance, you can store the time **3:57** as the floating-point number **3.95**.
- As a single **integer** representing **whole minutes**. In this case, the time **3:57** would be stored as the integer **237**.
- As a **pair of integers**, one representing the **hours** and one for the **minutes**.

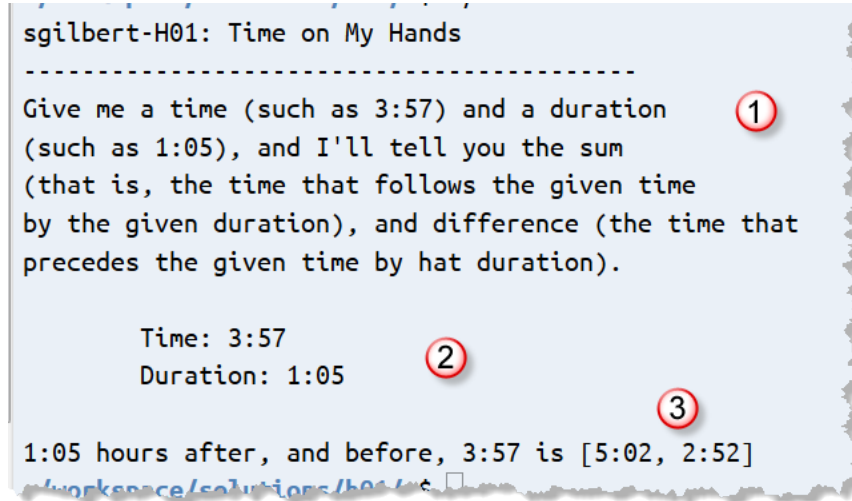
Each has its own benefits and drawbacks. The second is by far the easiest, and so we'll use that.

---

<sup>1</sup> Problem from Doug Cooper's Oh! Pascal, 3<sup>rd</sup> Edition, Chapter 2.

## 2. Mockup the Interactions

Here's what I'd like the interaction to look like:



I've split the interaction into three sections: a title and introduction, an input section, and an output section. Your actual solution will also have a section for processing, so that the basic form for the `run()` function will look like this:

```
int run()
{
    // 1. Title and introduction
    // - blank line
    // 2. Input section: prompt and input on same line
    // 3. Processing section - compute the results
    // - blank line
    // 4. Output section: test data inside brackets [ ]

    return 0;
}
```

## Your Turn

You should have enough information now to add the comments to the starter code and to use the `cout` object to produce the output shown in the mockup at the top of this section. Use the variables `STUDENT` and `ASSIGNMENT` in the first line, and don't hard-code the name and assignment that I have printed. In `H00` we entered the input and output values as **numeric literals**, such as `3.15`. You won't be able to do that with this problem. Instead, use **string literals**, like `"5:02"` to display the mocked-up input and output values.

## 3. The Input Section

Before you can do input, you'll need variables to store the data you receive from the user. Regardless of you choose to represent your data in your calculations, you'll need to get four values from the user:

- The hours and minutes for the current time. These both need to be integers.
- The hours and minutes for the duration. These are also integers.

### Changing Values through Input

Variables can be given values through **input**. In C++ you use the **cin** (*see-in*) object to read from the **standard input stream**. The **extraction, get-from, or input operator** (**>>**) reads a value from input and converts and stores the result in your variable.

You want to read an input that looks like this.

```
Time: 3:57
Duration: 1:05
```

Although this would be fairly difficult in Java, in C++ it is fairly simple. Here's a good **first draft**, with only one small problem:

```
// Prompt and read the input
cout << "    Time: ";
cin >> timeHours;
cin >> timeMinutes;

cout << "    Duration: ";
cin >> durationHours;
cin >> durationMinutes;
```

What's the problem? You haven't handled the colon separating the hours from the minutes! You can fix that in two ways.

#### Method 1 - Dummy Input

You don't want to keep the colon that appears between the two integer variables, but you still need to **read the data** to remove it from the stream. You can do that by introducing a "throw away" variable just for that purpose.

Because the colon is **not a number**, but a single character, the type of the variable should be **char**. This allows you to make your code a little more compact as well.

```
// Prompt and read the input
char discard; // won't keep this
cout << "    Time: ";
cin >> timeHours >> discard >> timeMinutes;

cout << "    Duration: ";
cin >> durationHours >> discard >> durationMinutes;
```

## Method 2 - Unformatted Character Input

Using the extraction operator is not the only way to use the **cin** object. The C++ input stream objects also support **unformatted input** using the traditional "method" syntax you should already be familiar with. Calling the member function (a method in Java terminology) **cin.get()** simply removes and returns one character from the input stream. Using this technique, you can rewrite your code as:

```
// Prompt and read the input
cout << "    Time: ";
cin >> timeHours;
cin.get(); // discard the colon
cin >> timeMinutes;

cout << "    Duration: ";
cin >> durationHours;
cin.get(); // discard next
cin >> durationMinutes;
```

## 4. Processing Some Time

Your next step is **processing**—organizing, analyzing, modifying and manipulating the input to produce our desired output. The simplest kind of processing: using **arithmetic expressions** to perform numerical calculations.

Before you get started, though, remember to **do some planning**. You know from the problem statement that if the user enters a time of **3:57** and a duration of **1:05**, then the **sum** should be **5:02** and the **difference** should be **2:52**.

Let's take a look at some other expected inputs and outputs. Here's an Excel spreadsheet with some inputs and expected outputs.

	A	B	C	D	E
1	<b>Time</b>	<b>Duration</b>	<b>Sum</b>	<b>Difference</b>	
2	3:57	1:05	5:02	2:52	
3	12:00	0:38	12:38	11:22	
4	12:00	1:15	1:15	10:45	
5	5:15	7:59	1:14	9:16	
6					
7					

## Planning the Processing

Just as we had four variables for our input, we'll need **four variables** for output. Let's call them: **sumHours**, **sumMinutes**, **diffHours** and **diffMinutes**

You can use **initialization**, **assignment** and simple integer arithmetic to add or subtract the appropriate time and duration values. As you do, it's important to **mentally trace** the results so you have rough idea whether you've done the calculations correctly.

Here's what happens. Note that I've marked the cells that **don't** work in light blue.

	A	B	C	D	E	F	G
1	<b>Time</b>	<b>Duration</b>	<b>Sum</b>	<b>Difference</b>	<b>Sum</b>	<b>Difference</b>	
2	3:57	1:05	5:02	2:52	4:62	2:52	
3	12:00	0:38	12:38	11:22	12:38	12:-38	
4	12:00	1:15	1:15	10:45	13:15	11:-15	
5	5:15	7:59	1:14	9:16	12:74	-2:-44	
6							

It's obvious that the **naïve** solution yields the **wrong answer**, at least for the addition in the given set of inputs. And, if the addition is wrong, you can be pretty sure that the subtraction is only **accidentally correct**, just like a "stopped clock is correct twice a day".

## Using a Different Approach

Keeping track of the minutes and hours separately **makes the calculations more complex** than they need to be. If you only deal with minutes, the calculations are almost simple. In any time, you can calculate the minutes by adding the hours x 60 to the existing minutes. Here's another spreadsheet that does that.



A	B	C	D	E	F	G	H
Time	Duration	Time-Min	Duration-Min	Sum	Difference	After	Before
3:57	1:05	237	65	302	172	5:02:00 AM	2:52:00 AM
12:00	0:38	720	38	758	682	12:38:00 PM	11:22:00 AM
12:00	1:15	720	75	795	645	1:15:00 PM	10:45:00 AM
5:15	7:59	315	479	794	-164	1:14:00 PM	#NUM!

This looks like it's correct except for one value (when the difference becomes negative). Let's handle that later.

## Start with Pseudocode

Before you start writing code, you should **plan your calculation** using **pseudocode**. Here's my pseudocode for the problem up to this point.

### Input section

*Prompt and read the time (time-hours and time-minutes)  
Prompt and read the duration (dur-hours and dur-minutes)*

### Processing section

```
time <- time-hours x 60 + time-minutes
duration <- dur-hours x 60 + dur-minutes
after <- time + duration
before <- time - duration
after-hours <- after / 60
after-minutes <- after % 60
before-hours <- before / 60
before-minutes <- before % 60
```

### Output section

*print before and after in hh:mm format*

You can keep your pseudocode in a separate document, but you'll find it easier to write as single-line comments into your source code. Then, you won't have to keep opening your text or notes to figure out what's next.

To convert your plan to code, you need to create some variables to hold the output and then write some **assignment statements** along with **arithmetic expressions** to store the values in the correct variables.

## 5. Output & Formatting

You're now ready to go on to the final **output section** where you display the results of all of our calculations. Your initial, "mocked-up" output section looks something like this:

```
cout << endl;
cout << "1:05" << " hours after, and before, "
    << "3:57" << " is [" << "5:02" << ", "
    << "2:52" << "]" << endl;
```

Now, all you have to do is replace each of these with an expression that represents our calculated values. For instance, to print the duration, "1:05", you just replace that string literal with the expression:

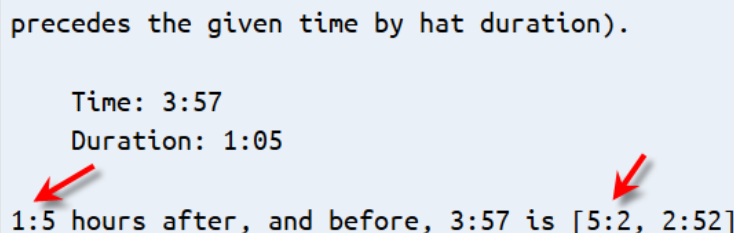
```
durationHours << ":" << durationMinutes
```

Similarly, to print the "after" value "5:02", just replace that literal with:

```
afterHours << ":" << afterMinutes
```

### Output Formatting

You'll notice that the output produces the correct values, but they are **formatted** incorrectly; you are missing the **0** when is less than **10**. If you try a few other examples, you'll see that you are also missing the leading zero, when the hours are zero.



```
precedes the given time by hat duration).
Time: 3:57
Duration: 1:05
1:5 hours after, and before, 3:57 is [5:2, 2:52]
```

Fix this by using some of the **output manipulators** defined inside the `<iomanip>` standard library header. The two manipulators you need are **setw** which changes the **column width** and **setfill** which affects how empty values in a column are filled, like this:

```
cout << setfill('0'); // only needed once
cout << setw(2) << after / 60 << ":" << setw(2) << after % 60;
```

Run the program again, (**make run**) and you'll see you now get the output you expect. Try a couple other examples from the previous section, and you'll see that they should work as well.

## 6. Testing Your Code

While everything looks pretty good up to this point, you **can't be sure** that your program is bug-free. **Testing** is the process of using many different (carefully crafted) inputs to see if you can flush a bug out of hiding.

As with the last chapter, I have used Excel to calculate about 25 different inputs along with the **expected outputs** for each of those input sets. To try your code against my test file, simply run your program by typing: **make test**

```
X Input of 2:30 13:27 : expected [3:57, 1:03] but found [15:57,
X Input of 2:54 14:23 : expected [5:17, 12:31] but found [17:17,
X Input of 3:18 15:19 : expected [6:37, 11:59] but found [18:37,
-----
INSTRUCTOR TESTS H01:sgilbert: ALL TESTS -- PASS 5/23 (22%).
-----
MTU0NjQ3NTc5MipzZ2lsYmVydDpIMDE6MjEuNzQl
```



### Yipee! Some Bugs!

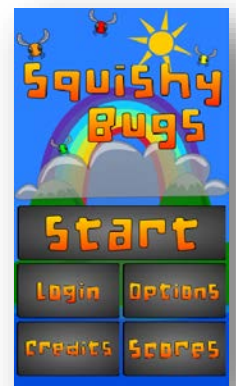
You might not be very happy to see a score of **5/23**, but that's the attitude you must adopt as to gain more success in programming. Every bug that your tests reveal tells you a little bit more about the logic in your program, and brings one more bug out of hiding **so that you can squish it**.

**How do you squish all of the bugs? One at a time.** Let's start with the one I've indicated here, where the after time is displayed incorrectly.

- One the first item, we **expect** a time of **3:57** but see **15:57**.

After a few moment's thought, it should be obvious that those are both the same time, the first using a 12 hour clock, and the second a 24.

How can we fix it? Just change the displaying expression **after / 60** to **after / 60 % 12**. Try it and you'll see that it fixes all but two of the "after" bugs and your score goes from 12% to 57%.



### Negative Remainders

Prior to C++11, division or remainder with **negative operands** was an implementation-defined operation. Some compilers would do it one way while others would do it another.



Also, when negative operands are used, the remainder operator is **not defined the same way that the mathematical modulus operation (Euclidian Division) is defined.**

For those reasons, it is easier to just make sure we don't end up with negative remainders. This is easily done by adding a day (or two or three) to **before** when calculating the difference. Go ahead and do that, and you'll find that your score should go to **87%.**

## The Midnight Hour

Now the only tests that fail are those that come in the **first hour after midnight.** This is not a calculation problem, but a human convention where we write **12:31** instead of **0:31.** A selection statement would make fixing this simple, but we won't have any selection statements until next week.

Can we do it with a simple calculation?

Sure. We need two additional statements that modify the **afterHours** and **beforeHours** variables. For **afterHours**, the pseudocode would be:

```
Add 11 to after-hours  
Find the remainder of dividing by 12  
Add 1 to the result
```

You can do all of this in one line, but remember to put the parentheses in the right place. Converting **beforeHours** is similar.

**Check it again and you'll see you got 100%. Use make submit to turn in your assignment. Use Canvas discussions or come to my office hours if you are having problems. Make sure you start early enough as well.**