Linköping University
Department of Computer and Information Science (IDA)
The UPP group 2021-08-10

Time

Prerequisites

You can write imperative programs in C++. Your program uses functions. Your functions are reusable and have a limited purpose. You can pass data to and from your functions through parameters and return statements. You always use parameter passing instead of global variables. You always use iteration, lookup or functions instead of complete enumeration. You write functions instead of repeating yourself. Your programs include only ever those headers that are used. You compile your programs with all warnings and according to the C++ standard. You can write proper names of functions, variables etc.

Background

You work in a team for SAAB and have gotten the assignment to construct a program module that represents a point in time. The intended users are your collegues that will use this module in their own code when writing *other* programs. This means that your code should not be a complete program, but instead just the functions and structs necessary for using the module. You must however write tests to prove to your colleagues that your module functions as expected.

Aim

When writing code you must be aware of what is prioritized. Some programs need to be fast and effective, others need to be written in such a way that other programmers can easily modifiy it etc. In this assignment you should focus on three aspects:

Correctness

It should not be possible for your colleagues to use your module incorrectly and there should not be any cases that produces faulty or incorrect results. Your code must be written in such a way that it never produces an incorrect results given any correct input. Incorrect usage of your module should *not* result in messages printed to the terminal.

If an error occurs due to incorrect usage then this should be reported to the user. It is important that the user can detect, filter and handle these errors directly in their code. This means that you must consider *how* your colleagues uses your code and how they should detect errors that occurs.

You must write tests that automatically verifies the correctness of your module. Every part of the module must be tested.

Usability

Your module should be as easy to use as possible. This means that your naming must be clear and you must make sure that your functions are called in the most obvious way possible. This also means that your functions should be written in a such a way that it can work in as many cases as possible. Since your target audience are other programmers, this also means that the parameters for functions and data members in structs should be as easy to use and understand as possible.

Constant (`const`) timepoints should be possible to use as long as they are not modified. Your code should be divided into an implementation file and a header file (which can be used by other programmers as a reference manual). It must be clear what code your colleagues should read and what is just implementation details not necessary to understand how the module is used.

Your code must follow C++ conventions as much as possible. Your module should not produce any errors or warnings with the flags supplied in the course.

Changeability

It should be possible to change the implementation of various functions without breaking other programs that uses your module. This means that all function parameters and return values should be independent of *how* your code is written. For example: A function named `get_hour` should return at what hour this timepoint occurs at, regardless of how the timepoint is actually implemented.

Your time struct will be implemented with three integers: one for hours, one for minutes and one for seconds. But your code should be written in such a way that this doesn't matter. You should be able to change your struct in such a way that it just uses seconds elapsed instead without this affecting the uses code. This would of course result in almost all functions having to be rewritten, but this change should not be noticable by other programmers. Meaning they will not have to change anything even though you changed a lot of things. **Note:** you are *not* meant to actually perform this change in the assignment.

Assignment: Time

You will create a data structure and all the functions that are needed to handle time. We defined time as three integers to represent time in the format `HH:MM:SS`. The minimal requirements that must be implemented for time are:

- Must be able to check if the time is valid or not. In other word if the values of hours are between the interval $[0, 23]$, minutes and seconds are between the interval $[0, 59]$. This should be done with a function called `is_valid()`.
- Possibility to get a time as a string. There must be a way to format the time as a 24-hour clock `"14:21:23"`, but also with am or pm `"02:21:23 am"`. The result must be a string in the format `HH:MM:SS[ p]`, where `p` is either am or pm. There must be a function `to_string()` that

return a string in the format given above. There must be an extra parameter to `to_string()` that determines whether it is printed in a 24-hour or 12-hour clock format.

- Must be able to check if the time is before or after noon (am or pm). This function must be called `is_am()`.
- Addition with a positive integer `N` to generate a new time `N` seconds into the future with `operator+`. Eg.

```
Time t{};
t + 5; // The result will have the time [00:00:05]. t is still unchanged
```

- Subtraction with a positive integer to give an earlier time with `operator-`.
- `operator++`, `operator--`

```
Time t{12, 40, 50};
t++; // t will have the time [12:40:51]
--t; // t will have the time [12:40:50]
```

Both operator must have the prefix- and postfix versions.

- Comparison between two `Time` objects with the usual comparison operators (there are six of them). Eg.

```
Time t1{00, 00, 01};
Time t2{12, 30, 40};
t1 > t2; // This is false;
t1 != t2; // This is true;
```

It is important to reuse code. Is it possible to implement some comparison operators by calling the others?

- Formatted output with `operator<<`. `Time` must be printed in the format `HH:MM:SS`.
- Formatted input with `operator>>`. You can always assume that a correct input will be in the format `HH:MM:SS`. The values must check for correct values. Your implementation must check that the read values are correct. Eg. Second and minute must be between 0-59, hour must be between 0-23. The `fail`-flag must be set if an incorrect value are read. `operator>>` normally only read until an error occur and the stop. If you choose to implement this in a different way, it is OK as long as you document it (by writing a comment in the header file).

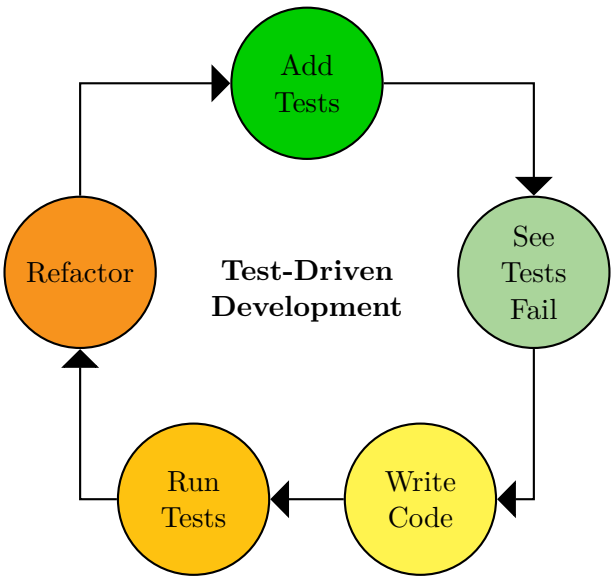
Guide

This section will guide you through test-driven development (TDD): how you find test cases and how you compile efficiently with Catch.

Test-driven development

You must implement this lab assignment with the TDD method - test-driven development. TDD is an iterative work process where you introduce small additions to your code one at a time. You begin by identifying a small addition to the code and then, instead of just implementing it directly, you think about how this addition is meant to be used in the code. Once you have decided on *how* this addition should be used, then you write a program that tests this addition as if it had already been implemented (meaning you haven't implemented it yet). This test program usually consist of multiple small tests that test exactly one thing. Once you have seen that all these tests *fail*, then you implement your addition step by step, one test case at a time until you see that all tests pass. This process is then repeated for each small part of the code until everything has been implemented. Every time the test program is run it must also execute all the *previous* tests. This is to make sure that everything still works after each modification of the code.

The advantage of this method is that your test program and your code will grow together and once the module is complete you have a collection of tests as well. This means that once you are done, you can already be fairly certain that your module does as it is intended since you have tested it as you went along.



When you write a test program you could do it with normal C++-code, for example with if-statements and simple prints. But that is becomes annoying in the long run so in this assignment you will use a framework called Catch. Catch is easy to learn and has good documentation. During the first teaching session you will get an introduction to both test-driven development and Catch.

Example of test-driven development

In this guide we will assume that you want to write a function that checks whether an integer is positive or not.

Start by thinking about how this function is used. Consider what values this function should return for different inputs. Are there any special cases we should make sure works?

For our function we might find it resonable that this function has the following declaration:

```
bool is_positive(int i);
```

So what we might want to test first is that this function returns `true` for positive values. Let's write some test cases:

```
CHECK( is_positive(1) );
CHECK( is_positive(100) );
```

Now the question is: are these tests enough to ensure that `is_positive` has the correct behaviour? **No**.

Consider this implementation:

```
bool is_positive(int i)
{
  return true;
}
```

This passes our tests, which means that the tests aren't good enough since it has the incorrect behaviour. The problem here is that we are not testing the negative cases. So let us add tests for that:

```
CHECK_FALSE( is_positive(-1) );
CHECK_FALSE( is_positive(-17) );
```

Now our dummy implementation of `is_positive` does not pass all tests. However we are not quite there yet. Consider this implementation:

```
bool is_positive(int i)
{
  return i > 0;
}
```

This implementation passes all our tests, but it has one case where it is incorrect, and that is for `0`. `0` is a positive integer, but this implementation considers it a negative integer. Therefore we should add a test for this as well:

```
CHECK( is_positive(0) );
```

Now we arrive at a correct implementation:

```
bool is_positive(int i)
{
  return i >= 0;
}
```

In reality we should write all these test cases before we even implement `is_positive`. The purpose of this demonstration was to show that you need to check both the positive and negative case as well as some corner cases.

Once you are done with this function you can move on to the next one and repeat this process until everything is done.

Compilation flags

```
std=c++17 -Wall -Wextra -Wpedantic -Weffc++ -Wold-style-cast
```

Compiling with Catch

The testing framework in `catch.hpp` is large and you will quickly notice that it is slow to compile. Because of this we recommend that you pre-compile the framework so that all subsequent compilations of your testcases can use the pre-compiled version to speed up compilation. You do this in two steps: create a main program that includes the framework and starts it (given as `text_main.cc`) and the link it with your testcases that you write in a separate file (for example `time_test.cc`). Below are the commands required. You must add the flags that are needed yourself (see above).

1. Compile the file `test_main.cc` but don't link it (don't create an executable file). You do this by adding the compilation flag `-c`:

   ```
   g++ -std=c++17 -c test_main.cc
   ```

   If everything succeeded you will get a file called `time_test.o`. This is what is known as an *object file* which contains pre-compiled code.

2. Add test cases to the separate file `time_test.cc` and your module implementation in `Time.h` and `Time.cc`. When you want to compile your test program you add the pre-compiled file to the command line instead of `test_main.cc`, that way it will be linked together with your module and test cases. This is done like this:

   ```
   g++ -std=c++17 test_main.o Time.cc time_test.cc
   ```

Useful tips and tricks

If you make changes to your files but those changes are not reflected in your executable this *might* be because you have accidentally pre-compiled a header (this happens when you compile a `.h` file). To check if that is the case, you can run the following command:

```
ls -1 *.gch
```

If you get any files, then you should remove those:

```
rm *.gch
```

Then you can compile. **Make sure to not mention any `.h` file in your compilation command**.

Some other useful commands are:

`grep "using namespace" *.h` to check if you are importing names in an header file (which you should *not* do!).

`grep "#include.*cc" *.cc` to check if you are including any implementation files. This is not something you should ever do.

For this assignment you are not allowed to include implementation files, nor are you allowed to import namespaces in header files. So make sure to perform these checks before handing in your solution.

To check if your header guards does their job, you can include the same header file twice. If your code still compiles after that, then your header guards are correct. For example:

```
#include "my_module.h"
#include "my_module.h"
```

Tips for testing

Here are some examples on things you are expected to test in this assignment. We will show you specific examples on how a selection of functions are supposed to work according to conventions. These examples aren't specifically written for this assignment, but the principle is the same. These tests are not exhaustive so it is still your responsibility to test everything in the assignment correctly. You will have to figure out how these examples are translated to the assignment (i.e. if there are more things that needs to be tested and if there are more places where the examples might be useful).

String conversion

The following declarations are assumed to exist for the module that is tested:

```
struct my_complex
{
  // ...
};

std::string to_string(my_complex const&, bool exp_form) const;
```

Below is an example of how this can be tested:

```
my_complex z {0, 16};
CHECK( to_string(z) == "0 + 16i" );
CHECK(( to_string(z, true) == "16e^1.5708i" ));
```

Input from stream

With stringstreams we can create a "fake" `std::cin` with predefined inputs:

```
std::istringstream iss{"1 + 2x "}; // fake input with faulty data
my_complex z {};
iss >> z;
CHECK(iss.fail()); // the input failed
CHECK( to_string(z) == "0 + i" ); // so z was not changed
```

```
std::istringstream iss{"1 + 2i "}; // fake input with correct data
my_complex z {};
iss >> z;
CHECK_FALSE(iss.fail()); // the input succeeded
CHECK( to_string(z) == "1 + 2i" ); // so z got the new value
```

Printing to streams

Stringstreams can also be used to create a "fake" `std::cout` so that we can retrieve what was printed as a string with the `.str()` function:

```
std::ostringstream oss {}; // fake "cout"
my_complex z {1, 2};
oss << z << endl;
CHECK( oss.str() == "1 + 2i" );
```

Increment

```
int i{0};
int j{i++};
CHECK( i == 1 );
CHECK( j == 0 );
```

```
int i{0};
int j{++i};
CHECK( i == 1 );
CHECK( j == 1 );
```

Addition with integers

```
my_complex a{0, 1};
my_complex b{5 + a}; // addition
my_complex c{a + 5}; // addition in reverse order
CHECK( to_string(a) == "0 + i" );
CHECK( to_string(b) == "5 + i" );
CHECK( to_string(c) == "5 + i" );
```