std::chrono

CMSC 240
All examples borrowed/modified from
C++ Crash Course by Josh Lospinoso
No Starch Press
The stdlib Chrono Library

- Provides a variety of clocks in the `<chrono>` header
- Useful for when you want to program something that depends on time or for timing your code
- Provides three clocks, all in the `std::chrono` namespace, with each providing a different guarantee
Aside: The stdlib Chrono Library

- `std::chrono::system_clock` is the system wide real-time clock
  - A.K.A. the *wall clock*
  - Provides elapsed time since an implementation specific start date
    - Most use January 1, 1970 at midnight
Aside: The stdlib Chrono Library

- `std::chrono::steady_clock` guarantees that its value will never decrease
  - Might seem absurd, but measuring time is complicated -- might have to deal with leap seconds and/or inaccurate clocks
- Aside: I once had to deal with real-world situation where triangle inequality failed!
  - So yes, this kind of stuff happens
Aside: The `stdlib Chrono Library`

- `std::chrono::high_resolution_clock` has the shortest *tick* period available
  - *tick* is the smallest atomic change that the clock can measure
    - i.e., the granularity of the clock
- Beware of situations where *tick* is, say, millisecond, but clock is only updated every half second!
  - Mostly a historical issue now
Aside: The stdlib Chrono Library

- Each clock supports the static member function `now()`, which returns a `time point` corresponding to the current value of the clock
- `time point` represents a moment in time
- `chrono` encodes time points using `std::chrono::time_point` type
Aside: The `std::chrono` Library

- Using `time_point` objects is relatively easy.
- They provide a `time_since_epoch()` method that returns the amount of time lapsed between the `time_point` and the clock’s `epoch`.
- This elapsed time is called a `duration`.
Aside: The stdlib Chrono Library

- epoch is an implementation defined reference point denoting the beginning of the clock
- UNIX epoch (or POSIX time) begins on January 1, 1970
- Windows epoch begins January 1, 1601
  - Corresponding to beginning of a 400 year Gregorian–calendar cycle
Aside: The stdlib Chrono Library

- An alternate method to obtain a duration from a `time_point` is to subtract two of them.
- A `std::chrono::duration` represents the time between two `time_point` objects.
- Durations expose a `count()` method that returns the number of clock ticks in the duration.
Each of the `auto` variables are `time_point` objects. And each of these exposes the `time_since_epoch()` method.
• time_since_epoch() returns a duration, and the count() method of that duration returns the number of ticks
Aside: The stdlib Chrono Library

Any clock has a `now()` method

```
now() ➔ time_point
```

Any `time_point` has a `time_since_epoch()` method

```
time_since_epoch() ➔ duration
```

Any `duration` has a `count()` method ➔ number of ticks
Aside: The stdlib Chrono Library

- duration objects can also be constructed directly
- std::chrono namespace contains helper functions for generating durations
- std::chrono::chrono_literals namespace offers User-defined literals for creating durations
Aside: The stdlib Chrono Library

<table>
<thead>
<tr>
<th>Helper function</th>
<th>Literal equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoseconds(3600000000000000)</td>
<td>3600000000000000ns</td>
</tr>
<tr>
<td>microseconds(36000000000)</td>
<td>36000000000us</td>
</tr>
<tr>
<td>milliseconds(360000)</td>
<td>3600000ms</td>
</tr>
<tr>
<td>seconds(3600)</td>
<td>3600s</td>
</tr>
<tr>
<td>minutes(60)</td>
<td>60m</td>
</tr>
<tr>
<td>hours(1)</td>
<td>1h</td>
</tr>
</tbody>
</table>

Note you don’t have to use those exact numerical values. Also, for example, ms is similar to appending L to a long value
Aside: The stdlib Chrono Library

```cpp
#include <chrono>
TEST_CASE("chrono supports several units of measurement") {
    using namespace std::_literals::chrono_literals;
    auto one_s = std::chrono::seconds(1);
    auto thousand_ms = 1000ms;
    REQUIRE(one_s == thousand_ms);
}
```
Aside: The stdlib Chrono Library

- Chrono also supplies the function template
  `std::chrono::duration_cast` which does pretty much what you’d expect: converts a duration from one unit to another (e.g., seconds to minutes)
  - And it works, pretty much how you’d expect
Aside: The stdlib Chrono Library

- `std::chrono::duration_cast`

```cpp
TEST_CASE("chrono supports duration_cast") {
    using namespace std::chrono;
    auto billion_ns_as_s = duration_cast<seconds>(10000000000ns);
    REQUIRE(billion_ns_as_s.count() == 1);
}
```
Aside: The stdlib Chrono Library

- Waiting: You can use durations to specify an amount of time for your program to wait
- stdlib provides additional concurrency primitives in the `<threads>` header
  - Contains the non-member function `std::this_thread::sleep_for`
  - `sleep_for` accepts a duration argument corresponding to how long you want your thread to wait (or “sleep”)

• stdlib provides additional concurrency primitives in the `<threads>` header
#include <thread>
#include <chrono>

TEST_CASE("chrono used to sleep") {
  using namespace std::_literals::chrono_literals;
  auto start = std::chrono::system_clock::now();
  std::this_thread::sleep_for(100ms);
  auto end = std::chrono::system_clock::now();
  REQUIRE(end - start >= 100ms);
}
So Let’s Use This

- Optimizing code requires accurate measurement (to determine how long a particular code path takes)
- Chrono is very useful for this
- The **Stopwatch** class defined in the following (user defined, not in a standard library) is an example of how you can measure time in a code path
- The idea: a **Stopwatch** object keeps a reference to a **duration** object
So Let’s Use This

• When the `Stopwatch` is constructed, the time (via `now()`) is recorded
• When the `Stopwatch` is destructed, the time since the start is recorded
• So, construct your `Stopwatch`, run your task, destruct your `Stopwatch`
Stopwatch

```cpp
struct Stopwatch {
    Stopwatch(std::chrono::nanoseconds& result)
        : result{ result }, start{ std::chrono::high_resolution_clock::now() } {}
    ~Stopwatch() {
        result = std::chrono::high_resolution_clock::now() - start;
    }

private:
    std::chrono::nanoseconds& result;
    const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};
```

- The `result` instance variable is a reference to a duration (with nanosecond granularity)
- `start` is a `time_point` for a `high_resolution_clock`
Stopwatch

When the Stopwatch is constructed, result parameter is assigned to the result instance variable

the time (via `now()`) is recorded
Stopwatch

struct Stopwatch {
    Stopwatch(std::chrono::nanoseconds& result)
        : result{ result }, start{ std::chrono::high_resolution_clock::now() } {}
    ~Stopwatch() {
        result = std::chrono::high_resolution_clock::now() - start;
    }

    private:
    std::chrono::nanoseconds& result;
    const std::chrono::time_point<std::chrono::high_resolution_clock> start;
};

- When the Stopwatch is destructed, result is assigned a duration that records the different between the current time and start
  - Current time is obtained via now()
#include <chrono>
#include <cstdio>

struct Stopwatch {
    Stopwatch(std::chrono::nanoseconds& result)
        : result{ result }, start{ std::chrono::system_clock::now() } {}
    ~Stopwatch() {
        result = std::chrono::system_clock::now() - start;
    }

    private:
    std::chrono::nanoseconds& result;
    const std::chrono::time_point<std::chrono::system_clock> start;
};

int main() {
    const size_t n = 100'000'000;
    std::chrono::nanoseconds elapsed;
    {
        Stopwatch stopwatch{ elapsed };
        volatile double result{ 1.23e45 };
        for (double i = 1; i < n; i++) {
            result /= i;
        }
    }
    auto time_per_addition = elapsed.count() / double{ n };
    printf(" Took %.5gns per division.", time_per_addition);
}

What’s with the apostrophes?
#include <chrono>
#include <cstdio>

struct Stopwatch {
    Stopwatch(std::chrono::nanoseconds& result)
        : result{ result }
        , start{ std::chrono::system_clock::now() } {}
    ~Stopwatch() {
        result = std::chrono::system_clock::now() - start;
    }

    private:
    std::chrono::nanoseconds& result;
    const std::chrono::time_point<std::chrono::system_clock> start;
};

int main() {
    const size_t n = 100'000'000;
    std::chrono::nanoseconds elapsed;
    {
        Stopwatch stopwatch{ elapsed };
        volatile double result{ 1.23e45 }; 
        for (double i = 1; i < n; i++) {
            result /= i;
        }
    }
    auto time_per_addition = elapsed.count() / double{ n };
    printf(" Took %gns per division.", time_per_addition);
}
Using Stopwatch

```cpp
#include <chrono>
#include <cstdio>

struct Stopwatch {
    Stopwatch(std::chrono::nanoseconds& result)
        : result{ result }, start{ std::chrono::system_clock::now() } {}
    ~Stopwatch() {
        result = std::chrono::system_clock::now() - start;
    }

private:
    std::chrono::nanoseconds& result;
    const std::chrono::time_point<std::chrono::system_clock> start;
};

int main() {
    const size_t n = 100'000'000;
    std::chrono::nanoseconds elapsed;
    {
        Stopwatch stopwatch{ elapsed };
        volatile double result{ 1.23e45 }; // What's with the volatile keyword?
        for (double i = 1; i < n; i++) {
            result /= i;
        }
    }
    auto time_per_addition = elapsed.count() / double{ n };
    printf("Took %gns per division.\n", time_per_addition);
}
```
According to the standard: [...] `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation.[...]

```cpp
int main() {
    const size_t n = 100'000'000;
    std::chrono::nanoseconds elapsed;
    {
        Stopwatch stopwatch{ elapsed };
        volatile double result{ 1.23e45 };
        for (double i = 1; i < n; i++) {
            result /= i;
        }
    }
    auto time_per_addition = elapsed.count() / double{ n };
    printf("Took %gns per division.\n", time_per_addition);
}
```
In English: The compiler can see that the value of \( n \) never changes, so it might try to optimize away the `for` loop (thus avoiding the conditional check on each iteration, which can involve fetching the value of the variable \( i \), comparing to \( n \), etc).

```c++
int main() {
    const size_t n = 100'000'000;
    std::chrono::nanoseconds elapsed;
    {
        Stopwatch stopwatch{ elapsed };
        volatile double result{ 1.23e45 };
        for (double i = 1; i < n; i++) {
            result /= i;
        }
    }
    auto time_per_addition = elapsed.count() / double{ n };
    printf("Took %gns per division.\n", time_per_addition);
}
```
In English: volatile says "Don’t do this. Though it looks like the value of \( n \) never changes, it may actually at times change through means of which you may not be aware and/or cannot detect."
In this particular example, we’re trying to time the iterations of the loop, so we don’t want the loop to be optimized out of the executable code. Since \texttt{result} is declared \texttt{volatile}, and appears in the loop, the compiler will not optimize out the loop.

Thanks to StackOverflow:
https://stackoverflow.com/questions/4437527/why-do-we-use-volatile-keyword