Clocks

This lesson gives a brief introduction to clocks and their usage in C++ with the help of interactive examples.

WE'LL COVER THE FOLLOWING

The fact that there are three different types of clocks begs the question: What are the differences?

- **std::chrono::sytem_clock**: is the system-wide real time clock (wall-clock). The clock has the auxiliary functions `to_time_t` and `from_time_t` to convert time points into calendar time
- **std::chrono::steady_clock**: is the only clock to provide the guarantee that you can not adjust it. Therefore, `std::chrono::steady_clock` is the preferred clock to wait for a time duration or until a time point
- **std::chrono::high_resolution_clock**: is the clock with the highest accuracy, but it can simply be an alias for the clocks `std::chrono::system_clock` or `std::chrono::steady_clock`

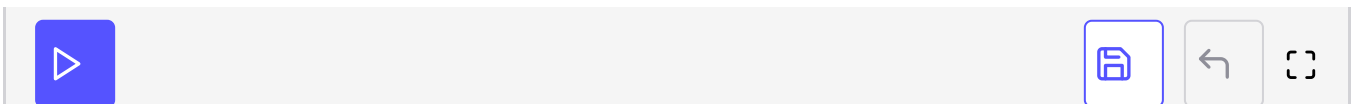
No guarantees about accuracy, starting point, and valid time range

The C++ standard provides no guarantee about the accuracy, the starting point, or the valid time range of the clocks. Typically, the starting point of `std::chrono:system_clock` is the 1.1.1970, the so called UNIX-epoch, while for `std::chrono::steady_clock` it is typically the boot time of your PC.

Accuracy and Steadiness

It is quite interesting to know which clocks are steady and what accuracy they provide. Steady means that the time points can not decrease. You can get the answers directly from the clocks.

```
// clockProperties.cpp

#include <chrono>
#include <iomanip>
#include <iostream>

using namespace std::chrono;
using namespace std;

template <typename T>
void printRatio(){
    cout << "  precision: " << T::num << "/" << T::den << " second " << endl;
    typedef typename ratio_multiply<T,kilo>::type MillSec;
    typedef typename ratio_multiply<T,mega>::type MicroSec;
    cout << fixed;
    cout << "             " << static_cast<double>(MillSec::num)/MillSec::den
                           << " milliseconds " << endl;
    cout << "             " << static_cast<double>(MicroSec::num)/MicroSec::den
                           << " microseconds " << endl;
}

int main(){

    cout << boolalpha << endl;

    cout << "std::chrono::system_clock: " << endl;
    cout << "  is steady: " << system_clock::is_steady << endl;
    printRatio<chrono::system_clock::period>();

    cout << endl;

    cout << "std::chrono::steady_clock: " << endl;
    cout << "  is steady: " << chrono::steady_clock::is_steady << endl;
    printRatio<chrono::steady_clock::period>();

    cout << endl;

    cout << "std::chrono::high_resolution_clock: " << endl;
    cout << "  is steady: " << chrono::high_resolution_clock::is_steady
         << endl;
    printRatio<chrono::high_resolution_clock::period>();

    cout << endl;

}
```



I show in lines 27, 33, and 39 whether or not each clock is steady. The function `printRatio` (lines 10 - 20) is more challenging to read. First, I display the accuracy of the clocks as a fraction with the unit in seconds. Additionally, I use the function template `std::ratio_multiply` and the constants `std::kilo` and `std::mega` to adjust the units to milliseconds and microseconds displayed as floating-point numbers. You can get the details of the calculation at compile time at cppreference.com.

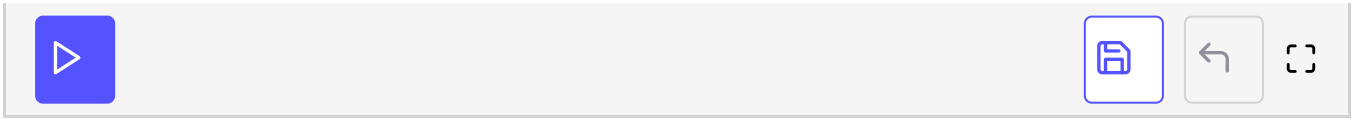
The output on Linux differs from that on Windows.
`std::chrono::system_clock` is far more accurate on Linux;
`std::chrono::high_resultion_clock` is steady on Windows. Although the C++ standard doesn't specify the epoch of the clock, you can calculate it.

Epoch

Thanks to the auxiliary function `time_since_epoch`, each clock returns how much time has passed since the epoch.

```
// now.cpp

#include <chrono>
#include <iomanip>
#include <iostream>

using namespace std::chrono;

template <typename T>
void durationSinceEpoch(const T dur){
    std::cout << "     Counts since epoch:  " << dur.count() << std::endl;
    typedef duration<double, std::ratio<60>> MyMinuteTick;
    const MyMinuteTick myMinute(dur);
    std::cout << std::fixed;
    std::cout << "     Minutes since epoch: "<< myMinute.count() << std::endl;
    typedef duration<double, std::ratio<60*60*24*365>> MyYearTick;
    const MyYearTick myYear(dur);
    std::cout << "     Years since epoch:   " << myYear.count() << std::endl;
}

int main(){

    std::cout << std::endl;

    system_clock::time_point timeNowSysClock = system_clock::now();
    system_clock::duration timeDurSysClock= timeNowSysClock.time_since_epoch();
    std::cout << "system_clock: " << std::endl;
```

```
    durationSinceEpoch(timeDurSysClock);

    std::cout << std::endl;


    const auto timeNowStClock = steady_clock::now();
    const auto timeDurStClock= timeNowStClock.time_since_epoch();
    std::cout << "steady_clock: " << std::endl;
    durationSinceEpoch(timeDurStClock);

    std::cout << std::endl;

    const auto timeNowHiRes = high_resolution_clock::now();
    const auto timeDurHiResClock= timeNowHiRes.time_since_epoch();
    std::cout << "high_resolution_clock: " << std::endl;
    durationSinceEpoch(timeDurHiResClock);

    std::cout << std::endl;

}
```



The variables `timeDurSysClock` (line 26), `timeNowStClock` (line 32), and `timeNowHiResClock` (line 39) contain the amount of time that has passed since the starting point of the corresponding clock. Without automatic type deduction with `auto`, the explicit types of the time point and time duration are extremely verbose to write. In the function `durationSinceEpoch` (lines 9 - 19), I display the time duration in different resolutions. First, I display the number of time ticks (line 11), then the number of minutes (line 15), with the years (lines 18) since the epoch at the end; all values depend on the clock used. For the sake of simplicity, I ignore leap years and assume that a year has 365 days.

`std::chrono::system_clock` and `std::chrono::high_resolution_clock` have the UNIX-epoch as starting point on my linux PC. The starting point of `std::chrono::steady_clock` is the boot time of my PC. While it seems that `std::high_resolution_clock` is an alias for `std::system_clock` on Linux, `std::high_resolution_clock` seems to be an alias for `std::steady_clock` on Windows. This conclusion is in accordance with the result from the previous subsection: *Accuracy and Steadiness*.

Thanks to the time library, you can put a thread to sleep. The arguments of the sleep and wait functions are time points or time durations.