Chapter 38. Boost.Timer

Boost.Timer provides clocks to measure code performance. At first, it may seem like this library competes with Boost.Chrono. However, while Boost.Chrono provides clocks to measure arbitrary periods, Boost.Timer measures the time it takes to execute code. Although Boost.Timer uses Boost.Chrono, when you want to measure code performance, you should use Boost.Timer rather than Boost.Chrono.

Since version 1.48.0 of the Boost libraries, there have been two versions of Boost.Timer. The second version of Boost.Timer has only one header file: `boost/timer/timer.hpp`. The library also ships a header file `boost/timer.hpp`. Do not use this header file. It belongs to the first version of Boost.Timer, which shouldn't be used anymore.

The clocks provided by Boost.Timer are implemented in the classes `boost::timer::cpu_timer` and `boost::timer::auto_cpu_timer`. `boost::timer::auto_cpu_timer` is derived from `boost::timer::cpu_timer` and automatically stops the time in the destructor. It then writes the time to an output stream.

Example 38.1 starts by introducing the class `boost::timer::cpu_timer`. This example and the following examples do some calculations to make sure enough time elapses to be measurable. Otherwise the timers would always measure 0, and it would be difficult to introduce the clocks from this library.

Example 38.1. Measuring time with `boost::timer::cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

Measurement starts when `boost::timer::cpu_timer` is instantiated. You can call the member function `format()` at any point to get the elapsed time. Example 38.1 displays output in the following format: `0.099170s wall, 0.093601s user + 0.000000s system = 0.093601s CPU (94.4%)`.

Boost.Timer measures wall and CPU time. The wall time is the time which passes according to a wall clock. You could measure this time yourself with a stop watch. The CPU time says how much time the program spent executing code. On today's multitasking systems a processor isn't available for a program all the time. A program may also need to halt and wait for user input. In these cases the wall time moves on but not the CPU time.

CPU time is divided between time spent in *user space* and time spent in *kernel space*. Kernel space refers to code that is part of the operating system. User space is code that doesn't belong to the operating system. User space includes your program code and code from third-party libraries. For example, the Boost libraries are included in user space. The amount of time spent in kernel space depends on the operating system functions called and how much time those functions need.

Example 38.2. Stopping and resuming timers

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.stop();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.resume();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

`boost::timer::cpu_timer` provides the member functions `stop()` and `resume()`, which stop and resume timers. In Example 38.2, the timer is stopped before the second `for` loop runs and resumed afterwards. Thus, the second `for` loop isn't measured. This is similar to a stop watch that is stopped and then resumed after a while. The time returned by the second call to `format()` in Example 38.2 is the same as if the second `for` loop didn't exist.

`boost::timer::cpu_timer` also provides a member function `start()`. If you call `start()`, instead of `resume()`, the timer restarts from zero. The constructor of `boost::timer::cpu_timer` calls `start()`, which is why the timer starts immediately when `boost::timer::cpu_timer` is instantiated.

Example 38.3. Getting wall and CPU time as a tuple

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;
  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);

  cpu_times times = timer.elapsed();
  std::cout << times.wall << '\n';
  std::cout << times.user << '\n';
  std::cout << times.system << '\n';
}
```

While `format()` returns the measured wall and CPU time as a string, it is also possible to receive the times in a tuple (see Example 38.3). `boost::timer::cpu_timer` provides the member function `elapsed()` for that. `elapsed()` returns a tuple of type `boost::timer::times`. This tuple has three member variables: **`wall`**, **`user`**, and **`system`**. These member variables contain the wall and CPU times in nanoseconds. Their type is `boost::int_least64_t`.

`boost::timer::times` provides the member function `clear()` to set **`wall`**, **`user`**, and **`system`** to 0.

Example 38.4. Measuring times automatically with `boost::timer::auto_cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <cmath>

using namespace boost::timer;

int main()
{
  auto_cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
}
```

You can measure the wall and CPU time of a code block with `boost::timer::auto_cpu_timer`. Because the destructor of this class stops measuring time and writes the time to the standard output stream, Example 38.4 does the same thing as Example 38.1.

`boost::timer::auto_cpu_timer` provides several constructors. For example, you can pass an output stream that will be used to display the time. By default, the output stream is **`std::cout`**.

You can specify the format of reported times for `boost::timer::auto_cpu_timer` and `boost::timer::cpu_timer`. Boost.Timer provides format flags similar to the format flags supported by Boost.Format or `std::printf()`. The documentation contains an overview of the format flags.