# 18.4 — Timing your code



When writing your code, sometimes you'll run across cases where you're not sure whether one method or another will be more performant. So how do you tell?

One easy way is to time your code to see how long it takes to run. C++11 comes with some functionality in the chrono library to do just that. However, using the chrono library is a bit arcane. The good news is that we can easily encapsulate all the timing functionality we need into a class that we can then use in our own programs.

Here's the class:

```
#include <chrono> // for std::chrono functions

class Timer
{
private:
	// Type aliases to make accessing nested type easier
	using Clock = std::chrono::steady_clock;
	using Second = std::chrono::duration<double, std::ratio<1> >;

	std::chrono::time_point<Clock> m_beg { Clock::now() };

public:
	void reset()
	{
		m_beg = Clock::now();
	}

	double elapsed() const
	{
		return std::chrono::duration_cast<Second>(Clock::now() -
m_beg).count();
	}
};
```

That's it! To use it, we instantiate a Timer object at the top of our main function (or wherever we want to start timing), and then call the elapsed() member function whenever we want to know how long the program took to run to that point.

```
#include <iostream>

int main()
{
    Timer t;

    // Code to time goes here

    std::cout << "Time elapsed: " << t.elapsed() << " seconds\n";

    return 0;
}
```

Now, let’s use this in an actual example where we sort an array of 10000 elements. First, let’s use the selection sort algorithm we developed in a previous chapter:

```
#include <array>
#include <chrono> // for std::chrono functions
#include <cstddef> // for std::size_t
#include <iostream>
#include <numeric> // for std::iota

const int g_arrayElements { 10000 };

class Timer
{
private:
    // Type aliases to make accessing nested type easier
    using Clock = std::chrono::steady_clock;
    using Second = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<Clock> m_beg{ Clock::now() };

public:

    void reset()
    {
        m_beg = Clock::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<Second>(Clock::now() - m_beg).count();
    }
};

void sortArray(std::array<int, g_arrayElements>& array)
{

    // Step through each element of the array
    // (except the last one, which will already be sorted by the time we get there)
    for (std::size_t startIndex{ 0 }; startIndex < (g_arrayElements - 1);
++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered this
iteration
        // Start by assuming the smallest element is the first element of this
iteration
        std::size_t smallestIndex{ startIndex };

        // Then look for a smaller element in the rest of the array
        for (std::size_t currentIndex{ startIndex + 1 }; currentIndex <
g_arrayElements; ++currentIndex)
        {
            // If we've found an element that is smaller than our previously found
smallest
            if (array[currentIndex] < array[smallestIndex])
            {
```

```
                // then keep track of it
                smallestIndex = currentIndex;
            }
        }

        // smallestIndex is now the smallest element in the remaining array
        // swap our start element with our smallest element (this sorts it into the
correct place)
        std::swap(array[startIndex], array[smallestIndex]);
    }
}

int main()
{
    std::array<int, g_arrayElements> array;
    std::iota(array.rbegin(), array.rend(), 1); // fill the array with values 10000
to 1

    Timer t;

    sortArray(array);

    std::cout << "Time taken: " << t.elapsed() << " seconds\n";

    return 0;
}
```

On the author's machine, three runs produced timings of 0.0507, 0.0506, and 0.0498. So we can say around 0.05 seconds.

Now, let's do the same test using std::sort from the standard library.

```
#include <algorithm> // for std::sort
#include <array>
#include <chrono> // for std::chrono functions
#include <cstddef> // for std::size_t
#include <iostream>
#include <numeric> // for std::iota

const int g_arrayElements { 10000 };

class Timer
{
private:
    // Type aliases to make accessing nested type easier
    using Clock = std::chrono::steady_clock;
    using Second = std::chrono::duration<double, std::ratio<1> >;

    std::chrono::time_point<Clock> m_beg{ Clock::now() };

public:

    void reset()
    {
        m_beg = Clock::now();
    }

    double elapsed() const
    {
        return std::chrono::duration_cast<Second>(Clock::now() - m_beg).count();
    }
};

int main()
{
    std::array<int, g_arrayElements> array;
    std::iota(array.rbegin(), array.rend(), 1); // fill the array with values 10000
to 1

    Timer t;

    std::ranges::sort(array); // Since C++20
    // If your compiler isn't C++20-capable
    // std::sort(array.begin(), array.end());

    std::cout << "Time taken: " << t.elapsed() << " seconds\n";

    return 0;
}
```

On the author's machine, this produced results of: 0.000693, 0.000692, and 0.000699. So basically right around 0.0007.

In other words, in this case, std::sort is 100 times faster than the selection sort we wrote ourselves!

Things that can impact the performance of your program

Timing a run of your program is fairly straightforward, but your results can be significantly impacted by a number of things, and it's important to be aware of how to properly measure and what things can impact timing.

First, make sure you're using a release build target, not a debug build target. Debug build targets typically turn optimization off, and that optimization can have a significant impact on the results. For example, using a debug build target, running the above std::sort example on the author's machine took 0.0235 seconds -- 33 times as long!

Second, your timing results may be influenced by other things your system may be doing in the background. Make sure your system isn't doing anything CPU, memory, or hard drive intensive (e.g. playing a game, searching for a file, running an antivirus scan, or installing a update in the background). Seemingly innocent things, like idle web browsers, can temporarily spike your CPU to 100% utilization when the active tab rotates in a new ad banner and has to parse a bunch of javascript. The more apps you can shut down before measuring, the less variance in your results you are likely to have.

Third, if your program uses a random number generator, the particular sequence of random numbers generated may impact timing. For example, if you're sorting an array filled with random numbers, the results will likely vary from run to run because the number of swaps required to sort the array will vary from run to run. To get more consistent results across multiple runs of your program, you can temporarily seed your random number generator with a literal value (rather than std::random_device or the system clock), so that it generates the same sequence of numbers with each run. However, if your program's performance is highly dependent on the particular random sequence generated, this can also lead to misleading results overall.

Fourth, make sure you're not timing waiting for user input, as how long the user takes to input something should not be part of your timing considerations. If user input is required, consider adding some way to provide that input that does not wait on the user (e.g. command line, from a file, having a code path that routes around the input).

Measuring performance

When measuring the performance of your program, gather at least 3 results. If the results are all similar, these likely represent the actual performance of your program on that machine. Otherwise, continue to take measurements until you have a cluster of similar results (and understand which other results are outliers). It's not uncommon to have one or more outliers due to your system doing something in the background during some of those runs.

If your results have a lot of variance (and aren't clustering well), your program is likely either being significantly affected by other things happening on the system, or by the effects of randomization within your application.

Because performance measurements are impacted by so many things (particularly hardware speed, but also OS, apps running, etc…), absolute performance measurements (e.g. "the program runs in 10 seconds") are generally not that useful outside of understanding how well the program runs on one particular machine you care about. On a different machine, that same program may run in 1 second, 10 seconds, or 1 minute. It's hard to know without actually measuring across a spectrum of different hardware.

However, on a single machine, relative performance measurements can be useful. We can gather performance results from several different variants of a program to determine which variant is the most performant. For example, if variant 1 runs in 10 seconds and variant 2 runs in 8 seconds, variant 2 is probably going to be faster on all similar machines regardless of the absolute speed of that machine.

After measuring the second variant, a good sanity check is to measure the first variant again. If the results of the first variant are consistent with your initial measurements for that variant, then the result of both variants should be reasonably comparable. For example, if variant 1 runs in 10 seconds, and variant 2 runs in 8 seconds, and then we measure variant 1 again and get 10 seconds, then we can reasonably conclude that the measurements for both variants were fairly measured, and that variant 2 is faster.

However, if the results of the first variant are no longer consistent with your initial measurements for that variant, then something has happened on the machine that is now affecting performance, and it will be hard to tell whether differences in measurement are due to the variant or due to the machine itself. In this case, it's best to discard the existing results and re-measure.