8.16 — 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 functional we need into a class that we can then use in our own programs.

Here's the class:

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

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

    std::chrono::time_point<clock_t> m_beg;

public:
    Timer() : m_beg(clock_t::now())
    {
    }

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

    double elapsed() const
    {
        return std::chrono::duration_cast<second_t>(clock_t::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.

```cpp
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:

```cpp
#include <iostream>
#include <array>
#include <chrono> // for std::chrono functions

const int g_arrayElements = 10000;

class Timer
```

```cpp
 8    {
 9    private:
10        // Type aliases to make accessing nested type easier
11        using clock_t = std::chrono::high_resolution_clock;
12        using second_t = std::chrono::duration<double, std::ratio<1> >;
13
14        std::chrono::time_point<clock_t> m_beg;
15
16    public:
17        Timer() : m_beg(clock_t::now())
18        {
19        }
20
21        void reset()
22        {
23            m_beg = clock_t::now();
24        }
25
26        double elapsed() const
27        {
28            return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
29        }
30    };
31
32    void sortArray(std::array<int, g_arrayElements> &array)
33    {
34
35        // Step through each element of the array
36        // (except the last one, which will already be sorted by the time we get there)
37        for (int startIndex = 0; startIndex < g_arrayElements - 1; ++startIndex)
38        {
39            // smallestIndex is the index of the smallest element we've encountered this iteration
40            // Start by assuming the smallest element is the first element of this iteration
41            int smallestIndex = startIndex;
42
43            // Then look for a smaller element in the rest of the array
44            for (int currentIndex = startIndex + 1; currentIndex < g_arrayElements; ++currentIndex)
45            {
46                // If we've found an element that is smaller than our previously found smallest
47                if (array[currentIndex] < array[smallestIndex])
48                    // then keep track of it
49                    smallestIndex = currentIndex;
50            }
51
52            // smallestIndex is now the smallest element in the remaining array
53            // swap our start element with our smallest element (this sorts it into the correct place)
54            std::swap(array[startIndex], array[smallestIndex]);
55        }
56    }
57
58    int main()
59    {
60        std::array<int, g_arrayElements> array;
61        for (int i = 0; i < g_arrayElements; ++i)
62            array[i] = g_arrayElements - i;
63
64        Timer t;
65
66        sortArray(array);
67
68        std::cout << "Time taken: " << t.elapsed() << " seconds\n";
69
70        return 0;
71    }
```

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.

```cpp
#include <iostream>
#include <array>
#include <chrono> // for std::chrono functions
#include <algorithm> // for std::sort

const int g_arrayElements = 10000;

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

    std::chrono::time_point<clock_t> m_beg;

public:
    Timer() : m_beg(clock_t::now())
    {
    }

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

    double elapsed() const
    {
        return std::chrono::duration_cast<second_t>(clock_t::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 (int 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
        int smallestIndex = startIndex;

        // Then look for a smaller element in the rest of the array
        for (int 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;
    for (int i = 0; i < g_arrayElements; ++i)
        array[i] = g_arrayElements - i;

    Timer t;
```

```
66
67      std::sort(array.begin(), array.end());
68
69      std::cout << "Time taken: " << t.elapsed() << " seconds\n";
70
71      return 0;
72  }
```

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!

A few caveats about timing

Timing is straightforward, but your results can be significantly impacted by a number of things, and it's important to be aware of what those things are.
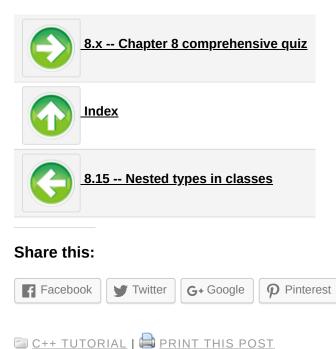
First, make sure you're using a release build target, not a debug build target. Debug build targets typically turn optimization off, and that optimization can have a significant impact on the results. For example, using a debug build target, running the above std::sort example on the author's machine took 0.0235 seconds -- 33 times as long!

Second, your timing results will be influenced by other things your system may be doing in the background. For best results, make sure your system isn't doing anything CPU or memory intensive (e.g. playing a game) or hard drive intensive (e.g. searching for a file or running an antivirus scan).

Then measure at least 3 times. If the results are all similar, take the average. If one or two results are different, run the program a few more times until you get a better sense of which ones are outliers. Note that seemingly innocent things, like web browsers, can temporary spike your CPU to 100% utilization when the site you have sitting in the background rotates in a new ad banner and has to parse a bunch of javascript. Running multiple times helps identify whether your initial run may have been impacted by such an event.

Third, when doing comparisons between two sets of code, be wary of what may change between runs that could impact timing. Your system may have kicked off an antivirus scan in the background, or maybe you're streaming music now when you weren't previously. Randomization can also impact timing. If we'd sorted an array filled with random numbers, the results could have been impacted by the randomization. Randomization can still be used, but ensure you use a fixed seed (e.g. don't use the system clock) so the randomization is identical each run. Also, 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.

Finally, note that results are only valid for your machine's architecture, OS, compiler, and system specs. You may get different results on other systems that have different strengths and weaknesses.

8.x -- Chapter 8 comprehensive quiz

Index

8.15 -- Nested types in classes

Share this:

Facebook   Twitter   G+ Google   Pinterest

33 comments to 8.16 — Timing your code