8.16 — Timing your code

BY ALEX ON JANUARY 4TH, 2018 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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_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.

```
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 <iostream>
```

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

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 pl
        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;

    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 <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 pl
        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;

    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!

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 temporarily 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

C++ TUTORIAL | PRINT THIS POST

61 comments to 8.16 — Timing your code

Connor
January 19, 2020 at 2:33 pm · Reply

Hi, I've done a bit more searching around on the chrono library and I came across some more clock types: system_clock and steady_clock. What is the difference between system_clock, steady_clock and high_resolution_clock? Thanks, loved the lesson :)

Alex
January 22, 2020 at 11:16 pm · Reply

https://solarianprogrammer.com/2012/10/14/cpp-11-timing-code-performance/ has a good summary of these three clocks.

Connor
January 28, 2020 at 10:59 am · Reply

Really helpful! Thank you :)

Ged
December 30, 2019 at 5:55 am · Reply

Why do we need to use const here?

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

nascardriver
December 30, 2019 at 5:58 am · Reply

Every member function that doesn't modify the object should be const. Non-const member functions can't be used with const objects.

manasra
October 16, 2019 at 12:48 pm · Reply

how to configure it please?

Jordy
September 7, 2019 at 5:57 am · Reply

Third to last paragraph: temporary instead of temporarily

Alex
September 10, 2019 at 8:41 am · Reply

Typo fixed, thanks!

Ares
May 19, 2019 at 2:01 pm · Reply

Alex, do you memorise all this stuff or do you copy the code from somewhere when you need it?
I can not imagine for the life of me remembering the class. Even if I memorise it today I would not be able to redo it in a month.

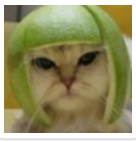
Alex
May 20, 2019 at 10:18 am · Reply

Copy it for sure!
IMO, C++ is 50% knowing things and 50% being able to remember where to look things up.

Ares
May 20, 2019 at 12:41 pm · Reply

Through out the entire tutorial up until now I wondered about this, "is he some wizard!?".

Perhaps devoting an article on this early on might be a good idea! It really made me feel stupid at times wondering whether or not this is something to memorise or not.



Alex
May 23, 2019 at 6:30 pm · Reply

"I Put on My Robe and Wizard Hat". :)

I'll find a convenient place to mention this.

Helliarc
April 6, 2019 at 4:27 pm · Reply

Ah, you guys did a race too! If you didn't see my comment before, my friends and I had a race I made from calculating prime numbers. I lost... But on std::sort I get: 0.10000 seconds. You must be using a beast single core CPU with a TON of cache! I'm on a Ryzen 2700x, definitely under-performs on a single core...

Dimitri
May 29, 2019 at 11:32 pm · Reply

Don't forget to change from Debug to Release. I have a 4-th generation i7 and was also surprised that my computer is 20 times weaker, lol :)) After changing to Release, the code was executed seven times faster than Alex (0.0001)

sonngo
December 18, 2018 at 6:51 am · Reply

Hey Alex, what is your computer specs? I have a notebook with i5-3210m, 4gb ram. Everytime I run the selection sort algorithm with 1000 double and the timer's output is always 0.

nascardriver
December 18, 2018 at 8:04 am · Reply

1000 elements is very few for a computer and it should take close to 0 time units. However, I find exactly 0 unlikely. Do you get positive numbers when you increase the array's size?

What's the output of

```
1 std::cout << sizeof(double) << '\n';
```

sonngo
February 5, 2019 at 7:46 am · Reply

It say 8.I have increased the array's size to 10000 element but it always output 0.

Wulfric Lee
August 29, 2019 at 4:09 am · Reply

So Alex uses 1000 elements actually, my specs are i5-8300H, 16GB DDR4, and my result is about 1.5 seconds, so when I saw Alex's 0.05 seconds I was like: "What???

Did Alex use a NASA computer to run this?"

nascardriver
August 29, 2019 at 4:23 am · Reply

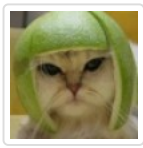
Sounds like you might be compiling in debug mode. Compile in release mode and enable optimizations (If that's not part of release mode already).

Gili
October 20, 2018 at 8:07 pm · Reply

I think I found a typo.

"we can easily encapsulate all the timing **functional** we need"

"functional" should be replaced by "functionality"



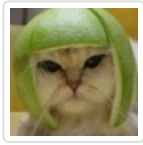
Alex
October 22, 2018 at 1:35 pm · Reply

Type fixed. Thanks!

Olga
August 4, 2018 at 12:02 pm · Reply

Additional, what is the meaning of this part?

```
1 return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
```



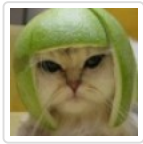
Alex
August 7, 2018 at 12:18 pm · Reply

This converts the time elapsed between when the clock was initialized and the current time into a double value representing seconds.

Olga
August 4, 2018 at 4:30 am · Reply

Can you explain please what these lines mean?

```
1 using clock_t = std::chrono::high_resolution_clock;
2 using second_t = std::chrono::duration<double, std::ratio<1> >;
3
4 std::chrono::time_point<clock_t> m_beg;
```



Alex
August 7, 2018 at 11:59 am · Reply

The using statements are acting as type aliases (see "type aliases" in the site index for more information). The third statement is defining a new member named m_beg of type std::chrono::time_point. You've seen examples of both the prefixing (e.g. std:: prefix on almost

everything) and the angled brackets (see the examples of std::array or std::vector. These are actually template instantiations -- we cover templates in chapter 13.

Sod
July 15, 2018 at 4:45 am · Reply

Instead of using chrono and actually time programs, won't it be much more sensible to count clocktics instead? (and get dynamic and hard disk calls separately, as thay are slower by changing by the machine)

Alex
July 17, 2018 at 4:18 pm · Reply

Not sure what you mean by clock ticks in this context -- ticks in reference to a clock typically just means a count of some time unit (e.g. microseconds). Are you instead trying to suggest we count via cpu cycles or some other cpu related metric?

DecSco
June 26, 2018 at 6:12 am · Reply

You are using single quotation marks around ' seconds\n'. It should be double quotation marks, I think.

Alex
June 29, 2018 at 2:27 pm · Reply

Indeed. Thanks for pointing that out!

Peter Baum
May 25, 2018 at 1:47 pm · Reply

Maybe it will be helpful to have a practical timing example which demonstrates some of the validity issues that we have been considering here. The following is code that tests two different implementations of the binary search from a previous lesson. Here is a test program to try to determine which of two approaches is better.

```
1  #include "stdafx.h"
2  #include <iostream>
3  #include <string>
4  #include <chrono> // for std::chrono functions
5  #include <array>
6  #include <random>
7
8  class Timer
9  {
10 private:
11     // Type aliases to make accessing nested type easier
12     using clock_t = std::chrono::high_resolution_clock;
13     using second_t = std::chrono::duration<double, std::ratio<1> >;
14
15     std::chrono::time_point<clock_t> m_beg;
16
17 public:
18     Timer() : m_beg {clock_t::now()} {}
```

```
19
20      void reset() { m_beg = clock_t::now(); }
21
22      double elapsed() const
23      {
24          return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
25      }
26  };
27
28
29  // Peter version
30    int binarySearch3(int * &array, int target, int min, int max)
31    { // binarySearch
32
33        // normalize min and max so that we know the target is > min and < max
34        if (target <= array[min]) // if min not normalized
35        { // target <= array[min]
36            if (target == array[min]) return min;
37            return -1;
38        } // end target <= array[min]
39          // min is now normalized
40
41        if (target >= array[max]) // if max not normalized
42        { // target >= array[max]
43            if (target == array[max]) return max;
44            return -1;
45        } // end target >= array[max]
46          // max is now normalized
47
48        while (min + 1 < max)
49        { // delta >=2
50          int tempi = min + ((max - min) >> 1); // point to index approximately in the middl
51          int atempi = array[tempi]; // just in case the compiler does not optimize this
52            if (atempi > target)max = tempi; // if the target is smaller, we can decrease ma
53          else if (atempi < target)min = tempi; // the target is bigger, so we can increase
54              else return tempi; // if we found the target, return with the index
55              // Note that it is important that this test for equality is last because it ra
56        } // end delta >=2
57        return -1; // nothing in between normalized min and max
58    } // end binarySearch
59
60
61
62
63  // Alex version
64  int binarySearch(int * &array, int target, int min, int max)
65  {
66      while (min <= max)
67      {
68          // implement this iteratively
69          int midpoint = min + (max - min) / 2; // this way of calculating midpoint avoids o
70
71          if (array[midpoint] > target)
72          {
73              // if array[midpoint] > target, then we know the number must be in the lower h
74              // we can use midpoint - 1 as the upper index, since we don't need to retest t
75              max = midpoint - 1;
76          }
77          else if (array[midpoint] < target)
78          {
79              // if array[midpoint] < target, then we know the number must be in the upper h
80              // we can use midpoint + 1 as the lower index, since we don't need to retest t
```

```
            min = midpoint + 1;
        }
        else
            return midpoint;
    }
    return -1;
}

constexpr int arraySize{ 1000000 }; // number of elements in the array
constexpr int spread = 200; // maximum spread of sorted integers
constexpr int startElement = 100; // value of element 0 of the array
constexpr int testCount = 5; // the number of tests to perform
/************************ MAIN ***************************/
int main()
{
    double temp0; // used to prevent std::cout from interferring with timing
    double temp1; // and used for percentage calculations
    double alexArray[testCount]{};
    double peterArray[testCount]{};
    std::mt19937 mt_rand(23); // create a pseudo random number generator
    int * sortedArray=new int[arraySize]; // array that holds sorted list of target intege
    sortedArray[0] = startElement; // starting element of sorted array
    // created the sorted array
    for (int i = 1; i < arraySize; ++i)
    { // create the sorted array
        sortedArray[i] = sortedArray[i - 1] + mt_rand() % spread; // on average, about spr
    } // end create the sorted array

    Timer t; // create the timing object
                    // first binary search method examined

    // timing tests start here *********
    for (int j = 0; j < testCount; ++j)
    { // for each test
        // determine the time to search for elements within the sorted array



    // second binary search method examined
        t.reset();
        for (int i = startElement; i < arraySize*spread / 2; ++i)// used to ignore first s
    //    for (int i = 0; i < arraySize*spread / 2; ++i) // use if you want to include spe
    //    for (int i = 0; i < spread; ++i) // special test
        {
            int index = binarySearch3(sortedArray, i, 0, arraySize - 1);
        }
        peterArray[j] = t.elapsed();// make sure the results are not interfered with by co

// first binary search method examined
        t.reset(); // just to make sure both trials start after the exact same relative st
        for (int i = startElement; i < arraySize*spread / 2; ++i)  // used to ignore first
    //    for (int i = 0; i < arraySize*spread / 2; ++i) // use if you want to include spe
    //    for (int i = 0; i < spread; ++i) // special test
        {
            int index = binarySearch(sortedArray, i, 0, arraySize - 1);
        }
        alexArray[j] = t.elapsed(); // make sure the results are not interfered with by co


    } // for each test


```

```
// output test statistics
    for (int j = 0; j < testCount; ++j)
    {
        temp0 = alexArray[j];
        temp1 = peterArray[j];
        std::cout << " Alex Binarysearch Time elapsed: " << temp0 << " seconds." << std::e
        std::cout << "Peter Binarysearch3 Time elapsed: " << temp1 << " seconds." << std::
        std::cout << "Normalized version is " << 100 * (temp0 - temp1) / temp0 << "% faste
    }




    std::system("pause");
    return 0;
}
```

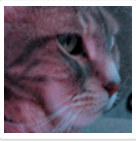
If I just run the program as a stand-alone .exe file (i.e., not within an IDE) and don't bother to disable the network or virus scanner, I get these results:

Microsoft Windows [Version 10.0.16299.431]
(c) 2017 Microsoft Corporation. All rights reserved.
Alex Binarysearch Time elapsed: 8.2431 seconds.
Peter Binarysearch3 Time elapsed: 6.84111 seconds.
Normalized version is 17.008% faster.
Alex Binarysearch Time elapsed: 8.23866 seconds.
Peter Binarysearch3 Time elapsed: 6.85477 seconds.
Normalized version is 16.7976% faster.
Alex Binarysearch Time elapsed: 8.79291 seconds.
Peter Binarysearch3 Time elapsed: 6.85149 seconds.
Normalized version is 22.0794% faster.
Alex Binarysearch Time elapsed: 8.23806 seconds.
Peter Binarysearch3 Time elapsed: 7.46674 seconds.
Normalized version is 9.36288% faster.
Alex Binarysearch Time elapsed: 8.25212 seconds.
Peter Binarysearch3 Time elapsed: 6.85163 seconds.
Normalized version is 16.9713% faster.

So from a practical point of view, should we conclude that the Normalized version of a binary search is best? Without additional information, should we definitely use that method?

Alex
June 1, 2018 at 3:29 pm · Reply

Yes, we can reasonably conclude that the normalized version is faster. Look how nice and clustered those times are -- with the exception of the 3rd Alex and 4th Peter, they're all within hundredths of a second per run each time.



Peter Baum
June 3, 2018 at 1:17 pm · Reply

Then should we update (or add to) the solution to 3a) of the comprehensive quiz for chapter 7 (http://www.learncpp.com/cpp-tutorial/7-x-chapter-7-comprehensive-quiz/) to show this better implementation of a binary search?

Alex
June 6, 2018 at 4:34 pm · Reply

We could, but the point of these tutorials is to teach C++ fundamentals, not be an archive of the best data structures and algorithms. As such, I generally favor simplicity over performance.

Peter Baum
June 7, 2018 at 12:27 pm · Reply

Hi Alex,
A couple of thoughts...
1. It isn't that much more complicated.
2. Perhaps, as a minimum, there should information so students aren't misled into thinking that the implementation is optimum. Same for the sort algorithm.

Anastasia
August 23, 2019 at 3:47 pm · Reply

I'm just curious whether the normalized version's speed boost compared to the Alex's variant is (to some extent) due to using right-shift (>>) bitwise operator as opposed to the ordinary division one?

Running the original version of the code above I've got the results close to the author's (~18-20% faster norm.version). Changing line 69 (Alex binarySearch) from

```
1 int midpoint = min + (max - min) / 2;
```

to

```
1 int midpoint = min + ((max - min) >> 1);
```

the difference in speed was reduced to no more than 4.7%.

Next step was to change line 50 (Peter binarySearch) so it uses division instead of the shift:
from

```
1 int tempi = min + ((max - min) >> 1);
```

to

```
1 int tempi = min + (max - min) / 2;
```

After that the normalized version became actually slower than Alex's one (~ -13.5%).

edit:wrong line numbers

nascardriver
August 24, 2019 at 2:17 am · Reply

gcc, clang and msvc (The 3 major compilers) all turn the division by 2 into a right shift. The time difference you're observing is caused by something else or you didn't enable compiler optimizations.
These are the little things that should be left to the compiler. Favor readability. If you want to divide, divide.

Anastasia
August 24, 2019 at 8:09 am · Reply

Yep, compiled with gcc 9.1.0 without optimizations. A bit surprising for me how big an impact on the speed one little operator has.

Peter Baum
May 25, 2018 at 12:55 pm · Reply

I previously posted about some of the problems I encountered trying to do valid program timing under Microsoft Windows. I did a little more work trying to see if I could run a timing program under the Microsoft Windows recovery disk OS to avoid some of the timing pitfalls that one encounters trying to use Windows 10. You start by creating a recovery disk or recovery USB and bring up that OS using one of the BIOS routines. It is possible to get to a command prompt (DOS box) from this free OS by a menu item under the troubleshooting menu. When I tried to run a timing .exe, I got the message "This version of [program name.exe] is not compatible with the version of Windows you're running." It appears that the OS being used for recovery in my case is actually Windows PE.

The issues then becomes:

1. Would Windows PE introduce its own set of timing issues?

2. How should your IDE be configured so that a program will run under that operating system?

3. Are there limits to the kind of program that will run under Windows PE that will make this approach unsuitable?

At https://docs.microsoft.com/en-us/windows-hardware/manufacture/desktop/winpe-create-apps I found some information that will help with issues 2 and 3. For example,

a) The program will not run if it uses MFC or ATL. MFC (Microsoft Foundation Classes) provide a C++ object-oriented wrapper over Win32 for rapid development of native desktop applications. ATL (Active Template Library) is a wrapper library that simplifies COM development and is used extensively for creating ActiveX controls.

b) You also have to link to the static run-time libraries and thus not use Msvcrt.dll.

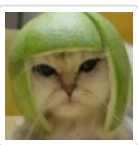
c) The project properties under Configuration Properties \ C/C++ RunTime Library must be set to Multi-threaded or Multi-threaded debug, not one of the .dll versions.

d) You cannot run the Common Language Runtime (CLR). See https://docs.microsoft.com/en-us/dotnet/standard/clr for more information about CLR.

So I don't know at this point how realistic using Windows PE is going to be. In any case, finding out more about this approach would be something of a project unless someone has already gone down that path.

Other approaches might use an actual real-time OS. Such an operating system would allow fine control of priorities and interrupts but the underlying OS might not support everything written in CPP with a particular IDE.

All that I have seen so far leaves this timing lesson in a very awkward position: Although we all recognize how important timing is for certain kinds of program development, we have not been given the tools to perform valid timing tests.



Alex
June 1, 2018 at 3:27 pm · Reply

I think all of the constraints you list above are achievable: You shouldn't be using MFC, ATL, or CLR anyway, and linking to the static run-times without multithreading is definitely doable. I think I cover how in appendix A.

Peter Baum
May 22, 2018 at 11:21 am · Reply

I don't think the constructor should initialize m_beg because we don't know for any particular compiler, if the results will be the same as a call to reset(). In other words, we should force the user to always call reset() to start the timing.

nascardriver
May 22, 2018 at 12:32 pm · Reply

Hi Peter!

Why would the results be different?
I partially agree, though. I think there should "start" and "stop" functionality.

Peter Baum
May 25, 2018 at 8:07 am · Reply

The results might differ because initialization using the constructor as part of the operation of creating an object may be different from the call to the start (reset) function. In particular, we don't know when the call to get the system clock information occurs in that creation sequence. Even something like obtaining the time within a different function calling sequence (which involves stack operations) or different page faulting might make a difference.

Peter Baum
May 21, 2018 at 4:34 pm · Reply

I think this lesson is extremely important because having the ability to evaluate the execution-time performance of routines is sometimes critical to the success of a computer program. Unfortunately, we face some very significant obstacles trying to do accurate timing, only some of which were addressed in this lesson. Here are some additional topics that might be considered. I'm sure there are experts out there who know much more about this than I do.

1. Running a program 3 or more times is insufficient to detect whether or not the timing is reliable. Consider a delay that takes effect 50% of the time. That means that 1/8 times or 12.5% of the time we can expect a situation where there is no detected delay 3 times in a row. We also have the same probability that the delay will occur 3 times in a row.

2. There are potential problems using a routine itself as an experimental control. This is effectively what is being done when you run the same routine several times and look for variations in the results. A better approach is to use relatively simple routines as controls that exercise possible problem areas. Thus, controls might stress machine language execution, memory, or I/O, and we can also use various combinations of these fundamental routines. If these simple controls produce consistent results, then we can be pretty sure that our basic timing platform is suitable.

3. We need practical advice about placing our machine in a state that is likely to produce the best results. One idea I had was to put my windows 10 machine in safe-mode. That got rid of some things like the network, but I was surprised at how much was left running, including Cortana! One can use the task manager to stop programs such as anti-virus software, but others are hard to delete (they automatically restart), can't be deleted because of permissions, or should not be deleted. Even in safe mode, I was seeing very obvious performance variations in the 0.5% range.

Can an expert give us some advice here?

4. We know that the timers used to measure time intervals are part of a hardware/software system that depends upon interrupts. There can be several timer interrupts and they can be interfered with by other hardware interrupts and interrupt service routines. However, the situation is worse than I thought. From https://www.osr.com/nt-insider/2014-issue3/windows-real-time/ I learned

"Because timers on Windows do not have direct support from hardware, it means that they are not very accurate. That is even more so on Windows 8 which introduces a new feature called "dynamic clock tick", where the system clock does not interrupt at fixed intervals but rather, when the operating system deems it necessary for power saving reasons. This has caused any measuring method dependent on a kernel timer to become unreliable. If a device should need accurate periodic attention from software, the hardware should come equipped with its own timer that can fire interrupts so that it can be serviced by software in a timely fashion. This is true despite the fact that the Windows 8.1 WDK has a new feature called high resolution timers."

5. Even if we were to solve the underlying timing problems and figured out some reasonable bounds for the error we could expect from the results of our experiment, there are other subtle potential sources of error. As an example, I was tripped up comparing two routines that I arranged so that one routine would be timed and the results output followed by the timing of the second routine and outputting its results. What I hadn't considered is that the output from the first routine might be asynchronously output while the second routine is running, making the second routine appear to run slower than it might otherwise. I also ended up doing a timer reset before the first routine because I could not be sure that the result would be the same as the initialization of the timer during timer object construction. Vectors doing garbage collection would be another potentially issue, along with things like page swaps, and the use of multiple cores. The experience left me wondering what other mistakes I might have made.

6. One of my complaints about C++ is that so much goes on that is invisible to the programmer. That means that optimizing a routine is difficult because it is hard to know what (perhaps obvious) optimizations will be applied to the code. Yes, we could try to look at the assembly language code, but not only is this inconvenient, it is sometimes difficult to follow if other kinds of optimization has been applied. As a practical matter, I think perhaps I should just assume that the compiler doesn't do much optimization.

7. Comparing two routines can also be made difficult if the algorithms are fast for some configurations and slow for others. Sometimes we just end up with "it depends" rather than being able to say one is faster than another.

It is tempting to run some timing tests and if we see a 5 or 10 percent difference and we have taken some precautions, then assume that the results have some validity. Putting on my scientist hat, I am not so sure.

nascardriver
May 22, 2018 at 8:06 am · Reply

Hi Peter!

If you haven't already, you might want to take a look at the Big O notation. It can be used to compare efficiencies of algorithms without even needing to compile the code.

References
* https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/ (I didn't read the article, but it seems like a good start)

Peter Baum
May 22, 2018 at 11:17 am · Reply

Hi nascardriver,

That is certainly a useful tool when comparing algorithms. That consideration is especially important when the orders are obviously different. However, at some point (especially if the orders are the same) we get down to specifics that relate to actual machine language execution times. In

fact, it is possible that the order difference at some large number or at infinity is completely irrelevant to a practical implementation.

Another thing that can be done is to look at machine instruction execution times. With cache and multiple execution units, that has gotten hard to do now. There may be software that will assist; anyone know?

My other thought is that perhaps a system repair disk might contain enough of a simplified operating system so that we could execute programs that only use the CLI. I haven't experimented with that yet but perhaps somebody has already tried it.

nascardriver
May 22, 2018 at 12:28 pm · Reply

Another thing to keep in mind is that your CPU might have one or another feature that allows algorithm A to perform better than algorithm B even though B performs better than A on another machine. I don't think timing code is a reliable way of comparing algorithms.
By "perform" I mean "run faster".

Peter Baum
May 25, 2018 at 8:00 am · Reply

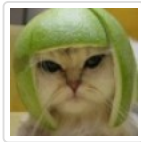
Hi nascardriver,
I agree with what you suggest about the performance on two different machines. However, we will be confronted with having to choose an algorithm to implement, and making a choice based on some kind of testing and documentation is better than giving up completely.

Peter Baum
May 19, 2018 at 9:45 am · Reply

1. In the timer class, why not use uniform initialization? If the argument is that you want it to compile regardless of the compiler used, that same argument suggests we shouldn't bother learning any of the new c++ capabilities and just stick to the old formats. Perhaps the way forward is to teach and use the new formats and make a separate note telling people how to make do if they aren't willing to obtain an uptodate compiler.

2. The code in main() is missing the units. Because of the duration cast used, it should be

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



Alex
May 20, 2018 at 1:09 pm · Reply

1) I learned to code before uniform initialization was a thing. Old habits die hard.
2) Added. Thanks for the suggestion!

Peter Baum
May 19, 2018 at 9:39 am · Reply

First sentence: "performant" isn't a word. You want to talk about execution speed here.

J Gahr
May 11, 2018 at 1:33 pm · Reply

Hello! In Visual Studio 2017, I ran your selection sort example, and then replaced std::array<>() with array[] to see how their times compared:

```
#include <array>
#include <chrono>

const int g_arrayElements = 10000;

class Timer
{
private:

    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(  int array[g_arrayelements]  /*std::array<int, g_arrayElements> &array*/)
{


    for (int startIndex = 0; startIndex < g_arrayElements - 1; ++startIndex)
    {

        int smallestIndex = startIndex;


        for (int currentIndex = startIndex + 1; currentIndex < g_arrayElements; ++currentIn
        {
            if (array[currentIndex] < array[smallestIndex])
                smallestIndex = currentIndex;
        }

        std::swap(array[startIndex], array[smallestIndex]);
    }
}

int main()
{
    //std::array<int, g_arrayElements> array;
    int array[g_arrayElements]; // This one seems to be much more performant than std::arra
    for (int i = 0; i < g_arrayElements; ++i)
```

```
56          array[i] = g_arrayElements - i;
57
58      Timer t;
59
60      sortArray(array);
61
62      std::cout << "Time taken: " << t.elapsed() << '\n';
63
64      return 0;
65  }
```

In release mode, the time taken by array[] was identical to std::array<>()'s time (roughly 0.12 seconds).

In debug mode, however, using array[] resulted in a consistent time of 0.12 seconds, and when I used std::array<>() the time taken was always hovering around 2.1 seconds.

I was wondering if this is a typical result that happens across most compilers and machines, or if this was specific to my compiler/machine. What do you guys think?

nascardriver
May 13, 2018 at 2:12 am · Reply

Hi J!

Debug mode is always slower, because compiler optimizations are usually disabled and your compiler might have added some extra checks.

Pierce
April 27, 2018 at 11:33 am · Reply

I'm assuming that if the timer outputs 0, then the timer is not set to have enough precision to actually hold the amount of seconds it took

Alex
April 29, 2018 at 11:26 am · Reply

C++ doesn't require the time to have a specific resolution, so maybe yours simply doesn't have a good enough resolution. You can see your resolution as follows:

```
1  std::cout << std::chrono::high_resolution_clock::duration::period::den << std::endl;
```

This shows you how many ticks per second your clock has.

Sébastien
January 8, 2018 at 2:55 am · Reply

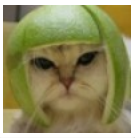
I was not able to run the code above although it compiles without any warning. std::chrono crashes the program even before it reaches the first line of main (used gdb to discover that). I use a Win 10 64x architecture(Core-i5 4200H).

Alex
January 8, 2018 at 5:47 pm · Reply

Interesting. Perhaps there's a bug in my code. If anybody else tries these examples, can you report back success/failure, including what your compiler was?

nascardriver
January 9, 2018 at 1:53 am · Reply

GCC 4.9.2 Working
GCC 6.3 Working
GCC 7.2.0 Working
Microsoft (R) C/C++ Optimizing Compiler 19.00.23506 Working

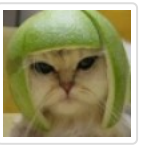


Alex
January 10, 2018 at 5:02 pm · Reply

Thanks! I appreciate the compiler checks.

Benjamin
January 5, 2018 at 12:52 pm · Reply

Some thoughts about this section:
The std::sort is deploying better algorithm than yours. Although std::sort sorting algorithm differs by compiler by compiler, it must have average complexity requirement of O(n*log(n)) by the ISOCPP. However, your sort implementation is a kind of selection sort, which causes average complexity(also best and worse) to be O(n^2). Roughly comparing, although the constant will be assumed to be 1, the complexity ratio in your case may be (yours):(std) = 10^8:40000. I think that that would be the one explanation why your implementation was slow, plus the compiler might have the optimized function itself.
Also, I think your array initialization was not good, as you rely on some garbage values which may not be garbage values on some machine or compiler even compiling mode. Thus, I think it is wise that you fix your array initialization using random function inside <random.h>.
Finally, I would be appreciated if you publicize the name of the compiler used on the compilation, as it would help everyone compare the algorithm implemented.



Alex
January 7, 2018 at 1:28 pm · Reply

You're correct in that my selection sort algorithm is o(n^2), whereas std::sort is o(n log n). This should account for the bulk of the difference in timing.

My array initialization looks fine to me. I initialize elements 0 through length-1 with integer values length-1 through 0 (essentially, the array starts reverse sorted). Originally I was randomizing the array, but you have to be careful with that so to ensure you get the same randomization across both runs -- that adds a bit of complexity that I thought muddled the example slightly.

This was done on Visual Studio 2017. I'm not sure what algorithm it's using. It might be https://en.wikipedia.org/wiki/Introsort

Peter Baum
June 3, 2018 at 2:37 pm · Reply

There are many different sorting algorithms and no one best algorithm. There is a great deal of information on the internet about sorting algorithms and the circumstances where you might choose one method over the other. One of the nicest visual displays that shows something about the different methods used by some of the common ones can be found at https://www.youtube.com/watch?v=QmOtL6pPcI0