What does "current time" mean?

Document number: P0342R1
Date: 2023-01-15
Project: Programming Language C++
Audience: SG1
Author: Mike Spertus
Reply-to: msspertu@amazon.com

Abstract

According to time.clock.req/2, calling `now()` returns a " `time_point` object representing the current point in time," but what is meant by "current point in time"? This is not only unclear in theory, but it also leads to problems in practice. This paper illustrates the problem and suggests some possible approaches for helping developers get the current time more effectively.

History

At the 2016 Oulo meeting, Evolution Working Group reviewed a prior revision of this paper that ambitiously proposed a solution to transparently provide correct behavior for clocks and provide standard library barriers for analogous problems. Although there were a variety of viewpoints expressed and no straw poll was taken, there were major concerns about the implementability of the approach taken in that paper, and the overall sentiment was not to proceed further on that form of the proposal.

The motivation for revisiting this question are twofold. Firstly, the chair of SG1 informed me in discussions during the 2022 Kona meeting that SG1 had a desire to look at questions of this nature and requested an updated revision. More generally, the challenges a programmer faces in the common task of getting a current timestamp are serious enough in our opinion to warrant looking more broadly for improvements to the current situation, whether through normative changes or simply a clarifying note in the standard or guidance to educators via SG20.

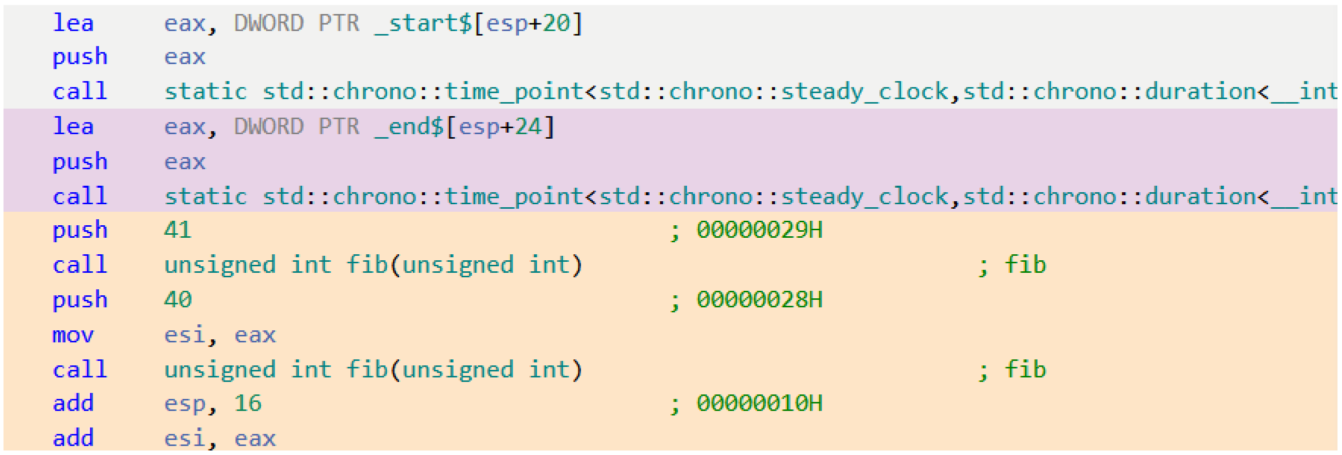
The problem

When teaching my students C++, I wanted to demonstrate how poor the performance of the naive implementation of the Fibonacci sequence is and ran the following code, which calls `chrono` 's `now()` function to get the current time.

```
#include<chrono>
#include<atomic>
#include<iostream>
using namespace std;

size_t
fib(size_t n)
{
  if (n == 0)
    return n;
  if (n == 1)
    return 1;
  return fib(n - 1) + fib(n - 2);
}

int const which{42};

int main()
```

```
{
  auto start = chrono::high_resolution_clock::now();
  auto result = fib(which);
  auto end = chrono::high_resolution_clock::now();
  cout << "fib(" << which << ") is " << result << endl;
  cout << "Elapsed time is "
       << chrono::duration_cast<chrono::milliseconds>(end - start).count()
       << "ms" << endl;

  return 0;
}
```

I was surprised to discover that the program printed an elapsed time of zero milliseconds on at least one compiler. Looking at the generated code at https://godbolt.org/z/vY5d9bref revealed that the compiler had reordered the code to get the end time prior to running the calculation.



```
    lea     eax, DWORD PTR _start$[esp+20]
    push    eax
    call    static std::chrono::time_point<std::chrono::steady_clock,std::chrono::duration<__int
    lea     eax, DWORD PTR _end$[esp+24]
    push    eax
    call    static std::chrono::time_point<std::chrono::steady_clock,std::chrono::duration<__int
    push    41                                  ; 00000029H
    call    unsigned int fib(unsigned int)                          ; fib
    push    40                                  ; 00000028H
    mov     esi, eax
    call    unsigned int fib(unsigned int)                          ; fib
    add     esp, 16                             ; 00000010H
    add     esi, eax
```

Fixing the problem in a standards-compliant way appears difficult. For example, inserting memory fences (a common way to force ordering) had no effect in this single-threaded program.

Several committee members suggested working around by breaking the code into multiple files, which worked in my case but is likely to be fragile in practice if non-local optimizations like whole program optimizations and link-time code generation become more prevalent, rendering the relationship between the current time and where it is requested in the code increasingly tenuous.

In the end, I was left with the feeling that the common programming task of getting a current timestamp is anything but simple and prone to errors and fragility even for experienced programmers.

Possible solutions

We consider several possible approaches that address this problem to an increasing degree.

No change to the standard but improved guidance

While the original paper did not result in a change to the standard, there was an explicit agreement on communicate the perils of relying on `now()` to get the current time and promoting superior benchmarking strategies by directing people to resources like Chandler Carruth's Tuning C++ talk from CppCon 2015.

Likewise, while I have started incorporating the perils of using `now()` into my courses as part of "what every programmer should know," I remain unsure what guidance to give on what can and cannot be relied on about the current time. Even coming up with guidance that could be disseminated by SG20 would be a great help for educators.

Editorial change to the standard

Along the lines of the previous item, inserting such guidance editorially into the standard via a clarifying note or example like the above could be very helpful.

The requirements for a clock in the table in time.clock.req/2 says `now()` returns a "`time_point` object representing the current point in time," without ever defining what is meant by "current point in time." More precise language that this is the time in the clock when `now()` is invoked and that reordering is consistent with the *as if* rule may also be helpful (this may or may not be fully editorial)

Ban reordering around `now()`, either absolutely or conditionally

It is not really clear to what extent `now()` is even meaningful if the compiler is free to move it anywhere relative to other code in the program. In general, an explicit request for the "current time" (which is the language currently sed without definition in the standard) might reasonably be expected by the typical programmer to occur after/before statements that are sequenced before/after the call to `now()` in the source.

Note that there were great concerns that would need be reviewed about whether such constraints could be implemented in the Oulu discussion. This note by Daveed Vandevoorde delves into the constraints placed on compilers by existing fences that may or may not be visible, which may provide some useful guideposts.

Add a sequence fence

We suspect that getting the current time is not the only place where these concerns arise. For example, it appears that reading a memory mapped sensor before and after the calculation to determine the induced temperature rise could be vulnerable to the same problems. To support such use cases, we could expose the underlying mechanism associated with the previous suggestion as some sort of *sequence fence*.

Note that this approach of a fence that blocks code reordering across it is similar to the approach recommended by the original paper, which was rejected due to implementation concerns from compiler vendors which would need to be mitigated before pursuing this direction. In addition to implementation concerns, there were concerns that this would be too low-level and confusing beyond encapsulating in `now()`. We believe that such a fence is in the spirit of the other (admittedly expert) fences that have been adopted in C++. To support the non-expert, We could also require (absolutely or conditionally) that `now()` invoke a sequence fence. There is also a question as to whether the behavior of this fence would be implementation-defined as in the previous revision or have normative requirements.