This Videogame Programmer Used the STL
(and You Will Never Guess What Happened Next)
Mathieu Ropert
@MatRopert

Talk title idea: "This videogame programmer used the STL and you will never guess what happened next"

Traduire le Tweet
12:35 PM · 18 avr. 2019 · Twitter Web Client

Voir l’activité sur Twitter

2 Retweets 60 J’aime
STL and videogames
The STL is sometimes seen as a strange and dangerous beast, especially in the game development industry. There is talk about performance concerns, strange behaviours, interminable compilations and weird decisions by a mysterious "committee". Is there any truth to it? Is it all a misconception?

I have been using the STL in a production videogame that is mostly CPU bound and in this talk we will unveil the truth behind the rumours. We will start by a discussion about the most common criticism against the STL and its idioms made by the gamedev community.

Then we will see a few practical examples through STL containers, explaining where they can do the job, where they might be lacking and what alternatives can be used.

Finally we will conclude with some ideas on how we can improve both the STL for game developers and also how to foster better discussion on the topic in the future.

At the end of this talk, attendees should have a solid understanding of why the STL is sometimes frowned upon, when it makes sense to look for alternatives to the standard and most importantly when it does not.

**Speakers**

**Mathieu Ropert**
Experienced Programmer, Paradox Development Studio
French C++ expert working on (somewhat) historical video games. Decided to upgrade his compiler once and has been blogging about build systems ever since. Past speaker at CppCon, Meeting C++ and ACCU. Used to run the Paris C++ User Group. Currently lives in Sweden.
READING IS FOR NERDS!
STL in videogames considered harmful
This Videogame Programmer Used the STL
(and You Will Never Guess What Happened Next)
THE C++ STANDARD TEMPLATE LIBRARY

P.J. PLUGER
ALEXANDER A. STEPANOV
MENG LEE
DAVID R. MUSSER
Standard Template Library

- Proposed in 1993 by Alex Stepanov
- Adopted in 1994
- Offers a set of generic containers and algorithms for C++
1994
“We don’t use the STL here”

Anonymous videogame programmer, 2019
Hello!

I am Mathieu Ropert

I’m a C++ developer at Paradox Development Studio where I make Europa Universalis.

You can reach me at:

✉️ mro@puchiko.net
🐦 @MatRopert
🌐 https://mropert.github.io
About this talk

- The case against STL
- STL containers in practice
- Frequently sought-after alternatives
- A word about debug performance
About not this talk

- Allocators
- Exceptions
- Build times
Is the **STL** so bad?

Or where the criticism is coming from
Common complaints

- “STL is unfamiliar”
- “STL is not supported on platform X”
- “STL is bloated”
- “STL performance isn’t that great”
STL familiarity

- STL been around for 25 years
- Popular C++ libraries adopted the same idioms (Boost, Abseil, Intel TBB...)
- Resources teaching Containers, Iterators and Algorithms are plenty
STL familiarity

- Stepanov’s approach on decoupling containers and algorithms is based on sound research
- We might need to study and teach the principles better in schools
Major vendors should provide a reasonably good implementation of the STL

As any software, they may have bugs or caveats

Keep up with updates, report issues
STL availability

- Vendors that won’t care about STL probably won’t care about C++ in general
- Chances are they will have broken standard support or subpar optimizations
- Consider using open source alternatives
STL bloat

- Standard additions may feel unnecessary or unwanted
- Vendor implementations may look over-complicated for what they are trying to achieve
STL bloat?

- STL, like C++, is designed for general purpose usage.
- C++ design principles dictate that unused features should not be added to the cost.
- Not always possible in practice, as the cost of multiple policies grows quite fast.
Vendor implementations may include additional debug features to help developers.

There is a build flag somewhere to turn them off.

Debug checks are not incompatible with optimizations.
The quest for **performance**

- Games need to run within a timebox
- Worst case scenarios and unpredictable latency matter a lot
- Common wisdom recommends low level languages for better control over performance
The quest for **performance**

- STL comes with some degree of abstraction
  - Templates
  - Iterators
  - Debug / checked iterators
  - Proxy iterators

- Requires a good optimizer to yield performance
```cpp
static void RawAccumlate(benchmark::State& state) {
    const auto v = generate_values<int>(10000);
    for (auto _ : state) {
        const int* p = v.data();
        const int sz = v.size();
        int sum = 0;
        for (int i = 0; i < sz; ++i)
            sum += p[i];

        benchmark::DoNotOptimize(sum);
    }
}

BENCHMARK(RawAccumlate);
```
```cpp
static void STLAccumlate(benchmark::State& state) {
    const auto v = generate_values<int>(10000);
    for (auto _ : state) {
        auto sum = std::accumulate(begin(v), end(v), 0);
        benchmark::DoNotOptimize(sum);
    }
}
BENCHMARK(STLAccumlate);
```
STL performance

clang / libc++
STL performance

clang / libstdc++
“That’s why I use C. C++ has bad performance without optimization!”

Anonymous videogame programmer, 2019
Released in 1994 too!
Performance in 2019

- The 80486 was the last x86 to run instructions sequentially
- Modern CPUs execute instructions out of order
- How does “low level” imperative C fare without optimization today?
Performance in 2019

```c
int accumulate( const int* ptr, int size )
{
    int sum = 0;
    for ( int i = 0; i < size; ++i)
        sum += ptr[i];
    return sum;
}
```
Performance in 2019

```c
__attribute__((optnone))
int accumulate_unoptimized( const int* ptr, int size )
{
    int sum = 0;
    for ( int i = 0; i < size; ++i)
    {
        sum += ptr[i];
    }
    return sum;
}
```
Performance in 2019

![Graph showing performance comparison between RawAccumulate and RawAccumulateUnoptimized. The graph indicates that RawAccumulateUnoptimized is 26 times slower than RawAccumulate.]}
Performance in 2019

The chart shows the performance ratio (CPU time / Noop time) for different configurations, with Lower is faster.
Performance and debug

- C++ abstractions will be slower than raw C with all optimizations turned off
- Both C and C++ are an order of magnitude slower when you disable optimizations
- Enabling even minimal optimizations yields enormous gains
Performance and debug

- Some vendors offer good or decent support for optimized debug builds (GCC, MSVC)
- There’s probably room for improvements
- Know your build flags!
All you need is std::vector
Containers overview

- Most commonly used containers
- Arrays and dynamic arrays
- Ordered associative containers
- Hash tables
std::vector

- Heap-allocated array that can be resized
- Go-to container in the STL
- Cheap to move and random access
- As fast as it gets to iterate over
# Not all CPU operations are created equal

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost in CPU Cycles</th>
<th>$10^0$</th>
<th>$10^1$</th>
<th>$10^2$</th>
<th>$10^3$</th>
<th>$10^4$</th>
<th>$10^5$</th>
<th>$10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Simple” register-register op (ADD, OR, etc.)</td>
<td></td>
<td>&lt;1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory write</td>
<td></td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bypass delay: switch between integer and floating-point units</td>
<td></td>
<td>0-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Right” branch of “if”</td>
<td></td>
<td>1-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point/vector addition</td>
<td></td>
<td>1-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplication (integer/float/vector)</td>
<td></td>
<td>1-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Return error and check</td>
<td></td>
<td>1-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 read</td>
<td></td>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLB miss</td>
<td></td>
<td>7-21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 read</td>
<td></td>
<td>L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>“Wrong” branch of “if” (branch misprediction)</td>
<td></td>
<td>10-20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point division</td>
<td></td>
<td>10-40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128-bit vector division</td>
<td></td>
<td>10-70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomics/CAS</td>
<td></td>
<td>15-30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C function direct call</td>
<td></td>
<td>15-30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer division</td>
<td></td>
<td>15-40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C function indirect call</td>
<td></td>
<td>20-50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ virtual function call</td>
<td></td>
<td>30-60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 read</td>
<td></td>
<td>30-70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main RAM read</td>
<td></td>
<td>100-150</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Not all CPU operations are created equal

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost in CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Simple&quot; register-register op (ADD, OR, etc.)</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Memory write</td>
<td>-1</td>
</tr>
<tr>
<td>Bypass delay: switch between integer and floating-point units</td>
<td>0-3</td>
</tr>
<tr>
<td>&quot;Right&quot; branch of &quot;if&quot;</td>
<td>1-2</td>
</tr>
<tr>
<td>Floating-point/vector addition</td>
<td>1-3</td>
</tr>
<tr>
<td>Multiplication (integer/float/vector)</td>
<td>1-7</td>
</tr>
<tr>
<td>Return error and check</td>
<td>1-7</td>
</tr>
<tr>
<td>L1 read</td>
<td>3-4</td>
</tr>
<tr>
<td>TLB miss</td>
<td>7-21</td>
</tr>
<tr>
<td>L2 read</td>
<td>10-12</td>
</tr>
<tr>
<td>&quot;Wrong&quot; branch of &quot;if&quot; (branch misprediction)</td>
<td>10-20</td>
</tr>
<tr>
<td>Floating-point division</td>
<td>10-40</td>
</tr>
<tr>
<td>128-bit vector division</td>
<td>10-70</td>
</tr>
<tr>
<td>Atomics/CAS</td>
<td>15-30</td>
</tr>
<tr>
<td>C function direct call</td>
<td>15-30</td>
</tr>
<tr>
<td>Integer division</td>
<td>15-40</td>
</tr>
<tr>
<td>C function indirect call</td>
<td>20-50</td>
</tr>
<tr>
<td>C++ virtual function call</td>
<td>30-60</td>
</tr>
<tr>
<td>L3 read</td>
<td>30-70</td>
</tr>
<tr>
<td>Main RAM read</td>
<td>100-150</td>
</tr>
</tbody>
</table>

Modern CPU caching can have a 1-100 impact on performance

O(n) operations on std::vector can outperform O(log n) on other containers

Rule of thumb: for small sets, brute force search through vector is faster than std::map
For read-intensive associative sets, consider a sorted vector.

- Prefer indexes to pointers or iterators for storing long term references.
std::vector limitations?

- None!
- std::vector is awesome!
- 😍👍
**std::vector limitations**

- Growth factor is neither specified nor configurable (most commonly 1.5 or 2)
- Standard specification prohibits small buffer optimization
- `std::vector<bool>` is a mess
std::array

- Stack-allocated array with fixed size
- C++11 addition
- O(1) random access and cache friendly layout
- O(n) to move, potentially as expensive as copy
**std::vector alternatives**

- std::vector with small buffer optimization
  - Boost’s boost::small_vector
  - Facebook’s folly::small_vector
  - Google’s absl::InlinedVector

- Avoid heap allocation for small sizes

- May be $O(n)$ on move (and invalidate iterators)
std::array limitations

- Fixed size, not capacity
- Not suitable for dynamic insertion
std::array alternatives

- Fixed capacity vector
  - Boost’s boost::static_vector
  - EA’s eastl::fixed_vector
  - Facebook’s folly::small_vector

- Proposed addition to the standard as P0843
  - WIP name is std::static_vector
**std::map and std::set**

- Classic sorted associative containers
- $O(\log n)$ access, insertion and erase
- Iterators remain valid upon insert and erase
- $O(1)$ move construction
std::map and std::set

- Almost always implemented as a R/B tree
- Data is not stored in a cache-friendly manner
- Lookup time is logarithmic, not constant
“STL map and set have terrible performance, don’t use them!”

Anonymous videogame programmer, 2019
std::map and std::set implementation

- Can we do better?
- Not really...
- ... unless we drop some constraints from the standard
std::map and std::set variants

- Drop the sorted requirement
- We get C++11’s std::unordered_set and std::unordered_map
- Average constant time on insert, erase and lookup
“STL unordered map and set are not using open addressing, don’t use them”

Anonymous videogame programmer, 2019
std::map and std::set variants

- Open addressing hash tables offer better cache performance

- Incompatible with standard requirements
  - Too high space/time tradeoff
  - Invalidate references even when no rehashing occurs
std::map and std::set variants

- Caching is not the main reason why STL hash tables are slow
- You can get good performance *and* follow the standard...
- As long as implementation doesn’t use modulo
**std::map and std::set variants**

![Graph showing benchmark results for different hash table implementations.](image-url)

**Malte Skarupke**

*You Can Do Better Than std::Unordered_Map New and Recent Improvements to Hash Tables*
How to make things better
C++ sucks, why is the committee so incompetent?!

12:00 PM - 1 Oct 2018

63 Retweets  255 Likes
The problem

- The Committee make specifications, not implementation
- C++ is a general purpose language, its defaults have to be sane for the 99%
- Social media rants are not a good way to get a point across
Burden of proof

- Common STL implementations are widely used and tested
- Have feature and performance tests to justify an alternative
- Revisit the comparison from time to time
“Good enough”

- Standard specifications cannot make unsafe assumptions
  - Reference stability
  - Memory overhead

- Target the most common use case

- Specific cases can benefit from specific implementations
“Good enough”

- Corollary: fit-to-purpose alternatives make poor defaults
- Remember the word of Donald Knuth
“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times”
Engaging with your peers

- Ranting on Twitter will not make C++ better
- Neither will a talk given only at GDC
- Make your voice heard where the rest of the C++ community is
- Meetups, conferences, ISO study groups
Engaging with your peers

- Progress goes much faster when people collaborate
- The bigger the sample, the better the results
- Don’t be afraid of talking to C++ developers outside of your field
Engaging with your peers

- Challenge your vendor quality of implementation if needed
- Publish your findings
- Provide reusable benchmarks
- Need help packaging? Ask me!
In conclusion

- STL aims to be a good enough default, as long as some optimizations are enabled
- Specific cases may warrant specific solutions
- Feedback is needed to improve the experience of all C++ developers
Furthermore, I think your build should be destroyed
Thanks!

Any questions?

You can reach me at

✉️ mro@puchiko.net

🐦 @MatRopert

🔗 @mropert

🌐 https://mropert.github.io
References

- *C Is Not a Low-level Language* – David Chisnall, ACM Vol. 16 No. 2 – March-April 2018
- *You Can Do Better than std::unordered_map* – Malte Skarupke, C++Now 2018
- *Fifty shades of debug* – Mathieu Ropert, August 3rd, 2019
Accumulate Benchmark:
http://quick-bench.com/Z-PZk-rBkKjhf50mIcoiwB2Ijdg