Document number: N4456
Date: 2015-04-12
Project: Programming Language C++, Embedded SG, Evolution WG
Authors: Michael Wong, Sean Middleditch, Nicolas Guillemot
Reply to: michaelw@ca.ibm.com

Towards improved support for games, graphics, real-time, low latency, embedded systems.

Introduction

In CPPCon 2014, an impromptu BoF was organized to gather interest for groups interested in improving C++ support for games, graphics, real-time, low-latency, and embedded systems (The Industry).This has evolved into a discussion group (unofficial-real-time-cxx) online where people of similar interest have gathered to gain some traction on the common problems face by this Industry.

There is some commonality between this yet-to-be-formed SG and the emerging Embedded SG by Daniel Gutson. There remains a possibility that these two groups can be folded.

This document summarizes the concerns of that industry as a jump-off point for a possible future Study Group

History

We started by looking at a paper from Paul Pedriana on N2771 "**EASTL -- Electronic Arts Standard Template Library".** While this paper was submitted in 2007, neither chairs of LWG recall reviewing this paper and no record of its review exists. The paper outlines a very detailed proposal of some of the changes required of the motivation for the creation of EASTL, separate from std::stl and their differences. It further outlines the changed and additional functionality of EASTL that was especially suitable for the gaming industry.

While much of this paper remains useful in understanding the ongoing pain-point of The Industry, some of it is no longer valid

- with the advance of C++11 and C++14 such as move semantics
- better Optimized debugging support by compilers (GCC's –Og and other vendors' related options),
- changes in the industry (a 'large' game is so much bigger than 2,000,000 lines of code these days, especially when you consider tools, middleware, plugins, etc.),
- or other library updates (modern allocators)

We feel that a holistic "here's a new STL and our problems" approach to any change seems a bit doomed. The biggest bits we have observed from EASTL and some of the authors' own experiences that are missing from current C++14 are the containers or small independent tweaks to the language or libraries.

The problems with C++ for The Industry

From the discussion in the external group (unofficial-real-time-cxx), we have gathered a number of pain points for the Industry:

*Game developers don't like global new/delete. (Point brought up by Scott Wardle)
- No alignment, no tracking of file/function/line for debugging (especially a problem for tracking middleware allocations). Solutions exist, but unsightly.

- Everybody wraps new/delete in unsightly macros because of the last points. The macros are unsightly because operator new has unorthodox syntax, especially new[].

- Game developers want separate memory budgets for every subsystem, so there's no such thing as a "global" new. I guess HPC people probably also ditch global new in favor of NUMA-aware allocators.

* Can't really do thread pools right in plain C++11 because good thread pool implementations need to be able to do tricky/interesting hacks to reduce scheduling latency. (Paraphrasing Scott Wardle)

* Concerned about std::function/lambdas doing allocations (Point brought up by Scott Wardle)

- I've heard of other game developers also cautiously doing measurements on std::function/lambdas to see how much you can do with them without them resorting to dynamic allocations.

* Concerned about variant proposals resorting to RTTI. RTTI is verboten for game developers and dynamic dispatch is not ideal, so this variant would be a write-off. Better implementations are possible. (Sean Middleditch's post)

* Would be great to be able to get vtable pointers as global objects without requiring an existing reference to the object. (Nicolas Fleury's post for a proposal)

- Maybe also interesting to think of how it could be possible to customize the way virtual functions work as a library solution? (discussion in Nicolas Fleury's post)

Re-examining N2771 (by Sean Middleditch) yielded the following issues. All references are from N2771.

1. std STL allocators are painful to work with and lead to code bloat and sub-optimal performance. This topic is addressed separately within N2771.

   C++11 simplified allocators and the polymorphic allocator proposal help here a lot. I've found that a lack of proper alignment support and for more advanced allocation flags (e.g., clear to 0) annoying but not show stoppers.

2. Useful STL extensions (e.g. slist, hash_map, shared_ptr) found in some std STL implementations are not portable because they don't exist in other versions of STL or are inconsistent between STL versions (though they are present in the current C++11 draft).

   All three of those extensions are in C++11. We are still missing some handy ones, though, like

flat_map.

3. STL lacks functionality that game programmers find useful (e.g. intrusive containers) and which could be best optimized in a portable STL environment. See Appendix item 16 of N2771

   This is still true.

4. Existing std STL implementations use deep function calls. This results in low performance with compilers that are weak at inlining, as is the currently prevalent open-source C++ compiler. See Appendix item 15 and Appendix item 10.

   All the major compilers today support inlining small accessor functions evne in debug builds. This item needs measurement, though.

5. Existing STL implementations are hard to debug. For example, you typically cannot browse the contents of a std::list container with a debugger due to std::list's usage of void pointers. On the other hand, EASTL allows you to view lists without incurring a performance or memory cost. See Appendix item 2.

   This is mostly a tools issue. Visual Studio for instance has a powerful debug visualizer feature and supports its own STL out of the box. GDB has an even more powerful debug visualizer and supports libstdc++ out of the box.

6. STL doesn't explicitly support alignment requirements of contained objects, yet non-default alignment requirements are common in game development. A std STL allocator has no way of knowing the alignment requirements of the objects it is being asked to allocate, aside from compiler extensions. Granted, this is part of the larger problem of the C++ language providing minimal support for alignment requirements. Alignment support is proposed for C++14.

   The C++11 alignment issue has no effect on free-store allocations. However, I think this might be less of a "we need explicit alignment" issue and more "the native alignment should include the SIMD/vector types of the CPU." (which in my experience yet is the major reason I have to work around default allocation alignments.)

7. STL containers won't let you insert an entry into a container without supplying an entry to copy from. This can be inefficient in the case of elements that are expensive to construct.

   Examples are emplace_back and move semantics.

8. STL implementations that are provided by compiler vendors for the most popular PC and console (box connected to TV) gaming platforms have performance problems. EASTL generally outperforms all existing STL implementations; it does so partly due to algorithmic improvements but mostly due to practical improvements that take into account compiler and hardware behavior. See Appendix item 20.

   Not the standard's fault... mostly.

   One could make the case that the standard could be stricter about some of these issues and not leave them all as QoI items. One that is a problem is small-object "optimizations" in things like std::string or std::function. This means we can't rely on what size of data will or will not trigger an allocation in a portable manner. It becomes worthwhile to rewrite them just to get a consistent cross-platform cross-vendor known semantics here.

9. Existing STL implementations are hard to debug/trace, as some STL implementations use cryptic variable names and unusual data structures and have no code documentation. See Appendix item 2.

   This is still a common issue. This has some up in some proposals for allowing certain headers to ignore user macros, which are mostly the reason that these cryptic issues exist.

10. STL containers have private implementations that don't allow you to work with their data structures in a portable way, yet sometimes this is an important thing to be able to do (e.g. node pools). See Appendix item 22.

    This is not quite as relevant anymore, e.g. vector::data(), but it needs more investigation.

11. Many current versions of std STL allocate memory in empty versions of at least some of their containers. This is not ideal and prevents optimizations such as container memory resets that can significantly increase performance in some situations. An empty container should allocate no memory.

    While "Many" is an overstatement, but "any" is a problem. This might be another issue where the standard needs to be stricter. I understand that some standard containers actually have to do this for move operations and the like, though, in order to not invalidate iterators. This might be one of the cases similar to unordered_map where a small oversight in the standardese hampered the usefulness of the container significantly but is now embedded in stone.

12. All current std STL algorithm implementations fail to support the use of predicate references, which results in inefficient hacks to work around the problem. See Appendix item 3.

There have been proposals for this, either directly or indirectly, but focus on reviewing those and helping them through would be a good future direction.

13. STL puts an emphasis on correctness before practicality and performance. This is an understandable policy but in some cases (particularly std::allocator) it gets in the way of usability and optimized performance.

    This is a hard one. I think the correctness is the right choice as the default. There are some cases where the standard could allow more dangerous operations; see the discussions on vector::push_back_no_grow() for instance.

Future directions

While this paper only lists an analysis of N2771, we do have some future directions (by Sean Middleditch)

- Adding flat_map or the like would involve straight-forward stand alone papers.

- The small changes or stricter requirements EASTL places on containers are common enough for our needs, e.g. requiring that containers never allocate memory on default construction. I know that there have been many arguments on the ISO lists recently about other simple changes (e.g. noexcept on move) that aren't acceptable because some library implementations allocate unique nodes for empty containers, which IMO should be addressed by a paper advocating why the cons outweigh the pros of such an implementation and why the standard should disallow them as the EASTL does (just like how C++11 disallowed COW strings, despite several major implementations use of them).

- Many other changes can be very small, minimal items, I think. The feature of splitting <algorithm> up into some small headers can be quite valuable and made fully backwards compatible by just having an <algorithm> that includes all the other algorithm headers.

- Some of the changes are obviated by compiler or tooling updates. Vectors using pointers as iterators for instance is a big issue for out-of-the-box debug builds in some implementations, but with a combination of modern debug visualizers and compilers smart enough to inline operations that have low debuggability impacts means that std::vector on most implementations is perfectly suitable once you tweak build options suitably (disabling iterator debugging, enabling inlining or the like on debug builds, etc.). The biggest problem I could see the standard addressing here is how vendor-specific all these parameters are, perhaps via some kind of _CPP_NO_DEBUG macro or the like that implementations are all recommended to interpret as "turn off all the crap you don't strictly need to be standards compliant."

- The ability to safely disable exceptions is another big one, but I don't see that having much chance of acceptance just yet. I'm led to believe that Google is aware of the

problems even outside of embedded/real-time and certainly Sony is as well, but the ISO committee will be a much harder nut to crack on this item IMO.

Other relevant WG21 papers

A number of current and past WG21 papers are also relevant towards better support for The Industry (by Nicolas Guillemot)

- Standard SIMD types (N3759).
  The aim to make the language aware of their existence would solve lots of alignment problems.

- N4237 Language Extensions for Vector loop level parallelism
- N4238 An Abstract Model of Vector Parallelism
- Static reflection (N3996).
  In our experience, games tend to use many enums (often auto-generated by tools) to communicate between systems. Being able to iterate over enums or automatically convert them to strings would be helpful for writing debuggers, loggers, profilers..

  Furthermore, this could make it possible to automatically convert data structures between SoA (Struct of Array) and AoS (Array of Struct), which would make it much easier to experiment and find which layouts of memory have optimal memory access patterns. It would be much more interesting to use std algorithms if they allowed writing layout-agnostic code to use with SoA/AoS. Such a feature would be even better if possible to use for SIMD types, ie. being able to easily change your data from xyzw xyzw xyzw xyzw to xxxx yyyy zzzz wwww.

  The importance of design considerations with SIMD was highlighted in a GDC talk this year (https://deplinenoise.files.wordpress.com/2015/03/gdc2015_afredriksson_simd.pdf), and from reading the slides you can see that one of the only viable options for these game developers is still just to use intrinsics, which could definitely use cleaning up from a language perspective.

Summary

While work continues as an ongoing proposal, we propose that items should be addressed individually and split up into separate work items. At least that evaluation should just be identifying the individual work items. I think the split works down to broad work groups as:

(1) new containers
  (a) inline lists and trees
  (b) fixed/flat trees
  (c) whole new containers like ring_buffers
  (d) whole new algorithms like radix_sort
(2) stricter requirements on existing containers and algorithms
  (a) no allocations for default-empty construction

(b) guaranteed cheap/trivial/noexcept move
(c) debug builds cannot break algorithm efficiency requirements (*cough* Microsoft's std::sort *cough*)
(3) misc. issues
(a) smaller headers, like <algorithm>
(b) QoI on container inspection
(c) ??? -fno-exceptions, -fno-rtti or equivalents should be guranteed to be supported by the library, possibly via feature-testing support for these features

Acknowledgement

This document has been the collaborative work of a google discussion group started by Sean Middleditch, based on a BoF at CPPCon 2014 by Michael Wong. We thank a number of people who have been involved in this discussion (some are known only by their handle)

Billy Baker
Daniel Gutson
Jason Meisel
Kieran O'Connor
Martin Slater
Matt Newport
Nicholas Baldwin
Nicolas Fleury
Nicolas Guillemot
Ryan Rohrer
Ryan Ruzich
Scott Wardle
Sean Middleditch
sittej
Vittorio Romeo