Document number: N4526
Revision: of N4456
Date: 2015-05-22
Project: Programming Language C++, Library Evolution WG, Evolution WG, SG14
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.

History

N4526: updated paper pre-Lenexa and adding Lenexa EWG feedback and creation of SG14
Lenexa: presented N4456 to EWG; There was support for creating SG14 Games Dev and low latency
N4456: initial paper based on discussions at and post CPPCon 2014 in https://groups.google.com/forum/#!forum/unofficial-real-time-cxx

Introduction

At CPPCon 2014, an impromptu BoF was organized 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 interests have gathered to gain some traction on the common problems faced 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: N2271 “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 describes a very detailed list of issues that motivated the creation of EASTL, and explains its differences from the std::stl. It further outlines the changes and additional functionality of EASTL that make it 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While much of this paper remains useful in understanding the ongoing pain-point of The Industry, some of it is no longer valid:

- C++11/C++14 features like move semantics and modern allocators solve many issues.
- Better optimized debugging support by compilers is now available. (e.g., GCC's –Og)
- The Industry has since changed (a 'large' game is so much bigger than 2,000,000 lines of code these days, especially when you consider tools, middleware, plugins, etc.)

We feel that a holistic "here's a new STL and our problems" approach to any change seems a bit doomed. Based on EASTL and some of the authors' own experiences, we feel that what is missing from the current C++14 is containers and small independent tweaks to the language or libraries.

The problems with C++ for The Industry

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

1. Game developers don't like global new/delete. [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.
2. Can't really do thread pools correctly in plain C++11, because good thread pool implementations need to be able to do tricky/interesting hacks to reduce scheduling latency. [Scott Wardle]
3. Concerned about std::function/lambdas doing allocations [Scott Wardle]
    - Game developers cautiously use std::function/lambdas to make sure they don't resort to dynamic allocations.
4. 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]
5. Would be great to be able to get vtable pointers as global objects without requiring an existing reference to the object. [Nicolas Fleury]
    - Also would be interesting if virtual functions' dynamic dispatch could be implemented as a library solution.

Re-examination of N2271 (EASTL)

Re-examining N2271 (by Sean Middleditch) yielded the following issues. All references are from N2271.

Original Problem Statement: std STL allocators are painful to work with and lead to code bloat and sub-optimal performance. This topic is addressed separately within N2271.
Response: C++11 simplified allocators, and the polymorphic allocator proposal helped here a lot. Lack of proper alignment support and more advanced allocation flags (e.g., clear to 0) are annoying, but not show stoppers.

Original Problem Statement: 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).
Response: All three of those extensions are in C++11. We are still missing some handy ones, though, like flat_map.

Original Problem Statement: 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 N2271
Response: This is still true.

Original Problem Statement: 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
Response: All the major compilers today support inlining small accessor functions even in debug builds. This item needs measurement, though.

Original Problem Statement: 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.
Response: 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.

Original Problem Statement: 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.
Response: 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.)

Original Problem Statement: 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.
Response: Examples are emplace_back and move semantics.

Original Problem Statement: 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.
Response: 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.

Original Problem Statement: 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.
Response: 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.

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

Original Problem Statement: 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.
Response: 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.

Original Problem Statement: 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.
Response: 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.

Original Problem Statement: 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.
Response: 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 N2271, we do have some recommended 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 (in my opinion) 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 things 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.

Other relevant WG21 papers

A number of current and past WG21 papers are also relevant towards better support for The Industry:

- N3759 - Standard SIMD types.
    - 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
- N3996 - Static reflection

Use cases for reflection

In our experience, games tend to use thousands of enums (often auto-generated by tools) to communicate between sub-systems. Being able to iterate over enums and automatically convert enums to strings (statically) would be helpful for writing debuggers, loggers, profilers.. which currently rely on machine-generated code or hand-written mappings.

Static reflection could also 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 iterate on designs to 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 layout design with SIMD was highlighted in a GDC talk this year (https://deplinenoise.files.wordpress.com/2015/03/gdc2015_afredriksson_simd.pdf). From reading the slides, you can see that one of the only viable options for these game developers is still just to use intrinsics (and other unnatural-looking code), which could definitely use cleaning up from a language perspective.

Summary

While work continues as an ongoing proposal, items should be addressed individually and split up into separate work items. We propose the following groupings of work items:

1. New containers and algorithms.
    - inline lists and trees

    - fixed/flat trees
    - whole new containers like ring_buffers
    - whole new algorithms like radix_sort
2. Stricter requirements on existing containers and algorithms.
    - no allocations for default-empty construction
    - guaranteed cheap/trivial/noexcept move
    - debug builds cannot break algorithm efficiency requirements (*cough* Microsoft's std::sort *cough*)
3. misc. issues
    - smaller headers, like <algorithm>
    - QoI on container inspection
    - -fno-exceptions, -fno-rtti or equivalents should be guaranteed to be supported by the library, possibly via feature-testing support for these features

The Future and the creation of SG14

N4456 was presented at Lenexa 2015 meeting. It received positive review and encouragement to start its own SG. The difficulty of Games developer in attending C++ Standard meeting due to their schedule was acknowledged. It was decided that there was enough support and interest to start a new SG. SG14 was created with the tentative name of Games Dev and Low Latency. I am interim chair until someone more appropriate is found or wants the job. Some people have asked about the specifics of low latency. I think I like to define that later. But since it is difficult to get the Gaming community to attend C++ Std meetings, we are proposing two special F2F meetings. One at CPPCon 2015, and one at GDC 2016 hosted and paid by Sony. The CPPCon will be a 2 day session. 1 day will be a special focus on Games presentation. The second day will be the official SG14 meeting attended by all interested as well as committee members.

A number of C++ Std committee members have agreed to attend both meetings to help triage papers before they are presented to the Feb and June C++ Standard meeting. Our intention is to triage proposals, initially of a high level nature, but becoming more specific at GTC.

Acknowledgements

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