Making C++ Better for Game Developers – Progress Report

Author: Patrice Roy, based on suggestions made from various experts from the game development domain, including Nicolas Fleury (Ubisoft), Gabriel Morin (EIDOS), Arthur O'Dwyer, Matt Bentley, Staffan Tjernstrom and others.

Reply to: patricer@gmail.com

Target audience: EWG, SG14

Contents

Abstract

Starting December 2019, SG14 has begun collecting information as to how C++ could be improved from the perspective of game developers. The requests that have been collected cam from prominent members of the game development community, starting with the greater Montréal area (Québec, Canada), a well-known hub for that application domain.

These requests were brought to several SG14 meetings to be discussed, categorized and for a selection to be made. SG14 is the C++ Standards Committee study group for low-latency, game, finance, and embedded systems programming. The intent of this process was to identify, from the set of requests that had been collected, those that would benefit this important subset of the C++ programmer community the most and turn these into individual papers to be discussed on their own merits.

This progress report aims to provide an overview of this overall effort. It establishes the guiding principles behind the set of requests. It also identifies the topics for papers know to come, which form an open set as we expect more to be added once this effort is made public, and it identifies those requests which have already been serviced through existing efforts.

Note: most of the suggestions in this document would not be major features of C++. The set of suggestions, however, can be seen as significant, aiming to address aspects of the language that (a) make the language more difficult to us or learn than it could be, (b) brings users to write workarounds or (c) hamper adoption of subsets of the language.

Guiding Principles

Contributors to this effort have committed to the following guiding principles:

- Things that simplify C++ are good
- Things that make C++ more teachable are good
- Avoid negative performance impacts
- Debugging matters

Not all suggestions made throughout this process fall into the purview of one or more of these guiding principles, but they all aim not to contravene these principles.

Things that simplify C++ are good

C++ is a rich but complex language. Some of the suggestions that stem from this effort aim to reduce the number of “gotchas” and pitfalls faced by C++ programmers and would reduce the number of workarounds and trickery involved in using C++ to write games.

Some specific remarks that have been made include:

- Things that make generic programming simpler are appreciated.
- One needs to understand how the code will run based on the source code.
- Unexpected side-effects are to be avoided.

Things that make C++ more teachable are good

The game programming industry is big and turnover rates make training new colleagues something important. Things that make C++ more teachable reduce costs, make professional

insertion easier, and help reduce debugging efforts (see also **Erreur ! Source du renvoi introuvable.**).

Avoid negative performance impacts

SG14 developers use C++ for many reasons, but control and performance characteristics are very high on the list. Things with negative impact on performance are unacceptable to them. This means that new features that could impact performance negatively need to be accompanied with an opt-out mechanism.

Debugging matters

For some of the suggestions in this document, availability only in so-called « debug » builds would be acceptable due to the costs expected in so-called « release » builds. Contributors know that the standard does not recognize this distinction but hope that we can find a way to make some of the more costly features available in a conditional manner.

- Contributors have discussed the importance of manageable « Debug/ -O0 » builds as today, they sometimes need to debug code built with « Release/ -O2/ -O3 » builds to make their programs fit into memory.
- Many have reported that design styles tend to change (monadic programming, functional programming, lazy execution) making it harder to grasp what's going on from the source code (it's « more magic »)
- Call stacks that are too deep make debugging harder

SG14 Process

The author has initially met with prominent members of the game development community to collect an initial set of requests; these meetings were started at the behest of said members themselves, and as such this document aims to carry their voice.

The principal author of this document then grouped the requests by topic to turn this set of ideas into an organized whole to help discussion.

Following this initial collection and grouping effort, this unnamed proto paper was brought to several SG14 meetings to progress through the set of requests found therein. For each suggestion, SG14 provided guidance:

- Is this feature something SG14 wants?
- If the suggestion is to be pursued, should it be pursued on its own or as part of a related group?
- Is this something that can be achieved with existing language facilities? If so, is it worthwhile to pursue the suggestion?
- Are there alternative approaches that would be preferable?

This led to questions being raised, requests being dropped, requests being modified, workarounds being identified, etc. This discussion effort is still ongoing, but sufficient progress has been made that the production of actual papers can begin.

Actions

For each suggestion / suggestion category / suggestion group in this document, we seek the following guidance from SG14:

- Is this something SG14 wants?
- If the suggestion is to be pursued, should it be pursued on its own or as part of a related group?
- Is this something that can be achieved with existing language facilities? If so, is it worthwhile to pursue the suggestion?
- Are there alternative approaches that would be preferable?

Requests by Category (Overview)

What follows is a set of tabular overviews of the effort so far. Requests have been categorized in one of the following groups:

- Compile-Time Computing
- Memory Allocation and Deterministic Behavior
- Attributes
- Move Semantics
- Handling Disappointment
- Pattern Matching
- Tooling and Ease-of-Coding
- Networking
- Parallel and Concurrent Computing
- Logging and I/O
- Numeric Computing
- Miscellaneous

In each table below, you will find:

- A (very brief) summary of what the request is
- Its status from the perspective of SG14: not pursued (the group was not convinced enough, or there exists a workaround already), adjusted (to be pursued but in a different form), pursued (papers incoming), to be discussed (not seen yet by SG14).

Compile-Time Computing

The set of compile-time computing aspects of C++ grows with each version of the standard. The following suggestions would help game developers perform optimizations that seems worthwhile to them.

Request	Status
Allow overloading based on constexpr arguments	Pursued (design space to be explored). Use-cases have been presented from different application domains. Has been discussed in the past: https://wg21.link/p1045
Static reflection	Pursued in order to make the important aspects of this feature for SG14 members

	known to the wider C++ programming community
Compile-time string interpolation	Pursued. Some progress has been made with https://wg21.link/p2741 but more is needed

Memory Allocation and Deterministic Behavior

Controlling dynamic memory allocation mechanisms closely is important for games in order to ensure acceptable performance, including more deterministic execution speed.

Note: in the suggestions below, SOO stands for "small object optimization".

Request	Status
SOO Thresholds	Pursued (design space exploration). Knowing the memory allocation threshold for SOO-enabled types (std::function, std::string and others), probably through compile-time traits to let programmers avoid resorting to dynamic memory allocation unwillingly and in a portable manner.
std::inplace_function	Pursued (candidate for freestanding). std::function can allocate if constructed from a function object of a size greater than an implementation-specific threshold, so some companies reject that type outright and roll out their own homemade version. For this reason, a std::inplace_function or equivalent, which never allocates, is desired. Note: this has been discussed by SG14 in the past[1] and there is implementation experience
SOO-Enabled vector	Pursued (might be covered already by static_vector<T>). A std::vector<T>/std::array<T,N> alternative that has a (potentially compile-time known) capacity and never allocates
External Buffer Vector	Pursued (might be solved by PMR vector and monotonic_buffer_resource). A vector that manages an externally provided buffer and switches to heap-allocated memory should that buffer's capacity not be sufficient
Intrusive Containers	Pursued (see if https://wg21.link/p0406 is appropriate or needs to be modified). Used extensively by many SG14 contributors, particularly intrusive lists.
InplaceContainer<Size> Inheriting from Container Pattern	Pursued (design space exploration required; might be solved through PMR containers). A set of containers (e.g.: inplace_vector<T,Sz>) that derive from standard containers and

[1] https://github.com/WG21-SG14/SG14/blob/master/Docs/Proposals/NonAllocatingStandardFunction.pdf

	expose the same interface but supply a fixed-size buffer to manage by default)
Heap-Free Functions	Pursued (candidates for freestanding). Add heap-free options to all situations that might lead to dynamic memory allocation (e.g.: passing client-allocated buffers). In some cases, that might simply be a matter of adding a function overload taking an array of std::byte as argument
"No RTTI" guarantees	Pursued. Many SG14 companies compile with RTTI turned off, but might still want to use PMR allocators; however, some implementations use dynamic_cast in their PMR types. Offering PMR with a "no-RTTI" guarantee, or at least a compile-time checkable guarantee would be desirable. Consider eliminating note [mem.res.private-note-1][2]
"Predictable lambdas"	Might be pursued (under exploration). Being able to declare a lambda on the stack, without initializing it right away, and having access to its constructor (some sort of placement new on an uninitialized lambda, kind of like an optional<lambda>)

Attributes

There are a number of suggestions related to attributes. All attribute names below are tentative (some of the names proposed, e.g. [[invalidate]] might conflict with other ongoing efforts in the language). Further exploration can change some of these from attributes to some other form (keyword, function, trait, etc.)

Request	Status
Support for User Attributes	Might be pursued (under exploration). Allowing users to implement their own attributes to replace macro-based tricks frequently found in game engines with something "in-language"
[[invalidate_dereferencing]]	Pursued. Annotate the pointer argument passed to realloc() with [[invalidate_dereferencing]]. The intent is that the compiler should consider *ptr to be invalid after the call to realloc(), but using ptr without dereferencing would still be valid. Would fix what some consider to be a "UB pitfall" with realloc(), while providing an attribute usable for user code wanting the

[2] https://eel.is/c++draft/mem.res.class#mem.res.private-note-1

	same optimization opportunities and semantics.
[[invalidate]]	Pursued (there is implementation experience). Annotate the pointer argument passed to free() with [[invalidate]]. The intent is that the compiler should consider both ptr and *ptr to be invalid after the call to free(). This would address what some consider to be "UB pitfalls" at compile-time, while providing an attribute usable for user code wanting the same optimization opportunities and semantics.
[[simd]]	Not pursued as such (work ongoing for the Parallelism TS).
[[no_copy]]	Might be pursued (under exploration). Annotate types and function arguments with [[no_copy]] if only move and RVO are acceptable. Type definition and function code can also evolve over time, making the guarantee at the function level is valuable. Note: part of the intent is to help with junior programmers who might not understand every intricacy of C++ value categories.
[[rvo]]	Might be pursued (under exploration). Annotate functions with [[rvo]] to ensure they only compile if used in a RVO situation. There might be a basis in tps://wg21.link/P2025 and in Clang's non-standard [[musttail]] attribute: https://reviews.llvm.org/D99517
[[side_effect_free]]	Pursued. Annotate functions with [[side_effect_free]] and make this checkable at compile-time. The intent would be to open up optimization opportunities such as automatic memoization. Prior work includes [[pure]] proposals and the [[conveyor]] suggestion for contracts.
[[trivially_relocatable]]	There is strong interest in a [[trivially_relocatable]] attribute such as the one in https://wg21.link/p1144 Note: some companies have their own is_memcopyable trait to simulate [[relocatable]].

Move Semantics

Request	Status
Move semantics are perceived as important but too easy to misuse	Might be pursued (under exploration): make it so a function taking const T&& as argument

	fails to compile or is warned about (too easy to write such a signature by copy-pasting from a copy constructor / copy assignment). Reported as a pain point by numerous contributors.

Handling Disappointment

Request	Status
So-called « Herbceptions » are looked upon favorably	Pursued. It's more than "herbceptions" however: this is a major and multi-faceted issue that requires a paper on its own

Pattern Matching

Request	Status
The switch-case style pattern matching (inspect) is looked upon favorably	This is more of a general support from SG14 for the general Pattern Matching features effort than a proposal on its own

Tooling and Ease-of-Coding

Game development companies typically develop tools to assist them and make them more productive. Even though C++ has not (traditionally) been known as the most "toolable" language, there are ways in which C++ could become better in that area. The items in this section include aspects which would make C++ easier to debug.

Note: recent progresses (std::mdspan, some vendors making it easier to avoid stepping through std::move() or std::forward()) have been noted and appreciated by SG14 contributors.

Request	Status
nameof operator	Pursued (exploration of design space required). See the nameof operator in C# and #[derive(Debug)] in Rust for inspiration (also https://github.com/Neargye/nameof). Note: might be solved by SG7 efforts
Better compile-time error detection	This is a general wish for things that will help compiler catch more errors at compile-time (there's hope that concepts will play a role there). Clarifying what the compiler "sees" and what it does not "see" would help wrote more "debuggable" code.
Conditional compilation	Pursued (exploration of design space required). Something that would replace #ifdef … #endif and would allow one's code to be checked for one platform while compiling for another (it seems to be a pain point for individuals writing code for multiple platforms).

Networking

Networking is something that every game engine has to implement by itself; a std:: version would be seen as something useful. Boost ASIO seems heavy; a replacement for C sockets would

be a huge win. Games would probably use the low-level std:: API for networking and use their own mechanisms on top of it, including their own asynchronous utilities.

Note: there have been discussions in WG21 as to whether it would be reasonable to provide a basic sockets replacement for C++ would be useful given all of the security concerns we have today. For games, the answer to this would be "yes". Not everyone needs security; some people just need fast and low-level. For embedded, a small and fast low-level interface would be most important (there's a stack everybody uses: https://en.wikipedia.org/wiki/LwIP).

Request	Status
A small, fast and low-level layer including sockets.	Pursued.

Parallel and Concurrent Computing

Request	Status
Compile-time Evaluated Thread-Safety	Might be pursued (under exploration):to allow enforcing Rust-inspired resource management in order to help validation non-thread-safe operations at compile-time
Naming, tracing and debugging	Might be pursued (under exploration): adding facilities to portably name mutexes and threads
Support of almost portable facilities	Might be pursued (under exploration): adding facilities to control thread priority and stack size. There has been work already, see https://wg21.link/p0484, https://wg21.link/p0320 and https://wg21.link/p2019 as well as *https://clang.llvm.org/docs/ThreadSafetyAnalysis.html*

Logging and I/O

Request	Status
Better logging facilities	Might be pursued (under exploration): some languages have optional attributes to know "who called you" which can be useful for logging. Note: std::stacktrace will help. Note: static reflection will help

Numeric Computing

Request	Status
Linear algebra	Pursued. SG14 supports the addition of foundational types for linear algebra (efforts are ongoing in that respect). Each game engine has its own version of such utilities, and so does each middleware, but there seems to be "holes" in most of them. In general, it would be good if what can be done in a language such as HLSL could be done directly in C++.

Opt-in UB on Unsigned Overflow	Might be pursued (under exploration): There is a need for an integral type (at least the 32 bits flavor) for which overflow would be UB

Miscellaneous

Request	Status
Forward Class Declarations with Inheritance	Might be pursued (under exploration): it would be useful to allow a forward class declaration specifying inheritance relationships when using pointer-to-base / pointer-to-derived conversions
"namespace class"	Might be pursued (under exploration): when defining a class' member functions in a .cpp file, repeating the class name everywhere can get tedious; reducing the noise would be useful
Constrained Construction	Might be pursued (under exploration): a syntax that would constrain the number of constructors involved at the call site (e.g.: construct(1) auto a = f();) might help protecting against performance losses resulting from unwanted conversions.
Flags-Only enums	Might be pursued (under exploration): enumerations that can only be flags, which could influence "stringification", particularly if two symbols have the same value. Note: workarounds have been proposed in the past, notably https://gpfault.net/posts/typesafe-bitmasks.txt.html, https://dalzhim.wordpress.com/2016/02/16/enum-class-bitfields/ and https://dalzhim.github.io/2017/08/11/Improving-the-enum-class-bitmask/
Member Functions of Enums	Might be pursued (under exploration): of particular interest would be conversion operators
Better Support of Arrays with enum-Based Strong Types	Might be pursued (under exploration): enum-based strong types and arrays mix unpleasantly, which blocks their adoption in some companies. See https://wandbox.org/permlink/dZvsd4MTz3WD7282
Making std::initializer_list Movable	Might be pursued (under exploration): this would allow such things as initializing a std::vector<std::unique_ptr<T>> with a pair of braces containing a sequence of calls to std::make_unique<T>(). Prior efforts include https://wg21.link/p0065
Explicit list-initialization	Might be pursued (under exploration): looking for fixes to the dichotomy between such situations as vector<int>(10,-1) and vector<int>{10,-1} which have been "gotchas" of C++ since C++11. Some forbid constructors taking initializer_list arguments for that reason

Efficient Downcasting	Might be pursued (under exploration): need for a way to downcast to the most-derived type at low cost, e.g.: using sorted vtables for statically linked .exe. Companies write their own currently but it's nonportable
Covariant Cloning	Might be pursued (under exploration): being able to have covariant return types based on unique_ptr<T> as well as on T*. There has been prior work in that regard[3]
Homogeneous Variadics	Might be pursued (under exploration): making it easier to write variadic packs where all elements are of the same type. Note : can be achieved in C++20 through techniques such as https://wandbox.org/permlink/f2TasMibAYysw2pM
Named Arguments	Might be pursued (under exploration): prior efforts include https://wg21.link/n4172. Made possible in part with designated initializers.
SoA to AoS	Pursued (design space to be explored): arrays of structs (AoS) make it easier to understand and structure classes but are often less efficient in terms of time and space usage than structs of arrays (SoA). A way to "transform" something expressed as an AoS into its SoA equivalent would be very useful
Unified Function Call Syntax	Might be pursued (under exploration): tooling and ease of use are motivating factors. Code editors tend to be better at assisting programmers with x.f(y) than they are with f(x, y). There have also been reports that free functions tend to be coded two or three times separately as programmers don't always find them, and end up rolling their own

[3] https://deque.blog/2017/09/08/how-to-make-a-better-polymorphic-clone/ which uses CRTP, https://www.fluentcpp.com/2017/09/12/how-to-return-a-smart-pointer-and-use-covariance/ which is a bit involved, and https://herbsutter.com/2019/10/03/gotw-ish-solution-the-clonable-pattern/ which requires metaclasses