Making C++ Better for Game Developers – Some Requests

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: SG14

Contents

Abstract

What follows collects requests made by prominent members of the Game Development community in meetings held in December 2019 and September 2020. These individuals are C++ users that love C++; they also would like the language to evolve in a way that would help write games. It also includes comments formulated in writing by SG14 contributors between September 7, 2021 and October 4th, 2021.

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.

Note: the grouping of suggestions by topic is the author's humble tentative to turn this set of ideas into something more organized. However, the author is highly fallible and the grouping is eminently perfectible.

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?

General Principles

Contributors to this document 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 in this document 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 provided in this document aim to reduce the number of "gotchas" and pitfalls faced by C++ programmers, and would reduce the amount of workarounds and trickery involved in using C++ to write games. In particular, things that make generic programming simpler are appreciated.

Things that make C++ more teachable are good

The games 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 Debugging matters).

Avoid negative performance impacts

Game 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 for game development.

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.

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.

Overloading based on constexpr arguments

One missing piece of the constexpr effort is the ability to know when a function argument is evaluated at compile time, which would allow overloading based on that fact.

Example:

```
template <class T> string MyFormat(constexpr const char*, T&&); // A
template <class T> string MyFormat(const char*, T&&);           // B
MyFormat("Some format {:d}", someArg); // calls A
MyFormat(RuntimeFmtString(), someArg); // calls B
```

Example:

```
class MyString {
   // ...
public:
   MyString(constexpr const char*); // A (e.g.: stores pointer directly)
   MyString(const char*); // B, uses normal code path (SSO, heap, etc.)
};
MyString s{ "some str" };   // ctor A
MyString s{ RuntimeStr() }; // ctor B
```

Example:

```
class CommandLineArguments {
   // ...
public:
   // first argument only compiles with string literal,
   // object only stores pointer
   void add(constexpr const char*, char, void(*)());
   // ...
};
// ...
CommandLineArguments cmd;
cmd.add("help", 'h', &someFunc);
// cmd.add(RuntimeString(), 'x', &otherFunc); // does not compile
```

Note: this has been proposed in the past (https://wg21.link/p1045) and was last discussed in XXX.

Note: there are workarounds for some use-cases, such as https://mpark.github.io/programming/2017/05/26/constexpr-function-parameters/

Note: could templates with NTTP be a workaround?

Note: could consteval be a better keyword in this case?

Note: possible issues with this have been signaled. [*Begin quote—"[The] example here reminds me of https://quuxplusone.github.io/blog/2021/08/07/p2266-field-test-results/#libreoffice-ostring and it has the same issue:*

```
struct MyString { MyString(constexpr const char *p) { assume the string's contents will never change } };

char buf[10] = "hello";

constexpr const char *p = buf;

int main() {

  MyString m = p;  // initializing m with a compile-time-constant pointer to some const chars

  strcpy(buf, "goodbye");

  std::cout << m;  // oops, the string's contents have changed

}
```

C++ does not give the programmer any tools to detect whether something is a string literal or not. There have been past proposals for something like `std::string_literal`, a magic library type similar to std::initializer_list<T>, so that functions could opt into taking only string literals instead of arbitrary arrays-of-char. Of course some people would want the string's value to be compile-time-manipulable, and others would want it to be "erased" (non-compile-time) in the same way as std::initializer_list<T> in order to avoid template bloat, and actually a lot of people (N4121, P1378; P0259 is related) seem to want the length of the literal to be compile-time-available but the character values to be "erased."."—End quote]

Static reflection

Static reflection in general is highly desirable. In particular:

- Ways to know on what line an instruction is (note: more than what is provided by std::source_location)
  - Note: some contributors have claimed that std::source_location already provides what is being requested here, so clarification of needs might be in order. Others claim they are different. We probably need more convincing examples.
- **std::variable_name** (inspired by the nameof operator of C# but different)
- **std::declaration_source_location**

[*Begin quote–"The addition of std::source_location in C++20 is great, but it is incomplete for our needs. Imagine you have your own std::vector-like class and you want to know what reserved size should be used for all the instances in your application:*

`template <typename T>`
`struct MyVec {`
`  MyVec(const char* varName =`
`  std::variable_name());`
`};`	

`struct MyClass {`
`  MyVec<int> m_MyVec;`
`};`
	`varName` would need to contain both `MyClass` and `m_MyVec`

`void foo() {`
`  MyVec<int> myVec;`
`}`
	`varName` would need to contain both `foo` and `myVec`

Here, "need" is the key point. One could also consider having ***std::declaration_source_location*** *and have the declaration line/file of* `m_MyVec` *and* `MyVec`*. In the end, we need the full information to fully identify the variable in question."–End quote*]

Note: some companies use C# with custom syntax to generate C++ code.

Note: some companies use macros to do so.

Note: generative code (e.g.: Herb Sutter's metaclasses?) would be welcome. Code generation seems quite common in game development companies.

Note: Nicolas Fleury presented at CppCon a way to work around this on Windows using _ReturnAddress and then using .pdb related tools:
https://www.youtube.com/watch?v=tD4xRNB0M_Q&t=1915s

Reflection on enums

Some needs targeting specifically reflection on enums:

- How many symbols are there in the enumeration?
- Are the values consecutive?
- A checked_cast to / from the underlying type

Reflection on classes and structs

Some needs targeting specifically reflection on classes and structs:

- Including static iteration on members

Note: this might be covered in part or totally by the efforts of SG7.

Compile-time string interpolation

There is a need for compile-time string interpolation given values known at compile-time. Example (strawman syntax; using '$' in C++ is probably a no-go, being very controversial; a prefix 'f' is used in Python, which might be an option):

```
template <int N>
   struct Facto {
      static_assert(N >= 0, $"{N} is negative"); // Ok: N known at compile-time
      enum : unsigned long long { value = N * Facto<N-1>::value };
   };
template <>
   struct Facto<0> {
      enum : unsigned long long { value = 1ULL };
   };
```

Ideally, this interpolation should follow the same format as std::format().

Run-time string interpolation has also been reported as important, ideally with the same syntax (e.g.: $"Val: {variable}" and $"Val: {constant}") as compile-time interpolation.

The upside of this feature would be in code refactoring, as it would alleviate the need to maintain the list of arguments involved in the string formatting operation separately from the string itself, leading to run-time errors when both are desynchronized.

Note: this might be covered in part by std::format() (https://wg21.link/p2216). I think it might have been fixed as a DR for C++20 (to be verified). If that is the case, it might be possible to resolve this problem through user-defined literals.

Note: if functions could return constexpr std::string objects, this problem could be solved by calling a constexpr std::format().

Traits

Note: this section has been struck by demand of the original contributors, and will be removed from future versions of the document unless SG14 requests otherwise.

~~Adding a std::is_complete<T> or a std::is_complete_type<T> traits. The use-case would be in static_assert where one would want to use sizeof(T), but when T is incomplete the sizeof operator does not compile and the static_assert message is not generated.~~

~~For example (https://wandbox.org/permlink/3JlASHwitUBMg5j4):~~

```
struct X;

template <class T>
   void f() {
      static_assert(sizeof(T) >= 4, "type too small");
   }
int main() {
   f<int>(); // probably ok
   f<double>(); // probably ok
   f<char>(); // not ok, but static_assert message is used
   f<X>(); // not ok, static_assert message not really used
}
```

~~Possible issue: the trait's value would be context-dependent, which might be a showstopper:~~

```
struct X;
static_assert(!std::is_complete_type_v<X>);
struct X{};
static_assert(std::is_complete_type_v<X>); // hum...
```

~~Note: some contributors have suggested that would be a bad idea. [*Begin quote—"[This] is a really really bad idea, for deep core-language reasons. See*~~

~~*https://stackoverflow.com/questions/1625105/how-to-write-is-complete-template*~~

~~*https://reviews.llvm.org/D108645*~~

~~*https://godbolt.org/z/nr3dPY7va*~~

~~*However, also notice that the OP's problem (failing to print a nice static_assert message) is solved in C++20 because you can write ad-hoc well-formedness constraints:*~~

```
~~  static_assert(requires{ sizeof(T); }, "type incomplete, or maybe not an object type");~~
~~  static_assert(sizeof(T) >= 4, "type too small");~~
```

Also, having an is_complete<T> trait wouldn't solve OP's problem, because what are you going to do?

static_assert(is_complete_v<T> && sizeof(T) >= 4, "type too small");

would still fail to print the nice message; splitting it into two static_asserts just gets back a lesser version of the C++20 solution. Lesser because e.g. [T=int()] is a type where sizeof(T) doesn't compile, and yet T is a complete type. If you're trying to check that sizeof(T) is well-formed, just check that; don't mess around with abstract ideas like "completeness" that don't fully capture the sense of what you need..” — End quote]

The suggested workaround has been received favorably by some contributors.

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”.

SOO Thresholds

Knowing the memory allocation threshold for SOO-enabled types (std::function, std::string and others), probably through compile-time traits, would be advantageous as it would allow programmers to avoid resorting to dynamic memory allocation unwillingly and in a portable manner.

Example (strawman syntax):

```
template <class F> std::function<void()> make_func(F f) {
   // only compiles if construction of a function<void()> from
   // an object of type F would not allocate
   static_assert(sizeof(F) <= soo_max_size_v<std::function<void()>>);
   return { f };
}
```

Example (strawman syntax):

```
template <class F> auto make_func(F f) {
   // returns a function<void()> if one can construct it without
   // allocating; fallback on a homemade "plan B" otherwise
   if constexpr(sizeof(F) <= soo_max_size_v<std::function<void()>>)
      return std::function<void()>{ f };
   else
      return my::inplace_function<void()>{ f }; // see below
}
```

Note: an exhaustive list of the SOO-enabled types might be useful here.

Note: an alternative approach would be to have a recognizable default value if soo_max_size<T> is applied on a non-SOO type.

Note: different approaches to the problem have been suggested. [*Begin quote—“I recommend changing your strawman syntax to std::function<void()>::max_soo_size(). There's no reason to use a type-trait here, because (A) this isn't a place for generic dispatch and (B) this isn't a place for user customization. The right precedent is something like unordered_map.bucket_count(): an idiosyncratic implementation detail that we're exposing on the bad assumption that someone cares about it.*

Why "bad assumption"? Because SOO is conditioned on more than just size.

https://quuxplusone.github.io/blog/2019/03/27/design-space-for-std-function/#sbo-affects-semantics

- Some vendors' std::function will store small objects on the heap anyway, unless they are nothrow_move_constructible.

- Some vendors' std::function will store small objects on the heap anyway, unless they are trivially_move_constructible and trivially_destructible.

So, just knowing that your lambda-or-whatever is physically small enough to fit doesn't tell you that the vendor actually will use SOO. So, if you pursue this at all, maybe the right interface would be a sort of "dry run constructor," here spelled `will_hold_inline`:

```
using F = std::function<void()>;

auto t = []() { puts("whatever lambda"); };

if (F::will_hold_inline(t)) {

    F f = t;

    assert(f.is_held_inline());

} else {

    F f = t;

    assert(!f.is_held_inline());

}.”—End quote]
```

Note: other alternative approaches have been mentioned. [*Begin quote—“It seems that what we care about is whether we allocate or not. The examples don’t attempt to do something specific with the threshold value. In this case, wouldn’t some kind of [[no_allocation]] attribute that causes compilation to fail if the called code contains any kind of allocation do the job? Since strings support constexpr now, building a small string whose value is known at compile time should as an example pass the [[no_allocation]] test.*

Perhaps a type trait is_allocating would also make sense… any class containing a new or malloc instruction would be disqualified unless that instruction is excluded by an if constexpr at compile time”—End quote]

std::inplace_function

Since std::function might allocate if constructed from a function object of a size greater than an implementation-specific threshold, some game development companies reject that type outright. However, since the functionality provided by std::function is used widely in games, game development companies tend to 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 (https://github.com/WG21-SG14/SG14/blob/master/Docs/Proposals/NonAllocatingStandardFunction.pdf) and there is implementation experience.

Note: question from a contributor: [*Begin quote—"You might link to the SG14 repo's implementation. ;) And I'd be interested to learn whether any of your contributors are actually using it in practice, and if not, why not, and how can we get them to switch to it. I mean, if people are avoiding an existing free library and rolling their own anyway, then how can we be sure that they wouldn't have the same reaction to a std::inplace_function after standardization? In which case, standardizing inplace_function would be wasted effort."—End quote*]

Note from a contributor: [*Begin quote—"As a data point (albeit from a different sphere), we use stdext::inplace_function in a number of our code-bases. Where it's not performance critical, std::function still gets used, which is mostly an internal education issue."—End quote*]

Containers

Game development companies seem to prefer their own containers to the set of containers provided by the C++ standard library. However, some additions to the standard library would be appreciated.

SOO-Enabled vector

There is a need for an SOO-enabled vector.

Note: there have been small_vector<T,N> proposals in the past, where N could be the SOO threshold. Would this be sufficient?

Note: see the discussion on static_vector<T> from David Stone at CppCon 2021[1]: which touches on a number of SG14-related concerns.

Note: need to differentiate between fixed_capacity_vector<T, Cap> (e.g.: Boost's static_vector, P0843, fixed capacity, "fails" when the capacity is exceeded — and defining the behavior on "failure" is ugly) and small_vector<T, SOOCap=SomeDefault> (LLVM's SmallString IIRC, infinite capacity, heap-allocates when the SOO is exceeded).

External Buffer Vector

Some companies report using their own flavor of vector that can manage an externally provided buffer, and switch to heap-allocated memory should that buffer's capacity not be sufficient. Like the small_vector<T,N> above (see SOO-Enabled vector), the N parameter would be the SOO threshold, and the container would switch from external buffer ownership to internal buffer ownership when allocating from the heap.

The general form would be (strawman names) flexible_vector<T> fv(buf) taking and externally defined buffer whose lifetime is under the control of client code (and could be on the stack). A small_flexible_vector<T,N> could internally use an alignas(T) std::byte[N*sizeof(T)] and derive from flexible_vector<T>, passing the internal array to its base class constructor. Having the same invariants for the base class and the derived class, this would not break Liskov's principle.

[1] https://www.youtube.com/watch?v=I8QJLGI0GOE

Note: this seems feasible since C++17 through std::pmr::vector using a std::pmr::monotonic_buffer_resource as storage. Does this suffice?

Note: question from a contributor: [*Begin quote—"Is this basically a way of "transferring ownership" of an allocation from the user into `vector`, and/or transferring ownership back out? Having a general-purpose interface for transferring allocations would in general be pretty great... but would also be a nightmare to specify, and would really lock down vendors into one specific implementation (which might not be bad).*

std::vector<char> v = {'a', 'b', 'c', 'd'};

*struct TakeResult { char *ptr; size_t len; size_t cap; }; // and also a copy of the relevant allocator?*

TakeResult tr = v.take(); // take ownership of the allocation

assert(v.empty());

std::string s;

s.adopt(std::move(tr)); // transfer ownership of the allocation to a string!

// actually, uh-oh, s needs to become null-terminated somehow

Or is this more like a std::pmr::vector attached to a monotonic_buffer_resource (which uses a fixed buffer first, if provided; and then switches to heap allocation)?."—End quote]

Intrusive Containers

Many game development companies use intrusive containers, particularly intrusive lists. A set of such containers has been proposed for standardization (e.g.: https://wg21.link/p0406), and seemed to be received favorably, but there does not seem to be recent progress on that front.

Note: was p0406 appropriate to SG14 needs? If that is the case, should we make a push for it? If not, what should be modified in order to make it more appropriate?

InplaceContainer<Size> Inheriting from Container Pattern

Contributors have reported a practice they use internally and that might be beneficial for some programming domains, in particular those concerned with low-latency.

The pattern involves sub-classing containers using the heap in order to add a form of static buffer. They use public inheritance to reduce impact on existing code, and ensure invariants are respected through the base class not owning its buffer whilst being able to allocate if necessary.

This leads to code such as :

```
template <typename T, int size>
  class inplace_vector : public vector<T> { /* ... */ };
```

… where the base class (vector<T> in this case) has to be able to use a buffer it does not own. Supposing for the sake of the example that the base class uses a bit to distinguish between its owning and non-owning representations, that class would act as a std::vector<T> if owning and

as an SOO-enabled container relying on a buffer of size*sizeof(T) bytes supplied at construction time by the derived class otherwise. Similar tricks can be used for e.g. an inplace_string where the SSO buffer is of a capacity controlled by user code.

Heap-Free Functions

Adding heap-free options to all situations that might lead to dynamic memory allocation (e.g.: passing client-allocated buffers) would help optimize execution speed in some occasions. In some cases, that might simply be a matter of adding a function overload taking an array of std::byte as argument.

Note: inplace_function could be considered a part of this suggestion.

[*Begin quote—"We've have seen in the past some C++11 date-time utilities using heap to our surprise. We need heap-free versions for everything we could use in the stdlib, as we do not mind to use alloca() or static-sized buffer on stack on our side."—End quote*]

"No-RTTI" Guarantees

Games typically compile with RTTI turned off, but might still want to use PMR allocators; however, some implementations use dynamic_cast in their PMR types, apparently. Offering PMR with a "no-RTTI" guarantee, or at least a compile-time checkable guarantee would be desirable

Note: [*Begin quote—"Yes, `std::pmr::memory_resource::do_is_equal` used to specifically mandate the use of dynamic_cast, but LWG3000 fixed that.*

https://cplusplus.github.io/LWG/issue3000

Unfortunately, I notice that there's still a non-normative note on the pure virtual method that encourages use of dynamic_cast in derived user types. Maybe someone should open an LWG issue to eliminate that note.

https://eel.is/c++draft/mem.res.class#mem.res.private-note-1."—End quote]

Predictable lambdas

There is a need to be 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>).

Note: EXAMPLE NEEDED

Note: [*Begin quote—" Perhaps like this (C++17):*

```
auto make_lambda(std::string s) { return [s](auto t) { return s + t + "!"; }; }

int main() {

    using LambdaType = decltype(make_lambda(""));

    alignas(LambdaType) char buffer[sizeof(LambdaType)];

    auto *lam = ::new ((void*)buffer) LambdaType( make_lambda("hello ") );
```

*(*lam)("world");*

lam->~LambdaType();

}

or like this (C++20) for captureless lambdas only:

int main() {

using LambdaType = decltype([](int x) { return x + 1; });

alignas(LambdaType) char buffer[sizeof(LambdaType)];

*auto *lam = ::new ((void*)buffer) LambdaType();*

*(*lam)(42);*

lam->~LambdaType();

}.”—End quote]

Attributes

There are a number of suggestions related to attributes. All attribute names below are tentative.

Support for User Attributes

There has been interest in allowing users to implement their own attributes. This could replace macro-based tricks frequently found in game engines with something “in-language”. We would probably need to provide a way to associate meaning with user attributes.

Note: static reflection might be a solution to this.

Note: a possible source of resistance to this suggestion would be fear that C++ would devolve into dialects, but given that this is intended to replace existing, macro-based tricks, this fear is probably unfounded (or at least there are counterarguments).

[[invalid_derefererencing]]

Annotate the pointer argument passed to realloc() with [[invalidate_dereferencing]], e.g.:

```
void *realloc([[invalidate_dereferencing]] void *ptr,
              size_t new_size );
```

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. This would fix what some consider to be a “UB pitfall” with realloc(), while providing an attribute usable for user code wanting the same optimization opportunities and semantics.

Note: the desired result is a compile-time error.

Note: this is currently QoI.

Note: even if standard library functions might do this, the intent is to allow user code to convey this meaning too.

Alternative names have been suggested: [[frees]] our [[deletes]]. Rationale (and criticism): [*Begin quote—“A better name for this attribute would be [[frees]] or [[deletes]], since the intent (at least in realloc's case) is that it takes ownership of the resource. `void free([[frees]] void *p)` should be annotated in the same way. However, this attribute is fairly useless in C++, because (A) we have so many different kinds of resources, and (B) we have RAII for this. If you're able to change your code to add a [[frees]] attribute, you're (in theory) also able to change it to make it take unique_ptr instead. unique_ptr's calling convention is worse than the raw-pointer calling convention, but that's an orthogonal topic, and we already have a non-standard attribute for that (apply [[trivial_abi]] to your codebase's unique_ptr class).”—End quote*]

Reply from another contributor: [*Begin quote—“Finding better names is of course possible, but there’s a semantic difference between the proposed [[invalidate_dereferencing]] and [[invalidate]]: in the first case, dereferencing the pointer after the call would be UB.”—End quote*]

[[invalidate]]

Annotate the pointer argument passed to free() with [[invalidate]], e.g.:

```
void free([[invalidate]] void *ptr);
```

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.

Note: the desired result is a compile-time error.

Note: this is currently QoI.

Note: even if standard library functions might do this, the intent is to allow user code to convey this meaning too.

Note from a contributor: [*Begin quote—“[The] [[invalidate_dereferencing]] and [[invalidate]] attributes are the same thing: they both say "this pointer gets freed."”—End quote*]

[[simd]]

Add [[simd]] on functions and arguments to get compile-time optimization guarantees.

Note: unclear how this would interact with the rest of the code (e.g.: could there be a [[simd]] and a non-[[simd]] version of the same function?).

Note: since we’re talking about attributes, “guarantees” might be too strong a word, but one could hope for a QoI warning if the attribute is not taken into account.

Note: EXAMPLE NEEDED

Note: there are efforts ongoing in the SIMD design space. Maybe they will meet the needs of these users

[[no_copy]]

Annotate types and function arguments with [[no_copy]] if only move and RVO are acceptable.

While it’s possible to define types to not be copyable at all, this is not always something making sense. For example, it might be reasonable in general to be able to pass a container by value to a function, but it might not be reasonable in some specific contexts; in a huge code base, one could want the [[no_copy]] guarantee at the function level instead of having to look at type definition. Type definition and function code can also evolve over time, making the guarantee at the function level is valuable.

Examples:

```
[[no_copy]] SomeContainer<SomeType> Foo();

[[no_copy]] SomeType Bar();
```

Note: maybe this is already covered by rvalue reference arguments and proper definition of copy and move special functions. Part of the intent, however, is to help with junior programmers who might not understand every intricacy of C++ value categories.

Note from a contributor: [*Begin quote—"[This] this is just "Delete your copy constructor.""—End quote*]

Another contributor: [*Begin quote—"I have found a StackOverflow question which illustrates a scenario where we would benefit from C++ offering a zero-copy guarantee:*

https://stackoverflow.com/questions/33872026/copy-elision-for-pass-by-value-arguments

Currently, there are copies in that example unless the compiler elides them under the as-if rule. I think our paper offers candidates to help guarantee the desired behavior:

[[no_copy]] Box box(Range(0.0,1.0),Range(0.0,2.0));

construct(3) Box box(Range(0.0,1.0),Range(0.0,2.0)); // one Box ctor, two Range ctors, no more

or change the Box ctor's syntax to Box(Range~ x, Range~ y) meaning "forward and elide x and y", where x and y become more "information as to how to construct an object" than actual objects.

My understanding is that C++ needs a form of "elision forwarding, such that when the compiler seesx(x) and y(y) in the Box ctor, instead of seeing named variables that do not require elision, sees instead that is has to perform a form of mathematical substitution sequence:

Box box(Range(0.0,1.0),Range(0.0,2.0)) --> x(Range(0.0,1.0)), y(Range(0.0,2.0))

… which would lead to the Range objects being directly constructed in the Box::x and Box::y member variables exactly as if prvalues had been provided.

We can do this with perfect forwarding, to the extent that we can preserve information to the effect that x and y are movable, but we lose "elidability" along the way. Having a simple syntax that preserves both informations along an arbitrary length call chain would be a gain."—End quote]

[[rvo]]

Annotate functions with [[rvo]] to ensure it only compiles if used in a RVO situation, e.g.:

```
[[rvo]] X f();

// ...

auto x0 = f(); // Ok

X x1;

// x1 = f(); // not Ok
```

[*Begin quote—"The addition of guaranteed return value optimization in C++17 is a good thing. However, the number of situations without RVO being done is too big in reality. Even if you write code that will properly apply RVO, someone can make a minor change to the code without realizing RVO will no more occur. What we would want are attributes so that such changes no more compile. The attribute does not tell the compiler to do RVO, but tells the compiler to not compile without it. The same logic can be applied to a [[no_copy]] attribute that would only compile with RVO or move semantics being used."—End quote*]

Note: while this can be done in some cases by removing copy and move from types, this is not always possible (returning a container with copy and move semantics), and it does not support evolving code over time.

```
[[rvo]] SomeContainer<SomeType> Foo();
```

Note: does https://wg21.link/P2025 provide an interesting basis?

Note from a contributor: [*Begin quote—"The example is solved by "Delete your copy and move assignment operators." Since C++17, we've had the new prvalue rules that make "copy elision" quite predictable, so it's also easy to work with non-movable types like std::lock_guard.*

However, it sounds like the text doesn't match the example. The text is talking about annotating a single function's definition (not its declaration/uses) and forcing the compiler to complain if the RVO is not used. Besides P2025, there's relevant prior art in Clang's non-standard [[musttail]] attribute:

https://reviews.llvm.org/D99517

where you annotate a specific return statement and say, "This return statement must get optimization X; otherwise, error."'—End quote]

Note from a contributor: [*Begin quote—"Sure, one can change the type, but that's not what happens "in real life". Say you return a std::vector from a function, what do you do? And for copyable types, the point is to ensure there are no copies _there_.*

It's a matter of actual production code, something that has to be kept in mind. Guidelines and coding standards have their uses, but they are limited in scope. For something "exotic" like the is_complete trait, workarounds are fine, but things like [[no_copy]] and [[rvo]] will benefit everyone."—End quote]

[[side_effect_free]]

Annotation 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.

Note: would compile-time check be a trait?

Note: might this attribute be spelled [[pure]]?

Note: there is implementation experience as most compilers offer such a feature. This attribute would make the feature portable.

Note from a contributor: [*Begin quote—" the problem here is that there are so many things "pure" might mean in C++. This is related to my point number (3), where we have a constexpr pointer to const chars and yet those chars' values can change at runtime. Same thing here. You might think `strlen` is a pure function, but in fact*

char g[] = "hello world";

void f() { g[5] = '\0'; }

int main() {

*const char *p = g;*

strlen(p); // returns 11

f();

strlen(p); // returns 5

}

So you have to start by figuring out what property the average programmer actually cares about, and then work on nailing down exactly how to specify that property. Specifying a "similar but less useful" property is... less useful. And in practice it turns out that the average programmer doesn't care about any of these properties, because calling strlen twice is plenty fast; and if calling strlen twice ever does end up being too slow, they'll just call it once and cache the value in a local variable. ;) Giving the compiler enough information to figure out that optimization on its own has a high cost (in terms of human brain cells to make sure you got it right) and a low benefit.

I'll add that any annotation that depends on human input like this will be misused. Remember my [[trivially_relocatable]] talk? Facebook had been enabling their trivially-relocatable trait for `std::list`, but `std::list` was not in fact trivially relocatable! This led to segfaults when resizing a vector<list<int>>. If you rely on the programmer to decide whether their function ComputeSomeExpensiveOperation() is [[pure]] or not (based on some standardese definition of purity that nobody at the company has ever read, let alone understood), they will mark too much stuff as [[pure]], and eventually it will bite them."—End quote]

Note: there have been [[pure]] proposals in the past. The recommendation I personally got after championing one in 2017 was to keep it simple and standardize existing practice (the equivalents in existing compilers). Anything that goes beyond that will probably hit a wall since there are "pure purists" who will react negatively.

[[trivially_relocatable]]

There is strong interest in a [[trivially_relocatable]] attribute such as the one championed by Arthur O'Dwyer in https://wg21.link/p1144

Note: some companies have their own is_memcopyable trait to simulate [[relocatable]].

Move semantics

Move semantics are perceived as important but too easy to misuse.

[*Begin quote—"A common mistake we noticed with std::move, is the following:*

```
MyClass(const MyClass& other) : m_Member(other.m_Member) {
   // ...
}
MyClass(const MyClass&& other) : m_Member(std::move(other.m_Member)) {
   // ...
}
```

instead of:

```
MyClass(const MyClass& other) : m_Member(other.m_Member) {
   // ...
}
MyClass(MyClass&& other) : m_Member(std::move(other.m_Member)) {
   //  ...
}
```

It's typically a copy-paste error. The problem is that the mistake is silent and results in copies. It could be fixed by using a non-const move (mutable_move? whatever the name, it must not compile with const&)."—End quote]

Contributors have signaled that there are situations where std::move() can be used, but no move is used at all, e.g.:

```
void Foo(some_type p) {
   auto lam = [p]() /*mutable*/ -> some_type { return std::move(p); };
   // ...
}
void Foo2(const some_type& p) {
   auto lam = [p /*= p*/]() mutable -> some_type { return std::move(p); };
   // ...
}
```

Note: some game development companies write their own std::move() replacement in their "Debug" builds to explicitly reject const T&&, e.g.:

```
namespace std {
   template<class _Ty>
```

```
     [[nodiscard]] constexpr remove_reference_t<_Ty>&&
        move(const _Ty&& _Arg) noexcept = delete;
   template<class _Ty>
      [[nodiscard]] constexpr remove_reference_t<_Ty>&&
         move(const _Ty& _Arg) noexcept = delete;
}
```

… which seems … unsatisfactory.

Note: a contributor wrote [*Begin quote—" IMHO a typoed "copy constructor" with signature `T(const T&&)` should be caught by clang-tidy or grep, not by any C++2b core-language change"—End quote*]. Others have reported to agree with that statement.

Handling Disappointment

So-called « Herbceptions » are looked upon favorably

[*Begin quote—"Herb Sutter's proposal for a new exception model is music to our ears. It would give us an exception model we could use"—End quote*]

On the question of "why optimizing exceptions might not suffice…"

As an aside, I had the occasion to discuss with a prominent member of the game development community as to the reasons why most games do not use exceptions. I mentioned a previous discussion I had had with another important game developer at a WG21 meeting, who had told me that even if exceptions were faster than using if statements and checking function return values, they would still not use them due to what they consider to be "hidden control flow".

On the issue of the exception cost model

What follows comes from a contributor:

[*Begin quote—"A commonly held belief among C++ developers is that exceptions are free when not thrown, since the stack unwinding tables are outside the functions. As a data point, on Rainbow Six Siege, if exceptions are enabled on all code with Clang 12, instead of just the few libraries using it, we measure a CPU performance regression of around 1%. This is too much.*

It is then up to debate how much this can still be improved or if this is really the result of missed optimization opportunities because of all possible code paths introduced by exceptions. As someone doing optimizations, I always have the mindset that not doing something is always faster than doing something quickly, so I'm afraid while we can measure a 1% cost for exception handling today, we can only reduce that to a certain amount and always be left with a measurable cost.

Then, there's when exceptions are thrown. At CppCon 2021, Bjarne Stroustrup described the C++ exception model as aiming for great performance when exceptions are not thrown, but expecting a cost when exceptions are thrown. What type of error handling does a video game? I would divide them in two: the exceptional situations coming from outside the game, and the exceptional situations inside the game simulation.

The exceptional situations coming from outside, like disconnecting a gamepad, network problems, online disconnections, etc. are all an acceptable fit for C++ exceptions, and on Rainbow Six Siege some of these libraries are using them internally. For some other such situations, like out-of-memory and disk access issues, video games typically just crash, because dealing with these is not worth it.

The aforementioned issues where C++ exceptions are an acceptable fit represent a very small part of the code being executed, let's say 1% (could be much more than 1% of overall code, while still being less than 1% of executed code). When it comes to exceptional situations inside the game simulation, this is where the C++ exceptions are not great. For that reason, we would prefer so-called Herbceptions, std::expected, or anything with performance matching return values with a very small scope overhead (or even an overhead reduced because of compiler complicity, like a compiler realizing std::expected can be smaller because the values stored in it are not using all the bits of the biggest type).

When it comes to bugs in our game, C++ exceptions are the exact opposite of what we need. We don't want to unwind the stack, we want instead to semantically freeze all threads and make a mini-dump as fast as possible, so we can understand what happened. With big code bases with numerous programmers in a heavy multi-threaded application, having the call stack of every thread when such issue occurs is important, and we will enrich dumps with more contextual information. We use platform exception handling (instead of C++ exceptions) for the process to survive for a short amount of time, create mini-dumps with additional information, and then kill the process.

Finally, we have the issue of semantics. C++ has opt-out exceptions, whereas we need opt-in exceptions. It's not only about the measurable cost of exception handling, it is also about the added complexity of additional possible code paths. If we were asked to use the existing exception model in 1% of our executed code, I think we could make that effort, even if I personally prefer "Herbceptions". To do this, we would need a standard where disabling exceptions is possible, even normal, in such a way that noexcept is the default and we can opt-in selectively. And we would need "doexcept" or "throws" instead of noexcept. Just specifying noexcept on a class is not sufficient for our needs; most of the code we write does not need to be exception-safe and does not need the added complexity that would come with it. An opt-in exception model would solve this problem".—End quote]

Quoting (and translating to English) freely:

[*Begin quote—"I'm not sure what that person meant by "hidden control flow", but I can comment on the general exception usage question. Our game is our religion, so the question we are always asking is: what will benefit our game? To me, a game is a simulation where error handling is less than 1% of the code we write. Thus, if exceptions become faster than if statements, that 1% becomes faster, but we can expect 99% of the code to be slower as compilers will need to inject stack unwinding in various places. If I could make a game faster by converting the codebase to exceptions, great, but even with significant optimizations, I doubt this would be possible: doing nothing is faster than doing something quickly.*

Then, we have the issue of semantics. C++ has opt-out exceptions, whereas we need opt-in exceptions. If we were asked to use the existing exception model in 1% of our code, I think we could make that effort, even if I personally prefer "Herbceptions". To do this, we would need a standard where disabling exceptions is possible, even normal, in such a way that noexcept is the default and we can opt-in selectively. And we would need "doexcept" or "throws" instead of noexcept, Just specifying noexcept on a class is not sufficient for our needs; the vast majority of the code we write does not need to be exception-safe, and does not need the added complexity that would come with it. An opt-in exception model would solve this problem.

"Survival" of our process is Ok for us, even in out-of-memory situations; we have lots of code that runs after an out-of-memory situation to perform diagnosis. However, this uses platform exception handling, not C++ exception handling, and we crash right after so I guess we don't really "survive" it. Practically speaking, we never want stack unwinding; instead, we want mini-dumps to diagnose individual threads including the (important) amount of information we add to crashes. In practice, we have a number of tools that let programmers add information to crash dumps, and we even do that in the released version of the game (although we do control

what information is allowed at that stage). We have a very wide definition of "irrecoverable cases"; for example, std::vector::operator[](invalid_index) is irrecoverable to me: I prefer to crash and see who called me with an incorrect argument.

Our assert is a deliberate crash; we have a "soft" assert that does not crash but we use it a lot less."—End quote]

Note: do we want to push for configurable defaults with error management?

Note: the quote above is generally representative of what has been reported to the author. However, there have been nuances depending on the company:

- The "opt-in" requested for exceptions seems favored by many. Some have insisted on the importance of the code being "clearly opt-in" for programmers to know it's important to write exception-safe code in regions where it counts.
- Some question the "1%" estimation in the quote, estimating the portion of error handling code to be higher, so this might vary depending on company culture and practice.
- A reported upside of exceptions in game development is the capability to bring better information when a problem occurs. Call chains that return Booleans through a number of layered function calls tend to convey that information less clearly (note: would std::expected<T,E> or something similar solve this? Note: would std::stacktrace help in this regard?); the case of an object pool failing to spawn due to insufficient capacity instead of an object pool failing to spawn without providing context for the error has been mentioned.

A contributor adds: [*Begin quote—"'99% of the code to be slower" — IMO this is repeating a common misconception. The executable will be bigger due to stack-unwinding tables, but those tables go in .rodata, not .text; that shouldn't noticeably affect the speed of the non-throwing code. There's no "ooh, this next bit might throw, so set up some handlers in preparation." That just isn't how EH works anymore. It's all static precomputed tables, kept off to the side: "oops, this line seems to have thrown; let me consult my data tables and see what I'm supposed to do now."'"—End quote*]

Other contributors have replied that the previous comment (exceptions having no noticeable impact on speed) is measurably wrong. This is, as ever, a heated topic. The key point seems to be (to this author) that a significant subset of C++ users have reasons not to use exceptions, and that the language could make their lives easier.

Pattern Matching

The switch-case style pattern matching (inspect) is looked upon favorably.

Note: there have been quite a number of proposals in that area since I collected information, so I suppose the interest in such a feature still prevails, but I do not know what would be the preferred syntax. It's clearly an active area of research.

Tooling and Ease-of-Coding

Game development companies typically have a number of 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.

[*Begin quote—“Garnered from watching the various twitter/reddit conversations that've cropped up around game dev in c++, is that the main problem is the standard and the compilers tend to ignore the fact that many of the 'high level' constructs do not optimize at all in debug mode. The concept repeated by many is that -O2 will solve things. But for games and any other field involving very large executables which require running in realtime debug mode, it doesn't. It is more of an attitudinal disfunction than an actual technical problem.*

Promoting std::fill over memset is one such example. Take using std::fill vs memset to zero memory under GCC8:

Results in debug mode (-O0):

--

Benchmark Time CPU Iterations

--

memory_filln 87209 ns 87139 ns 8029

memory_fill 94593 ns 94533 ns 7411

memory_memset 8441 ns 8434 ns 82833

==

Results in -O3 (clang at -O2 is much the same):

--

Benchmark Time CPU Iterations

--

memory_filln 8437 ns 8437 ns 82799

memory_fill 8437 ns 8437 ns 82756

memory_memset 8436 ns 8436 ns 82754

==

This is the same in clang. Clang optimizes to memset at -O2, GCC8 at -O3. That's a factor of ten difference in speed until you reach -O3. Granted memset is very limited in what values it can propagate. But the way that C++ is often communicated is that, no, you should use these high level constructs, not these terrible 'c-style' functions, and there will be zero cost - which is not true.

I often see vector naively promoted as being 'a better array' in a similar way, which of course it's not, their usages are different.

Ideally students of C++ coming into the industry should understand what creates performance - not coming from a viewpoint that high level is better, and then having to have those attitudes torn asunder by industry vets. The effort to promote C++ as a high level language with zero-cost abstractions does not gel with the games industry, or most of the industries focussed around performance. My reading on the general perception is that the direction which C++ is heading (high level abstractions attempting to cater to every scenario) does not match the expectations of the workforce."—End quote]

A nameof operator

There have been requests for an equivalent to the nameof operator found in C#. Ideally yielding contextual information such as point of declaration, calling function and such (these might be in part solved with upcoming static reflection features, work from SG7). It might be valuable to separate context information such as enclosing namespace or class from the rest.

Note: for the basic "nameof" functionality, see the C# language[2]. For the additional information that could be provided, see Static reflection.

Compile-time Error Detection

Things that help catch more errors at compile-time are looked upon favorably. There is hope that concepts will help in that regard.

Note: contributors have asked for more information as to what do compiler not see at compile-time but really should see.

Conditional Compilation

There is a need for something similar to if constexpr but that would allow removing #ifdef in multiplatform / multi-target code.

As an example, C# allows annotating a member function with [Conditional(opt)] to make that function conditionally defined, yet syntactically validated. The reported use-case for such a feature would be logging features and debug-only code. One could want some variable to be only defined when some compile-time conditions are met (such variables would typically be static).

Note: a contributor asked [*Begin quote—"Given that [we] already have `#ifdef`, what's the motivation for a new conditional-compilation facility? Why does it need replacing?"—End quote*]

Note from a contributor: [*Begin quote—"In C#, [Conditional("DEBUG")] causes the annotated function to compile, but actual calls to the annotated function are omitted if the symbol DEBUG doesn't exist. It's superior to #ifdef from a maintanability point of view since one doesn't need to*

[2] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/nameof

build all combinations of platforms and targets (in practice, impossible) to ensure a refactoring didn't break one of them. See also https://stackoverflow.com/a/3788719/1077814

Honestly I'd remove the reference to « if constexpr » from this section and make it just about adding the equivalent of a [Conditional("DEBUG")] feature to C++."—End quote]

Networking

Networking is something that every game engine has to implement by itself; a std:: version would be seen as something useful.

However, Boost ASIO seems heavy to many; at least providing a replacement for C sockets would be a huge win. Indeed, 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”.

Note from a contributor: [*Begin quote—“The feedback on "ASIO or not to ASIO?" seems less opinionated than I'd wish for. It sounds like the contributors don't realize that (AFAIK) Networking will be ASIO by default, and so it's not terribly useful to say "well, I don't need all of ASIO... really I just need some standard BSD-sockets stuff... I don't ask for much... never mind me..." :) because if you let the Networking group just do their own thing, I'm pretty sure you're not going to get any BSD-sockets stuff. It's all going to be io_contexts and executors and crap. (In particular, you seem to be conflating "the low-level std:: API for networking" with "the thing gamedevs will actually build on top of." What if the low-level std:: API is ASIO, and gamedevs don't want to build on top of ASIO? Would they continue to ignore it and build their own stuff next to it?) If you positively want sockets, speak louder. And if you positively want ASIO, also speak louder. And if different contributors want both, say that. And if nobody cares, say that. But it's hard to tell what position you're taking at the moment.”—End quote*]

Parallel and Concurrent Computing

Compile-time Evaluated Thread-Safety

There's currently no way in C++ to force resource management "Rust-Style". In a function, it would be useful to identify in code arguments that acquire or borrow a resource (read_only, read_write). The idea of a "Borrow Checker" seems interesting.

This can help in:

- Validating non-thread-safe operations at compile-time (Burst in C#, with the Unity Engine, does some of that).
- Writing "const classes" to create immutable views with language support (instead of through programming tricks and techniques). In Unity, one can indicate the access modes on a per-variable basis, which can help in validation and optimization.

Note: constexpr classes have been proposed for C++23 and rejected: https://wg21.link/p2350

In general, there is a desire for better facilities to debug multithreaded code. There exist various vendor-specific tools of good quality; the desire if for ways to enforce some checks in the language.

Naming, Tracing and Debugging

It's often useful to be able to name resources involved in concurrent code. Thus:

- We need a way to name a standard mutex. It is unpleasant to do so non-portably.
- We need a way to name a standard thread. It is unpleasant to do so non-portably.

It would in general be useful to provide structures that are left to implementation to provide more information to thread creation, including thread name, priority and stack size.

Note: https://wg21.link/p0484 and https://wg21.link/p0320 have done some work in that respect, and more recently https://wg21.link/p2019. There has been resistance to the question of controlling a thread's stack size through these efforts, so explaining this need more convincingly might be important

It would be useful to have some way to get some metadata from a mutex in order to know what's "protected" and what is not (Valgrind does this, it seems).

Note from a contributor: [*Begin quote—"Honestly, I've worked on at least two codebases that named their threads, and it was not unpleasant at all. It was like, you write one ten-line function `void set_current_thread_name(const std::string& name)` — or honestly, copy it from the last time you had to write it — and then you never look at that code again. It is the exact opposite of unpleasant. Yes, of the function's ten lines, about five are #ifdefs; but that's not a problem at all. Write once, never look at it again."—End quote*]

Note from a contributor: [*Begin quote—"Stack size is fundamentally different from name and priority and affinity, because you cannot change it on an existing thread; it needs to be a parameter to std::thread's constructor (or something like that). Name, etc., can all be set from within the thread after it's started running. You should reflect that difference in your discussion."—End quote*]

Note from a contributor: [*Begin quote—“"some metadata from a mutex" — This is basically Clang's thread-safety attributes?*

https://clang.llvm.org/docs/ThreadSafetyAnalysis.html

I've never used them nor have any special knowledge of them, but it sounds like what you want. It would be good to know whether any contributors have used them, whether they suffice or are lacking in some way (besides "not being standard yet"), etc.”—End quote]

Logging / IO

Many have asked for better logging facilities.

[*Begin quote—"In C# one can have optional attributes such as "who called you?" which can be useful for logging purposes. Knowing the context (class, function, namespace) at point-of-call is useful (std::source_location?). Ideally, it would be possible to go back three-to-four levels in a class' sequence; a stack trace might not be sufficient (std::stacktrace?)"—End quote*]

Note from a contributor: [*Begin quote—"Notice that adding logging to a function probably renders it non-[[pure]]."—End quote*]. However, as noted previously, that point is probably moot given past proposals on the subject.

Note: see also Static reflection.

Miscellaneous

A number of suggestions made do not fit well in the categories above.

Classes

Some convenience features with respect to classes might make coding more pleasant.

Forward Class Declarations with Inheritance

In some cases, it would be useful to be able to specify inheritance relations in a forward class declaration, e.g.:

```
class X : public Y;
```

This would allow using the forward-declared class in situations where a pointer or a reference to the base class is expected.

Example:

```
class X : ManagedObject;

class Y: X;

void Foo(ManagedObject*);

void Bar(Y *y) { Foo(y); }  // compiles with incomplete type
```

Note from a contributor: [*Begin quote—"As long as each base class is a complete type, I think this can be implemented. If you allow incomplete base classes, then it can't be done (or at least the results are not useful to the programmer).*

struct E {};

struct A;

struct B : E, A {};

*B *b = f();*

*A *a = b; // this cannot be compiled because we don't know the offset of B's A subobject: it depends on whether A derives from E or not*

There's also a syntactic problem: You can't express "I forward-declare that this class type exists and has no base classes." This is analogous to the problem C89 had in distinguishing between `int f();` and `int f(void);`: we've already gobbled up the obvious syntax and made it mean "this exists and I'm not saying anything about its base classes."."—End quote].

Note: a contributor has questioned whether this remained as relevant with the advent of modules.

namespace class

When defining a class' member functions in a .cpp file, repeating the class name everywhere can get tedious. If one could replace this:

```
class X {
```

```
   static const std::string S;
public:
   using type = int;
   X(type);
   type f() const;
};
// ... in the .cpp file
const std::string X::S = "...";
X::X(type) {
}
X::type X::f() const { // or auto X::f() const -> type
   return {};
}
```

… with that:

```
class X {
   static const std::string S;
public:
   using type = int;
   X(type);
   type f() const;
};
// ... in the .cpp file
namespace class X {
   const std::string S = "...";
   X(type) {
   }
   type f() const { // or auto f() const -> type
      return {};
   }
}
```

… it could reduce the noise somewhat.

Constrained Construction

An alternative to [[rvo]] (see [[rvo]]) would be to be able to constrain the number of constructors involved at the call site. Something such as (strawman syntax):

```
construct(1) auto a = f();
```

… where if there was more than one constructor involved in the call chain leading to the construction of object a, code would not compile.

Note: contributors have asked for more information as to how this would be useful to game developers or to those who develop for embedded systems.

Note: a contributor stated [*Begin quote—"Simply put, we're looking for ways to protect some important performance properties of our code against accidental upstream modifications. We want compilation to fail at the call site if someone (employee or vendor) accidentally modified a class somewhere that suddenly causes extra copies and allocations to happen. Otherwise small problems like this pile up and become big problems that are only caught much later when profiling, and we lost the context for the problematic change"—End quote*]

Enumerations

Note: see also Reflection on enums.

Flags-only Enums

There is a desire for enumerations that can only be flags (inspired by the flag attribute in C#). This could influence "stringification", particularly if two symbols have the same value.

It would also be useful to have ways to know define if non-power-of-two values are acceptable for a given enum type.

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 on Enums

There have been requests to allow member functions on enums. Of particular interest would be conversion operators:

```
enum class X : unsigned char {
   operator bool() const {
      return static_cast<std::underlying_type_t<T>>(*this) != 0;
   }
};
```

Better Support of Arrays with enum-Based Strong Types

It is the case that enum-based strong types and arrays mix unpleasantly, which blocks their adoption in some companies, as https://wandbox.org/permlink/dZvsd4MTz3WD7282 shows:

```
#include <utility>

#include <array>

int main() {

   using namespace std;

   [[maybe_unused]] byte b0{ 0 }; // ok

   // byte b1[]{ 0, 0 }; // nope

   [[maybe_unused]] byte b1[2]{ }; // ok

   [[maybe_unused]] array<byte,1> b2; // ok

   // array<byte,1> b3{ 0 }; // not ok

   [[maybe_unused]] array<byte,1> b4{ byte{} }; // ok

   [[maybe_unused]] array<byte,1> b5{ {} }; // ok

}
```

Note: contributors have offered criticism of std::byte and similar constructs and suggested using unsigned char instead.

Note: contributors have suggested workarounds, such as a macro, e.g.:

```
#define B(x) std::byte(x)

std::byte array[] = { B(0), B(1), B(42), B(0); };
```

… or a user-defined literal:

```
auto operator""_b(int v) { return std::byte(v); }

std::byte array[] = { 0_b, 1_b, 42_b, 0_b };
```

Improvements to std::initializer_list

It's now known that std::initializer_list is a tool with some rough edges, which might benefit from some "smoothening".

Making std::initializer_list Movable

There have been requests for the addition of move operations on std::initializer_lists. 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>().

Note: this has been proposed in the past, in https://wg21.link/p0065

Explicit list-initialization

People have been 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.

[*Begin quote—"We had to forbid the usage of std::initializer_list as its greedy nature can cause regressions in code:*

```
struct MyString {
   MyString(~~char~~ int chr, int count);
   // ...
};
MyString indent{' ', 4};
// Cannot add MyString(std::initializer_list<int>) anymore
```

It's not only here that adding a std::initializer_list could cause regression in current code. We work in an organic environment. Code gets integrated between branches, both inside the same project (i.e. game) or different projects. It means we cannot search for all usages of { } for constructors before introducing a new one. To fix the issue, we use that simple class instead:

```
template<class T>
struct explicit_init_list {
   std::initializer_list<T> m_InitList;
   constexpr explicit_init_list(std::initializer_list<T> initList)
      : m_InitList(initList) {}
    constexpr const T* begin() const { return m_InitList.begin(); }
    constexpr const T* end() const { return m_InitList.end(); }
    constexpr std::size_t size() const { return m_InitList.size(); }
};
```

"*—End quote*]

Note: contributors have suggested never using braces unless one really wants list initialization, but admitting this can be hard to enforce over a large codebase.

Downcasting

There is a need for a way to downcast to the most-derived type at low cost. [*Begin quote—"We would really like to have sorted vtables for statically linked .exe and have a fast down_cast operator"—end quote*].

Additional context: suppose a statically linked executable. In a single inheritance hierarchy, a fast down_cast can be a single load to obtain the value of the vtable ptr; if the vtables are sorted in memory according to the class hierarchy, the down_cast implementation can be as simple looking if the vtable pointer is pointing in the appropriate range:

```
template <typename T, typename BaseT>
T* down_cast(BaseT *obj) {
   T* possible_result =  static_cast<T*>(obj);
   ptrdiff_t vtable_ptr = __get_vtable_ptr(possible_result);
   // note: linker inserts values for first and last vtable ptrs
   return (vtable_ptr >= __first_vtable_ptr<Type> &&
           vtable_ptr <= __last_vtable_ptr<Type>) ? possible_result : nullptr;
}
```

Note: some companies have their own RTTI with homemade tricks to perform dest = down_cast<T>(src) which yields nullptr in the case of an invalid cast.

Note: contributors have asked for an actual example, and have asked if this corresponds to the workaround mentioned (note that this uses typeid, which requires standard RTTI):

```
template<class MDT, class Base>
   MDT *downcast(Base *p) {
      return static_cast<MDT*>(
         typeid(*p) == typeid(MDT) ? p : nullptr
      );
}
```

Reply from a contributor: [*Begin quote—"No, that's not it. We can achieve assembly code limited to a load of a vtable, but vtables have to be sorted beforehand so it has to be wired in the compiler to be portable (we write it per-compiler)."—end quote*]

Covariant Cloning

Currently, C++ supports covariant return types, but this does not extend well to smart pointers. The main need would be for std::unique_ptr (it's unclear if shared_ptr needs this), in order to replace this:

```
struct B {
   virtual B *clone() const { // could cause a leak
      return new B{ *this };
   }
   virtual ~B() = default;
   // ...
};
struct D : B {
   // ok (covariant return type), but could cause a leak
   D *clone() const override {
      return new D{ *this };
   }
   // ...
};
```

… which could leak, with this:

```
struct B {
   virtual std::unique_ptr<B> clone() const { // ok
      return std::make_unique<B>( *this );
   }
   virtual ~B() = default;
   // ...
};
struct D : B {
   // would not compile in C++20
   std::unique_ptr<D> clone() const override { // not ok
      return std::make_unique<D>( *this );
   }
   // ...
};
```

… which would be safe but is currently illegal.

Note: there have been discussions of this in the past (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 – something C++ does not have as of C++20).

Homogeneous Variadics

There are cases where one wants to express the fact that a pack's members all have to be of the same type. Something like:

```
template <class T> void f(T...) { /* ... */ }
```

Note: this can be achieved already, e.g. with

```
#include <concepts>

template <class T, class ...Ts> requires (std::same_as<T, Ts> && ...)
   void f(T, Ts...){ /* ... */ }

int main() {
   f(3, 4, 5); // ok
   // f(3, 4.0, 5); // not ok
}
```

… in C++20: https://wandbox.org/permlink/f2TasMibAYysw2pM as well as with std::conjunction<T…> in C++17.

More mandatory elisions and (N)RVO

There is support for the addition of mandatory elision cases as expressed in https://wg21.link/p2025

Named arguments

There is support for named arguments, for which https://wg21.link/n4172 has been mentioned.

Note: this need might be partly covered by the addition of designated initializers to C++20. See https://wg21.link/p0329 for details.

SoA to AoS

The fact that 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) is well-known to the game development community.

If there was a way to "transform" something expressed as an AoS into its SoA equivalent, it would be an appreciated feature of the language.

Note: contributors have reacted warmly to this, and have stated this section deserves more attention. An example of strawman syntax to simplify discussion would help.

Unified call syntax

Interest in some uniform call syntax has been brought up in a number of discussions.

Tooling and ease of use have been mentioned as 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.

Game engines tend to use member functions more than free functions, apparently.

Note: might Deducing this be helpful here? https://wg21.link/p0847

Note: some contributors think this ship had sailed or sunk already.

Numeric Computing / Linear Algebra

There is support for efforts in providing foundational types for linear algebra, such as what is proposed by https://wg21.link/p1385

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.

Fast transpose for matrices is seen as important. Conversions should be mostly cost-free. Functions for min, max, bool functions on vectors, etc. are seen as important. In general, it would be good if what can be done in a language such as HLSL could be done directly in C++.

Note: contributors have asked for some precision on what "bool functions on vectors" means.

Opt-in UB on Unsigned Overflow

There is a need for an integral type (at least the 32 bits flavor) for which overflow would be UB. Most game companies use their own aliases for types; the intent here would be to change that alias from a "classic" unsigned integral where overflow is defined to this new type, to see if performance gains could be achieved.

Note from a contributor: [*Begin quote—"I believe this can be done in a library. Has any contributor actually experimented with this? If someone provided them with a header-only library with a UB-on-overflow `ouint32`, `ouint16`, etc., would they actually try it out and provide feedback?."—End quote*]

Note from a contributor: [*Begin quote—"After testing on godbolt with a colleague, it seems not as many optimization opportunities as we initially thought are possible when unsigned does not support overflow. We found a single optimization that could occur, and it was not present with common usages of unsigned. While I think it makes sense semantically to provide an unsigned without overflow, and 99%+ of our usages of unsigned do not need overflow, it's not clear there's a real gain here.*

It might be too early to remove this from the document, but implementing it in Clang could confirm if it is actually useful. If I can't justify implementing myself in Clang, I wonder If we should propose it to the standard, unless we do it for semantics, not performance"—End quote]