Making C++ Better for Game Developers – Some Requests

Author: Patrice Roy (based on suggestions made from various experts from the game development domain)

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.

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, to 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?

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)
- **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 you 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.

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):

```
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().

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: run-time string interpolation would be appreciated too. If the same syntax (e.g.: $"Val: {variable}" and $"Val: {constant}") could be used for both compile-time and run-time string interpolation, it would be ideal.

Note: this might be covered in part by std::format() (https://wg21.link/p2216)

Traits

Adding a std::is_complete<T> or a std::is_complete_type_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
}
```

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 };
}
```

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 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 implementations experience.

Containers

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

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 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.

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.

Heap-Free Functions

Adding heap-free options to all situations that might lead to dynamic memory allocation (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

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

Attributes

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

Suport 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”.

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.

[[invalid_refererencing]]

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().

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

Note: this is currently QoI.

[[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 *ptr to be invalid after the call to free().

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

Note: this is currently QoI.

[[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: 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.

Note: maybe this is already covered by rvalue reference arguments and proper definition of copy and move special functions

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

[[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]]?

[[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]

Note: some game development companies specialize std::move() to explicitly reject const T&&.

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".

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. This, 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) amout 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: 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?); 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.

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.

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.

A nameof operator

An equivalent of 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). 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. 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.

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).

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.

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.

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).

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: 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
```

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.

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

}
```

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 chr, int count);

   // ...

};

MyString indent{' ', 4};

// Cannot add MyString(std::initializer_list) 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*]

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*].

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

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
   virtual 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
   virtual std::unique_ptr<D> clone() const { // 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

Forwarder (WIP)

There is support for a simpler way to express forwarding. The T~(waving the value along) simplified syntax has been mentioned.

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.

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.

"#ifdef-like" if constexpr

The fact that code in the not-taken branch of an if constexpr construct has to be valid is sometimes annoying. There have been reports of "sadness" using macro expansions or trying to replace SFINAE tricks with if constexpr. There might be room to explore here; something that would be distinct from if constexpr, however, as if constexpr being as it is today is seen as something very useful.

Note: might concept overloading suffice here?

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 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++.

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.